Automatic Representation Changes in Problem Solving

Eugene Fink

June 1999

CMU-CS-99-150

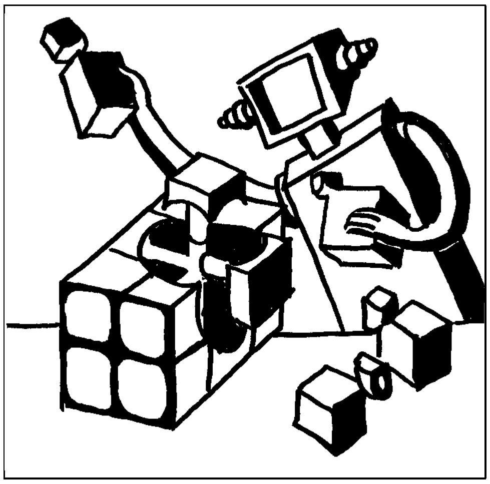
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Herbert A. Simon, Jaime G. Carbonell, Manuela M. Veloso,
and Richard E. Korf (University of California at Los Angeles).

The work was sponsored by the Defense Advanced Research Projects Agency (DARPA) via the Navy, under grant F33615-93-1-1330, and the Air Force Research Laboratory, under grant F30602-97-1-0215. The author's views and conclusions should not be interpreted as policies of DARPA or the U.S. government.

Keywords: Machine learning, problem solving, planning, automatic problem reformulation, primary effects, abstraction, PRODIGY.

Contents



Abstract

The purpose of our research is to enhance the efficiency of AI problem solvers by automating representation changes. We have developed a system that improves description of input problems and selects an appropriate search algorithm for each given problem.

Motivation Researchers have accumulated much evidence of the importance of appropriate representations for the efficiency of AI systems. The same problem may be easy or difficult, depending on the way we describe it and on the search algorithm we use. Previous work on automatic improvement of problem description has mostly been limited to the design of individual learning algorithms. The user has traditionally been responsible for the choice of algorithms appropriate for a given problem.

We present a system that *integrates* multiple description-changing and problem-solving algorithms. The purpose of our work is to formalize the concept of representation, explore its role in problem solving, and confirm the following general hypothesis:

An effective representation-changing system can be constructed out of three parts:

- *a library of problem-solving algorithms;*
- *a library of algorithms that improve problem description by static analysis and learning;*
- *a top-level control module that selects appropriate algorithms for each given problem.*

Representation-changing system We have supported this hypothesis by building a system that improves representations in the PRODIGY problem-solving architecture. The library of problem solvers consists of several search engines available in the PRODIGY architecture. The library of description changers comprises novel algorithms for selecting primary effects, generating abstractions, and discarding irrelevant elements of a problem encoding. The control module chooses and applies appropriate description changers, stores and reuses new descriptions, and selects problem solvers.

Improving problem description The implemented system includes seven static-analysis and learning algorithms for improving description of a given problem. First, we formalize the notion of primary effects of operators, and give two algorithms for identifying primary effects. Second, we extend the theory of abstraction search to the PRODIGY domain language, and describe two techniques for abstracting preconditions and effects of operators. Third, we present auxiliary algorithms that enhance the power of abstraction, by identifying relevant features of a problem and generating partial instantiations of operators.

Top-level control We define a space of possible representations of a given problem, and view the task of changing representation as search in this space. The top-level control mechanism guides the search, using statistical analysis of previous results, user-coded control rules, and general heuristics. First, we formalize the statistical problem involved in finding an effective representation and derive a solution to this problem. Then, we describe control rules for selecting representations, and present a mechanism for the synergetic use of statistical techniques, control rules, and heuristics.

Acknowledgments

The reported work was a result of my good fortune of being a student at Carnegie Mellon University, which gave me freedom and support to pursue my inquiries. I gratefully acknowledge the help of my advisors, co-workers, and friends, who greatly contributed to my work and supported me during six long years of my graduate studies.

My greatest debt is to Herbert Alexander Simon, Jaime Guillermo Carbonell, and Maria Manuela Magalhães de Albuquerque Veloso, who provided so much stimulating ideas, guidance, and advice to my research that it is no longer possible to segregate the ideas of the thesis exploration into those that were theirs, those of the author, and those developed jointly.

They guided me through my years in the graduate school, and provided invaluable help and support in all aspects of my work, from the strategic course of my research to minute details of implementation and writing. They helped me to broaden my knowledge and, at the same time, stay focussed on my thesis research.

I am grateful to Derick Wood, Qiang Yang, and Jo Ebergen, who guided my research before I entered Carnegie Mellon. Derick Wood and Qiang Yang also supervised my work during the three summers that I spent away from Carnegie Mellon, after entering the Ph.D. program. They taught me research and writing skills, which proved invaluable in my thesis work.

I am also grateful to my undergraduate teachers of math and computer science, especially to my advisor Robert Rosebrugh, who led me through the woods of my coursework and encouraged me to pursue a graduate degree.

I would like to thank my first teachers of science, Maria Yurievna Filina, Nikolai Moiseevich Kuksa, and Alexander Sergeevich Golovanov. Back in Russia, they introduced me into the wonderful world of mathematics, developed my taste for learning and research, and gave an impetus to my career.

I am thankful to Richard Korf for his valuable comments on my thesis research and writing. I have also received thorough feedback from Catherine Kaidanova, Henry Rawley, and Karen Haigh.

I have done my work in the stimulating research environment of the PRODIGY group. I fondly remember my meetings and discussions with members of the group, including Jim Blythe, Daniel Borrajo, Michael Cox, Rujith DeSilva, Rob Driskill, Karen Haigh, Vera Kettnaker, Craig Knoblock, Erica Melis, Steven Minton, Alicia Pèrez, Paola Rizzo, Yury Smirnov, Peter Stone, and Mei Wang[1]. My special thanks are to Jim Blythe, who helped me to understand PRODIGY code and adapt it for my system. I also thank Yury Smirnov for his feedback and constructive criticism.

Svetlana Vainer, a graduate student in mathematics, aided me in constructing the statistical model used in my system. She helped me to acquire the necessary background and guided me through the forest of statistical derivations. Evgenia Nayberg, a fine-arts student, assisted me in designing some illustrations.

I have received a lot of support and encouragement from fellow graduate students, Claudson Bornstein, Tammy Carter, Karen Haigh, Nevin Heintze, Bob Monroe, Henry Rawley, and Po-Jen Yang[1]. I am especially thankful to Nevin Heintze for his help to adapt to the environment of the Computer Science Department during my first year of studies, and for encouraging me to write a thesis proposal during my third year; to Karen Haigh, for her support in understanding the American culture; and to Henry Rowley and Bob Monroe, for their help with innumerable software problems.

I am grateful to my parents, who provided help and support all through my studies, in spite of their original negative attitude toward my immigration and intensive scientific work.

Finally, I thank my friends outside of the Computer Science. Natalie Gurevich, Alex Gurevich, and Lala Matievsky played especially important role in my life. They encouraged me to immigrate from Russia and helped me to form my priorities and objectives. I have received much support and encouragement from my other ex-Russian friends, Catherine Kaidanova, Natalia Kamneva, Alissa Kaplunova, Alexander Lakher, Irina Martynov, Alex Mikhailov, Evgenia Nayberg, Michael Ratner, and Viktoria Suponiskaya[1]. I am also thankful to my Canadian friends, Elverne Bauman, Louis Choiniere, Margie Roxborough, Marty Sulek, Alison Syme, and Linda Wood[1], who helped me to learn and accept the culture of Canada and the United States.

[1]The names are in the alphabetical order.

Part I

Introduction

Chapter 1

Motivation

Could you restate the problem? Could you restate it still differently?
— George Polya [1957], *How to Solve It.*

The performance of all reasoning systems crucially depends on problem representation: the same problem may be easy or difficult to solve, depending on the way we describe it. Researchers in psychology, cognitive science, artificial intelligence, and many other areas have accumulated much evidence on the importance of appropriate representations for human problem solvers and AI systems.

In particular, psychologists have found out that human subjects often simplify hard problems, by changing their representation, and that the ability to find an appropriate problem reformulation is a crucial skill for mathematicians, physicists, economists, and experts in many other areas. AI researchers have demonstrated the impact of changes in problem description on the performance of search systems, and showed the need for automating problem reformulation.

Although researchers have long realized the importance of effective representations, they have done little investigation in this area, and the notion of "good" representations has remained at an informal level. The human user has traditionally been responsible for providing appropriate problem descriptions, as well as for selecting search algorithms that effectively use these descriptions.

The purpose of our work is to automate the process of revising problem representation in AI systems. We formalize the concept of representation, explore its role in problem solving, and develop a system that evaluates and improves representations in the PRODIGY problem-solving architecture.

The work on the system for changing representations has consisted of two main stages, described in Parts II and III. First, we outline a framework for the development of algorithms that improve problem descriptions, and apply it to designing several novel algorithms. Second, we construct an *integrated AI system*, which utilizes the available description improvers and PRODIGY problem-solving algorithms. The system is named $\mathcal{S}$HAPER, for its ability to change the shape of problems and their search spaces. We did not plan this name to be an acronym; however, it may be retroactively deciphered as *Synergy of Hierarchical Abstraction, Primary Effects, and other Representations.* The central component of the $\mathcal{S}$HAPER system is a top-level control module, which selects appropriate algorithms for each given problem.

We begin by explaining the concept of representations in problem solving (Section 1.1), illustrating their impact on problem complexity (Section 1.2), and reviewing the previous work on representation changes (Section 1.3). We then outline our approach to the automation of representation improvements (Section 1.4) and give a summary of the main results (Section 1.5).

1.1 Representations in problem solving

Informally, a *problem representation* is a certain view of a problem and approach to solving it, which determines the efficiency of search for a solution. Scientists have considered different formalizations of this concept, and its exact meaning varies across research contexts. The representation of a problem in an AI system may include the initial encoding of the problem, data structures for storing relevant information, production rules for drawing inferences about the problem, and heuristics that guide the search for a solution.

We explain the meaning of representation in our research and introduce related terminology. First, we give informal examples that illustrate this notion (Section 1.1.1). Second, we review several alternative formalizations (Section 1.1.3) and define the main notions used in the work on the $\mathcal{S}$HAPER system (Section 1.1.2). Third, we discuss the role of representation in problem solving (Section 1.1.4).

1.1.1 Informal examples

We consider two examples that illustrate the reasons for using multiple representations. In Section 1.2, we will give a more technical example, which involves representation changes in the PRODIGY architecture.

Representations in geometry

Mathematicians have long developed the art of constructing and fine-tuning sophisticated representations, which is one of their main tools for addressing complex research tasks [Polya, 1957]. For example, when a scientist works on a hard geometry problem, she usually tries multiple approaches, such as pictorial reasoning, analytical techniques, trigonometric derivations, and computer simulations.

These approaches differ not only in the problem encoding, but also in the operations for transforming its encoding, as well as in related mental structures and high-level strategies [Qin and Simon, 1992]. For example, the mental techniques for analyzing geometric sketches are very different from the methods for solving trigonometric equations.

A mathematician may have to try many alternative representations of the given problem, and go back and forth among promising approaches [Kaplan and Simon, 1990]. For instance, she may consider several pictorial representations, then try analytical techniques, and then give up on her analytical model and go back to one of the pictures.

If several different representations provide useful information about the problem, the mathematician may use them in parallel and combine the resulting inferences. This synergetic use of alternative representations is a standard mathematical technique. In particular, proofs of geometric results often include equations along with pictorial arguments.

Search for an appropriate representation is based on two main processes: retrieval or construction of candidate representations, and evaluation of their utility. The first process may involve look-up of a matching representation in the library of available strategies, modification of an "almost" matching representation for use with the new problem, or development of a completely new approach. For example, the mathematician may re-use an old sketch, draw a new one, or even devise a new framework for solving this type of problems.

After constructing a new representation, the mathematician estimates its usefulness for solving the problem. If the representation does not look promising, she may prune it right away, or store it as a back-up alternative; for example, she may discard the sketches that clearly do not help. Otherwise, she tries to use the representation in problem solving and evaluates the usefulness of the resulting inferences.

To summarize, different representations of a given problem support different inference techniques, and the choice among them determines the effectiveness of the problem-solving process. Construction of an appropriate representation may be a difficult task, which requires search in a certain space of alternative representations.

Driving directions

We next give an example of representation changes in everyday life, and show that the choice of representation may be important for simple tasks. In this example, we consider the use of directions for driving to an unfamiliar place.

Most drivers employ several standard techniques for describing a route, such as a sketch of the streets that form the route, pencil marks on a city map, and verbal directions for reaching the destination. When a driver chooses one of these techniques, she commits to certain mental structures and strategies. For instance, if the driver navigates by a map, then she has to process pictorial information and use imagery for matching it to the real world. On the other hand, the execution of verbal instructions requires discipline in following the described steps and attention to the relevant landmarks.

When the driver selects a representation, she should consider her goals, the effectiveness of alternative representations for achieving these goals, and the related trade-offs. For instance, she may describe the destination by its address, which is a convenient way for recording it in a notebook or quickly communicating to others; however, the address alone may not be sufficient for finding the place without a map. The use of accurate verbal directions is probably the most convenient way for reaching the destination, without stopping to look at the map. On the other hand, the map may help to identify points of interest close to the route; moreover, it becomes an invaluable tool if the driver gets lost.

If an appropriate representation is not available, the driver may construct it from other representations. For example, if she has the destination address, then she may find a route on the map, and then write down directions that help to follow this route. When people consider these representation changes, they often weigh the expected simplification of the task against the cost of performing the changes. For instance, even if the driver believes that written directions facilitate the trip, she may decide they are not worth the writing effort.

A representation...

- includes a machine language for the description of reasoning tasks and a specific encoding of a given problem in this language [Amarel, 1968].
- is the space expanded by a solver algorithm during its search for a solution [Newell and Simon, 1972].
- is the state space of a given problem, formed by all legal states of the simulated world and transitions between them [Korf, 1980].
- "consists of both data structures and programs operating on them to make new inferences" [Larkin and Simon, 1987, page 67].
- determines a mapping from the behavior of an AI system on a certain set of inputs to the behavior of another system, which performs the same task on a similar set of inputs [Holte, 1988].

Figure 1.1: Different definitions of representation, in artificial intelligence and cognitive science; note that these definitions are *not* equivalent, and thus they give rise to different formal models.

To summarize, this example shows that people employ multiple representations not only for complex problems, but also for routine tasks. When people repeatedly perform some task, they develop standard representations and routine techniques for constructing them. Moreover, the familiarity with the task facilitates the selection among available representations.

1.1.2 Alternative definitions of representation

Even though AI researchers agree in their intuitive understanding of representation, they have not yet developed a standard formalization of this notion. We review several formal models, used in artificial intelligence and cognitive science, and discuss their similarities and differences; in Figure 1.1, we summarize the main definitions.

Problem formulation

Amarel [1961; 1965; 1968] was first to point out impact of representation on the efficiency of search algorithms. He considered some problems of reasoning about actions in a simulated world, and discussed their alternative formulations in the input language of a search algorithm. The discussion included two types of representation changes: modifying the encoding of a problem and translating it to different languages.

In particular, he demonstrated that a specific formulation of a problem determines its *state space,* that is, the space of possible states of the simulated world and transitions between them. We illustrate this notion in Figure 1.7 (page 17), which shows the full space of the Tower-of-Hanoi puzzle. Amarel pointed out that the efficiency of problem-solving algorithms depends on the size of the state space, as well as on the allowed transitions, and that change of a description language may help to reveal hidden properties of the simulated world.

Van Baalen [1989] adopted a similar view in his doctoral work on a theory of representation design. He defined a representation as a mapping from concepts to their syntactic description in a formal language and implemented a program that automatically improves descriptions of simple reasoning tasks.

Problem space

Newell and Simon [1972] investigated the role of representation in human problem solving. In particular, they observed that the human subject always encodes a given task in a *problem space,* that is, "some kind of space that represents the initial situation presented to him, the desired goal situation, various intermediate states, imagined or experienced, as well as any concepts he uses to describe these situations to himself" (*Human Problem Solving,* page 59).

They defined a representation as the subject's problem space, which determines partial solutions considered by the human solver during his search for a complete solution. This definition is applicable not only to human subjects but also to AI systems, since all problem-solving algorithms are based on the same principle of searching among partial solutions.

Observe that the problem space may differ from the state space of the simulated world. In particular, the subject may disregard some of the allowed transitions and, on the other hand, consider impossible world states. For instance, when people work on hard versions of the Tower of Hanoi, they sometimes attempt illegal moves [Simon *et al.*, 1985]. Moreover, the problem solver may abstract from the search among world states and use some alternative view of partial solutions (for example, see the textbook by Rich and Knight [1991]). In particular, the search algorithms in the PRODIGY architecture explore the space of transition sequences (see Section 2.2), which is very different from the space of world states.

State space

Korf [1980] described a formal framework for changing problem representations and used it in designing a system for automatic improvement of the initial representation. He developed a language for describing search problems, and defined a representation as a specific encoding of a given problem in this language. The encoding includes the initial state of the simulated world and operations for transforming the state; hence, it defines the state space of the problem.

Korf has pointed out the correspondence between the problem encoding and the resulting state space, which allowed him to view a representation as a space of named states and transitions between them. This view underlies his techniques for changing representation. In particular, he has defined a representation change as a transformation of the state space, and considers two main types of transformations, called *isomorphism* and *homomorphism.* An isomorphic representation change involves renaming the states without affecting the structure of the space. On the other hand, a homomorphic transformation is a reduction of the space, by abstracting some states and transitions.

Observe that Korf's notion of representation does *not* include the behavior of a problem-solving algorithm. Since performance depends not only on the state space, but also on the search strategies, a representation in his model does not uniquely determine the efficiency of problem solving.

Data and programs

Simon suggested a general definition of representation as "data structures and programs operating on them," and used it in the analysis of reasoning with pictorial representations [Larkin and Simon, 1987]. When describing the behavior of human solvers, he viewed their initial encoding of a given problem as a "data structure," and the available productions for modifying it as "programs." Since the problem encoding and rules for changing it determine the subject's search space, this view is similar to the earlier definition in *Human Problem Solving.*

If we apply Simon's definition in other research contexts, the notions of data structures and programs may take different meanings. The general concept of "data structures" encompasses any form of a system's input and internal representation of related information. Similarly, the term "programs" may refer to any strategies and procedures for processing a given problem and constructing its solution.

In particular, when considering an AI architecture with several search engines, we may view the available engines as "programs" and the information passed among them as "data structures." We will use this approach to formalize representation changes in the PRODIGY system.

System's behavior

Holte [1988] developed a framework for analysis and comparison of learning systems, which included rigorous mathematical definitions of task domains and their alternative representations. He considered representations of domains rather than specific problems, which distinguished his view from the earlier definitions.

A domain in Holte's framework includes a set of certain elementary entities, a collection of primitive functions that describe the relations among entities, and legal compositions of primitive functions. For example, we may view the world states as elementary objects and transitions between them as primitive functions. A domain specification may include not only a description of reasoning tasks, but also a particular behavior of an AI system on these tasks.

A representation is a mapping between two domains that encode the same reasoning tasks. This mapping may describe a system's behavior on two different encodings of a problem. Alternatively, it may show the correspondence between the behavior of two different systems that perform the same task.

1.1.3 Representations in the Shaper system

The previous definitions of representation have been aimed at the analysis of its role in problem solving, but researchers have not applied theoretical results to automating representation changes. Korf utilized his formal model in the development of a general-purpose

A **problem solver** is an algorithm that performs some type of reasoning task. When we invoke this algorithm, it inputs a given problem and performs search for a solution, which results in either solving the problem or reporting a failure.

A **problem description** is an input to a problem solver. In most systems, it includes a list of allowed operations, available objects, initial state of the world, logical statement describing the goals, and possibly some heuristics for guiding the search.

A **domain description** is the part of the problem description that is common for a certain class of problems. It usually does *not* include specific objects, initial state, and goal.

A **representation** is a domain description with a problem solver that uses this description. A representation change may involve improving a description, selecting a new solver, or both.

A **description changer** is an algorithm for improving domain descriptions. When we invoke the changer algorithm, it inputs a given domain and modifies its description.

A **system for changing representations** is an AI system that automatically improves domain descriptions and matches them with appropriate problem solvers.

Figure 1.2: Definitions of the main objects in the SHAPER system; these notions underlie our formal model, aimed at developing an AI architecture for automatic representation changes.

system for improving representations, but encountered several practical shortcomings of the model, which prevented complete automation of search for appropriate representations.

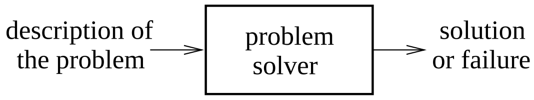
Since the main purpose of our work is the construction of a fully automated system, we develop a different formal model, which facilitates the work on SHAPER. We follow Simon's view of representation as "data structures and programs operating on them;" however, the notion of data structures and programs in the SHAPER system differs from their definition in the research on human problem solving [Newell and Simon, 1972]. We summarize our terminology in Figure 1.2.

The SHAPER system uses PRODIGY search algorithms, which play the role of "programs" in Simon's definition. We illustrate the functions of a solver algorithm in Figure 1.3(a): given a problem, the algorithm searches for its solution, and either finds some solution or terminates with a failure. In Chapter 7, we will discuss two main types of failures: exhausting the available search space and reaching a time bound.

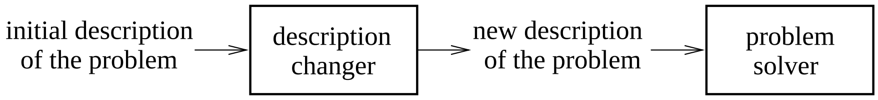
A *problem description* is an input to the solver algorithm, which encodes a certain reasoning task. This notion is analogous to Amarel's "problem formulation," which is a part of his definition of representation. The solver's input must satisfy certain syntactic and semantic rules, which form the *input language* of the algorithm.

When the initial description of a problem does not obey these rules, we have to translate it into the input language before applying the solver [Paige and Simon, 1966; Hayes and Simon, 1974]. If a description satisfies the language rules but causes a long search for solution, we may need to modify it for efficiency reasons.

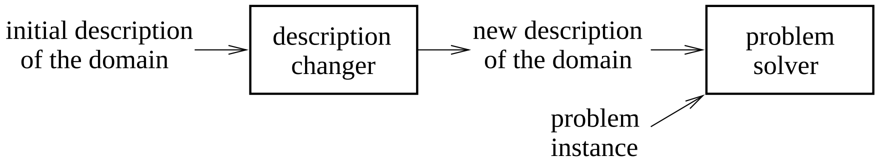
A *description-changing algorithm* is a procedure for converting the initial problem description into an input to a problem solver, as illustrated in Figure 1.3(b). The conversion may serve two goals: (1) translating the problem into the input language of the problem



(a) Use of a problem-solving algorithm.



(b) Changing the problem description before application of a problem solver.



(c) Changing the domain description.

Figure 1.3: Description changes in problem solving: a changer algorithm generates a new description and then a solver algorithm uses it to search for a solution.

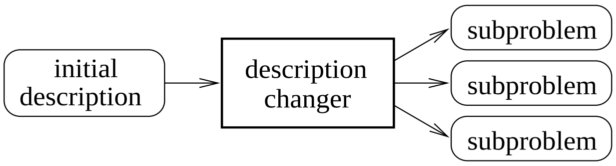
solver and (2) improving performance of the solver.

The $\mathcal{S}$HAPER system performs only the second type of description changes. In Figure 1.4, we show the three main categories of these changes: decomposing the initial problem into smaller subproblems, enhancing the description by adding relevant information, and replacing the original problem encoding with a more appropriate encoding.

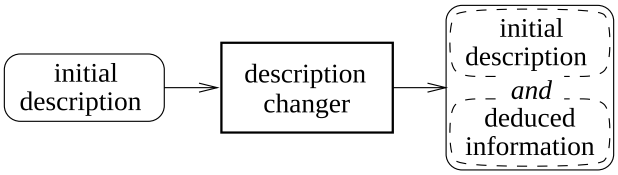
Note that there are no clear-cut boundaries between these categories. For example, suppose that we apply some *abstraction procedure* (see Section 1.2), which determines the relative importance of different problem features, and then uses important features in constructing an outline of a solution. We may view it as enhancement of the initial description with the estimates of importance. Alternatively, we may classify abstraction as decomposition of the original problem into two subproblems: constructing a solution outline and then turning it into a complete solution.

A problem description in the PRODIGY architecture consists of two main parts, a *domain description* and *problem instance.* The first part comprises the properties of a simulated world, which is called the *problem domain.* For example, if we apply PRODIGY to solve the Tower-of-Hanoi puzzle (see Section 1.2.1), then the domain description specifies the legal moves in this puzzle. The second part encodes a particular reasoning task, which includes an initial state of the simulated world and a goal specification. For example, a problem instance in the Tower-of-Hanoi Domain consists of the initial positions of all disks and their desired final positions.

The PRODIGY system first parses the domain description and converts it into internal structures that encode the simulated world. Then, PRODIGY uses this internal encoding in processing specific problem instances. Observe that the input description determines the



(a) Decomposing a problem into subproblems: We may often simplify a reasoning task by breaking it into smaller subtasks. For example, a driver may subdivide search for an unfamiliar place into two stages: getting to the appropriate highway exit and then finding her way from the exit. In Section 1.2.2, we will give a technical example of problem decomposition, based on an abstraction hierarchy.



(b) Enhancing a problem description: If some important information is not explicit in the initial description, we may deduce it and add to the description. If the addition of new information affects the problem-solving behavior, we view it as a description change. For instance, a mathematician may enhance a geometry sketch by an auxiliary construction, which reveals hidden properties of the geometric object. As another example, we may improve performance of PRODIGY by adding control rules.

(c) Replacing a problem description: If the initial description contains unnecessary data, then improvements may include not only addition of relevant information, but also deletion of irrelevant data. For example, a mathematician may simplify her sketch by erasing some lines. In Section 5.3, we will describe a technique for detecting irrelevant features of PRODIGY problems.

Figure 1.4: Main categories of description changes: We may (a) subdivide the original problem into smaller reasoning tasks, (b) extend the initial description with additional knowledge, and (c) replace the given problem encoding with a more effective encoding.

system's internal encoding of the domain; thus, the role of domain descriptions in our model is similar to that of "data structures" in the general definition.

When using a description changer, we usually apply it to the domain encoding and utilize the resulting new encoding for solving multiple problem instances (see Figure 1.3c). This strategy reduces the computational cost of description improvements, since it allows us to amortize the running time of the changer algorithm over several problems.

A *representation* in the SHAPER system consists of a domain description *and* a problem-solving algorithm that operates on this description. Observe that, if the algorithm does not make any random choices, then the representation uniquely defines the search space for every problem instance. This observation relates our definition to Newell and Simon's view of representation as a search space.

We use this definition in our work on a *representation-changing system,* which automates the two main tasks involved in improving representations. First, it analyzes and modifies the initial domain description, with the purpose of improving the search efficiency. Second, it selects an appropriate solver algorithm for the modified domain description.

1.1.4 The role of representation

Researchers have used several different frameworks for defining and investigating the concept of representation. Despite these differences, most investigators have reached consensus on the main qualitative conclusions:

- The choice of a representation affects the complexity of a given problem; both human subjects and AI systems are sensitive to changes in the problem representation

- Finding the right approach to a given problem is often a difficult task, which may require a heuristic search in a space of alternative representations

- Human experts employ advanced techniques for construction and evaluation of new representations, whereas amateurs often try to utilize the original problem description

Alternative representations differ in explicit information about properties of the problem domain. Every representation hides some features of the domain, and highlights other features [Newell, 1965; Van Baalen, 1989; Peterson, 1994]. For example, when a mathematician describes a geometric object by a set of equations, she hides visual features of the object and highlights some of its analytical properties.

Explicit representation of important information enhances performance of problem-solving systems. For instance, if a student of mathematics cannot solve some problem, the teacher may help her by pointing out the relevant features of the task [Polya, 1957]. As another example, we may improve efficiency of an AI system by encoding useful information in control rules [Minton, 1988], macro operators [Fikes *et al.*, 1972], or an abstraction hierarchy [Sacerdoti, 1974].

On the other hand, explicit representation of irrelevant data may have a negative effect. In particular, when a mathematician tries to utilize some *seemingly* relevant properties of a given problem, she may attempt a wrong approach.

If we provide irrelevant information to an AI system and do *not* mark this information as unimportant for the current task, then the system attempts to use it, which takes extra computation and often leads to exploring useless branches of the search space. For example, if we allow use of unnecessary extra operations, then the branching factor of search increases, which usually results in a larger search time [Stone and Veloso, 1994].

Since problem-solving algorithms differ in their use of available information, they perform efficiently with different domain descriptions. Moreover, the utility of explicit knowledge about the domain may depend on a specific problem instance. We usually cannot find a "universal" description, which works well for all solver algorithms and problem instances. The task of constructing good descriptions has traditionally been left to the user.

The relative performance of solver algorithms also depends on specific problems. Most analytical and experimental studies have shown that different search techniques are effective for different classes of problems, and no solver algorithm can consistently outperform all its competitors [Minton *et al.*, 1994; Stone *et al.*, 1994; Knoblock and Yang, 1994; Knoblock and Yang, 1995; Smirnov, 1997]. To ensure efficiency, the user has to make an appropriate selection among the available algorithms.

To address the representation problem, researchers have designed a number of learning and static-processing algorithms, which deduce hidden properties of a given domain, and use it to improve the domain description. For example, they constructed systems for learning control rules [Mitchell *et al.*, 1983; Minton, 1988], replacing operators with macros [Fikes *et al.*, 1972; Korf, 1985a], abstracting unimportant features of the domain [Sacerdoti, 1974; Knoblock, 1993], and reusing past problem-solving episodes [Hall, 1987; Veloso, 1994].

These algorithms are themselves sensitive to changes in problem encoding, and their ability to learn useful information depends on the initial description. For instance, most systems for learning control rules require a certain generality of predicates in the domain encoding, and become ineffective if we use too specific or too general predicates [Etzioni and Minton, 1992; Veloso and Borrajo, 1994].

As another example, abstraction algorithms are very sensitive to the description of available operators [Knoblock, 1994]. If the operator encoding is too general, or the domain includes unnecessary operations, then they fail to construct an abstraction hierarchy. In Section 1.2, we will illustrate such failures and discuss related description improvements.

To ensure the effectiveness of learning algorithms, the user normally has to perform two manual tasks. First, she needs to decide which algorithms are appropriate for the current domain. Second, she may have to adjust the initial domain description for the selected algorithms. An important next step in AI research is to develop a system that automatically accomplishes these tasks.

1.2 Examples of representation changes

All AI systems are sensitive to description of the input problems. If we use an inappropriate domain encoding, then even simple problems may become hard or unsolvable. Researchers have noticed that novices often construct ineffective domain descriptions, because intuitively appealing encodings are often inappropriate for AI problem solving.

On the other hand, expert users prove proficient in finding good descriptions; however, the construction of a proper domain encoding is often a difficult task, which requires not only familiarity with the system, but also creativity and experimentation with alternative encodings. The user usually begins with a semi-effective description and tunes it, based on the results of problem solving. If the user does not provide a good domain encoding, then automatic improvements are essential for efficient problem solving.

To illustrate the need for description changes, we present a puzzle domain, whose standard encoding is inappropriate for PRODIGY (Section 1.2.1). We then show modifications to the initial encoding that drastically improve efficiency (Sections 1.2.1–1.2.4) and discuss the choice of an appropriate problem solver (Section 1.2.5). The $\mathcal{S}$HAPER system is able to perform these improvements automatically.

1.2.1 Tower-of-Hanoi Domain

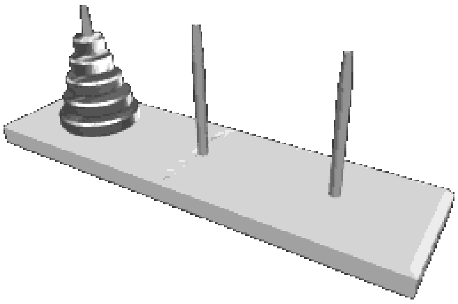
We consider the Tower-of-Hanoi puzzle, shown in Figure 1.5, which has proved difficult for most problem-solving algorithms, as well as for human subjects. It long served as one of the tests for AI systems, but gradually acquired a negative connotation of a "toy" domain. We utilize this puzzle to illustrate basic description changes in the $\mathcal{S}$HAPER system; however, we will use larger domains for empirical evaluation of the system.

The puzzle consists of three vertical pegs and several disks of different sizes. Every disk has a hole in the middle, and we may stack several disks on a peg (see Figure 1.5a). The rules allow us to move disks from peg to peg, one disk at a time; however, the rules do *not* allow placing any disk above a smaller one. In Figure 1.7, we show the complete state space of the three-disk puzzle.

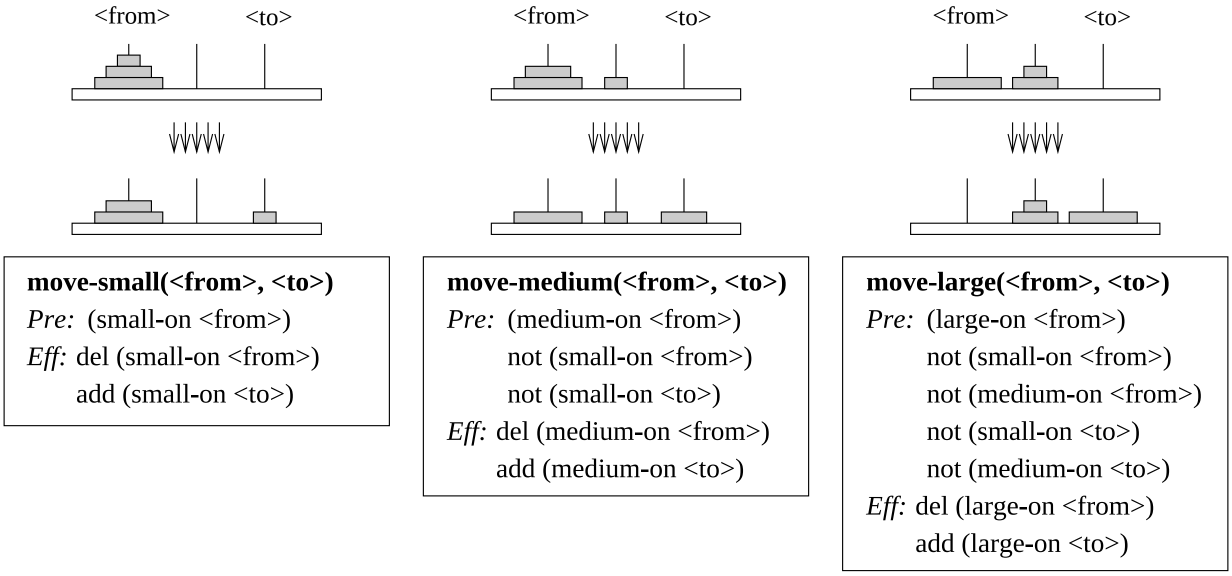
When using a classical AI system, we have to specify predicates for describing states of the simulated world (for example, see the AI textbook by Nilsson [1980]). If the Tower-of-Hanoi puzzle has three disks, then we may describe its states with three predicates, which denote the positions of the disks: (small-on <peg>), (medium-on <peg>), and (large-on <peg>), where <peg> is a variable that denotes an arbitrary peg. We obtain literals describing a specific state by substituting the appropriate constants for variables. For instance, the literal (small-on peg-1) means that the small disk is on the first peg.

The legal moves are encoded by production rules for modifying the world state, which are called *operators.* The description of an operator consists of *precondition predicates,* which must hold before its execution, and *effects*, which specify predicates that are added to or deleted from the world state upon the execution. In Figure 1.5(b), we give an encoding of all allowed moves in the three-disk puzzle. This encoding is based on the PRODIGY domain language, described in Sections 2.2 and 2.3; however, we slightly deviate from the exact PRODIGY syntax, in order to improve readability.

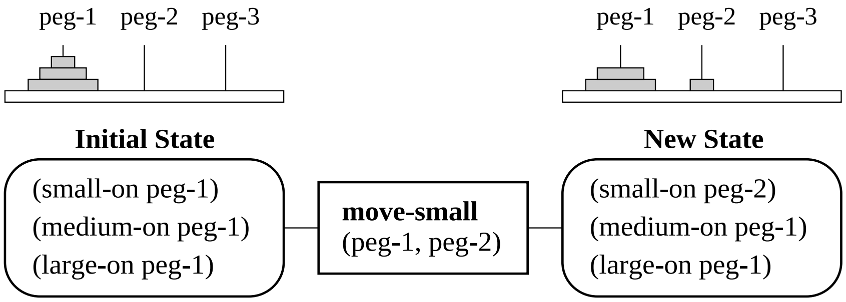
The <from> and <to> variables in the operator description denote arbitrary pegs. When a problem-solving algorithm uses an operator, it instantiates the variables with specific constants. For example, if the solver algorithm needs to move the small disk from peg-1 to peg-2, then it can execute the operator **move-small**(peg-1,peg-2) (see Figure 1.5c). The precondition of this operator is (small-on peg-1), that is, the small disk must initially be on peg-1. The execution results in deleting (small-on peg-1) from the current state and adding



(a) Tower-of-Hanoi puzzle.



(b) Encoding of operations in the three-disk puzzle.



(c) Example of executing an instantiated operator.

Figure 1.5: Tower-of-Hanoi Domain and its encoding in the PRODIGY architecture. The player may move disks from peg to peg, one at a time, without ever placing a disk on top of a smaller one. The traditional task is to move all disks from the left-hand peg to the right-hand (see Figure 1.6).

	number of a problem						mean
	1	2	3	4	5	6	time
without abstraction	2.0	34.1	275.4	346.3	522.4	597.4	296.3
using abstraction	0.5	0.4	1.9	0.3	0.5	2.3	1.0

Table 1.1: PRODIGY performance on six problems in the three-disk Tower-of-Hanoi Domain. We give running times in seconds, for problem solving without and with the abstraction hierarchy.

(small-on peg-2).

In Figure 1.6, we show the encoding of a classic problem in the Tower-of-Hanoi Domain, which requires moving all three disks from peg-1 to peg-3, and give the shortest solution to this problem. The initial state of the problem corresponds to the left corner of the state-space triangle in Figure 1.7, whereas the goal state is the right corner.

1.2.2 Constructing an abstraction hierarchy

Most AI systems solve a given problem by exploring the space of partial solutions, rather than expanding the problem's state space. That is, the nodes in their search space represent incomplete solutions, which may not correspond to paths in the state space (for example, see the review article by Weld [1994]).

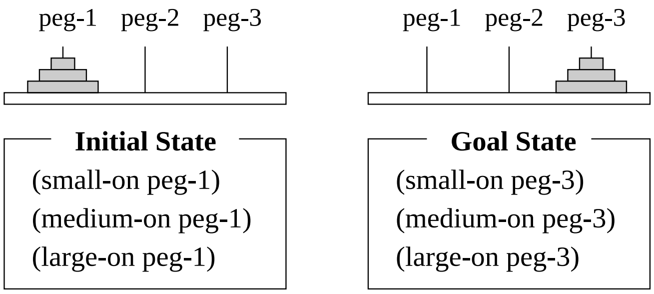
This strategy allows efficient reasoning in large-scale domains, which have intractable state spaces; however, it causes a major inefficiency in the Tower-of-Hanoi Domain. For example, if we apply PRODIGY to the problem in Figure 1.6, then the system considers more than hundred thousand partial plans during its search for a solution, which takes ten minutes on a Sun 5 computer.

We may significantly improve performance by using an *abstraction hierarchy* [Sacerdoti, 1974], which enables the system to subdivide problems into simpler subproblems. To construct a hierarchy, we assign different levels of importance to predicates in the domain encoding. In Figure 1.8(a), we give the standard hierarchy for the three-disk Tower of Hanoi.

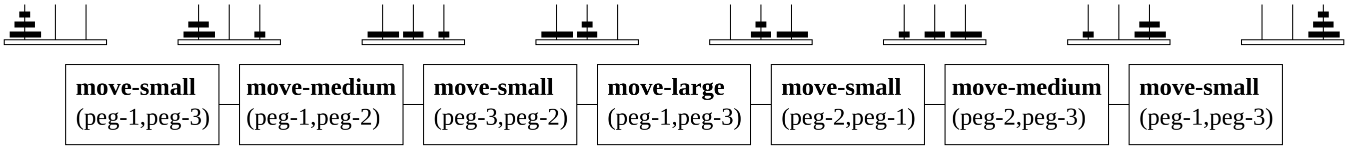
The system first constructs an abstract solution at level 2 of the hierarchy, ignoring the positions of the small and medium disk. We show the state space of the abstracted puzzle in Figure 1.8(b) and its solution in Figure 1.8(c). Then, PRODIGY steps down to the next lower level and inserts operators for moving the medium disk. At this level, the system *cannot* add new **move-large** operators, which limits its search space. We give the level-1 search space in Figure 1.8(d) and the corresponding solution in Figure 1.8(e). Finally, the system shifts to the lowest level of the hierarchy and inserts **move-small** operators, thus constructing the complete solution (see Figures 1.8f and 1.8g).

In Table 1.1, we give the running times for solving six Tower-of-Hanoi problems, without and with abstraction. We have obtained these results using a Lisp implementation of the PRODIGY search, on a Sun 5 machine; they show that abstraction drastically reduces search.

Knoblock [1994] has investigated abstraction problem solving in PRODIGY and developed the ALPINE system, which automatically assigns importance levels to predicates. We have extended Knoblock's technique and implemented the *Abstractor* algorithm, which serves as one of the description changers in the $\mathcal{S}$HAPER system.



(a) Encoding of the problem.



(b) Shortest solution.

Figure 1.6: Example of a problem instance in the Tower-of-Hanoi Domain: We need to move all three disks from `peg-1` to `peg-3`. The optimal solution to this problem comprises seven steps.

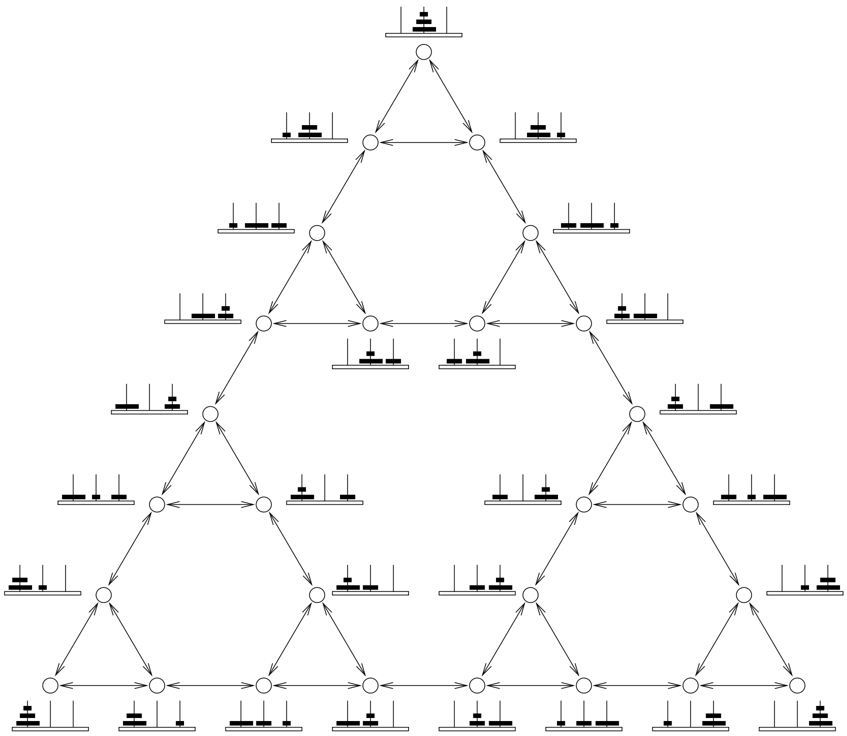
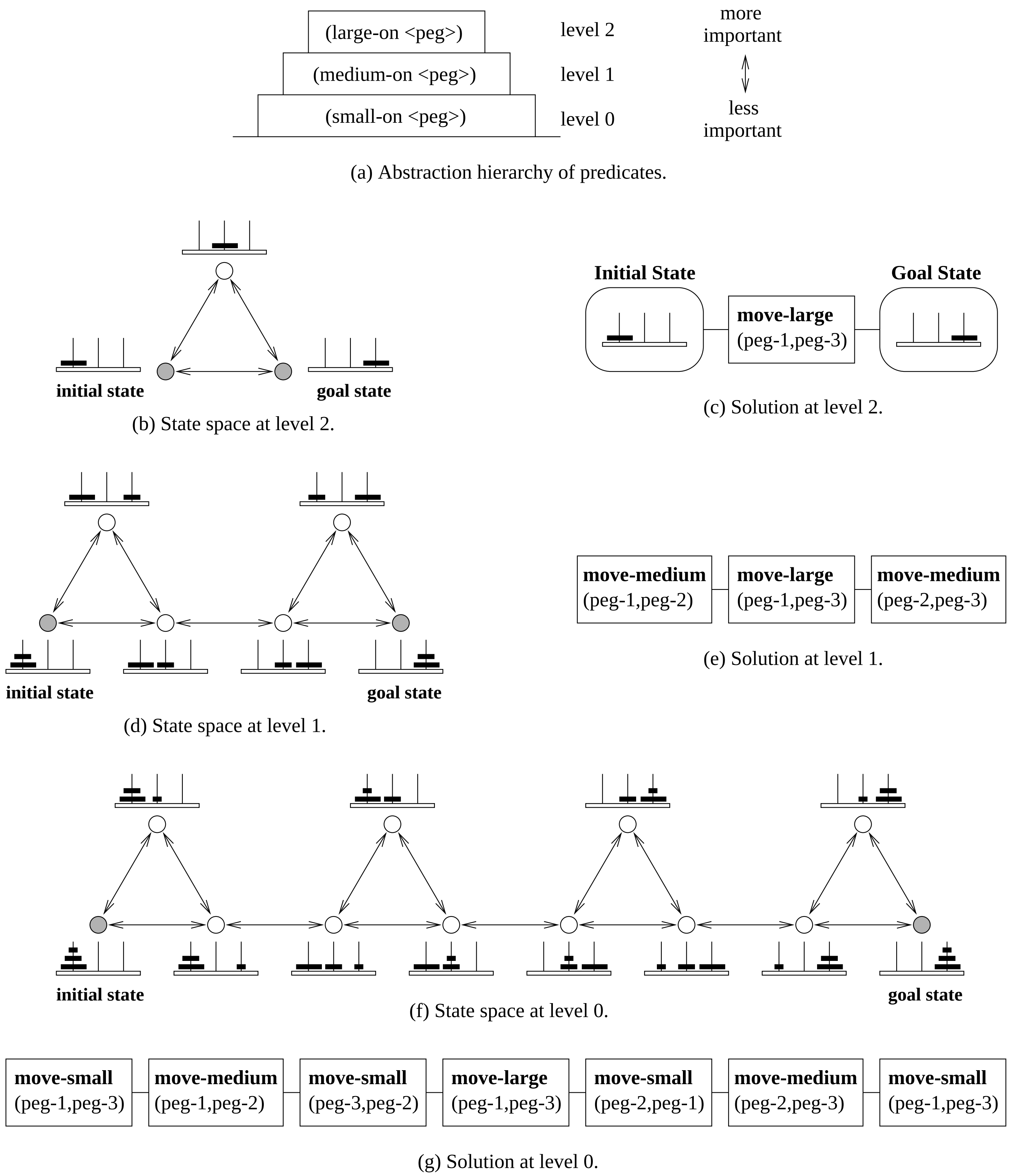


Figure 1.7: State space of the three-disk Tower of Hanoi: We illustrate all possible configurations of the puzzle (circles) and legal transitions between them (arrows). The initial state of the problem in Figure 1.6 is the left corner of the triangle, and the goal is the right corner.



(a) Abstraction hierarchy of predicates.

(b) State space at level 2.

(c) Solution at level 2.

(d) State space at level 1.

(e) Solution at level 1.

(f) State space at level 0.

(g) Solution at level 0.

Figure 1.8: Abstraction problem solving in the Tower-of-Hanoi Domain, with a three-level hierarchy (a). First, the system disregards the small and medium disk, and solves the simplified one-disk puzzle (b, c). Then, it inserts the missing movements of the medium disk (d, e). Finally, it steps down to the lowest level of abstraction and adds **move-small** operators (f, g).

	number of a problem						mean
	1	2	3	4	5	6	time
without primary effects	85.3	1.1	505.0	> 1800.0	31.0	172.4	> 432.4
using primary effects	0.5	1.2	16.6	144.8	77.5	362.5	100.5
primaries and abstraction	0.5	0.3	2.5	0.2	0.2	3.1	1.1

Table 1.2: PRODIGY performance in the extended Tower-of-Hanoi Domain, which allows two-disk moves. We give the running times in seconds for three different domain descriptions: without primary effects, with primary effects, and using primary effects along with abstraction. The results show that primary effects not only reduce the search, but also allow the construction of an effective abstraction hierarchy.

1.2.3 Selecting primary effects

Suppose that we deviate from the standard rules of the Tower-of-Hanoi puzzle and allow moving two disks together (see Figure 1.9a). The new operators enable us to construct shorter solutions for most problems. For example, we can move all three disks from `peg-1` to `peg-3` in three steps (Figure 1.9b).

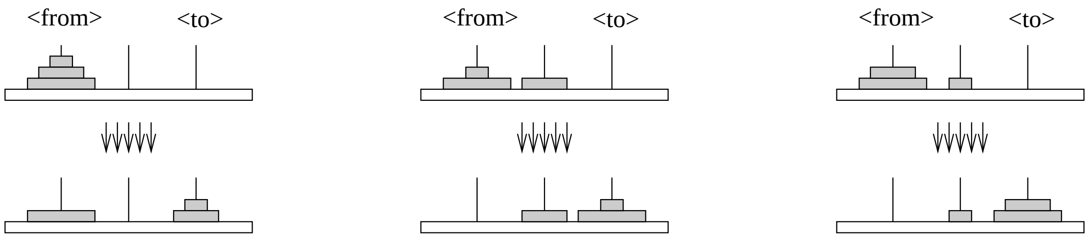
This change in the rules simplifies the puzzle for humans, but it makes most problems harder for the PRODIGY system. The availability of extra operations results in a higher branching factor, thus increasing the size of the expanded search space. Moreover, *Abstractor* fails to generate a hierarchy for the domain with two-disk moves. In Table 1.2, we give the results of using the extended set of operators to solve the six sample problems (see the first row of running times). For every problem, we set a 1800-second limit for the search time, and the system ran out of time on problem 4.

To reduce the branching factor, we may select *primary effects* of some operators and force the system to use these operators only for achieving their primary effects. For example, we may indicate that the main effect of the **move-sml-mdm** operator is the new position of the medium disk. That is, if the system's only goal is moving the small disk, then it must not consider this operator. Note that an inappropriate choice of primary effects may compromise completeness, that is, make some problems unsolvable; we will describe techniques for ensuring completeness in Section 3.2.

In Figure 1.9(c), we list the primary effects of the two-disk moves. The use of the selected primary effects improves the system's performance on most sample problems (see the middle row of Table 1.2); more importantly, it enables *Abstractor* to build the three-level hierarchy, which reduces search by two orders of magnitude (see the last row).

The $\mathcal{S}$HAPER system includes an algorithm for selecting primary effects, called *Margie*[1], which automatically performs this description change. The *Margie* algorithm is integrated with *Abstractor*: it chooses primary effects with the purpose of improving the quality of abstraction.

[1]The *Margie* procedure (pronounced *mär'gē*) is named after my friend, Margie Roxborough. Margie and her husband John invited me to stay at their place during the Ninth CSCSI Conference in Vancouver, where I gave a talk on related research results. This algorithm is *not* related to the MARGIE system (*mär'jē*) for parsing and paraphrasing simple stories in English, implemented by Schank *et al.* [1975].

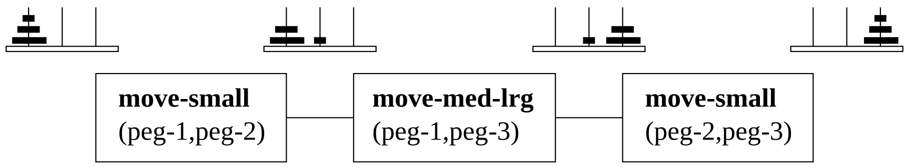


move-sml-mdm(<from>, <to>)
Pre: (small-on <from>)
(medium-on <from>)
Eff: del (small-on <from>)
add (small-on <to>)
del (medium-on <from>)
add (medium-on <to>)

move-sml-lrg(<from>, <to>)
Pre: (small-on <from>)
(large-on <from>)
not (medium-on <from>)
not (medium-on <to>)
Eff: del (small-on <from>)
add (small-on <to>)
del (large-on <from>)
add (large-on <to>)

move-med-lrg(<from>, <to>)
Pre: (medium-on <from>)
(large-on <from>)
not (small-on <from>)
not (small-on <to>)
Eff: del (medium-on <from>)
add (medium-on <to>)
del (large-on <from>)
add (large-on <to>)

(a) Encoding of two-disk moves.

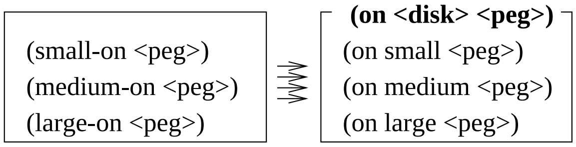


(b) Solution to the example problem.

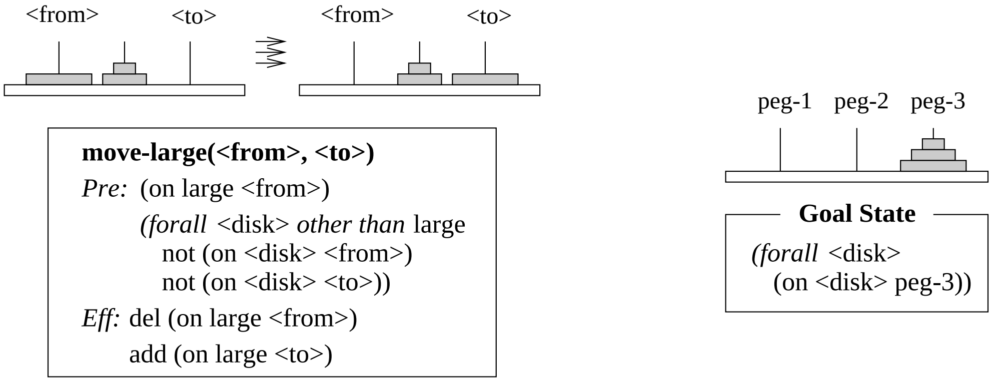
operators	primary effects
move-sml-mdm(<from>,<to>)	del (medium-on <from>), add (medium-on <to>)
move-sml-lrg(<from>,<to>)	del (large-on <from>), add (large-on <to>)
move-mdm-lrg(<from>,<to>)	del (large-on <from>), add (large-on <to>)

(c) Selection of primary effects.

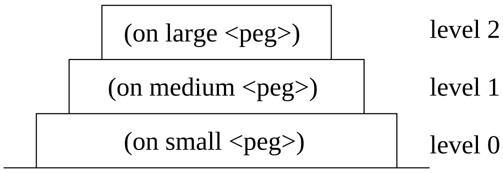
Figure 1.9: Extension to the Tower-of-Hanoi Domain, which includes operators for moving two disks at a time, and the selected primary effects of these additional operators.



(a) Replacing the predicates with a more general one.



(b) Examples of using the general predicate in the encoding of operators and problems.



(c) Hierarchy of partially instantiated predicates.

Figure 1.10: General predicate (on <disk> <peg>) in the encoding of the Tower-of-Hanoi Domain. This predicate enables the user to utilize quantifications in describing operators and goals, but it causes a failure of the *Abstractor* algorithm. The system has to generate partial instantiations of (on <disk> <peg>) before invoking *Abstractor*.

1.2.4 Partially instantiating operators

The main drawback of the *Abstractor* algorithm is its sensitivity to syntactic features of a domain encoding. In particular, if the domain includes too general predicates, then *Abstractor* may fail to construct a hierarchy.

For instance, suppose that the human user replaces the predicates (small-on <peg>), (medium-on <peg>), and (large-on <peg>) with a more general predicate (on <disk> <peg>), which allows greater flexibility in encoding operators and problems (see Figure 1.10a). In particular, it enables the user to utilize universal quantifications (see the examples in Figure 1.10b).

Since the resulting description contains only one predicate, the abstraction algorithm cannot generate a multi-level hierarchy. To remedy this problem, we may construct partial instantiations of (on <disk> <peg>) and apply *Abstractor* to build a hierarchy of these instantiations (see Figure 1.10c). We have implemented a description-changing algorithm, called

		number of a problem						mean
		1	2	3	4	5	6	time
SAVTA	with depth bound	0.51	0.27	2.47	0.23	0.21	3.07	1.13
	w/o depth bound	0.37	0.28	0.61	0.22	0.21	0.39	0.35
SABA	with depth bound	5.37	0.26	> 1800.00	2.75	0.19	> 1800.00	> 601.43
	w/o depth bound	0.45	0.31	1.22	0.34	0.23	0.51	0.51

Table 1.3: Performance of SAVTA and SABA in the extended Tower-of-Hanoi Domain, with primary effects and abstraction. We give running times in seconds, for search with and without a depth bound. The data suggest that SAVTA without a time bound is the most efficient among the four search techniques, but this conclusion is not statistically significant.

Refiner, that generates partially instantiated predicates for improving the effectiveness of *Abstractor*.

1.2.5 Choosing a problem solver

The efficiency of search depends not only on the domain description, but also on the choice of a solver algorithm. To illustrate importance of a solver, we consider the application of two different search strategies, called SAVTA and SABA, to problems in the extended Tower-of-Hanoi Domain.

Veloso and Stone [1995] developed these two strategies for guiding PRODIGY search (see Section 2.2.5). Experiments have shown that the relative performance of SAVTA and SABA varies across domains, and the choice between them may be essential for efficient problem solving. We consider two different modes of using each strategy: with a bound on the search depth and without limiting the depth. A depth bound helps to prevent an extensive exploration of inappropriate branches in the search space; however, it also results in pruning some solutions from the search space, which may have a negative effect on performance. Moreover, if a bound it too tight, it may lead to pruning *all* solutions, thus compromising the system's completeness.

In Table 1.3, we give the results of applying SAVTA and SABA to six Tower-of-Hanoi problems. These results show that SAVTA without a time bound is more effective than the other three techniques; however, the evidence is *not* statistically significant. In Section 8.4, we will describe a method for estimating the probability that a selected problem-solving technique is the best among the available techniques. We may apply this method to determine the chances that SAVTA without a time bound is indeed the most effective among the four techniques; its application gives the probability estimate of 0.47.

If we apply $\mathcal{S}$HAPER to many Tower-of-Hanoi problems, then it can accumulate more data on performance of the candidate strategies before adopting one of them. The system's control module combines exploitation of the past performance information with collecting additional data. First, the $\mathcal{S}$HAPER system applies heuristics for rejecting inappropriate search techniques; then, the system experiments with promising search algorithms, until it accumulates enough data for identifying the most effective algorithm.

Note that we have *not* accounted for solution quality in evaluating PRODIGY performance in the Tower-of-Hanoi Domain. If the user is interested in near-optimal solutions, then $\mathcal{S}$HAPER has to analyze the trade-off between running time and solution quality, which may result in selecting different domain descriptions and search strategies. For instance, a cost bound reduces the length of generated solutions, which may be a fair payment for the increase in search time. As another example, search without an abstraction hierarchy usually yields better solutions than abstraction problem solving. In Chapter 8, we will describe a statistical technique for evaluating trade-offs between speed and quality.

1.3 Related work

We next summarize previous results on representation changes, which include psychological experiments (Section 1.3.1), AI techniques for reasoning with multiple representations (Sections 1.3.2 and 1.3.3), and theoretical frameworks (Section 1.3.4).

We will later describe some other directions of past research, related to specific aspects of our work. In particular, we will give the history of the PRODIGY architecture in Section 2.1.1, outline the previous research on abstraction problem solving in Section 4.1.1, and review the work on automatic evaluation of problem solvers in Section 8.1.1.

1.3.1 Psychological evidence

The choice of an appropriate representation is one of the main themes of Polya's famous book *How to Solve It*. Polya showed that the selection of an effective approach to a problem is a crucial skill for a student of mathematics. Gestalt psychologists also paid particular attention to reformulation of problems [Duncker, 1945; Ohlsson, 1984].

Recent explorations in cognitive science have yielded much evidence that confirms Polya's pioneering insight. Researchers have demonstrated that changes in a problem description affect the problem difficulty, and that performance of human experts in many areas depends on their proficiency in constructing a representation that fits a given task [Gentner and Stevens, 1983; Simon, 1989; Gentner and Stevens, 1983].

Newell and Simon [1972] studied the role of representation during their investigation of human problem solving. They observed that human subjects always construct some representation of a given problem before searching for a solution: "Initially, when a problem is first presented, it must be recognized and understood. Then, a problem space must be constructed or, if one already exists in LTM, it must be evoked. Problem spaces can be changed or modified during the course of problem solving" (*Human Problem Solving*, page 809).

Simon [1979; 1989] continued the investigation of representations in human problem solving and studied their role in a variety of cognitive tasks. In particular, he tested the utility of different mental models in the Tower-of-Hanoi puzzle. Simon [1975] noticed that most subjects gradually improved their mental representations, in the process of solving the puzzle.

Hayes and Simon [1974; 1976; 1977] investigated the effect of isomorphic changes in task description on subjects' reasoning. Specifically, they analyzed hard isomorphs of the Tower of Hanoi [Simon *et al.*, 1985] and found out that "changing the written problem instructions, without disturbing the isomorphism between problem forms, can affect by a factor of two the times required by subjects to solve a problem" (*Models of Thought,* volume I, page 498).

Larkin and Simon [1981; 1987] explored the role of multiple mental models in solving physical and mathematical problems, with a particular emphasis on pictorial representations. They observed that mental models of skilled scientists differ from those of novices, and that expertly constructed models are crucial for solving scientific problems. Qin and Simon [1992] came to a similar conclusion during their experiments on the use of imagery in understanding special relativity.

Kook and Novak [1991] also explored alternative representations in physics. They implemented the APEX program, which used multiple representations of physics problems, handled incompletely specified tasks, and performed transformations among several types of representations. Their conclusions about the significance of appropriate representation in expert reasoning agreed with Simon's results.

Kaplan and Simon [1990] explored representation changes in solving the Mutilated-Checkerboard problem. They noticed that the participants of their experiments first tried to utilize the initial representation and then searched for a more effective approach. Kaplan [1989] implemented a production system that simulated the representation shift of successful human subjects.

Tabachneck [1992] studied the utility of pictorial reasoning in economics, and showed that the right mental models are essential for economics problems. She then implemented the CaMeRa production system, which modeled human reasoning with multiple representations [Tabachneck-Schijf *et al.*, 1997].

Boehm-Davis *et al.* [1989], Novak [1995], and Jones and Schkade [1995] recently demonstrated the importance of representations in software development. Their results confirmed that people are sensitive to changes in problem description, and that improvement of the initial description is often a difficult task.

The reader may find a detailed review of past results in Peterson's [1996] collection of recent articles on reasoning with multiple representations. It includes several different views on the role of representation, as well as evidence on importance of representation changes in physics, mathematics, economics, and other areas.

1.3.2 Automating representation changes

AI researchers recognized the significance of representation back in the early Sixties, in the very beginning of their work on automated reasoning systems. In particular, Amarel [1960; 1968; 1971] discussed the effects of representation on the behavior of search algorithms, using the Missionaries-and-Cannibals problem to illustrate his main points. Newell [1965; 1966] showed that the complexity of reasoning in some games and puzzles strongly depends on the representation and emphasized that "hard problems are solved by finding new viewpoints; i.e., new problem spaces" (*On the Representations of Problems,* page 19).

Later, Newell with several other researchers implemented the Soar system [Laird *et al.*, 1987; Tamble *et al.*, 1990; Newell, 1992], capable of utilizing multiple descriptions of a problem domain to facilitate search and learning; however, their system did *not* generate new representations. The human operator was responsible for constructing domain descriptions and providing guidelines for their effective use.

Larkin *et al.* [1988] took a similar approach in their work on the FERMI expert system, which accessed several different representations of task-related knowledge and used them in parallel. This system required the human operator to provide appropriate representations of the input knowledge. The authors of FERMI encoded "different kinds of knowledge at different levels of granularity" and demonstrated that "the principled decomposition of knowledge according to type and level of specificity yields both power and cross-domain generality" (FERMI*: A Flexible Expert Reasoner with Multi-Domain Inferencing*, page 101).

Research on automatic change of domain description has mostly been limited to design of separate learning algorithms that perform specific types of improvements. Examples of these special-purpose algorithms include systems for replacing operators with macros operators [Korf, 1985b; Mooney, 1988; Cheng and Carbonell, 1986; Shell and Carbonell, 1989], changing the search space by learning heuristics [Newell *et al.*, 1960; Langley, 1983] and control rules [Minton *et al.*, 1989b; Etzioni, 1993; Veloso and Borrajo, 1994; Pérez, 1995], generating abstraction hierarchies [Sacerdoti, 1974; Knoblock, 1994], and replacing a given problem with a similar simpler problem [Hibler, 1994].

The authors of these systems have observed that utility of most learning techniques varies across domains, and their blind application may worsen efficiency in some domains; however, researchers have not automated selection among available learning systems and left it as the user's responsibility.

1.3.3 Integrated systems

The Soar architecture comprises a variety of general-purpose and specialized search algorithms, but it does *not* have a top-level procedure for selecting an algorithm that fits a given task. The classical problem-solving systems, such as SIPE [Wilkins, 1988], PRODIGY [Carbonell *et al.*, 1990; Veloso *et al.*, 1995], and UCPOP [Penberthy and Weld, 1992; Weld, 1994], have the same limitation: they allow alternative search strategies, but do not include a central mechanism for selecting among them.

Wilkins and Myers [1995; 1998] have recently addressed this problem and constructed the Multiagent Planning Architecture, which supports integration of multiple planning and scheduling algorithms. The available search algorithms in their architecture are arranged into groups, called *planning cells.* Every group has a top-level control procedure, called a *cell manager,* which analyzes an input problem and selects an algorithm for solving it. Some cell managers are able to break the given task into subtasks and distribute them among several algorithms.

The architecture includes advanced software tools for incorporating diverse search algorithms, with different domain languages; thus, it allows a synergetic use of previously implemented AI systems. Wilkins and Myers have demonstrated the effectiveness of their architecture in constructing centralized problem-solving systems. In particular, they have developed several large-scale systems for Air Campaign Planning.

Since the Multiagent Planning Architecture allows the use of diverse algorithms and domain descriptions, it provides an excellent testbed for the study of search with multiple representations; however, its current capabilities for automated representation changes are very limited.

First, the system has no general-purpose control mechanisms, and the human operator has to design and implement a specialized manager algorithm for every planning cell. Wilkins and Myers have used fixed selection strategies in the implemented cell managers, and have not augmented them with learning capabilities.

Second, the system has no tools for the inclusion of algorithms that improve domain descriptions, and the user must either hand-code all necessary descriptions or incorporate a mechanism for changing descriptions into a cell manager. The authors of the architecture have implemented several specialized techniques for decomposing an input problem into subproblems, but have not considered other types of description changes.

Minton [1993a; 1993b; 1996] has investigated the integration of constraint-satisfaction programs and designed the MULTI-TAC system, which combines a number of generic heuristics and search procedures. The system's top-level module explores the properties of a given domain, selects appropriate heuristics and search strategies, and combines them into an algorithm for solving problems in this domain.

The main component of MULTI-TAC's top-level module is an inductive learning mechanism, which tests the available heuristics on a collection of problems and utilizes the accumulated data to select among them. It guides a beam search in the space of the allowed combinations of heuristics and search procedures.

The system synthesizes efficient constraint-satisfaction algorithms, which usually perform on par with manually configured programs and exceed the performance of fixed general-purpose strategies. The major drawback is significant learning time: when the top-level module searches for an efficient algorithm, it tests candidate procedures on hundreds of sample problems.

Yang *et al.* [1998] have recently begun development of an architecture for integration of AI planning techniques. Their architecture, called PLAN++, comprises tools for implementing, modifying, and re-using the main elements of planning systems. The purpose is to modularize typical planning algorithms, construct a large library of search modules, and use them as building blocks for new algorithms.

Since the effectiveness of most planning techniques varies across domains, the authors of PLAN++ intend to design software tools that enable the user to select appropriate modules and configure them for specific domains. The automation of these tasks is one of the main open problems, which is closely related to our work on representation improvements.

1.3.4 Theoretical results

Researchers have developed theoretical frameworks for some special cases of description changes, including abstraction, replacement of operators with macros, and learning control rules; however, they have done little study of the common principles that underlie different types of changer algorithms. The results in developing a formal model of representation are also limited, and the general notion of useful representation changes has remained at an informal level.

Most theoretical models are based on the analysis of a generalized search space. When investigating some description change, researchers usually identify its effects on the search space of a solver algorithm, and estimate the resulting reduction in the algorithm's running time. This technique helps to determine the main desirable properties of a new description.

In particular, Korf [1985a; 1985b; 1987] investigated the effects of macro operators on the state space, and demonstrated that well-chosen macros may exponentially reduce the search time. Etzioni [1992] analyzed the search space of backward-chaining algorithms and showed that inappropriate choice of macro operators or control rules may worsen performance. He then derived a condition under which macros reduce the search, and compared the utility of macro operators with that of control rules.

Cohen [1992] developed a mathematical framework for analysis of macro-operator learning, explanation-based generation of control rules, and chunking. He applied the learning theory to analyze mechanisms for saving and reusing solution paths, and described a series of learning algorithms that provably improve performance.

Knoblock [1991; 1994] explored the benefits and limitations of abstraction, identified conditions that ensure search reduction, and used them in developing an algorithm for automatic generation of abstraction hierarchies. Bacchus and Yang [1992; 1994] lifted some of the assumptions underlying Knoblock's analysis and presented a more general evaluation of abstraction search. They studied the effects of backtracking across abstraction levels, demonstrated that it may impair efficiency, and described a technique for avoiding it.

Giunchiglia and Walsh [1992] proposed a general model of reasoning with abstraction, which captured and generalized most of the previous frameworks. They defined an abstraction as a certain mapping between axiomatic formal systems, investigated the properties of this mapping, and classified the main abstraction techniques.

A generalized model for improving representation was suggested by Korf [1980], who formalized representation changes based on the notions of isomorphism and homomorphism of state spaces (see Section 1.1.3). Korf utilized the resulting formalism in his work on automatic representation improvements; however, his model did not address "a method for evaluating the efficiency of a representation relative to a particular problem solver and heuristics to guide the search for an efficient representation for a problem" (*Toward a Model of Representation Changes*, page 75), whereas such heuristics are essential for developing an effective representation-changing system.

1.4 Overview of the approach

The review of previous work has shown that the results are still very limited. The main open problems are (1) design of AI systems capable of performing a wide range of representation changes and (2) development of a unified theory of reasoning with multiple representations.

The work on the SHAPER system is a step toward addressing these two problems. We have developed a framework for evaluating available representations, and formalized the task of finding an appropriate representation as search in the space of alternative domain descriptions and matching solver algorithms. We have applied this framework to designing a system that automatically performs several types of representation improvements.

The system comprises a collection of problem-solving and description-changing algorithms, and a top-level control module that selects and invokes appropriate algorithms (see

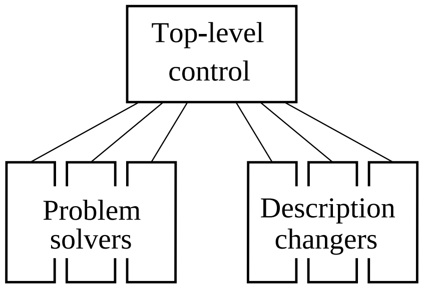


Figure 1.11: Integration of solver and changer algorithms: The $\mathcal{S}$HAPER system includes a top-level module, which analyzes a given problem and selects appropriate algorithms for solving it.

Figure 1.11). The most important result of our research is the control mechanism for intelligent selection among available algorithms and representations. We use it to combine multiple learning and search algorithms into an integrated AI system.

We now explain the main architectural decisions that underlie $\mathcal{S}$HAPER (Section 1.4.1), outline our approach to development of changer algorithms (Section 1.4.2), and briefly describe search in a space of representations (Section 1.4.3).

1.4.1 Architecture of the system

According to our definition, a system for changing representations has to perform two main functions: (1) improvement of the initial problem description and (2) selection of an algorithm for solving the problem. The key architectural decision underlying the $\mathcal{S}$HAPER system is the distribution of the first task among multiple changer algorithms. For instance, the description improvement in Section 1.2 has involved three algorithms, *Refiner*, *Margie*, and *Abstractor*.

The centralized use of separate algorithms differentiates our system from Korf's mechanism for improving representations. It also differs from the implementation of description-improving cell managers in the Multiagent Planning Architecture.

Every changer algorithm explores a certain space of modified descriptions, until finding a new description that improves the system's performance. For example, *Margie* searches among alternative selections of primary effects, whereas *Refiner* explores a space of different partial instantiations.

The top-level module of the $\mathcal{S}$HAPER system coordinates the application of description-changing algorithms. It explores a more general space of domain descriptions, using changer algorithms as operators for expanding nodes in this space. The system thus combines the low-level search by changer algorithms with the centralized high-level search. This two-level search prevents a combinatorial explosion in the number of candidate representations, described by Korf [1980].

The other major function of the top-level procedure is selection of problem-solving algorithms for the constructed domain descriptions. To summarize, $\mathcal{S}$HAPER consists of three main parts, illustrated in Figure 1.11:

Library of problem solvers: The system uses search algorithms of the PRODIGY

Efficiency: The primary criterion for evaluating representations in the $\mathcal{S}$HAPER system is the efficiency of problem solving, that is, the average running time of the solver algorithm.

Near-completeness: A change of representation must not cause a significant violation of completeness, that is, most solvable problems should remain solvable after the change. We measure the completeness violation by the percentage of problems that become unsolvable.

Solution quality: A representation change should not result in significant decline of solution quality. We usually define the quality as the total cost of operators in a solution, and evaluate the average increase in solution costs.

Figure 1.12: Main factors that determine the quality of a representation; we have developed a general utility model that unifies these factors and enables the system to make trade-off decisions.

architecture. We have composed the solver library from several different configurations of PRODIGY's general search mechanism.

Library of description changers: We have implemented seven algorithms that compose $\mathcal{S}$HAPER's library of changers. They include procedures for selecting primary effects, building abstraction hierarchies, generating partial and full instantiations of operators, and identifying relevant features of the domain.

Top-level control module: The functions of the control mechanism include selection of description changers and problem solvers, evaluation of new representations, and reuse of the previously generated representations. The top-level module comprises statistical procedures for analyzing past performance, heuristics for choosing algorithms in the absence of past data, and tools for manual control.

The control mechanism does *not* rely on specific properties of solver and changer algorithms, and the user may readily add new algorithms; however, all solvers and changers must use the PRODIGY domain language and access relevant data in the central data structures of the PRODIGY architecture.

We evaluate the utility of representations along three dimensions, summarized in Figure 1.12: the efficiency of search, the number of solved problems, and the quality of the resulting solutions [Cohen, 1995]. The work on changer algorithms involves decisions on trade-offs among these factors. We allow a moderate loss of completeness and decline of solution quality in order to improve efficiency. In Section 7.3, we will describe a general utility function that unifies the three evaluation factors.

Observe that, when evaluating the utility of a new representation, we have to account for the overall time for improving a domain description, selecting a problem solver, and using the resulting representation to solve given problems. We consider the system effective only if this overall time is smaller than the time for solving the problems with the initial description and some fixed solver algorithm.

1.4.2 Specifications of description changers

The current version of SHAPER includes seven description changers. We have already mentioned three of them: an extended version of the ALPINE abstraction generator, called *Abstractor*; the *Margie* algorithm, which selects primary effects; and the *Refiner* procedure, which generates partial instantiations of operators.

Every changer algorithm in the SHAPER system performs a specific *type* of description change and serves a certain *purpose*. For example, *Margie* selects primary effects of operators, with the purpose of increasing the number of levels in the abstraction hierarchy.

When implementing a changer algorithm, we must decide on the *type and purpose* of the description changes performed by the algorithm. The choice of a type determines the space of alternative descriptions explored by the algorithm, whereas the purpose specification helps to develop techniques for search in this space. We also need to analyze interactions of the new algorithm with other changer and solver algorithms. Finally, we have to identify the parts of the domain description that compose the algorithm's input.

These decisions form a high-level specification of a changer algorithm. We use specifications to summarize and systematize the main properties of description changers, thus separating them from implementation techniques. These summaries of main decisions have proved a useful development tool and facilitated our work. In Part II, we will give specifications for all seven description changers.

When making the high-level decisions, we must ensure that they define a useful class of description changes and lead to an efficient algorithm. We compose a specification from the following five parts:

Type of description change. When designing a new algorithm, we first have to decide on the type of description change. For instance, the *Abstractor* algorithm improves the domain description by *generating an abstraction hierarchy.* As another example, *Margie* is based on *selecting primary effects of operators.*

In Figure 1.13, we summarize the types of description changes used in SHAPER. Note that this list is only a small sample from the space of approaches to improving domain descriptions. We will briefly discuss some other improvements in Section 6.4.1.

Purpose of description change. Every changer algorithm in the SHAPER system serves a specific purpose, such as reducing the branching factor of search, constructing an abstraction hierarchy with certain properties, or improving the effectiveness of other description changers. We express this purpose by a heuristic function for evaluating the quality of a new description. For example, we may evaluate the performance of *Margie* by the number of levels in the resulting abstraction hierarchy: the more levels, the better. We also specify certain constraints that describe necessary properties of newly generated descriptions. For example, the *Margie* algorithm must preserve the completeness of problem solving, which limits its freedom in selecting primary effects.

To summarize, we view the purpose of a description improvement as maximizing a specific evaluation function, while satisfying certain constraints. This formal specification of the purpose shows exactly in which way we improve description, and helps to evaluate the results of applying the changer algorithm. Note that the overall effectiveness of our approach depends on the choice of an appropriate evaluation function, which must correlate with the

Selecting primary effects of operators: Choosing certain "important" effects of operators, and using operators only for achieving their important effects (Sections 3.4.1 and 3.5).

Generating an abstraction hierarchy: Decomposing the set of predicates in a domain encoding into several subsets, according to their "importance" (Sections 4.1, 4.2, and 5.1).

Generating more specific operators: Replacing an operator with several more specific operators, which together describe the same actions. We generate specific operators by instantiating variables in the original operator description (Section 3.4.2).

Generating more specific predicates: Replacing a predicate in a domain encoding with more specific predicates, which together describe the same set of literals (Section 4.3).

Identifying relevant features (literals): Determining which features of a domain description are relevant to the current task and ignoring the other features (Section 5.3).

Figure 1.13: Types of description changes in the current version of $\mathcal{S}$HAPER; we plan to add more techniques in the future (see the list of other description improvements in Figure 6.7, page 227).

resulting efficiency improvements.

Use of other algorithms. The description-changing algorithm may use subroutine calls to some problem solvers or other description changers. For example, *Margie* calls the *Abstractor* algorithm to construct hierarchies for the chosen primary effects. It repeatedly invokes *Abstractor* for alternative selections of primary effects, until finding a satisfactory selection.

Required input. We identify the elements of domain description that must be a part of the changer's input. For example, *Margie* has to access the description of all operators in the domain.

Optional input. Finally, we specify the additional information about the domain that may be used in changing description. If this information is available, then the changer algorithm utilizes it to generate a better description; otherwise, the algorithm uses some default assumptions. The optional input may include restrictions on the allowed problem instances, useful knowledge about domain properties, and advice of the human user.

For example, we may specify constraints on the allowed problems as an optional input to the *Margie* algorithm. Then, *Margie* passes these constraints to *Abstractor*, which utilizes them in building hierarchies (see Section 5.3). As another example, the user may pre-select some primary effects of operators. Then, *Margie* preserves this pre-selection and chooses additional primary effects (see Section 5.1).

We summarize the specification of the *Margie* algorithm in Figure 1.14; however, this specification does *not* account for advanced features of the PRODIGY domain language. In Section 5.1, we will extend it and describe the implementation of *Margie* in the PRODIGY architecture.

Type of description change: Selecting primary effects of operators.

Purpose of description change: Maximizing the number of levels in *Abstractor*'s hierarchy, while ensuring completeness of problem solving.

Use of other algorithms: The *Abstractor* algorithm, which constructs hierarchies for the selected primary effects.

Required input: Description of all operators in the domain.

Optional input: Restrictions on the allowed goal literals; pre-selected primary and side effects.

Figure 1.14: Simplified specification of the *Margie* algorithm. We use specifications to summarize the main properties of changers, thus abstracting them from the details of implementation.

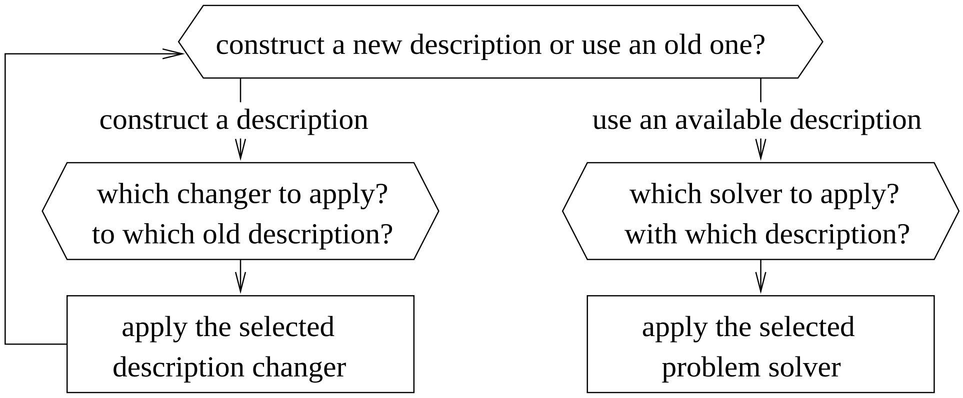


Figure 1.15: Top-level decisions in the SHAPER system: The control module applies changer algorithms to improve the domain description, and then chooses an appropriate solver algorithm.

1.4.3 Search in the space of representations

When the SHAPER system inputs a new problem, the control module has to determine a strategy for solving it, which involves several high-level decisions (see Figure 1.15):

1. Can SHAPER solve the problem with the initial domain description? Alternatively, can the system re-use one of the previously generated descriptions?

2. If not, what are the necessary improvements to the initial description? Which changer algorithms can make these improvements?

3. Which of the available search algorithms can efficiently solve the given problem?

These decisions guide the construction of a new representation, which consists of an improved description and matching solver. For example, if we encode the Tower-of-Hanoi Domain using the general predicate (on <disk> <peg>) and allow the two-disk moves, the control procedure will apply three description changers, *Refiner*, *Margie*, and *Abstractor*, and then select an effective problem solver (see Figure 1.16).

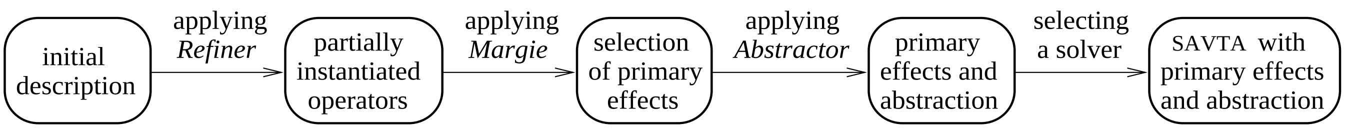


Figure 1.16: Representation changes in the Tower-of-Hanoi Domain (see Section 1.2): The top-level module applies three description changers and then identifies the most effective problem solver.

When $\mathcal{S}$HAPER searches for an effective representation, it performs two main tasks: generating new representations and evaluating their utility (see Figure 1.17). The first task involves improving the available descriptions and pairing them with appropriate solvers.

The top-level module selects and applies changer algorithms, using them as basic steps for expanding a space of alternative descriptions (see Figure 1.18a). After generating a new domain description, the top-level procedure chooses solver algorithms for this description. If the procedure identifies several matching algorithms, then it pairs each of them with the new description, thus constructing several representations (see Figure 1.18b).

The system evaluates the available representations in two steps (see Figure 1.17). First, it applies heuristics for estimating their relative utility and eliminates ineffective representations. We have provided several general heuristics and implemented a mechanism that enables the user to add domain-specific heuristics. Second, the system collects experimental data on performance of the remaining representations, and applies statistical analysis to select the most effective domain description and solver algorithm.

The exploration of the representation space is computationally expensive, because it involves execution of changer algorithms and evaluation of representations on multiple test problems. We therefore need to develop effective heuristics that guide the search in this space and allow the construction of good representations in feasible time.

1.5 Extended abstract

The main results of our work on $\mathcal{S}$HAPER include development of several description changers, construction of a general-purpose control module, and empirical evaluation of the system in the PRODIGY architecture. The presentation of these results is organized in four parts, as shown in Figure 1.19.

Part I includes the motivation for our research (Chapter 1) and description of the PRODIGY search (Chapter 2). In Part II, we present algorithms for using primary effects and abstraction, as well as auxiliary procedures that improve the performance of these algorithms.

Then, in Part III, we describe the top-level control mechanism, which explores a space of alternative representations. Finally, in Part IV, we give the results of testing the $\mathcal{S}$HAPER system and compare its performance to that of fixed description changers.

We now give a more detailed overview of the main results and summarize the material of every chapter. In Figure 1.19, we illustrate the dependencies among the contents of different chapters.

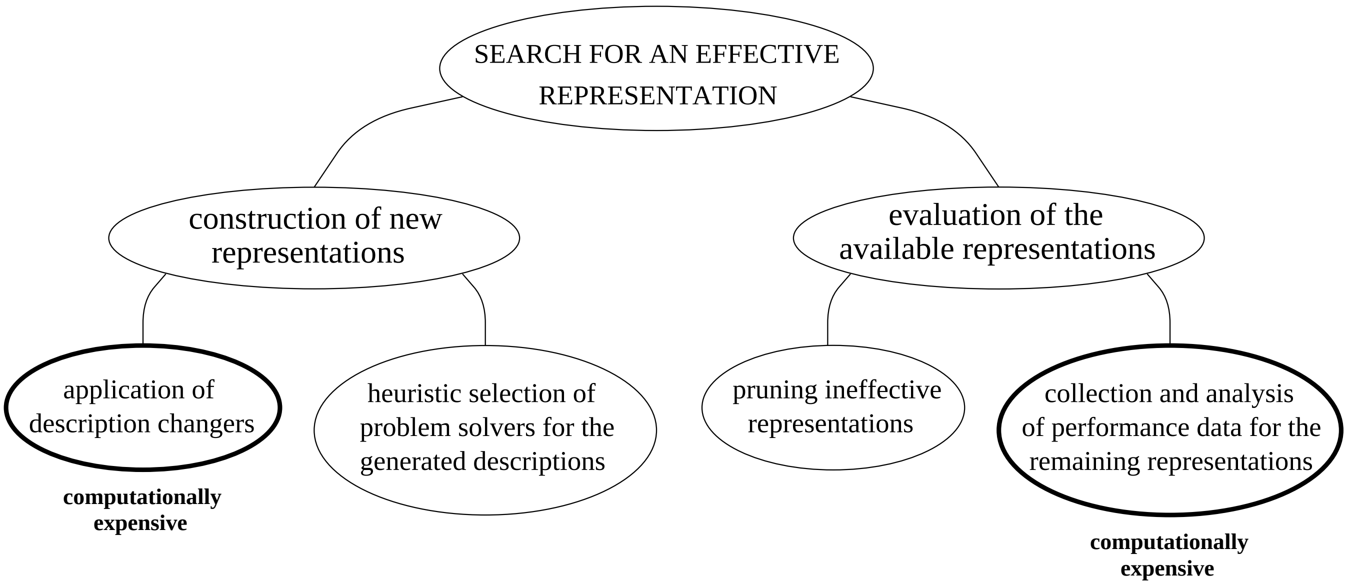


Figure 1.17: Main operations involved in exploring the space of representations: The $\mathcal{S}$HAPER system interleaves generation of new representations with testing of their performance.

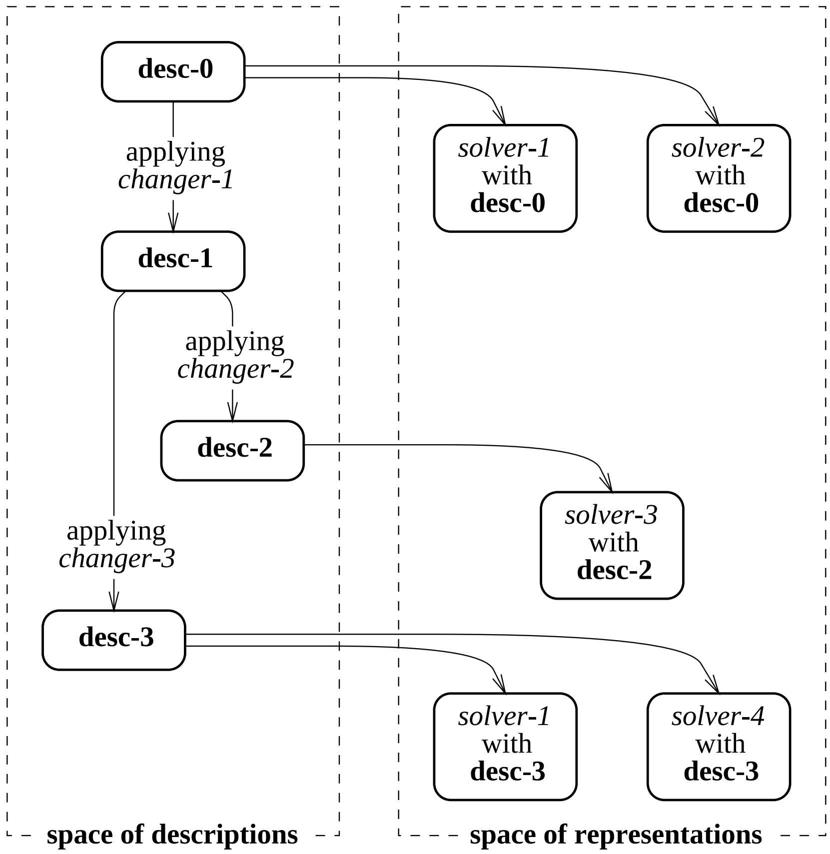


Figure 1.18: Expanding the representation space: The control module invokes changer algorithms to generate new domain descriptions, and combines solvers with the resulting descriptions.

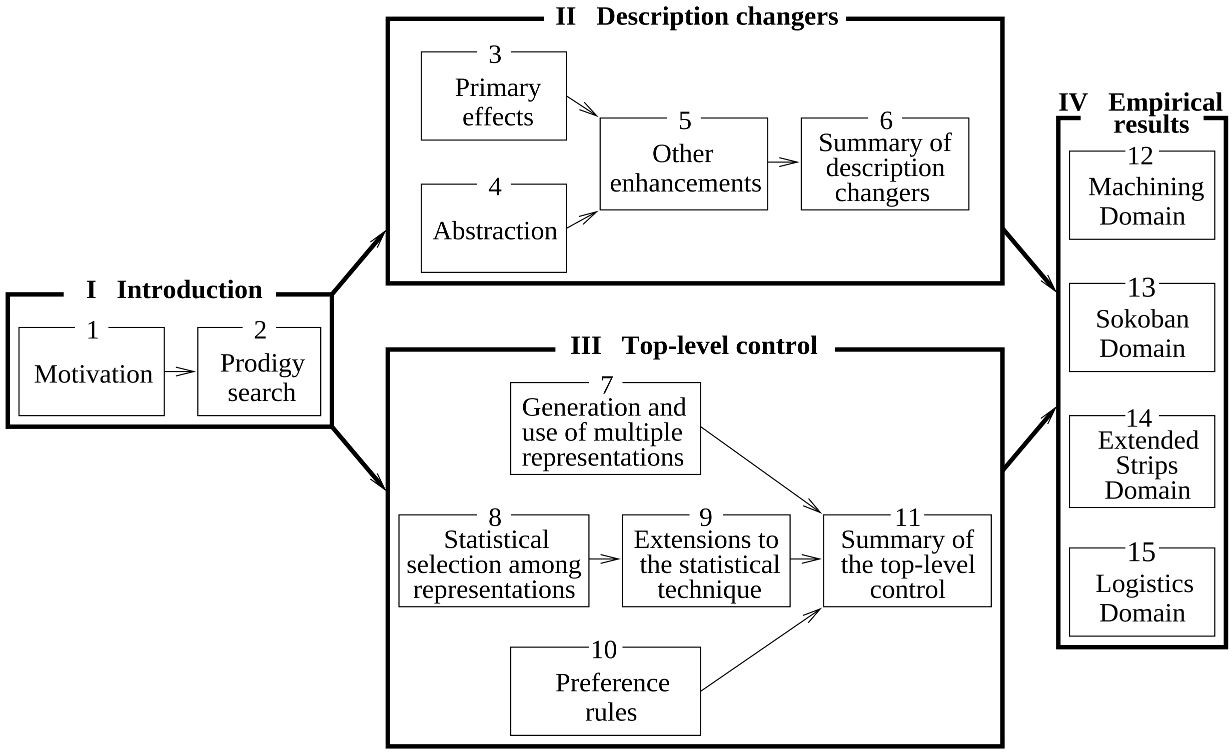


Figure 1.19: Reader's guide: Dependencies among the material of different chapters. The large rectangles show the four main parts of the presentation, whereas the small rectangles are chapters.

Part I: Introduction

The purpose of the introduction is to explain the problem of automatic representation changes, motivate the work on this problem, and present the background results that underlie our explorations.

We have emphasized importance of finding appropriate representations, described previous research on automating this task, and summarized the goals of our work. In Chapter 2, we will describe the PRODIGY architecture, which serves as a testbed for the development and evaluation of our system for changing representations.

Chapter 1: Motivation

We have explained the concept of representation, discussed its role in problem solving, and argued the need for an AI system that generates and evaluates multiple representations. We have also reviewed past work on representation changes, identified the main research problems, and outlined our approach to addressing some open problems.

To illustrate the role of representation, we have given an example of representation changes in PRODIGY, using the Tower-of-Hanoi puzzle. This example has demonstrated the functions of the $\mathcal{S}$HAPER system, which improves PRODIGY domain representations.

Chapter 2: Prodigy search

The PRODIGY system is based on a combination of goal-directed reasoning with simulation of operator execution. Researchers have implemented a series of search algorithms that utilize this technique; however, they have provided few formal results on the common principles underlying the developed algorithms.

We formalize the PRODIGY search, elucidate some techniques for improving its efficiency, and show how different strategies for controlling search complexity give rise to different versions of the system. In particular, we demonstrate that PRODIGY is not complete and discuss advantages and drawbacks of its incompleteness. We then develop a complete algorithm, which is almost as fast as PRODIGY and solves a wider range of problems.

Part II: Description changers

We investigate two techniques for reducing complexity of goal-directed search: identifying primary effects of operators and generating abstraction hierarchies. These techniques enable us to develop a collection of efficiency-improving algorithms, which compose $\mathcal{S}$HAPER's library of description changers.

We test the developed algorithms in several domains and demonstrate that they enhance performance of the PRODIGY system. We also discuss the drawbacks of the implemented speed-up techniques and point out some restrictions on their use.

Chapter 3: Primary effects

The use of primary effects of operators allows us to improve search efficiency and solution quality in many domains. We formalize this technique and provide analytical and empirical evaluation of its effectiveness.

First, we present a criterion for choosing primary effects, which guarantees efficiency and completeness, and describe algorithms for automatic selection of primary effects. Second, we experimentally demonstrate their effectiveness in two backward-chaining systems, PRODIGY and ABTWEAK.

Chapter 4: Abstraction

We describe abstraction for the PRODIGY system, present algorithms for automatic generation of abstraction hierarchies, and give empirical confirmation of their effectiveness in reducing search.

First, we review Knoblock's ALPINE system, which constructs hierarchies for a limited domain language, and extend it for the advanced language of PRODIGY. Second, we give an algorithm that improves effectiveness of the abstraction generator, by partially instantiating predicates in the domain encoding.

Chapter 5: Other enhancements

We present two techniques for enhancing utility of primary effects and abstraction. First, we describe a synergy of the abstraction generator with the procedure for selecting primary effects, and demonstrate that it leads to better hierarchies.

Second, we give an algorithm for adjusting the domain description to given classes of problems. It identifies the features of the simulated world that are relevant to a specific problem type, and then the system utilizes them to choose appropriate primary effects and abstraction.

Chapter 6: Summary of work on description changers

We review the main results of the work on description-changing algorithms in the SHAPER system. In particular, we summarize the interactions among the implemented algorithms and discuss the role of problem-specific information in improving domain descriptions.

In addition, we outline some directions for future research, which include implementation of other changer algorithms and work on a general theory of description changes. First, we consider several unexplored types of description improvements and give examples of their application in PRODIGY. Second, we discuss some steps toward a general framework for developing and evaluating changer algorithms.

Part III: Top-level control

We develop a system, called SHAPER, for the automatic generation and use of multiple domain descriptions. When SHAPER faces a new problem, it first improves the problem description and then selects an appropriate solver algorithm.

The system's central part is the top-level control module, which chooses appropriate changer and solver algorithms, stores and re-uses descriptions, and accumulates performance data. We describe the structure of the control module and its access to other parts of the system, present techniques for automatic selection among the available algorithms and domain descriptions, and discuss the main limitations of the implemented system.

Chapter 7: Generation and use of multiple representations

We lay a theoretical groundwork for a synergy of multiple description changers and problem solvers in a unified system. First, we discuss the task of improving domain descriptions and selecting appropriate solvers, formalize it as search in the space of available representations, and define the main elements of the representation space. Second, we develop a utility model for evaluating a representation-changing system. Third, we identify the limitations of our theory and their effects on the SHAPER system.

We apply the theoretical results to constructing the system's "control center," which provides access to the available algorithms. It consists of data structures and procedures that support exploration of the representation space. The control center forms the intermediate layer between the system's algorithms and the top-level decision mechanism.

Chapter 8: Statistical selection among representations

We consider the task of choosing among the available representations and formalize the statistical problem involved in evaluating their performance. We then present a learning algorithm that gathers performance data, evaluates representations, and chooses an appropriate representation for each given problem. The algorithm also selects a time bound for search with the chosen representation, and interrupts the solver upon reaching the bound.

We give results of applying this algorithm to select among PRODIGY problem solvers. We also describe controlled experiments with artificially generated performance data.

Chapter 9: Extensions to the statistical technique

We extend the statistical procedure to account for properties of specific problems, which allow a more accurate performance evaluation. We test it in the PRODIGY system and on artificially generated data, and demonstrate the resulting improvement in selection accuracy.

The extended learning algorithm accounts for problem-specific utility functions, adjusts the performance data to the estimated problem sizes, and utilizes information about similarity among problems.

Chapter 10: Preference rules

We describe heuristic rules for identifying an effective representation, which supplement the statistical evaluation. We then present techniques for resolving conflicts among multiple rules and for synergy of rules with the statistical algorithm.

The human operator may provide rules that encode initial knowledge about relative performance of different representations. She may also use rules for implementing additional learning mechanisms, as well as for controlling the trade-off between exploitation and exploration in statistical learning.

Chapter 11: Summary of work on the top-level control

We present some extensions to the control module, and then summarize the key results of developing the $\mathcal{S}$HAPER system. The major implemented extensions include (1) a mechanism for selecting among changer algorithms and (2) a collection of tools for optional user participation in the top-level control.

First, we describe the extension mechanisms and discuss their role. Then, we review the main parts of the top-level control and summarize the techniques underlying each part. Finally, we list the limitations of $\mathcal{S}$HAPER and point out related directions of future work.

Part IV: Results

We give the results of applying the $\mathcal{S}$HAPER system to a variety of problems in four different domains: a model of a machine shop (Chapter 12), Sokoban puzzle (Chapter 13), extended STRIPS world (Chapter 14), and PRODIGY Logistics Domain (Chapter 15).

The experiments have confirmed that the system's control module almost always selects the right description changers and problem solvers, and that its performance is *not* sensitive to features of specific domains.

Chapter 2

Prodigy search

Newell and Simon [1961; 1972] developed the *means-ends analysis* technique during their work on the General Problem Solver (GPS), back in the early days of artificial intelligence. Their technique combined goal-directed reasoning with forward chaining from the initial state. The authors of later systems [Fikes and Nilsson, 1971; Warren, 1974; Tate, 1977] gradually abandoned forward search and began to rely exclusively on backward chaining.

Researchers investigated several types of backward chainers [Minton *et al.*, 1994] and discovered that *least commitment* improves the efficiency of goal-directed reasoning, which gave rise to TWEAK [Chapman, 1987], ABTWEAK [Yang *et al.*, 1996], SNLP [McAllester and Rosenblitt, 1991], UCPOP [Penberthy and Weld, 1992; Weld, 1994], and other least-commitment problem solvers.

Meanwhile, PRODIGY researchers extended means-ends analysis and designed a family of problem solvers based on the combination of goal-directed backward chaining with simulation of operator execution. The underlying strategy is a special case of bidirectional search [Pohl, 1971]. It has given rise to several versions of the PRODIGYsystem, including PRODIGY1, PRODIGY2, NOLIMIT, PRODIGY4, and FLECS.

The developed algorithms keep track of the domain state that results from executing parts of the currently constructed solution, and use the state to guide the goal-directed reasoning. Least commitment proved ineffective for this search technique, and Veloso developed an alternative strategy, based on instantiating all variables as early as possible.

Experiments have demonstrated that PRODIGY search is an efficient procedure, a fair match to least-commitment systems and other successful problem solvers. Moreover, the PRODIGY architecture has proved a valuable tool for the development of learning techniques, and researchers have used it in constructing a number of systems for the automated acquisition of control knowledge.

We have utilized this architecture in the work on the representation changes and constructed the $\mathcal{S}$HAPER system as an extension to PRODIGY. In particular, $\mathcal{S}$HAPER's library of problem solvers is based on PRODIGY search algorithms. We therefore describe these algorithms before presenting $\mathcal{S}$HAPER.

First, we review the past work on the PRODIGY system and discuss advantages and drawbacks of the developed search techniques (Section 2.1). Then, we describe the foundations of these techniques and their use in different versions of PRODIGY (Section 2.2), as well as

version	year	authors
PRODIGY1	1986	Minton and Carbonell
PRODIGY2	1989	Carbonell, Minton, Knoblock, and Kuokka
NOLIMIT	1990	Veloso and Borrajo
PRODIGY4	1992	Blythe, Wang, Veloso, Kahn, Perez, and Gil
FLECS	1994	Veloso and Stone

Table 2.1: Main versions of the PRODIGY architecture. The work on this problem-solving architecture continued for over ten years, and gave rise to a series of novel search strategies.

the main extensions to the basic search engine (Sections 2.3 and 2.4). Finally, we report the results of a joint investigation with Blythe on the completeness of the PRODIGY search technique (Section 2.5).

2.1 PRODIGY system

The PRODIGY system went through several stages of development, over the course of ten years, and gradually evolved into an advanced architecture, which supports a variety of search and learning techniques. We give a brief history of its development (Section 2.1.1) and summarize the main features of the resulting search engines (Section 2.1.2).

2.1.1 History

The history of the PRODIGY architecture (see Table 2.1) began circa 1986, when Minton and Carbonell implemented PRODIGY1, which became a testbed for their work on control rules [Minton, 1988; Minton *et al.*, 1989a]. They concentrated on explanation-based learning of control knowledge and left few records of the original search engine.

Minton, Carbonell, Knoblock, and Kuokka used PRODIGY1 as a prototype in their work on PRODIGY2 [Carbonell *et al.*, 1990], which supported an advanced language for describing problem domains [Minton *et al.*, 1989b]. They demonstrated the system's effectiveness in scheduling machine-shop operations [Gil, 1991; Gil and Pérez, 1994], planning a robot's actions in an extended STRIPS world [Minton, 1988], and solving a variety of smaller problems.

Veloso [1989] and Borrajo developed the next version, called NOLIMIT, which significantly differed from its predecessors. In particular, they added new branching points, which made the search near-complete, and introduced object types for specifying possible values of variables. Veloso demonstrated the effectiveness of NOLIMIT on the previously designed PRODIGY domains, as well as on large-scale transportation problems.

Blythe, Wang, Veloso, Kahn, Pérez, and Gil developed a collection of techniques for enhancing the search engine and built PRODIGY4 [Carbonell *et al.*, 1992]. In particular, they provided an efficient technique for instantiating operators [Wang, 1992], extended the use of inference rules, and designed advanced data structures to improve the efficiency of the low-level implementation. They also implemented a friendly user interface and tools for adding new learning mechanisms.

Veloso and Stone [1995] implemented the FLECS algorithm, an extension to PRODIGY4 that included an additional decision point and new strategies for exploring the search space, and demonstrated that their strategies often improved the efficiency.

The PRODIGY architecture provides ample opportunities for the application of speed-up learning, and researchers have used it to develop and test a variety of techniques for the automated efficiency improvement. Minton [1998] designed the first learning module for the PRODIGY architecture, which automatically generated control rules. He demonstrated the effectiveness of integrating learning with PRODIGY search, which stimulated work on other efficiency-improving techniques.

In particular, researchers have designed modules for explanation-based learning [Etzioni, 1990; Etzioni, 1993; Pérez and Etzioni, 1992], inductive generation of control rules [Veloso and Borrajo, 1994; Borrajo and Veloso, 1996], abstraction search [Knoblock, 1993], and analogical reuse of problem-solving episodes [Carbonell, 1983; Veloso and Carbonell, 1990; Veloso and Carbonell, 1993a; Veloso and Carbonell, 1993b; Veloso, 1994]. They also investigated techniques for improving the quality of solutions [Pérez and Carbonell, 1993; Pérez, 1995], learning unknown properties of the problem domain [Gil, 1992; Carbonell and Gil, 1990; Wang, 1994; Wang, 1996], and collaborating with the human user [Joseph, 1992; Stone and Veloso, 1996; Cox and Veloso, 1997a; Cox and Veloso, 1997b; Veloso *et al.*, 1997].

The reader may find a summary of PRODIGY learning techniques in the review papers by Carbonell *et al.* [1990] and Veloso *et al.* [1995]. These research results have been major contributions to the study of machine learning; however, they have left two notable gaps. First, PRODIGY researchers tested the learning modules *separately,* without exploring their synergetic use. Even though preliminary attempts to integrate learning with abstraction gave positive results [Knoblock *et al.*, 1991a], the researchers have not pursued this direction. Second, there has been no automated techniques for deciding when to invoke specific learning modules. The user has traditionally been responsible for the choice among available learning systems. Addressing these gaps is among the goals of our work on the $\mathcal{S}$HAPER system.

2.1.2 Advantages and drawbacks

The PRODIGY architecture is based on two major design decisions, which differentiate it from other problem-solving systems. First, it combines backward chaining with simulated execution of relevant operators. Second, it fully instantiates operators in early stages of search, whereas most classical systems delay the commitment to a specific instantiation.

The backward-chaining procedure selects operators relevant to the goal, instantiates them, and arranges them into a partial-order solution. The forward chainer simulates the execution of these operator and gradually constructs a total-order sequence of operators. The system keeps track of the simulated world state, which would result from executing this sequence.

The problem solver utilizes the simulated world state in selecting operators and their instantiations, which improves the effectiveness of goal-directed reasoning. In addition, PRODIGY learning modules use the state to identify reasons for successes and failures of the search algorithm.

Since PRODIGY uses fully instantiated operators, it efficiently handles a powerful domain language. In particular, it supports the use of disjunctive and quantified preconditions, conditional effects, and arbitrary constraints on the values of operator variables [Carbonell *et al.*, 1992]. The solver utilizes the knowledge of the world state in choosing appropriate instantiations.

On the flip side, early commitment to full instantiations and specific execution order leads to a large branching factor, which results in gross inefficiency of breadth-first search. The problem solver uses depth-first search and relies on heuristics for selecting appropriate branches of the search space, which usually leads to finding suboptimal solutions. If the heuristics prove misleading, the solver expands wrong branches and may fail to find a solution. When a problem has no solution, a large branching factor becomes a major handicap: PRODIGY cannot exhaust the available space in reasonable time.

A formal comparison of PRODIGY with other search systems is still an open problem; however, multiple experimental studies have confirmed that PRODIGY search is an efficient strategy [Stone *et al.*, 1994]. Experiments also revealed that PRODIGY and backward chainers perform well in different domains. Some tasks are more suitable for execution simulation, whereas others require standard backward chaining. Veloso and Blythe [1994] identified some domain properties that determine which of the two strategies is more effective.

Kambhampati and Srivastava [1996a; 1996b] investigated common principles underlying PRODIGY and least-commitment search. They developed a framework that generalizes these two types of goal-directed reasoning and combines them with direct forward search. They implemented the Universal Classical Planner (UCP), which can use all these search strategies; however, the resulting general algorithm has many branching points, which give rise to an impractically large search space. The main open problem is development of heuristics that would effectively use the flexibility of UCP to guide the search.

Blum and Furst [1997] constructed GRAPHPLAN, which uses the domain state in a different way. They implemented propagation of constraints from the initial state of the domain, which enables their system to identify some operators with unsatisfiable preconditions. The system then discards these operators and uses backward chaining to construct a solution from the remaining operators. GRAPHPLAN performs forward constraint propagation prior to the search for a solution. Unlike PRODIGY, it does *not* use forward search from the initial state.

The relative performance of PRODIGY and GRAPHPLAN also varies from domain to domain. The GRAPHPLAN algorithm has to generate and store all possible instantiations of all operators before searching for a solution, which often causes a combinatorial explosion; thus, PRODIGY usually faster than GRAPHPLAN in large-scale domains. On the other hand, GRAPHPLAN wins in small-scale domains that require extensive search.

Researchers recently applied PRODIGY to robot navigation and discovered that its execution simulation is useful for interleaving search with real execution. In particular, Blythe and Reilly [1993a; 1993b] explored techniques for planning routes of a household robot in a simulated environment. Stone and Veloso [1996] constructed a mechanism for user-guided interleaving of problem solving and execution.

Haigh and Veloso [1996; 1997; 1998a; 1998b] built a system that navigates XAVIER, a real robot at Carnegie Mellon University. Haigh [1998] integrated this system with XAVIER's low-level control procedures, and demonstrated its effectiveness in planning and guiding the robot's high-level actions.

Their interleaving algorithms begin the real-world execution before PRODIGY completes the search for a solution, thus eliminating some backtracking points in the search space. This strategy involves the risk of bringing the robot to a deadend or even into an inescapable trap. To avoid such traps, Haigh and Veloso restricted the use of their system to domains with reversible actions.

2.2 Search engine

We next describe the basics of PRODIGY search; the description is based on the results of joint work with Veloso on formalizing the main principles underlying the PRODIGY system [Fink and Veloso, 1996]. All versions of the system are based on the algorithm described here; however, they differ from each other in the decision points used for backtracking, and in the general heuristics for guiding the search.

We present the foundations of the PRODIGY domain language (Section 2.2.1), encoding of intermediate incomplete solutions (Section 2.2.2), and the algorithm that combines backward chaining with execution simulation (Sections 2.2.3 and 2.2.4). We delay the discussion of techniques for handling disjunctive and quantified preconditions until Section 2.3. After describing the search engine, we discuss differences among the main versions of PRODIGY (Section 2.2.5).

2.2.1 Encoding of problems

We define a *problem domain* by a set of object types and a library of operators that act on objects of these types. The PRODIGY language for describing operators is based on the STRIPS domain language [Fikes and Nilsson, 1971], extended to express conditional effects, disjunctive preconditions, and quantifications.

An operator is defined by its *preconditions* and *effects*. The preconditions of an operator are the conditions that must be satisfied before its execution. They are represented by a logical expression with negations, conjunctions, disjunctions, and universal and existential quantifiers. The effects are encoded as a list of predicates added to or deleted from the current state of the domain upon the execution.

We may specify conditional effects, also called *if-effects,* whose outcome depends on the domain state. An if-effect is defined by its *conditions* and *actions*. If the conditions hold, the effect changes the state, according to its actions. Otherwise, it does not affect the state.

The effect conditions are represented by a logical expression, in the same way as operator preconditions; however, their meaning is somewhat different. If the preconditions of an operator do not hold in the state, then the operator cannot be executed. On the other hand, if the conditions of an if-effect do not hold, we may execute the operator, but the if-effect does not change the state.

The actions of an if-effect are predicates, to be added to or deleted from the state; that is, their encoding is identical to that of unconditional effects. We refer to both unconditional effects and if-effect actions as *simple effects*. When we talk about "effects" without explicitly referring to if-effects, we mean simple effects.

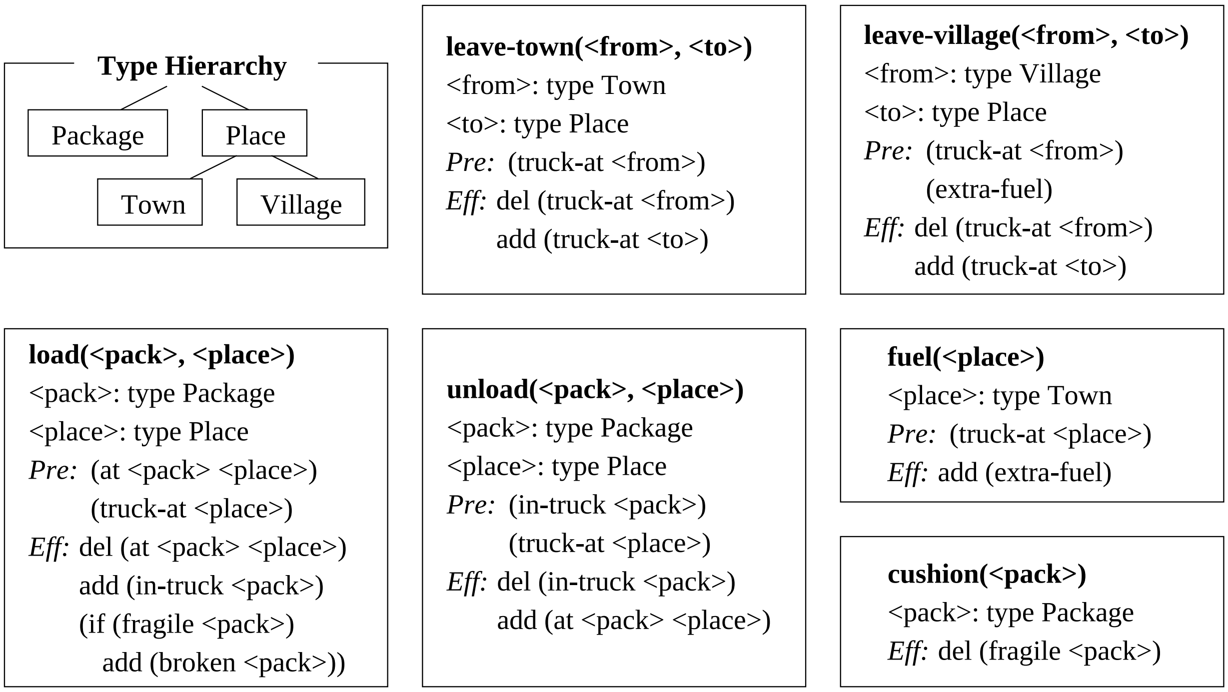


Figure 2.1: Encoding of a simple trucking world in the PRODIGY domain language. The Trucking Domain is defined by a hierarchy of object types and a library of six operators.

In Figure 2.1, we give an example of a simple domain. Note that the syntax of this domain description slightly differs from the PRODIGY language [Carbonell *et al.*, 1992], for the purpose of better readability. The domain includes two types of objects, Package and Place. The Place type has two subtypes, Town and Village. We use types to limit the allowed values of variables in the operator description.

A truck carries packages between towns and villages. The truck's fuel tank is sufficient for only one ride. Towns have gas stations, so the truck can refuel before leaving a town. On the other hand, villages do not have gas stations; if the truck comes to a village without a supply of extra fuel, it cannot leave. To avoid this problem, the truck can get extra fuel in any town.

We have to load packages before driving to their destination and unload afterwards. If a package is fragile, it gets broken during loading. We may cushion a package by soft material, which removes the fragility and prevents breakage.

A *problem* is defined by a list of object instances, an *initial state*, and a *goal statement*. The initial state is a set of literals, whereas the goal statement is a condition that must hold after executing a solution. A *complete solution* is a sequence of instantiated operators that can be executed from the initial state to achieve the goal. We give an example of a problem in Figure 2.2. The task in this problem is to deliver two packages from town-1 to ville-1. We may solve it as follows: "**load**(pack-1,town-1), **load**(pack-2,town-1), **leave-town**(town-1,ville-1), **unload**(pack-1,ville-1), **unload**(pack-2,ville-1))."

The initial state may include literals that cannot be added or deleted by operators, called *static literals*. For example, if the domain did not include the **fuel** operator, then (extra-fuel)

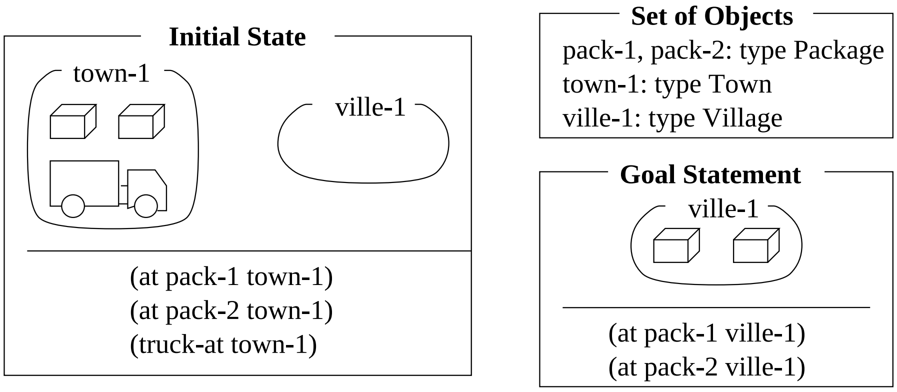


Figure 2.2: Encoding of a problem in the Trucking Domain, which includes a set of object instances, initial world state, and goal statement; the task is to deliver two packages from `town-1` to `ville-1`.

would be a static literal. If all instantiations of a predicate are static literals, we say that the predicate itself is static. Since no operator sequence can affect these literals, the goal statement should be consistent with the static elements of the initial state. Otherwise, the problem is unsolvable and the system reports failure without search.

2.2.2 Incomplete solutions

Given a problem, most problem-solving systems begin with the empty set of operators and modify it until a solution is found. Examples of modifications include adding an operator, instantiating or constraining a variable in an operator, and imposing an ordering constraint. The intermediate sets of operators are called *incomplete solutions.* We view them as nodes in the search space of the solver algorithm. Each modification of a current incomplete solution gives rise to a new node, and the number of possible modifications determines the branching factor of the search.

Researchers have explored a variety of structures for representing an incomplete solution. In particular, it may be a sequence of operators [Fikes and Nilsson, 1971] or a partially ordered set [Tate, 1977]. Some problem solvers fully instantiate the operators, whereas other solvers use the unification of operator effects with the corresponding goals [Chapman, 1987]. Some systems mark relations among operators by causal links [McAllester and Rosenblitt, 1991], and others do not explicitly maintain these relations.

In PRODIGY, an incomplete solution consists of two parts, a total-order *head* and tree-structured *tail* (see Figure 2.3). The root of the tail's tree is the goal statement G, the other nodes are fully instantiated operators, and the edges are ordering constraints.

The tail is built by a backward chainer, which starts from the goal statement and adds operators, one by one, to achieve goal literals and preconditions of previously added operators. When the algorithm adds an operator to the tail, it *instantiates* the operator, that is, replaces all the variables with specific objects. The preconditions of a fully instantiated operator are a conjunction of literals, where every literal is an instantiated predicate.

The head is a sequence of instantiated operators that can be executed from the initial state. It is generated by the execution-simulating algorithm described in Section 2.2.3. The

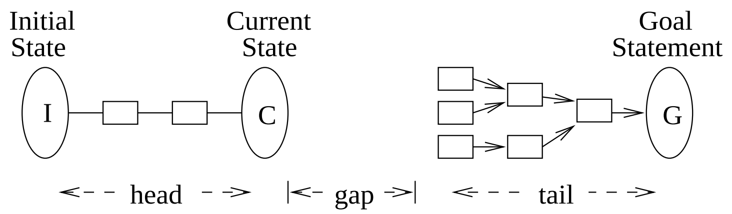


Figure 2.3: Representation of an incomplete solution: It consists of a total-order head, which can be executed from the initial state, and a tree-structured tail constructed by a backward chainer. The current state C is the result of applying the head operators to the initial state I.

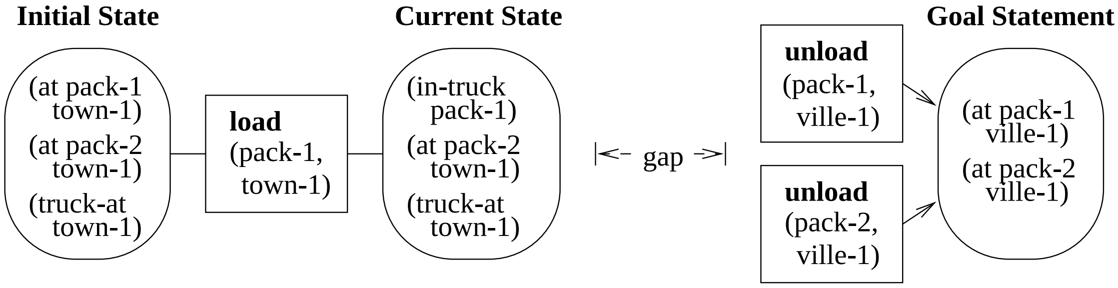


Figure 2.4: Example of an incomplete solution in the Trucking Domain: The head consists of a single operator, **load**; the tail comprises two **unload** operators, linked to the goal literals.

state C achieved by executing the head is the *current state*. In Figure 2.4, we illustrate an incomplete solution for the example trucking problem.

Since the head is a total-order sequence of operators that do not contain variables, the current state C is uniquely defined. The backward chainer, responsible for the tail, views C as its initial state. If the tail operators cannot be executed from the current state C, then there is a "gap" between the head and tail. The purpose of problem solving is to bridge this gap. For example, we can bridge the gap in Figure 2.3 by a sequence of two operators, "**load**(pack-2,town-1), **leave-town**(town-1,ville-1)."

2.2.3 Simulating execution

Given an initial state I and a goal statement G, PRODIGY begins with the empty head and tail, and modifies them, step by step, until it builds a complete solution. Thus, the initial incomplete solution has no operators and its current state is the same as the initial state, $C = I$.

At each step, PRODIGY can modify the current incomplete solution in one of two ways (see Figure 2.5). First, it can add an operator to the tail (operator t in the picture), to achieve a goal literal or a precondition of another operator. Tail modification is a job of the backward-chaining algorithm, described in Section 2.2.4.

Second, PRODIGY can move some operator op from the tail to the head (operator x in the picture). The preconditions of op must be satisfied in the current state C. The operator op becomes the last operator of the head, and the current state is modified according to the effects of op. The search algorithm usually has to select among several operators that can

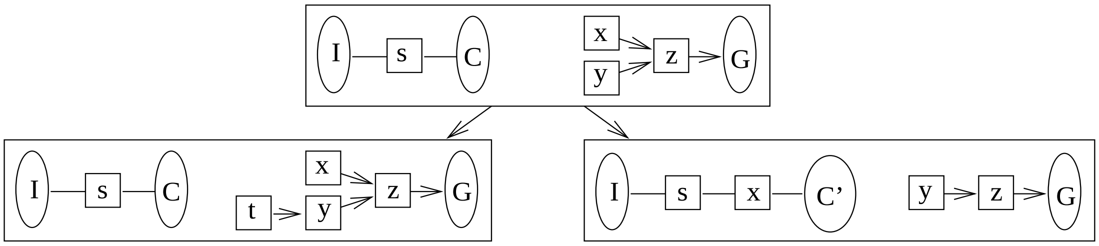


Figure 2.5: Modifying an incomplete solution: PRODIGY either adds a new operator to the tail tree (left), or moves one of the previously added operator to the head (right).

be moved to the head; thus, it needs to decide on the order of executing these operators.

Intuitively, we may imagine that the system executes the head operators in the real world and has already changed the world from its initial state I to the current state C. If the tail contains an operator whose preconditions are satisfied in C, then PRODIGY applies this operator and further changes the state. Because of this analogy with the real-world changes, moving an operator from the tail to the head is called the *application* of the operator; however, this term refers to *simulating* an operator application. Even if the execution of the head operators is disastrous, the world does not suffer: the search algorithm backtracks and tries an alternative execution sequence.

When the system applies an operator to the current state, it begins with the deletion effects, and removes the corresponding literals from the state; then, it performs addition of new literals. Thus, if the operator adds and deletes the same literal, the net result is adding it to the state.

For example, suppose that the current state includes the literal (truck-at town-1), and PRODIGY simulates the application of **leave-town**(town-1,town-1), whose effects are "del (truck-at town-1)" and "add (truck-at town-1)." The system first removes this literal from the state description, and then adds it back. If the system processed the effects in the opposite order, it would permanently remove the truck's location, thus obtaining an inconsistent state.

An operator application is the only way of updating the head. The system never inserts a new operator directly into the head, which means that it uses only goal-relevant operators in the forward chaining. The search terminates when the head operators achieve the goals; that is, the goal statement G is satisfied in C. If the tail is not empty at that point, it is dropped.

2.2.4 Backward chaining

We next describe the backward-chaining procedure that constructs the tree-structured tail of an incomplete solution. When the problem solver invokes this procedure, it adds a new operator to the tail, for achieving either a goal literal or a precondition literal of another tail operator. Then, it establishes a link from the newly added operator to the literal achieved by this operator, and adds the corresponding ordering constraint. For example, if the incomplete solution is as shown in Figure 2.4, then the procedure may add the operator **load**(pack-2,ville-

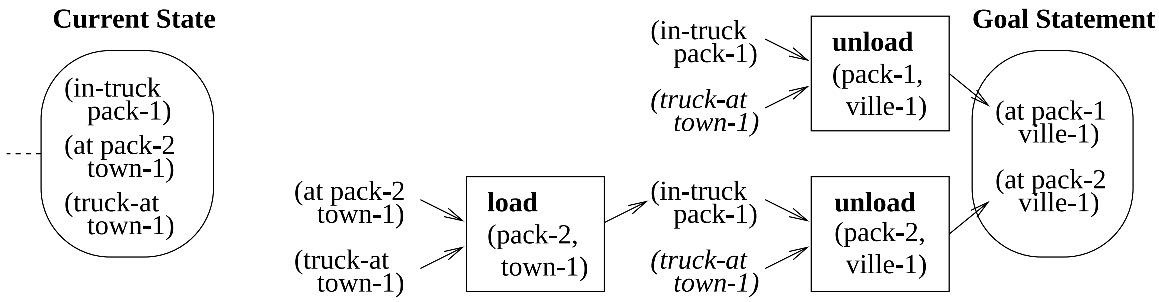


Figure 2.6: Example of the tail in an incomplete solution to a trucking problem. First, the backward chainer adds the **unload** operators, which achieve the two goal literals. Then, it inserts **load** to achieve the precondition (in-truck pack-1) of **unload**(pack-2,ville-1).

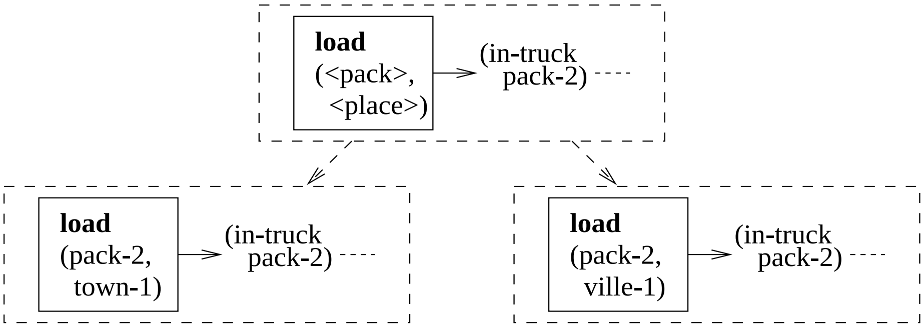


Figure 2.7: Instantiating a newly added operator: If the set of objects is as shown in Figure 2.2, PRODIGY can generate two alternative versions of **load** for achieving the subgoal (in-truck pack-2).

1) to achieve the precondition (in-truck pack-2) of **unload**(pack-2,ville-1) (see Figure 2.6). If the backward chainer uses an if-effect of an operator to achieve a literal, then the effect's conditions are added to the preconditions of the instantiated operator.

PRODIGY *tries to achieve a literal only if it is not true in the current state C and has not been linked to any tail operator.* Unsatisfied goal literals and preconditions are called *subgoals.* For example, the tail in Figure 2.6 has two identical subgoals, marked by italics.

Before inserting an operator into the tail, the solver fully instantiates it, that is, substitutes all free variables of the operator with specific object instances. Since the PRODIGY domain language allows the use of disjunctive and quantified preconditions, instantiating an operator may be a difficult problem. The system uses a constraint-based matching procedure that generates all possible instantiations [Wang, 1992].

For example, suppose that the backward chainer uses an operator **load**(<pack>,<place>) to achieve the subgoal (in-truck pack-2) (see Figure 2.7). First, PRODIGY instantiates the variable <pack> with the instance pack-2 from the subgoal literal. Then, it has to instantiate the other free variable, <place>. Since the domain has two places, town-1 and ville-1, the variable has two possible instantiations, which give rise to different branches in the search space (Figure 2.7).

In Figure 2.8, we summarize the search algorithm, which explores the space of incomplete solutions. The *Operator-Application* procedure builds the head and maintains the current state, whereas *Backward-Chainer* constructs the tail.

The algorithm includes five decision points, which give rise to different branches of the search space. It can backtrack over the decision to apply an operator (Line 2a), and over the choice of an "applicable" tail operator (Line 1b). It also backtracks over the choice of a subgoal (Line 1c), an operator that achieves it (Line 2c), and the operator's instantiation (Line 4c). We summarize the decision points in Figure 2.9.

Note that the first two choices (Lines 2a and 1b) enable the problem solver to consider different orderings of head operators. These two choices are essential for solving problems with interacting subgoals; they are analogous to the choice of ordering constraints in least-commitment algorithms.

2.2.5 Main versions

The algorithm in Figure 2.9 has five decision points, which allow flexible selection of operators, their instantiations, and order of their execution; however, these decisions give rise to a large branching factor. The use of built-in heuristics, which eliminate some of the available choices, may reduce the search space and improve the efficiency. On the negative side, such heuristics prune some solutions and may direct the search to a suboptimal solution, or even prevent finding any solution. Determining appropriate restrictions on the solver's choices is one of the major research problems.

Even though the described algorithm underlies all PRODIGY versions, from PRODIGY1 to FLECS, the versions differ in their use of decision points and built-in heuristics. Researchers investigated different trade-offs between flexibility and reduction of branching. They gradually increased the number of available decision points, from two in PRODIGY1 to all five in FLECS. We outline the use of the backtracking mechanism and its evaluation in the PRODIGY architecture.

The versions also differ in some features of the domain language, in the use of learning modules, and in the low-level implementation of search mechanisms. We do not discuss these differences; the reader may learn about them from the review article by Veloso *et al.* [1995].

PRODIGY1 and PRODIGY2

The early versions of PRODIGY had only two backtracking points: the choice of an operator (Line 2c in Figure 2.8) and the instantiation of the selected operator (Line 4c). The other three decisions were based on fixed heuristics, which did not give rise to multiple search branches. The algorithm preferred operator application to adding new operators (Line 2a), applied the tail operator that had been added last (Line 1b), and achieved the first unsatisfied precondition of the last added operator (Line 1c). This algorithm generated suboptimal solutions and sometimes failed to find any solution.

For example, consider the PRODIGY2 search for the problem in Figure 2.2. The solver adds **unload**(pack-1,ville-1) to achieve (at pack-1 ville-1), and **load**(pack-1,town-1) to achieve the precondition (in-truck pack-1) of **unload** (see Figure 2.10a). Then, it applies **load** and adds **leave-town**(town-1,ville-1) to achieve the precondition (truck-at ville-1) of unload (Figure 2.10b). Finally, PRODIGY applies **leave-town** and **unload** (Figure 2.10c), thus bringing only one package to the village.

Base-PRODIGY

1a. If the goal statement G is satisfied in the current state C, then return the head.

2a. Either

(i) *Backward-Chainer* adds an operator to the tail,

(ii) or *Operator-Application* moves an operator from the tail to the head.

Decision point: Choose between (i) and (ii).

3a. Recursively call *Base*-PRODIGY on the resulting incomplete solution.

Operator-Application

1b. Pick an operator op, in the tail, such that

(i) there is no operator in the tail ordered before op,

(ii) and the preconditions of op are satisfied in the current state C.

Decision point: Choose one of such operators.

2b. Move op to the end of the head and update the current state C.

Backward-Chainer

1c. Pick a literal l among the current subgoals.

Decision point: Choose one of the subgoal literals.

2c. Pick an operator op that achieves l.

Decision point: Choose one of such operators.

3c. Add op to the tail and establish a link from op to l.

4c. Instantiate the free variables of op.

Decision point: Choose an instantiation.

5c. If the effect that achieves l has conditions,

then add them to the operator preconditions.

Figure 2.8: Foundations of the PRODIGY search algorithm: The *Operator-Application* procedure simulates execution of operators, whereas *Backward-Chainer* selects operators relevant to the goal.

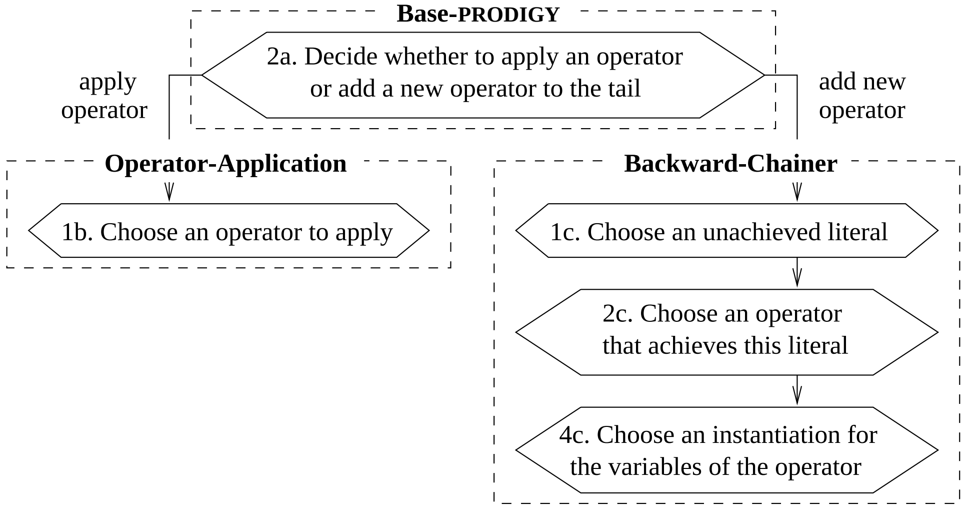


Figure 2.9: Main decision points in the PRODIGY search engine, summarized in Figure 2.8. Every decision point allows backtracking, thus giving rise to multiple branches of the search space.

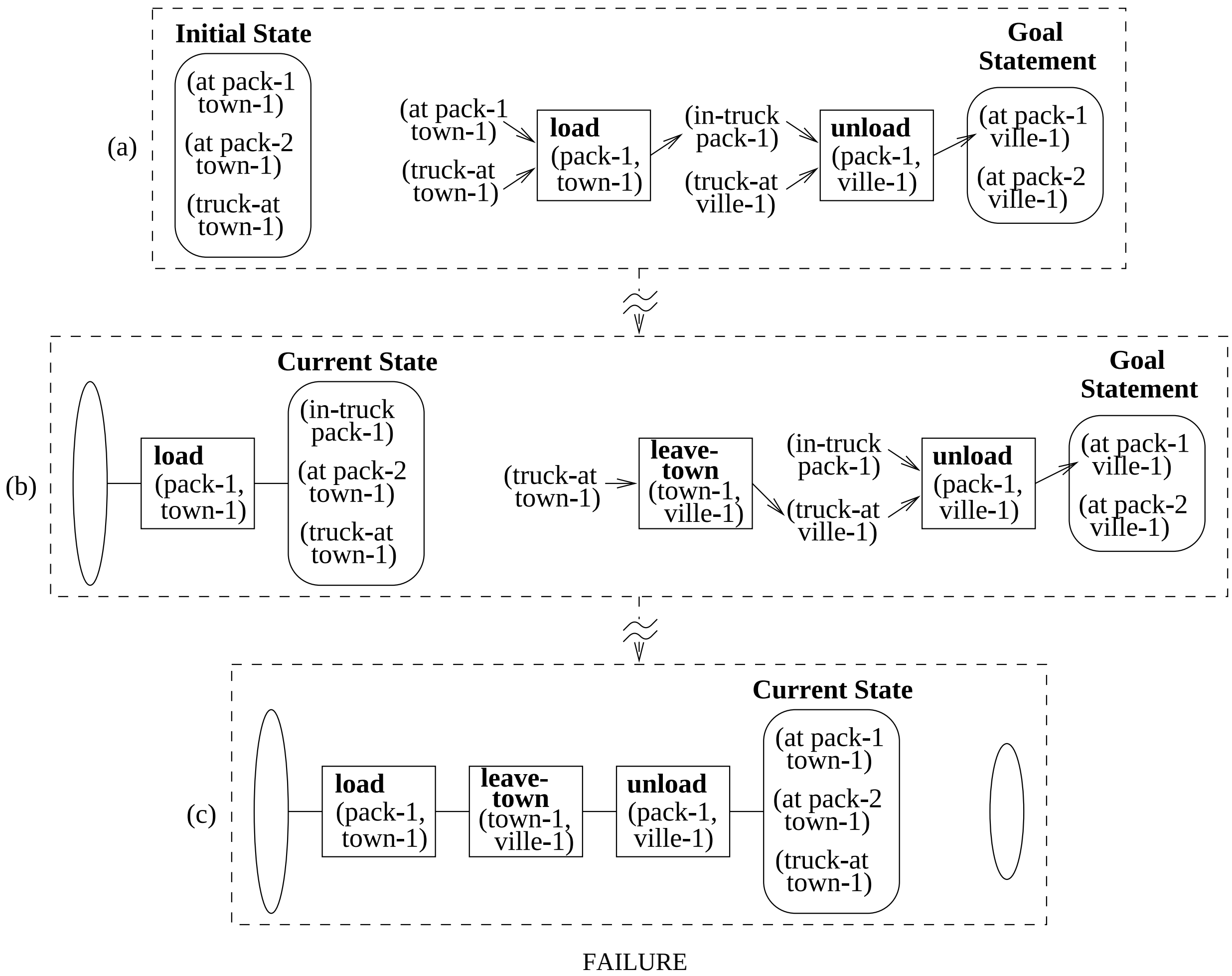


Figure 2.10: Incompleteness of PRODIGY2: The system fails to solve the trucking problem in Figure 2.2. Since the PRODIGY2 search algorithm always prefers the operator application to adding new operators, it cannot load both packages before driving the truck to its goal destination.

Since the algorithm uses only two backtracking points, it does not consider loading two packages before the ride or getting extra fuel before leaving the town; thus, it fails to solve the problem. This example demonstrates that the PRODIGY2 system is *incomplete,* that is, it may fail on a problem that has a solution. The user may improve the situation by providing domain-specific control rules, which enforce different choices of subgoals in Line 1c. Note that PRODIGY2 does *not* backtrack over these choices, and an inappropriate control rule may cause a failure. This approach often allows the enhancement of performance; however, it requires the human operator to assume the responsibility for completeness and solution quality.

NOLIMIT and PRODIGY4

During the work on the NOLIMIT system, Veloso added two more backtracking points, delaying the application of tail operators (Line 2a) and choosing a subgoal (Line 1c), and

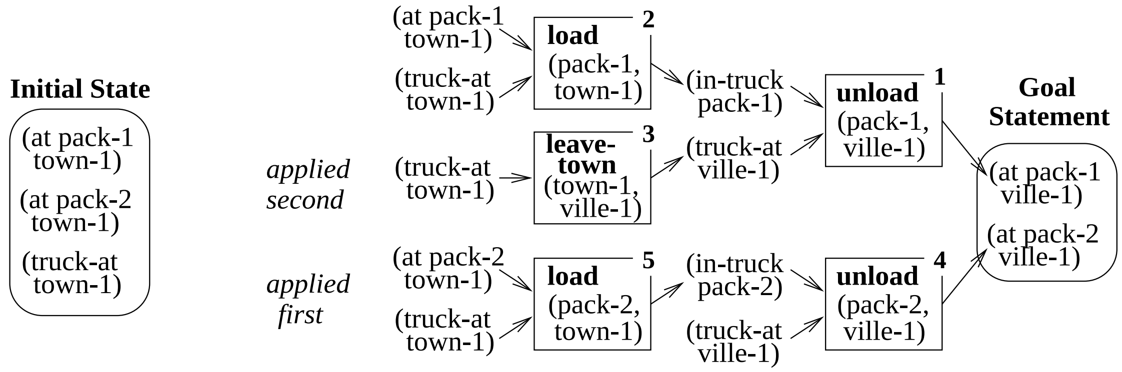


Figure 2.11: Example of inefficiency in the PRODIGY4 system; the boldface numbers, in the upper right corners of operators, mark the order of adding operators to the tail. Since the PRODIGY4 algorithm always applies the last added operator, it attempts to apply **leave-town** before one of the **load** operators, which leads to a deadend and requires backtracking.

later PRODIGY4 inherited these points. On the other hand, PRODIGY4 makes no decision in Line 1b: it always applies the last added operator. The absence of this decision point does not rule out any solutions, but sometimes negatively affects the search time.

For instance, if we use PRODIGY4 to solve the problem in Figure 2.2, it may generate the tail shown in Figure 2.11, where the numbers show the order of adding operators. We could now solve the problem by applying the two **load** operators, the **leave-town** operator, and then both **unload** operators; however, the solver cannot use this application order. The system applies **leave-town** before one of the **load** operators, which leads to a deadend. It then has to backtrack and construct a new tail, which allows the right order of applying the operators.

FLECS

The FLECS algorithm has all five decision points, but it does not backtrack over the choice of a subgoal (Line 1c), which means that only four points give rise to multiple search branches. Since backtracking over these points may produce an impractically large space, Veloso and Stone [1995] implemented general heuristics that further limit the space.

They experimented with two versions of the FLECS algorithm, called SAVTA and SABA, which differ in their choice between adding an operator to the tail and applying an operator (Line 2a). SAVTA prefers to apply tail operators before adding new ones, whereas SABA tries to delay their application.

Experiments have demonstrated that the greater flexibility of PRODIGY4 and FLECS usually gives an advantage over PRODIGY2, despite the larger branching factor. The relative effectiveness of PRODIGY4, SAVTA, and SABA depends on the specific domain, and the right choice among these algorithms is often essential for performance [Stone *et al.*, 1994].

Veloso has recently fixed a minor bug in the implementation of SABA, which sometimes led to inappropriate search decisions; however, she has not yet reported empirical evaluation of the corrected algorithm. Note that we employed the original version of SABA in experiments with the $\mathcal{S}$HAPER system, since the bug has been found after the completion of our work.

cushion(<pack>, <place>)
<pack>: type Package
<place>: type Place
Pre: (or (at <pack> <place>)
(in-truck <pack>))
Eff: del (fragile <pack>)

(a) Disjunction.

Goal Statement

(exists <pack> of type Package
(at <pack> ville-1))

(b) Existential quantification.

Goal Statement

(forall <pack> of type Package
(at <pack> ville-1))

(c) Universal quantification.

Figure 2.12: Examples of disjunction and quantification in the PRODIGY domain language. The user may utilize these constructs in precondition expressions and goal statements.

2.3 Extended domain language

The PRODIGY domain language is an extension of the STRIPS language [Fikes and Nilsson, 1971]. The STRIPS system used a limited description of operators, and PRODIGY researchers added several advanced features, which allowed encoding of large-scale domains. The new features included complex preconditions and goal expressions (Section 2.3.1), inference rules, (Section 2.3.2), and flexible use of object types (Section 2.3.3).

2.3.1 Extended operators

The PRODIGY domain language allows complex logical expressions in operator preconditions, if-effect conditions, and goal statements. They may include not only negations and conjunctions, but also disjunctions and quantifications. The language also enables the user to specify the costs of operators, which serve as a measure of solution quality.

Disjunctive preconditions

To illustrate the use of disjunction, we consider a variation of the **cushion** operator, given in Figure 2.12(a). In this example, we can cushion a package when it is inside or near the truck.

When PRODIGY instantiates an operator with disjunctive preconditions, it generates an instantiation for one element of the disjunction and discards all other elements. For example, if the solver has to cushion pack-1, it may choose the instantiation (at pack-1 town-1), which matches (at <pack> <place>), and discard the other element, (in-truck <pack>).

If the initial choice does not lead to a solution, the solver backtracks and considers the instantiation of another element. For instance, if the selected version of the **cushion**(o)perator has proved inadequate, PRODIGYmay discard the first element of the conjunction, (at <pack> <place>), and choose the instantiation (in-truck pack-1) of the other element.

Quantified preconditions

We illustrate the use of quantifiers in Figures 2.12(b) and 2.12(c). In the first example, the solver has to transport *any* package to ville-1. In the second example, it has to deliver *all* packages to the village.

When the problem solver instantiates an existential quantification, it selects one object of the specified type. For example, it may decide to deliver pack-1 to ville-1, thus replacing the goal in Figure 2.12(b) by (at pack-1 ville-1). If the chosen object does not lead to a solution, PRODIGY backtracks and tries another one. When instantiating a universally quantified expression, the solver treats it as a conjunction over all matching objects.

Instantiated operators

The PRODIGY language allows arbitrary logical expressions, which may contain multiple levels of negations, conjunctions, disjunctions and quantifications. When adding an operator to the tail, the problem solver generates all possible instantiations of its preconditions and chooses one of them. If the solver backtracks, it chooses an alternative instantiation. Every instantiation is a conjunction of literals, some of which may be negated; it has *no* disjunctions, quantifications, or negated conjunctions.

Wang [1992] has designed an advanced algorithm for generating possible instantiations of operators and goal statements. In particular, she developed an efficient mechanism for pruning inconsistent choices of objects and provided heuristics for selecting the most promising instantiations.

Costs

The use of operator costs allows us to measure the quality of complete solutions. We assign nonnegative numerical costs to instantiated operators, and define a solution cost as the sum of its operator costs. The lower the cost, the better the solution.

The authors of the original PRODIGY architecture did not provide support for operator costs, and usually measured the solution quality by the number of operators. Pérez [1995] has implemented a mechanism for using operator costs during her exploration of control rules for improving solution quality; however, she did not incorporate costs into the main version.

We re-implemented the cost mechanism during the work on the $\mathcal{S}$HAPER system. In Figure 2.13, we give an example of cost encoding. For every operator, the user specifies a Lisp function, whose arguments are operator variables. Given specific object instances, the function returns the corresponding cost, which must be a nonnegative real number. If the operator cost does not depend on the instantiation, it may be specified by a number rather than a function. If the user does not encode a cost, then by default it is 1.

The example in Figure 2.13(a) includes two cost functions, called *leave-cost* and *load-cost.* We give pseudocode for these functions (Figure 2.13b) and their real encoding in the PRODIGY system (Figure 2.13c).

The cost of driving between two locations is linear in the distance, determined by the *miles* function. The user may specify distances by a matrix or by a list of initial-state

leave-town(<from>, <to>)
<from>: type Town
<to>: type Place
...
Cost: leave-cost(<from>,<to>)

load(<pack>, <place>)
<pack>: type Package
<place>: type Place
...
Cost: load-cost(<place>)

cushion(<pack>)
<pack>: type Package
...
Cost: 5

(a) Use of cost functions and constant costs.

leave-cost*(<from>,<to>)*
Return 0.2 · *miles(<from>,<to>)* + 5.

load-cost*(<place>)*
If *<place>* is of type Village,
then, return 4; esle, return 3.

(b) Pseudocode of the cost functions.

```
(defun leave-cost (<from> <to>)
  (+ (* 0.2 (miles <from> <to>)) 5))

(defun load-cost (<place>)
  (if (eq (type-name (prodigy-object-type <place>)) 'Village)
    4 3)
```

(c) Actual LISP functions.

Figure 2.13: Encoding of operator costs: The user may specify a constant cost value or, alternatively, a Lisp function that inputs operator variables and returns a nonnegative real number. If the description of an operator does not include a cost, PRODIGY assumes that it is 1.

literals, and should provide the appropriate look-up procedure. The loading cost depends on the location type; it is larger in villages. Finally, the cost of the **cushion** operator is constant.

When the problem solver instantiates an operator, it calls the corresponding function to determine the cost of the resulting instantiation. If the returned value is negative, the system signals an error. Note that, since incomplete solutions consist of fully instantiated operators, the solver can determine the cost of every intermediate solution.

2.3.2 Inference rules

The PRODIGY language supports two mechanisms for changing the domain state, operators and inference rules, which have identical syntax but differ in semantics. Operators encode actions that change the world, whereas rules point out implicit properties of the world state.

Example

In Figure 2.14, we show three inference rules for the Trucking Domain. In this example, we have made two modifications to the original domain description (see Figure 2.1). First, the **cushion** operator adds (cushioned <pack>) instead of deleting (fragile <pack>), and the **add-fragile** rule indicates that uncushioned packages are fragile. Thus, the user does not have to specify fragility in the initial state.

Second, the domain includes the type County and the predicate (within <place> <county>). Note that this predicate is static, that is, it is not an effect of any operator or inference rule. We use the **add-truck-in** rule to infer the county of the truck's current location. For example, if the truck is at town-1, and town-1 is within county-1, then the rule adds (truck-in

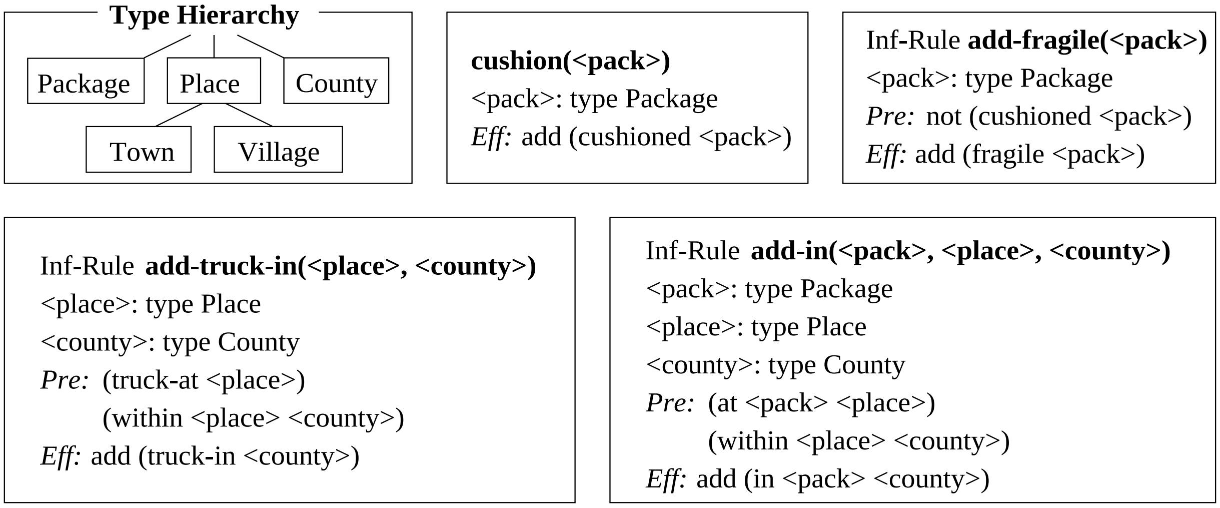


Figure 2.14: Encoding of inference rules in the PRODIGY domain language. These rules point out indirect results of changing the world state; their syntax is identical to that of operators.

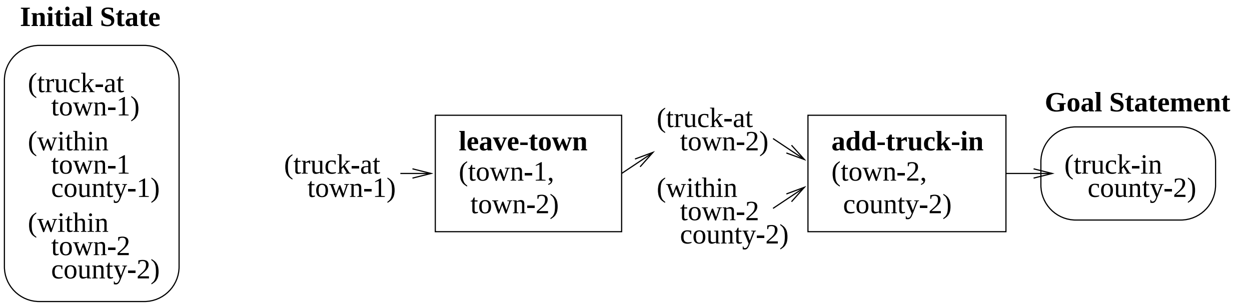


Figure 2.15: Use of an inference rule in backward chaining: PRODIGY links the **add-truck-in** rule to the goal literal, and then adds **leave-town** to achieve the rule's precondition (truck-at town-2).

county-1) to the current state. Similarly, we use add-in to infer the current county of each package.

Use of inferences

The encoding of inference rules is the same as that of operators, which may include disjunctive and quantified preconditions, and if-effects; however, the rules have no costs and their use does not affect the overall solution cost.

The use of inference rules is also similar to that of operators: the problem solver adds an instantiated rule to the tail, for achieving the selected subgoal, and applies the rule when its preconditions hold in the current state. We illustrate it in Figure 2.15, where the solver uses the **add-truck-in** rule to achieve the goal, and then adds **leave-town** to achieve a rule's precondition.

If the system applies an inference rule and *later* adds an operator that invalidates the rule's preconditions, then it removes the rule's effects from the state. For example, the inference rule in Figure 2.16(a) adds (truck-in town-2) to the state. If the system then applies

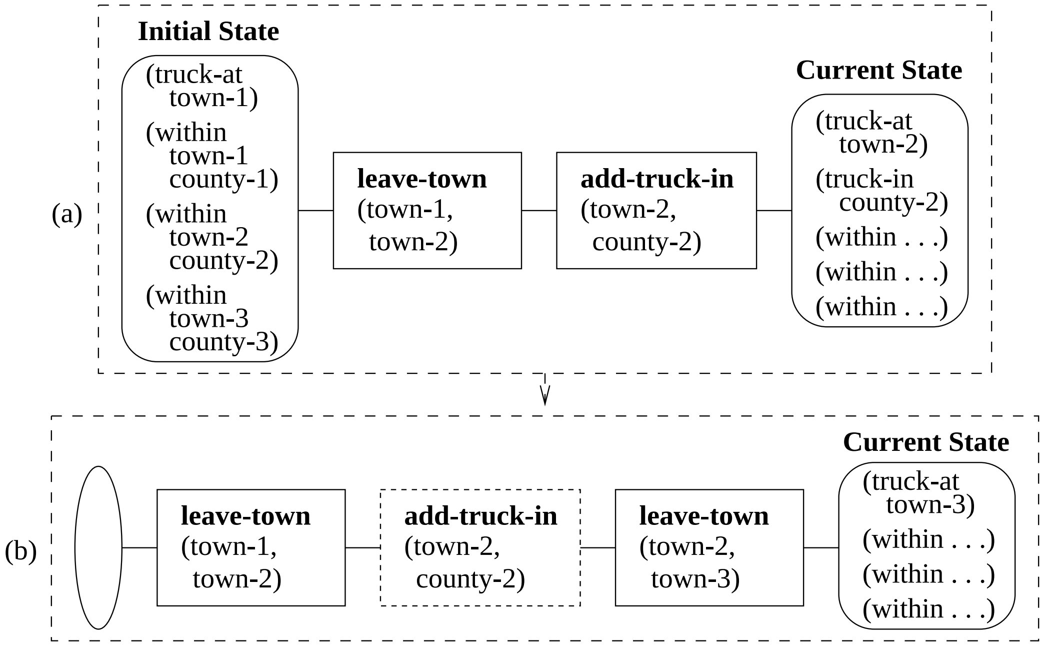


Figure 2.16: Cancelling the effects of an inference rule upon the negation of its preconditions: When PRODIGY applies **leave-town**(town-2,town-3), it negates the precondition (truck-at town-2) of the **add-truck-in** rule; hence, the system removes the effects of this rule from the current state.

leave-town (Figure 2.16b), it negates the preconditions of **add-truck-in** and, hence, cancels its effects. This semantics differs from the use of operators, whose effects remain in the state, unless deleted by opposite effects of other operators.

Eager and lazy rules

The *Backward-Chainer* algorithm selects rules at its discretion and may disregard unwanted rules. On the other hand, if some inference rule has an undesirable effect, it should be applied regardless of the solver's choice. For example, if pack-1 is not cushioned in the initial state, the system should immediately add (fragile pack-1) to the state.

When the user encodes a domain, she has to mark all rules that have unwanted effects. When the preconditions of a marked rule hold in the current state, the system applies it at once, even if it is not in the tail. The marked inference rules are called *eager rules*, whereas the others are *lazy rules.* Note that *Backward-Chainer* may use both eager an lazy rules, and the only special property of eager rules is their forced application in the matching states. If the user wants *Backward-Chainer* to disregard some eager rules, she may provide a control rule that prevents their use in the tail.

Truth maintenance

When the PRODIGY system applies an operator or inference rule, it updates the current state and then identifies the previously applied rules whose preconditions no longer hold. If the system finds such rules, it modifies the state by removing their effects. If some rules that remain in force have if-effects, the system must check the conditions of every if-effect, which may also lead to modification of the state. Next, PRODIGY looks for an eager rule whose conditions hold in the resulting state. If the system finds such a rule, then it applies the rule and further changes the state.

If inference rules interact with each other, then this process may involve a chain of rule applications and cancellations. It terminates when the system gets to a state that does not require applying a new eager rule or removing effects of old rules. This chain of state modifications, which does not involve search, is similar to the firing of productions in the Soar system [Laird *et al.*, 1986; Golding *et al.*, 1987].

Blythe designed an efficient truth-maintenance procedure, which keeps track of all applicable inference rules and controls the described forward chaining. The solver invokes this procedure after each application of an operator or inference rule from the tail.

If the user provides inference rules, she has to ensure that the resulting inferences are consistent. In particular, a rule must not negate its own preconditions. If two rules may be applied in the same state, they must not have opposite effects. If a domain includes several eager rules, they should not cause an infinite cyclic chain of forced application. The PRODIGY system does *not* check for such inconsistencies, and an inappropriate rule set may cause unpredictable results.

2.3.3 Complex types

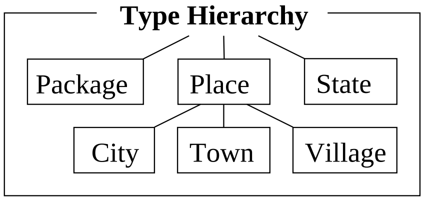
We have already explained the use of a type hierarchy (see Figure 2.1), which defines object classes and enables the user to specify the allowed values of variables in operator preconditions, conditions of if-effects, and goal statements. For example, the possible values of the <from> variable in **leave-town** include all towns, but not villages. The early versions of the PRODIGY system did not support a type hierarchy. Veloso designed a typed domain language during her work on NOLIMIT, and the authors of PRODIGY4 further developed the mechanism for using types.

A type hierarchy is a tree, whose nodes are called *simple types*. For instance, the hierarchy in Figure 2.1 has five simple types: Package, Town, Village, Place, and the root type that includes all objects. We have illustrated the use of simple types in the operator encoding; however, they often do not provide sufficient flexibility.

For example, consider the type hierarchy in Figure 2.17 and suppose that truck may get in extra fuel in a town or city, but not in a village. We cannot encode this constraint with simple types, unless we define an additional type. The PRODIGY language includes a mechanism for defining complex constraints, through disjunctive and functional types.

Disjunctive types

We illustrate the use of a disjunctive type in Figure 2.17, where it specifies the possible values of <from> and <place>. The user specifies a disjunctive type as a set of simple types; in our example, it includes Town and City. When the problem solver instantiates the corresponding variable, it uses an object that belongs to any of these types. For instance, the system may use the **leave-town** operator for departing from a town or city.



leave-town(<from>, <to>)
<from>: type (or Town City)
<to>: type Place
Pre: (truck-at <from>)
Eff: del (truck-at <from>)
add (truck-at <to>)

fuel(<place>)
<place>: type (or Town City)
Pre: (truck-at <place>)
Eff: add (extra-fuel)

Figure 2.17: Disjunctive type: The <from> and <place> variables are declared as (or Town City), which means that they may be instantiated with objects of two simple types, Town and City.

leave-town(<from>, <to>)
<from>: type (or Town City)
<to>: type Place
connected(<from>,<to>)
Pre: (truck-at <from>)
Eff: del (truck-at <from>)
add (truck-at <to>)

(a) Use of a functional type.

connected*(<from>, <to>)*
If *<from> = <to>,*
then return False;
else, return True.

(b) Pseudocode for the function.

```
(defun connected (<from> <to>)
   (not (eq <from> <to>)))
```

(c) Actual LISP function.

Figure 2.18: Functional type: When PRODIGY instantiates **leave-town**, it ensures that <from> and <to> are connected by a road. The user has to implement a boolean Lisp function for testing the connectivity. We give an example function, which defines the fully connected graph of roads.

Functional types

We give an example of a functional type in Figure 2.18, where it limits the values of the <to> variable. The description of a functional type consists of two parts: a simple or disjunctive type, and a boolean test function. The system first identifies all objects of the specified simple or disjunctive type, and then eliminates the objects that do not satisfy the test function. The remaining objects are the valid values of the declared variable. In our example, the valid values of the <to> variable include all places that have road connections with the <from> location.

The boolean function is an arbitrary Lisp procedure, whose arguments are the operator variables. The function *must* input the variable described by the functional type. In addition, it may input variables declared *before* this functional type; however, the function cannot input variables declared after it. For example, we use the <from> variable in limiting the values of <to>; however, we cannot use the <to> variable as an input to a test function for <from>, because of the declaration order.

For instance, if every place is connected with every other place except itself, then we use the test function given in Figure 2.18(b). The domain encoding must include a Lisp implementation of this function, as shown in Figure 2.18(c).

Use of test functions

When the system instantiates a variable with a functional type, it identifies all objects of the specified simple or disjunctive type, prunes the objects that do not satisfy the test function, and then selects an object from the remaining set. If the user specifies not only functional types but also control rules, which further limit suitable instantiations, then the generation of instantiated operators becomes a complex matching problem. Wang [1992] investigated it and developed an efficient matching algorithm.

Test functions may use any information about the current incomplete plan, other nodes in the search space, and the global state of the system, which allows unlimited flexibility in constraining operator instantiations. In particular, they enable us to encode functional effects, that is, operator effects that depend on the current state.

Generator functions

The system also supports the use of *generator functions* in the specification of variable types. These functions generate and return a set of allowed values, instead of testing the available values. The user has to specify a simple or disjunctive type along with a generator function. When the system uses the function, it checks whether all returned objects belong to the specified type and prunes the extraneous objects.

In Figure 2.18, we give an example that involves both a test function, called *positive,* and a generator function, *decrement.* In this example, the system keeps track of the available space in the trunk. If there is no space, it cannot load more packages. We use the generator function to decrement the available space after loading a package. The function always returns one value, which represents the remaining space.

When the user specifies a simple or disjunctive type used with a generator function, she may define a numerical type that includes infinitely many values. For instance, the Trunk-Space type in Figure 2.18 may comprise all natural numbers. On the other hand, the generator function always returns a finite set. The PRODIGY manual [Carbonell *et al.*, 1992] contains a more detailed description of infinite types.

2.4 Search control

The efficiency of problem solving depends on the search space and the order of expanding nodes of the space. The nondeterministic PRODIGY algorithm in Figure 2.32 defines the search space, but does not specify the exploration order. The algorithm has several decision points (see Figure 2.33), which require heuristics for selecting appropriate branches of the search space.

The PRODIGY architecture includes a variety of search-control mechanisms, which combine general heuristics, domain-specific experience, and advice by the human user. Some of the basic mechanisms are an integral part of the search algorithm, hard-coded into the system; however, most mechanisms are optional, and the user can enable or disable them at her discretion.

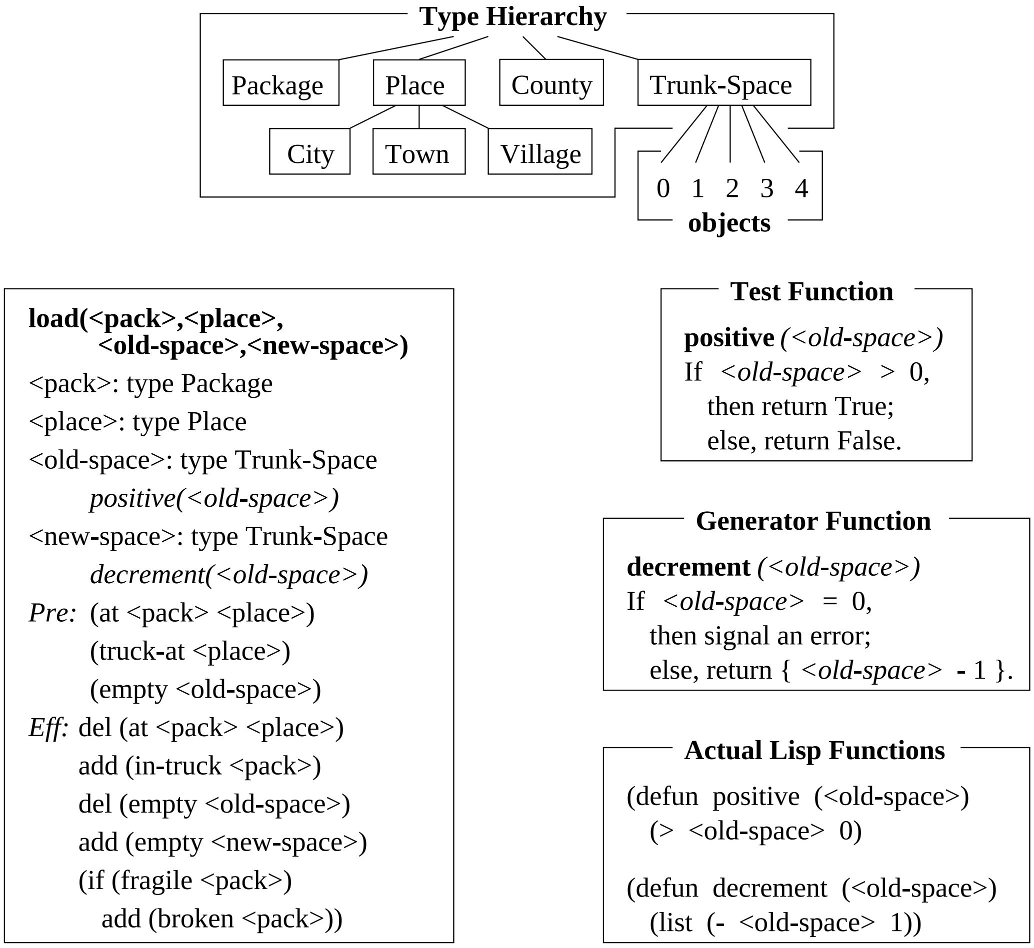


Figure 2.19: Generator function: The user provides a Lisp function, called *decrement*, which generates instances of <new-space>; these instances must belong to the specified type, Trunk-Space.

We outline some control mechanisms, including heuristics for avoiding redundant search (Section 2.4.1), main knobs for adjusting the search strategy (Section 2.4.2), and the use of control rules to guide the search (Section 2.4.3). The reader may find an overview of other control techniques in the article by Blythe and Veloso [1992], which explains dependency-directed backtracking in PRODIGY, the use of limited look-ahead, and some heuristics for choosing appropriate subgoals and instantiations.

2.4.1 Avoiding redundant search

We describe three basic techniques for eliminating redundant branches of the search space. These techniques improve the performance in almost all domains and, hence, they are hard-coded into the search algorithm, which means that the user cannot turn them off.

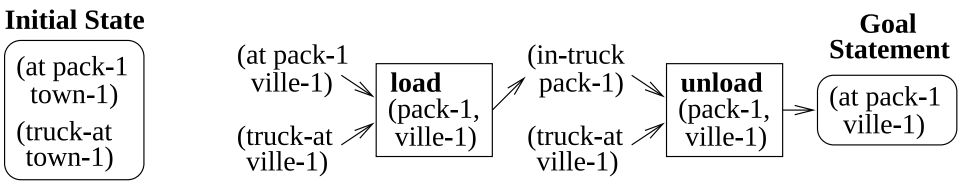


Figure 2.20: Goal loop in the tail: The precondition (at pack-1 ville-1) of the **load** operator is the same as the goal literal; hence, the solver has to backtrack and choose another operator.

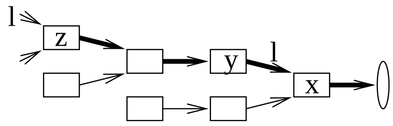


Figure 2.21: Detection of goal loops: The backward changer compares the precondition literals of a newly added operator z with the links between z and the goal statement. If some precondition l is identical to one of the link literals, then the solver backtracks.

Goal loops

We present a mechanism that prevents PRODIGY from running in simple circles. To illustrate it, consider the problem of delivering pack-1 from town-1 to ville-1 (see Figure 2.20). The solver first adds **unload**(pack-1,ville-1) and then may try to achieve its precondition (in-truck pack-1) by **load**(pack-1,ville-1); however, the precondition (at pack-1 ville-1) of **load** is identical to the goal and, hence, achieving it is as difficult as solving the original problem.

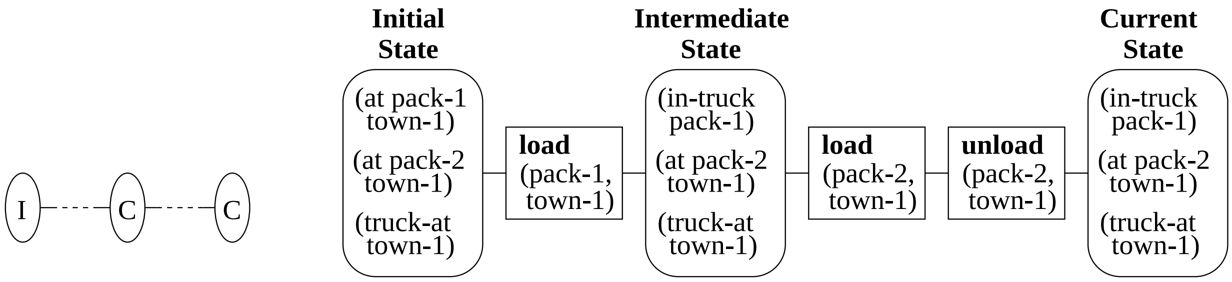
We call it a *goal loop,* which arises when a precondition of a newly added operator is identical to the literal of some link on the path from this operator to the goal statement. We illustrate it in Figure 2.21, where thick links mark the path from a new operator z to the goal. The precondition l of z makes a loop with an identical precondition of x, achieved by y.

When the problem solver adds an operator to the tail, it compares the operator's preconditions with the links between this operator and the goal. If the solver detects a goal loop, it backtracks and tries either a different instantiation of the operator or an alternative operator that achieves the same subgoal. For example, the solver may generate a new instantiation of the **load** operator, **load**(pack-1,town-1).

State loops

The problem solver also watches for loops in the head of an incomplete solution, called *state loops.* Specifically, it verifies that the current state differs from all previous states. If the current state is identical to some earlier state (see Figure 2.22a), then the solver discards the current incomplete solution and backtracks.

We illustrate a state loop in Figure 2.22, where the application of two opposite operators, **load** and **unload**, leads to a repetition of an intermediate state. The solver would detect this redundancy and either delay the application of **unload** or use a different instantiation.



(a) State loop. (b) Example of a loop.

Figure 2.22: State loops in the head of an incomplete solution: If the current state C is the same as one of the previous states, then the problem solver backtracks. For example, if PRODIGY applies **unload**(pack-2,town-1) immediately after **load**(pack-2,town-1), then it creates a state loop.

Satisfied links

Next, we describe the detection of redundant tail operators, illustrated in Figure 2.23. In this example, PRODIGY is solving the problem in Figure 2.2, and it has constructed the tail shown in Figure 2.23(a). Note that the literal (truck-in ville-1) is a precondition of two different operators in this solution, **unload**(pack-1,ville-1) and **unload**(pack-2,ville-1). Thus, they introduce two identical subgoals, and the solver adds two copies of the operator **leave-town**(town-1,ville-1) to achieve these subgoals.

Such situations arise because PRODIGY links each tail operator to only one subgoal, which simplifies the maintenance of links. When the solver applies an operator, it detects and skips redundant parts of the tail. For example, suppose that it has applied the two **load** operators and one **leave-town**, as shown in Figure 2.23(b). The precondition (truck-in ville-1) of the tail operator **unload** now holds in the current state, and the solver skips the tail operator **leave-town**, linked to this precondition.

When a tail operator achieves a precondition that holds in the current state, we call the corresponding link *satisfied.* We show this situation in Figure 2.24(a), where the precondition l of x is satisfied, which makes the dashed operators redundant.

The problem solver keeps track of satisfied links, and updates their list after each modification of the current state. When the solver selects a tail operator to apply (line 1b in Figure 2.8) or a subgoal to achieve (line 1c), it ignores the tail branches that support satisfied links. Thus, it would not consider the dashed operators in Figure 2.24 and their preconditions.

If the algorithm applies the operator x, it discards the dashed branch that supports a precondition of x (Figure 2.24b). This strategy allows the deletion of redundant operators from the tail. Note that the solver discards the dashed branch only *after* applying x. If it decides to apply some other operator before x, it may delete l, in which case dashed operators become useful again.

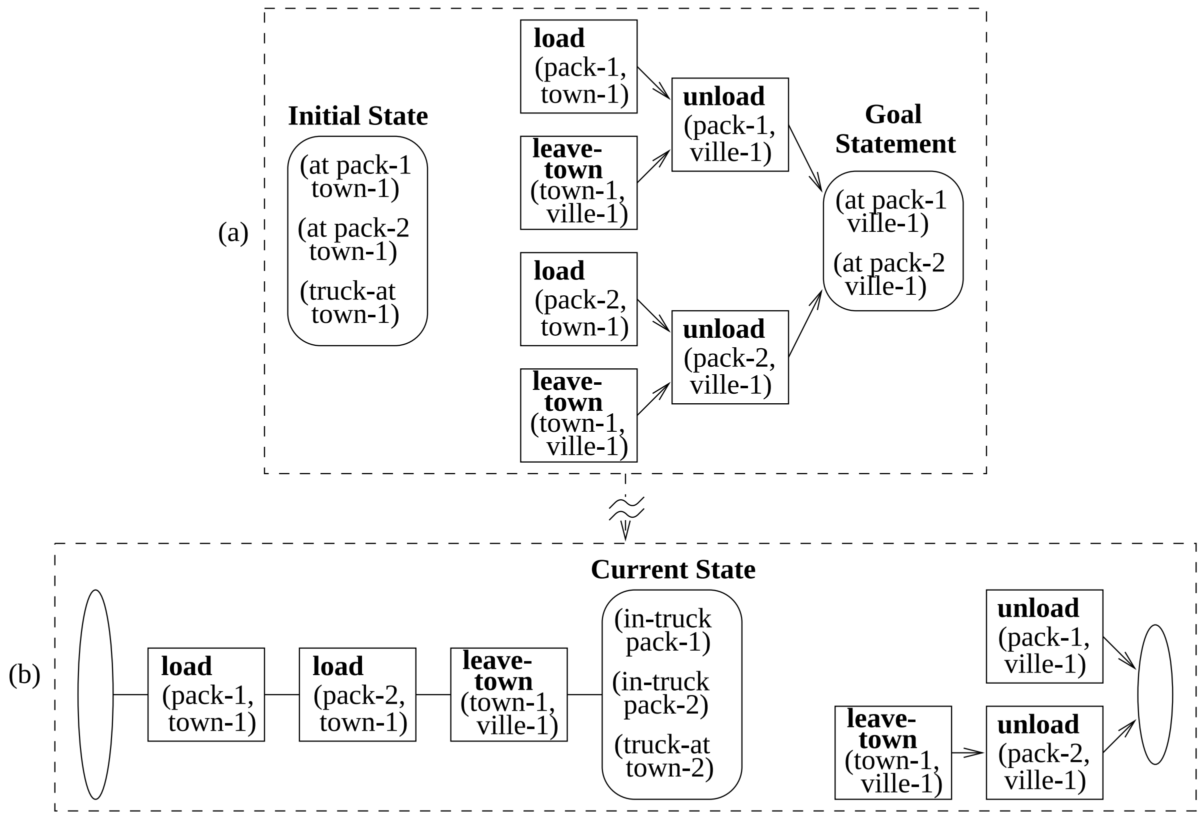


Figure 2.23: Satisfied link: After the solver has applied three operators, it notices that all preconditions of **unload**(pack-2,ville-1) hold in the current state; hence, it omits the tail operator **leave-town**, which is linked to a satisfied precondition of **unload**.

2.4.2 Knob values

The PRODIGY architecture includes several knob variables, which allow the user to adjust the search strategy to a current domain. and changes in their values may have a drastic impact on performance. Some of the knobs are numerical values, such as the search depth, whereas others specify the choices among alternative techniques and heuristics. We list some of the main knob variables, which were used in our experiments with the $\mathcal{S}$HAPER system.

Depth limit

The user usually limits the search depth, which results in backtracking upon reaching the pre-set limit. If the system explores all branches of the search space to the specified depth and does not find a solution, then it terminates with failure. Note that the number of operators in a solution is proportional to the search depth; hence, limiting the depth is equivalent to limiting the solution length.

After adding operator costs to the PRODIGY language, we provided a knob for limiting the solution cost. If the system constructs a partial solution whose cost is greater then the limit, it backtracks and considers an alternative branch. If the user bounds both search depth and solution cost, the solver backtracks upon reaching either limit.

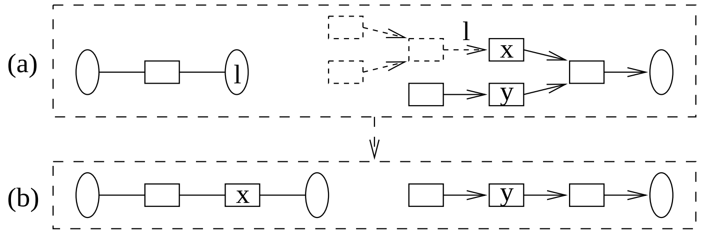


Figure 2.24: Identification of satisfied links: The solver keeps track of all link literals that are satisfied in the current state, and disregards the tail operators that support these satisfied literals.

The effect of these bounds varies across domains and specific problems. Sometimes, they improve not only solution quality but also efficiency, by preventing a long descent into a branch that has no solutions. In other domains, they cause an extensive search instead of fast generation of a suboptimal solution. If the search space has no solution within the specified bound, then the system fails to solve the problem, which means that a depth limit may cause a failure on a solvable problem.

Time limit

By default, the problem solver runs until it either finds a solution or exhausts the available search space. If it takes too long, the user may enter a keyboard interrupt or terminate the execution.

Alternatively, she may pre-set a time limit before invoking the solver and then the system automatically interrupts the search upon reaching this limit. We will analyze the role of time limits in Chapter 7.

The user may also bound the number of expanded nodes in the search space, which causes an interrupt upon reaching the specified node number. If she limits both running time and node number, then the search terminates after hitting either bound.

Search strategies

The system normally uses depth-first search and terminates upon finding any complete solution. The user has two options for changing this default behavior. First, PRODIGY allows breadth-first exploration; however, it is usually much less efficient than the default strategy. Moreover, some heuristics and learning modules do not work with breadth-first search.

Second, the user may request *all* solutions to a given problem. Then, the solver explores the entire search space and outputs all available solutions, until it exhausts the space, gets a keyboard interrupt, or reaches a time or node bound. The system also allows search for an *optimal* solution. This strategy is similar to the search for all solutions; however, when finding a new solution, the system reduces the cost bound and then looks only for better solutions. If the solver gets an interrupt, it outputs the best solution found by that time.

(a) Select Rule

If (truck-at <to>) is the current subgoal
and **leave-town**(<from>,<to>) is used to achieve it
and (truck-at <place>) holds in the current state
and <place> if of type Town
Then select instantiating <from> with <place>

(b) Reject Rule

If (truck-at <place>) is a subgoal
and (in-truck <pack>) is a subgoal
Then reject the subgoal (truck-at <place>)

Figure 2.25: Examples of control rules, which encode domain-specific heuristics for guiding PRODIGY search. The user may provide rules that represent her knowledge about the domain. Moreover, the system includes several mechanism for automatic construction of control heuristics.

2.4.3 Control rules

The efficiency of depth-first search crucially depends on the heuristics for selecting appropriate branches of the search space, as well as on the order of exploring these branches. The PRODIGY architecture provides a general mechanism for specifying search heuristics, in the form of control rules. These rules usually encode domain-specific knowledge, but they may also represent general domain-independent techniques.

A *control rule* is an *if-then* rule that specifies appropriate branching decisions, which may depend on the current state, subgoals, and other features of the current incomplete solution, as well as on the global state of the search space. The PRODIGY domain language provides a mechanism for hand-coding control rules. In addition, the architecture includes several learning mechanisms for automatic generation of domain-specific rules. The development of these mechanisms has been one of the main goals of the PRODIGY project.

The system uses three rule types, called select, reject, and prefer rules. A *select rule* points out appropriate branches of the search space. When its applicability conditions match the current incomplete solution, the rule generates one or more promising choices. For example, consider the control rule in Figure 2.25(a). When the problem solver uses the **leave-town** operator for moving the truck to some destination, the rule indicates that the truck should go there directly from its current location.

A *reject rule* determines inappropriate choices and removes them from the search space. For instance, the rule in Figure 2.25(b) indicates that, if PRODIGY has to load the truck and drive it to a certain place, then it should delay driving until after loading.

Finally, a *prefer rule* specifies the order of exploring branches, without pruning any of them. For example, we may replace the select rule in Figure 2.25 with an identical prefer rule, which would mean that the system should first try going directly from the truck's current location to the destination, but keep the other options open for later consideration. For some problems, this rule is more appropriate than the more restrictive select rule.

At every decision point, the system identifies all applicable rules and uses them to make appropriate choices. First, it uses an applicable select rule to choose candidate branches of the search space. If the current incomplete solution matches several select rules, the system arbitrarily selects one of them. If no select rules are applicable, then all available branches become candidates. Next, PRODIGY applies all reject rules that match the current solution and prunes every candidate branch indicated by at least one of these rules. Note that select and reject rules sometimes prune branches that lead to a solution; hence, they may prevent the system from solving some problems.

After using select and reject rules to prune branches at the current decision point, PRODIGY applies prefer rules to determine the order of exploring the remaining branches. If the system has no applicable prefer rules, or the applicable rules contradict each other, then it relies on general heuristics for selecting the exploration order.

If the system uses numerous control rules, matching of their conditions at every decision point may take significant time, which sometimes defeats the benefits of the right selection [Minton, 1990]. Wang [1992] has implemented several techniques that improve the matching efficiency; however, the study of the trade-off between matching time and search reduction remains an open problem.

2.5 Completeness

A search algorithm is *complete* if it finds a solution for every solvable problem. This notion does not involve a time limit, which means that an algorithm may be complete even if it takes an impractically long time for some problems.

Even though researchers used the PRODIGY search engine in multiple studies of learning and search, the question of its completeness had remained unanswered for several years. Veloso demonstrated the incompleteness of PRODIGY4 in 1995. During the work on $\mathcal{S}$HAPER, we further investigated completeness issues, in collaboration with Blythe.

The investigation showed that, to date, all PRODIGY algorithms had been incomplete; moreover, it revealed the specific reasons for their incompleteness. Then, Blythe implemented a complete solver by extending the PRODIGY4 search engine. We compared it experimentally with the incomplete system, and demonstrated that the extended algorithm is almost as efficient as PRODIGY and solves a wider range of problems [Fink and Blythe, 1998]. We now report the results of this work on completeness.

We have already shown that PRODIGY1 and PRODIGY2 do not interleave goals and sometimes fail to solve very simple problems. NOLIMIT, PRODIGY4, and FLECS use a more flexible strategy, and they fail less frequently. Veloso and Stone [1995] proved the completeness of FLECS using simplifying assumptions, but their assumptions hold only for a limited class of domains.

The incompleteness of PRODIGY is not a major handicap. Since the search space of most problems is very large, a complete exploration of the space is not feasible, which makes any problem solver "practically" incomplete. If incompleteness comes up only in a fraction of problems, it is a fair payment for efficiency.

If we achieve completeness *without compromising efficiency*, we get two bonuses. First, we ensure that the system solves every problem whose search space is sufficiently small for complete exploration. Second, incompleteness may occasionally rule out a simple solution to a large-scale problem, causing an extensive search instead of an easy win. If a solver is complete, it does not rule out any solutions and is able to find such a simple solution early in the search.

The incompleteness of means-ends analysis in PRODIGY comes from two sources. First, the problem solver does not add operators for achieving preconditions that are true in the current state. Intuitively, it ignores potential troubles until they actually arise. Sometimes, it is too late and the solver fails because it did not take measures earlier. Second, PRODIGY ignores the conditions of if-effects that do not achieve any subgoal. Sometimes, such effects negate goals or preconditions of other operators, which may cause a failure.

We achieve completeness by adding crucial new branches to the search space. The main challenge is to minimize the number of new branches, in order to preserve efficiency. We describe a method for identifying the crucial branches, based on the use of the information learned in failed old branches, and give an extended search algorithm (Sections 2.5.1 and 2.5.2). We believe that this method will prove useful for developing complete versions of other search algorithms.

The extended domain language of PRODIGY has two features that aggravate the completeness problem (Section 2.5.3), which are not addressed in the extended algorithm. First, eager inference rules may mislead the goal-directed reasoner and cause a failure. Second, functional types allow the reduction of *every* computational task to a PRODIGY problem, and some problems are undecidable.

We prove that the extended algorithm is complete for domains that have no eager inference rules and functional types (Section 2.5.4). Then, in Section 2.5.5, we give experimental results on the relative performance of PRODIGY4 and the extended solver. We conclude with the summary and discussion of the main results (Section 2.5.6).

2.5.1 Limitation of PRODIGY means-ends analysis

GPS, PRODIGY1, and PRODIGY2 were not complete because they did not explore all branches in their search space. The incompleteness of later algorithms has a deeper reason: they do not try to achieve tail preconditions that hold in the current state.

For example, suppose that the truck is in town-1, pack-1 is in ville-1, and the goal is to get pack-1 to town-1. The only operator that achieves the goal is **unload**(pack-1,town-1), so PRODIGY begins by adding it to the tail (see Figure 2.26a). The precondition (truck-at town-1) of **unload** is true in the initial state. The problem solver may achieve the other precondition, (in-truck pack-1), by adding **load**(pack-1,ville-1). The precondition (at pack-1 ville-1) of **load** is true in the initial state, and the other precondition is achieved by **leave-town**(town-1,ville-1), as shown in Figure 2.26(a).

Now all preconditions are satisfied, and the solver's only choice is to apply **leave-town** (Figure 2.26b). The application leads straight into an inescapable trap, where the truck is stranded in ville-1 without a supply of extra fuel. The algorithm may backtrack and consider different instantiations of **load**, but they will eventually lead to the same trap.

To avoid such traps, a solver must sometimes add operators for achieving literals that are true in the current state and have not been linked with any tail operators. Such literals are called *anycase subgoals*. The challenge is to identify anycase subgoals among the preconditions of tail operators.

A simple method is to view all preconditions as anycase subgoals. Veloso and Stone [1995] considered this approach in building a complete version of their FLECS search algorithm; however, it proved to cause an explosion in the number of subgoals, leading to gross inefficiency.

Kambhampati and Srivastava [1996b] used a similar approach to ensure the completeness of the Universal Classical Planner. Their system may add operators for achieving precondi-

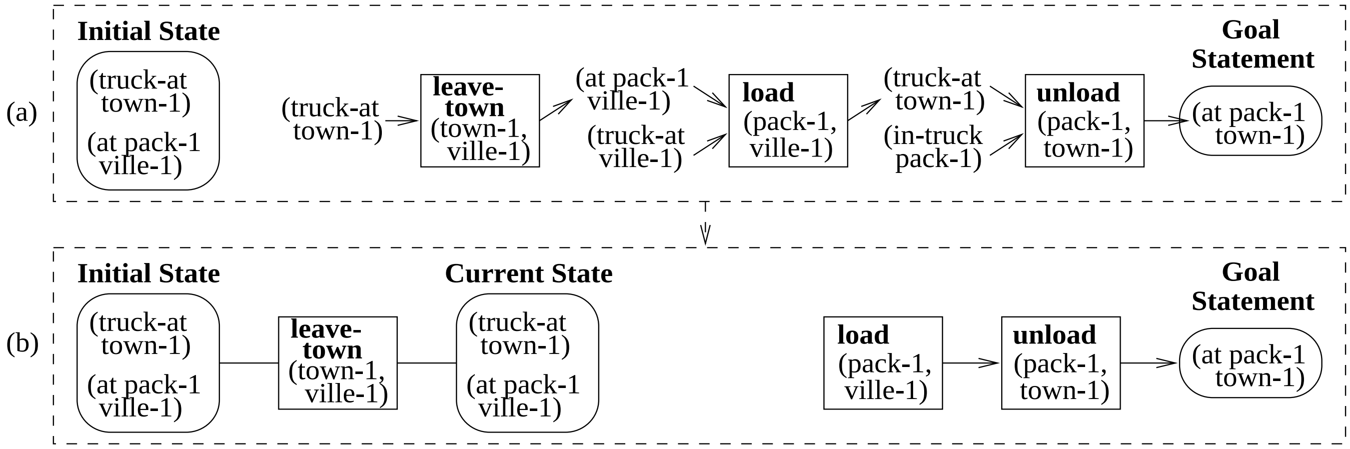


Figure 2.26: Incompleteness of means-ends analysis in the PRODIGY system: The solver does *not* consider fueling the truck before the application of **leave-town**(town-1,ville-1). Since the truck cannot leave ville-1 without extra fuel, PRODIGY fails to find a solution.

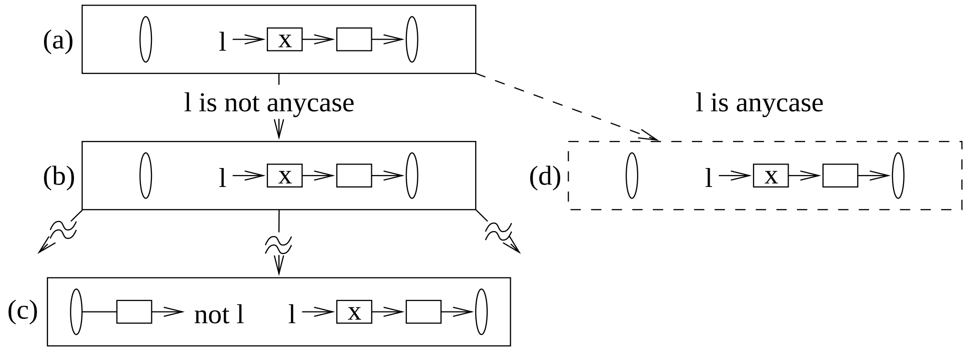


Figure 2.27: Identifying an anycase subgoal: When PRODIGY adds a new operator x (a), the preconditions of x are *not* anycase subgoals (b). If some application negates a precondition l of x (c), the solver marks l as anycase and expands the corresponding new branch of the search space (d).

tions that are true in the current state, if the preconditions are not explicitly linked to the corresponding literals of the state. Even though this approach is more efficient than viewing all preconditions as anycase subgoals, it considerably increases branching and often makes search impractically slow.

A more effective solution is based on the *use of information learned in failed branches of the search space.* Let us look again at Figure 2.26. The problem solver fails because it does not add any operator to achieve the precondition (truck-at town-1) of **unload**, which is true in the initial state. The solver tries to achieve this precondition only when the application of **leave-town** has negated it; however, after the application, the precondition can no longer be achieved.

We see that means-ends analysis may fail when some precondition is true in the current state, but is later negated by an operator application. We use this observation to identify anycase subgoals: *a precondition or a goal literal is an anycase subgoal if, at some point of the search, an application negates it.*

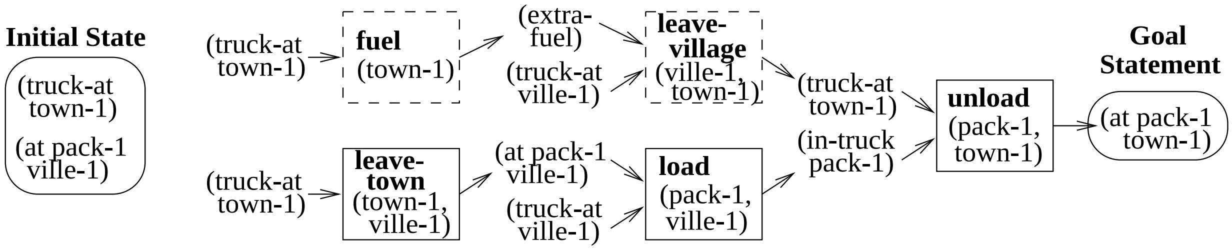


Figure 2.28: Achieving the subgoal (track-at town-1), which is satisfied in the current state. First, the solver constructs a three-operator tail, shown by solid rectangles. Then, it applies the **leave-town** operator and marks the precondition (truck-at town-1) of **unload** as an anycase subgoal. Finally, PRODIGY backtracks and adds the two dashed operators, which achieve this subgoal.

In Figure 2.27, we illustrate a technique for identifying anycase subgoals in Figure 2.27. Suppose that the problem solver adds an operator x, with a precondition l, to the tail (node a in the picture). The solver creates the branch where l is not an anycase subgoal (node b). If, at some descendent, an application of some operator negates l and if it was true before the application, then l is marked as anycase (node c). If the solver fails to find a solution in this branch, it eventually backtracks to node a. If l is marked as anycase, the problem solver creates a new branch, where l is an anycase subgoal (node d).

If several preconditions of x are marked as anycase, the solver creates the branch where they all are anycase subgoals. Note that, during the exploration of this new branch, the algorithm may mark some other preconditions of x as anycase. If it again backtracks to node a, then it creates a branch where the newly marked preconditions are also anycase subgoals.

Let us see how this mechanism works for the example problem. The solver first assumes that the preconditions of **unload**(pack-1,town-1) are not anycase subgoals. It builds the tail shown in Figure 2.26 and applies **leave-town**, negating the precondition (truck-at town-1) of **unload**. The solver then marks this precondition as anycase.

Eventually, the algorithm backtracks, creates the branch where (truck-at town-1) is an anycase subgoal, and uses the operator **leave-village**(ville-1,town-1) to achieve this subgoal (see Figure 2.28). The problem solver then constructs the tail shown in Figure 2.28, which leads to the solution "**fuel**(town-1), **leave-town**(town-1,ville-1), **load**(pack-1,ville-1), **leave-village**(ville-1,town-1), **unload**(pack-1,town-1)" (note that the precondition (truck-at ville-1) of **leave-village** is satisfied after applying **leave-town**).

When the solver identifies the set of all satisfied links (Section 2.4.1), it does not includes anycase links into this set; hence, it never ignores the tail operators that support anycase links. For example, consider the tail in Figure 2.28: the anycase precondition (truck-at town-1) of **unload** holds in the state, but the solver does *not* ignore the operators that support it.

We also have to modify the detection of goal loops, described in Section 2.4.1. For instance, consider again Figure 2.28: the precondition (truck-at town-1) of **fuel** makes a loop with the identical precondition of **unload**; however, the solver should *not* backtrack.

Since this precondition of **unload** is an anycase subgoal, it must not cause goal-loop backtracking. We use Figure 2.21 to generalize this rule: if the precondition l of x is an anycase subgoal, then the identical precondition of z does *not* make a goal loop.

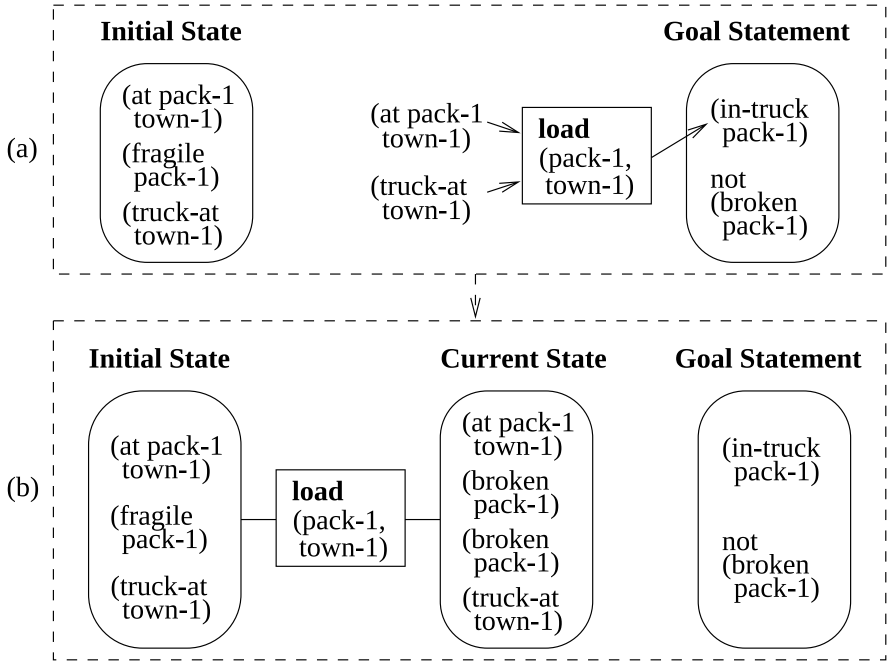


Figure 2.29: Failure because of a clobber effect: The application of the **load** operator results in the breakage of the package, and no further actions can undo this damage.

2.5.2 Clobbers among if-effects

In Figure 2.29, we illustrate another source of incompleteness: the use of if-effects. The goal is to load fragile pack-1, without breaking it. The problem solver adds **load**(pack-1,town-1) to achieve (in-truck pack-1). The preconditions of **load** and the goal "not (broken pack-1)" hold in the current state (Figure 2.29a), and the solver's only choice is to apply **load**. The application causes the breakage of pack-1 (Figure 2.29b), and no further search improves the situation. The solver may try other instances of **load**, but they also break the package.

The problem arises because an effect of **load** negates the goal "not (broken pack-1);" we call it a *clobber effect.* The application reveals the clobber, and the solver backtracks and tries to find another instance of **load**, or another operator, which does not cause clobbering. If the clobber effect has no conditions, backtracking is the only way to remedy the situation.

If the clobber is an if-effect, we can try another alternative: negate its conditions [Pednault, 1988a; Pednault, 1988b]. It may or may not be a good choice; perhaps, it is better to apply the clobber and then re-achieve the negated subgoal. For example, if we had a means for repairing a broken package, we could use it instead of cushioning. We thus need to add a new decision point, where the algorithm determines whether it should negate a clobber's conditions.

Introducing this new decision point for every if-effect will ensure completeness, but may considerably increase branching. We avoid this problem by identifying potential clobbers among if-effects. We detect them in the same way as anycase subgoals. *An effect is marked as a potential clobber if it actually deletes some subgoal in one of the failed branches.* The deleted subgoal may be a literal of the goal statement, an operator precondition, or a condition of

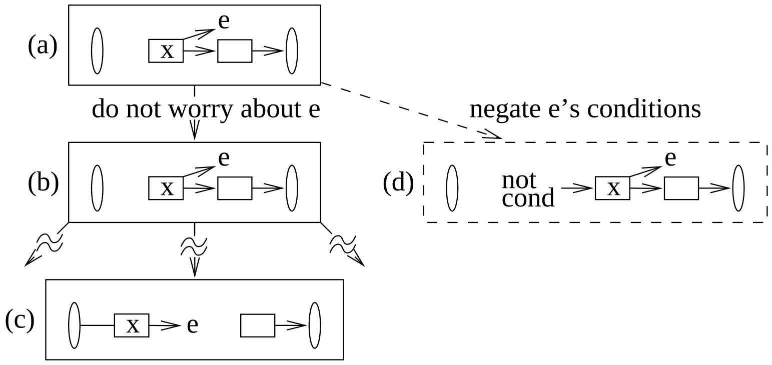


Figure 2.30: Identifying a clobber effect. When the solver adds a new operator x (a), it does *not* try to negate the conditions of x's if-effects (b). When applying x, PRODIGY checks whether if-effects negate any subgoals that were true before the application (c). If an if-effect e of x negates some subgoal, the solver marks e as a clobber and adds the respective branch to the search space (d).

an if-effect that achieves another subgoal. Thus, we again use information learned in failed branches to guide the search.

We illustrate the mechanism for identifying clobbers in Figure 2.30. Suppose that the problem solver adds an operator x with an if-effect e to the tail, and that this operator is added for the sake of some other of its effects (node a in Figure 2.30); that is, e is not linked to a subgoal. Initially, the problem solver does not try to negate e's conditions (node b). If, at some descendent, x is applied and its effect e negates a subgoal that was true before the application, then the solver marks e as a potential clobber (node c). If the solver fails to find a solution in this branch, it backtracks to node a. If e is now marked as a clobber, the solver adds the negation of e's conditions, *cond*, to the operator's preconditions (node d). If an operator has several if-effects, the solver uses a separate decision point for each of them.

In the example of Figure 2.29, the application of **load**(pack-1,town-1) negates the goal "not (broken pack-1)" and the problem solver marks the if-effect of **load** as a potential clobber (see Figure 2.31). Upon backtracking, the solver adds the *negation* of the clobber's condition (fragile pack-1) to the preconditions of **load**. The solver uses **cushion** to achieve this new precondition and generates the solution "**cushion**(pack-1), **load**(pack-1,town-1)."

We have implemented the extended search algorithm, called RASPUTIN[1], which achieves anycase subgoals and negates the conditions of clobbers. Its main difference from PRODIGY4 is the backward-chaining procedure, summarized in Figure 2.32. We show its main decision points in Figure 2.33, where thick lines mark the points absent in PRODIGY.

2.5.3 Other violations of completeness

The PRODIGY system has two other sources of incompleteness, which arise from the advanced features of the domain language. We have not addressed them in our work; hence, the use of these language features may violate the completeness of the extended algorithm.

[1]The Russian mystic Grigori Rasputin used the biblical parable of the Prodigal Son to justify his debauchery. He tried to make the story of the Prodigal Son as complete as possible, which is similar to our goal. Furthermore, his name comes from the Russian word *rasputie*, which means *decision point*.

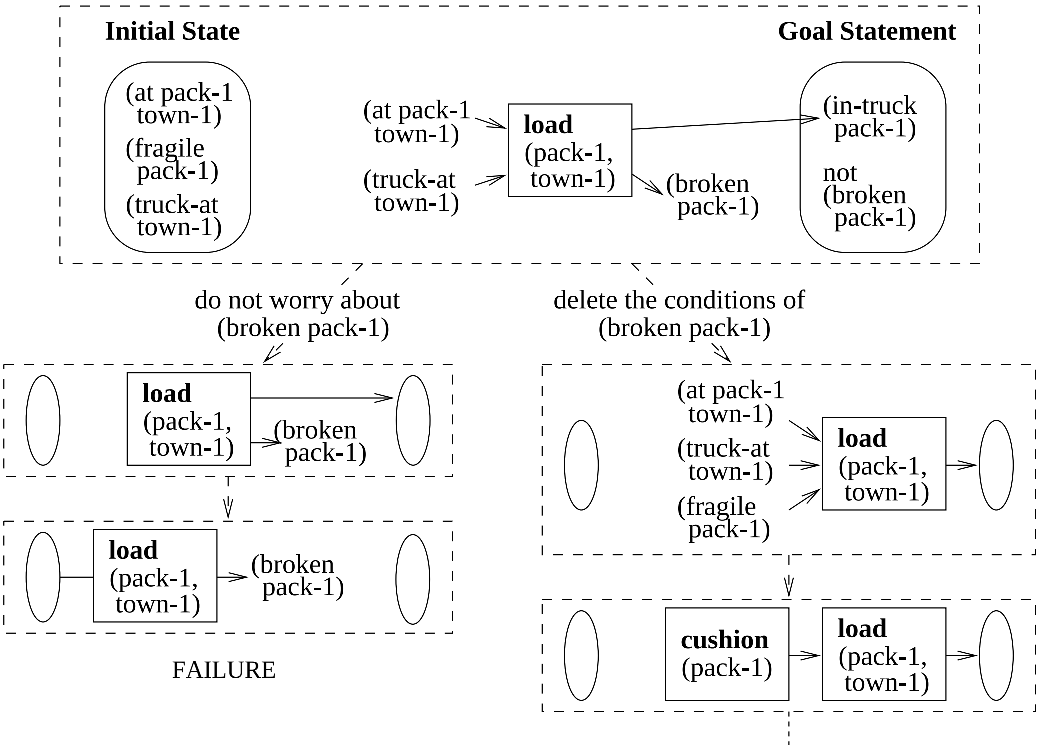


Figure 2.31: Negating a clobber. When **load** is applied, its if-effect clobbers one of the goal literals. The solver backtracks and adds the **cushion** operator, which negates the conditions of this if-effect.

Eager inference rules

If a domain includes eager rules, the system may fail to solve simple problems. For example, consider the rules in Figure 2.14 and the problem in Figure 2.34(a), and suppose that **add-truck-in** is an eager rule. The truck is initially in town-1, within county-1, and the goal is to leave this county.

Since the preconditions of **add-truck-in** hold in the initial state, the system applies it at once and adds (truck-in county-1), as shown in Figure 2.34(b). If we applied the operator **leave-town**(town-1,town-2), it would negate the rule's preconditions, which would cause the deletion of (truck-in county-1). In other words, the application of **leave-town** would immediately solve the problem.

The system does not find this solution because it inserts new operators only when they achieve some subgoal, whereas the effects of **leave-town** do *not* match the goal statement. Since the domain has no operators with a matching effect, the problem solver terminates with failure.

To summarize, the solver sometimes has to negate the preconditions of eager rules that have clobber effects, but it does not consider this option. We plan to implement the negation of clobber rules as in the future.

RASPUTIN-**Back-Chainer**
1c. Pick a literal l among the current subgoals.
    Decision point: Choose one of the subgoal literals.
2c. Pick an operator or inference rule *step* that achieves l.
    Decision point: Choose one of such operators and rules.
3c. Add *step* to the tail and establish a link from *op* to l.
4c. Instantiate the free variables of *step*.
    Decision point: Choose an instantiation.
5c. If the effect that achieves l has conditions,
        then add them to *step*'s preconditions.
6c. Use data from the failed descendants to identify anycase preconditions of *step*.
    Decision point: Choose anycase subgoals among the preconditions.
7c. If *step* has if-effects not linked to l, then:
        use data from the failed branches to identify clobber effects;
        add the negations of their conditions to the preconditions.
    Decision point(s): For every clobber, decide whether to negate its conditions.

Figure 2.32: Backward-chaining procedure of the RASPUTIN problem solver; it includes new decision points (lines 6c and 7c), which ensure completeness of PRODIGY means-ends analysis.

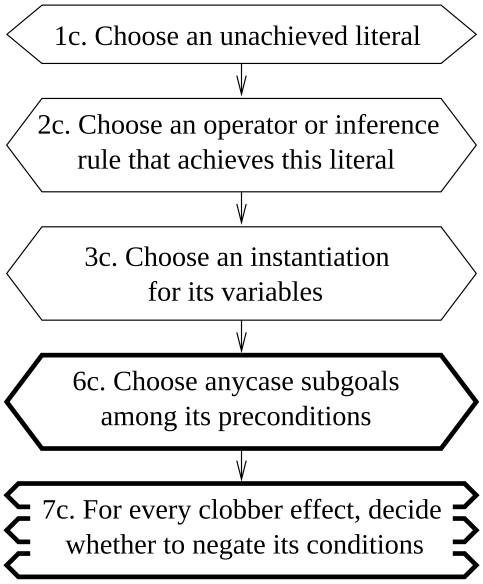
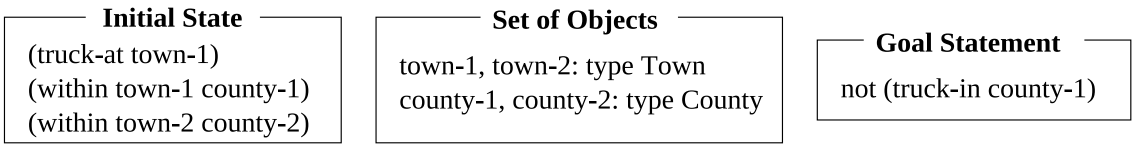
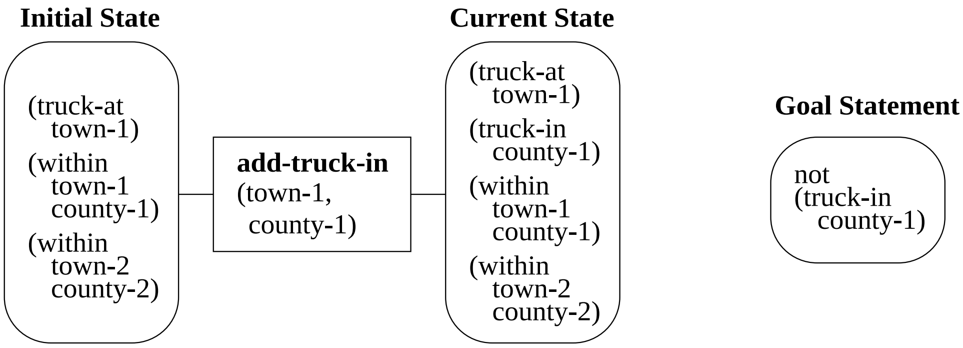


Figure 2.33: Main decision points in RASPUTIN's backward-search procedure, summarized in Figure 2.32; thick lines show the decision points that differentiate it from PRODIGY (see Figure 2.8).



(a) Example problem: the truck has to leave county-1.



(b) Application of an inference rule.

Figure 2.34: Failure because of an eager inference rule: The PRODIGY solver does *not* attempt to negate the preconditions of the rule **add-truck-in**(town-1,county-1), which clobbers the goal.

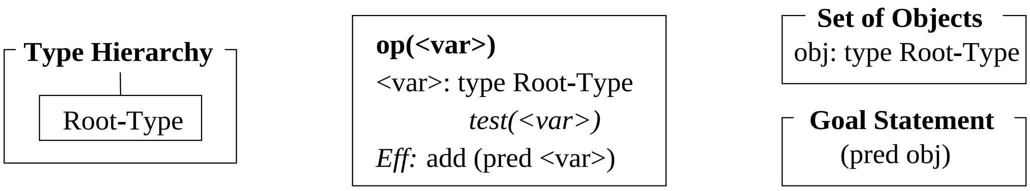


Figure 2.35: Reducing an arbitrary decision task to a PRODIGY problem; the test function is a Lisp procedure that encodes the decision task. Since functional types allow encoding of undecidable problems, they cause incompleteness of PRODIGY search.

Functional types

We next show that functional types enable the user to encode every computational decision task as a PRODIGY problem. Consider the artificial domain and problem in Figure 2.35. If the object obj satisfies the test function, then the solver uses the operator **op**(obj) to achieve the goal; otherwise, the problem has no solution.

The test function may be any Lisp program, which has full access to all data in the PRODIGY architecture. This flexibility allows the user to encode any decision problem, including undecidable and semi-decidable tasks, such as the halting problem.

Thus, we may use functional types to specify undecidable PRODIGY problems, and the corresponding completeness issues are beyond the scope of classical search. A related open problem is defining restricted classes of useful functions, which do not cause computational difficulties, and ensuring completeness for these functions.

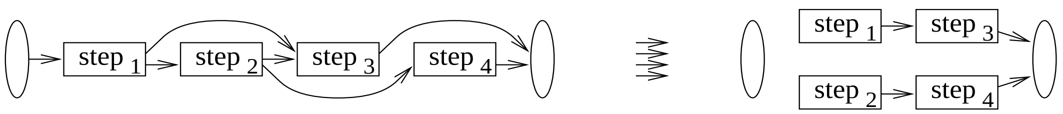


Figure 2.36: Converting a shortest complete solution into a tail: We link every subgoal to the last operator or inference rule that achieves it, and use these links to construct a tree-structured tail.

2.5.4 Completeness proof

If a domain has no eager inference rules, then the extended problem solver is complete. To prove it, we show that, *for every solvable problem, some sequence of choices in the solver's decision points leads to a solution.*

Suppose that a problem has a (fully instantiated) solution "$step_1, step_2, step_3, ..., step_n$," where every step is either an operator or a lazy inference rule, and that no other solution has fewer steps. We begin by defining clobber effects, subgoals, and justified effects in the complete solution.

A *clobber* is an if-effect such that (1) its conditions do not hold in the solution *and* (2) if we applied its actions anyways, they would make the solution incorrect.

A *subgoal* is a goal literal or precondition such that either (1) it does not hold in the initial state *or* (2) it is negated by some prior operator or inference rule. For example, a precondition of $step_3$ is a subgoal if either it does not hold in the initial state or it is negated by $step_1$. Every subgoal in a correct solution is achieved by some operator or inference rule; for example, if $step_1$ negates a precondition of $step_3$, than $step_2$ must achieve it.

A *justified effect* is the last effect that achieves a subgoal or negates a clobber's conditions. For example, if $step_1$, $step_2$, and $step_3$ all achieve some subgoal precondition of $step_4$, then the corresponding effect of $step_3$ is justified, since it is the last among the three.

If a condition literal in a justified if-effect does not hold in the initial state, or if it is negated by some prior step, we consider it a subgoal. Note that the definition of such a subgoal is recursive: we define it through a justified effect, and a justified effect is defined in terms of a subgoal in some step that comes *after* it.

Since we consider a shortest solution, each step has at least one justified effect. If we link each subgoal and each clobber's negation to the corresponding justified effect, we may use the resulting links to convert the solution into a tree-structured tail, as illustrated in Figure 2.36. If a step is linked to several subgoals, we use only one of the links in the tail.

We now show that the extended algorithm can construct this tail. If no subgoal holds in the initial state and the solution has no clobber effects, then the tail construction is straightforward. The nondeterministic algorithm creates the desired tail by always calling *Backward-Chainer* rather than applying operators, choosing subgoals that correspond to the links of the desired solution (see Line 1 in Figure 2.32), selecting the appropriate operators and inference rules (Line 2), and generating the right instantiations (Line 4).

If some subgoal literal holds in the initial state, the problem solver first builds a tail that has no operator linked to this subgoal. Then, the application of some step negates the literal, and the solver marks it as an anycase subgoal. The algorithm can then backtrack to the point *before the first application* and choose the right operator or inference rule for achieving the subgoal. Similarly, if the solution has a clobber effect, the algorithm can detect it by applying operators and inference rules. The problem solver can then backtrack to the point before the applications and add the right step for negating the clobber's conditions. Note that, even if the solver always makes the right choice, it may have to backtrack for every subgoal that holds in the initial state and also for every clobber effect.

Eventually, the algorithm constructs the desired tail and no head. It may then produce the complete solution by always deciding to apply, rather than adding new tail operators, and selecting applicable steps in the right order.

2.5.5 Performance of the extended solver

We tested the RASPUTIN solver in three domains and compared its performance with that of PRODIGY4. We present data on the relative efficiency of the two problem solvers and demonstrate that RASPUTIN solves more problems than PRODIGY4.

We first give experimental results for the PRODIGY Logistics Domain [Veloso, 1994]. The task in this domain is to construct plans for transporting packages by vans and airplanes. The domain consists of several cities, each of which has an airport and postal offices. We use airplanes for carrying packages between airports, and vans for delivery from and to post offices within cities. This domain has no if-effects and does not give rise to situations that require achieving anycase subgoals; thus, PRODIGY4 performs better than the complete algorithm.

We ran both problem solvers on fifty problems of various complexities. These problems differed in the number of cities, vans, airplanes, and packages. We randomly generated initial locations of packages, vans, and airplanes, and destinations of packages. The results are summarized in Figure 2.37(a), where each plus (+) denotes a problem instance. The horizontal axis shows PRODIGY's running time and the vertical axis gives RASPUTIN's time on the same problems. Since PRODIGY wins on all problems, all pluses are above the diagonal. The ratio of RASPUTIN's to PRODIGY's time varies from 1.20 to 1.97; its mean is 1.45.

We ran similar tests in the PRODIGY Process-Planning Domain [Gil, 1991], which also does not require negating effects' conditions or achieving anycase subgoals. The task in this domain is to construct plans for making mechanical parts with specified properties, using available machining equipment. The ratio of RASPUTIN's to PRODIGY's time in this domain is between 1.22 and 1.89, with the mean at 1.39.

We next show results in an extended version of our Trucking Domain. We now use multiple trucks and connect towns and villages by roads. A truck can go from one place to another only if there is a road between them. We experimented with different numbers of towns, villages, trucks, and packages. We randomly generated road connections, initial locations of trucks and packages, and destinations of packages.

In Figure 2.37(b), we summarize the performance of PRODIGY and RASPUTIN on fifty problems. The twenty-two problems denoted by pluses (+) do not require the clobber negation or anycase subgoals. PRODIGY outperforms RASPUTIN on these problems, with a mean ratio of 1.27.

The fourteen problems denoted by asterisks (∗) require the use of anycase subgoals or the negation of clobbers' conditions for finding an efficient solution, but can be solved inefficiently without it. RASPUTIN wins on twelve of these problems and loses on two. The ratio of

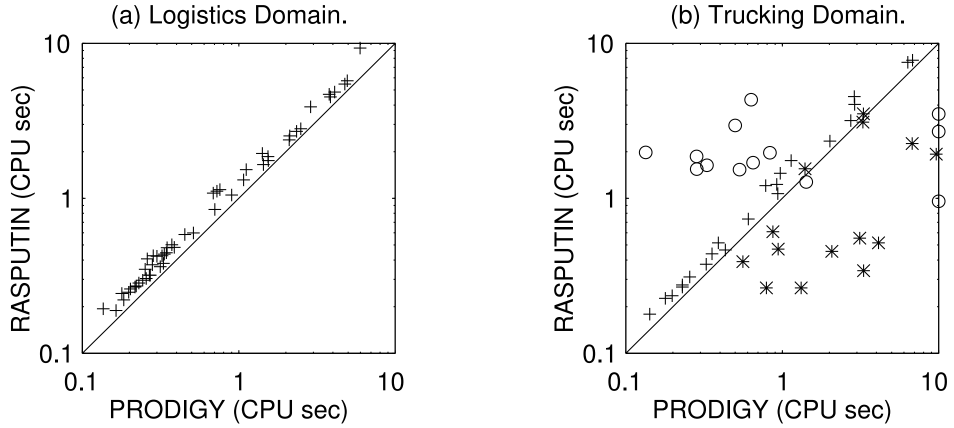


Figure 2.37: Comparison of RASPUTIN and PRODIGY, in the (a) Logistics world and (b) Trucking Domain. The horizontal axes give the search times of the PRODIGY solver, whereas the vertical axes show the efficiency of RASPUTIN on the same problems. Pluses (+) and asterisks (∗) mark the problems solved by both algorithms, whereas circles (o) are the ones solved only by RASPUTIN.

PRODIGY's to RASPUTIN's time varies from 0.90 to 9.71, with the mean at 3.69. This ratio depends on the number of required anycase subgoals: it grows with the number of such subgoals.

Finally, the circles (o) show the sixteen problems that cannot be solved without anycase subgoals and the negation of clobbers. PRODIGY hits the 10-second time limit on some of these problems and terminates with failure on the others, whereas RASPUTIN solves all of them.

2.5.6 Summary of completeness results

We have extended PRODIGY search, in collaboration with Blythe. The new algorithm is complete for a subset of the PRODIGY domain language. The full language includes two features that may violate completeness: eager inference rules and functional types. To our knowledge, the extended solver is the first complete algorithm among general-purpose problem-solving systems that use means-ends analysis. It is about 1.5 times slower than PRODIGY4 on the problems that do not require negating clobbers' conditions and achieving anycase subgoals; however, it solves problems that PRODIGY cannot solve.

We developed the extended solver in three steps. First, we identified the specific reasons for incompleteness of previous systems. Second, we added new decision points to eliminate these reasons, without significantly increasing the search space. Third, we implemented a search algorithm that explores the branches of the old search space first, and extends the search space only after failing to find a solution in the old space's branches. We conjecture that this three-step approach may prove useful for enhancing other incomplete algorithms.

The extended solver uses information from failed branches in its decisions, which means that it must perform depth-first search. We cannot use breadth-first or best-first search; however, breadth-first search in PRODIGY is impractically slow anyways, due to a large branching factor.

Part II

Description changers

Chapter 3

Primary effects

The use of primary effects in goal-directed search is an effective approach to reducing the problem-solving time. The underlying idea is to identify important effects of every operator and apply operators only for achieving their important effects. Intuitively, an effect is *not* important if it may be achieved by some other, cheaper operator.

For example, the main effect of lighting a fireplace is to heat the house; we call it a *primary effect*. If we have lamps in the house, then illumination is not an important result of using the fireplace. We may view it as a *side effect*, which means that we would not light the fireplace just for illuminating the room.

If a problem solver considers only the operators whose primary effects match the current subgoal, and ignores operators with matching side effects, then it explores fewer branches of the search space, which may result in a significant efficiency improvement. Researchers have long recognized the advantages of this technique and incorporated it in a number of AI systems.

For example, Fikes and Nilsson [1971; 1993] used primary effects to improve the quality of solutions generated by the STRIPS planner. Wilkins [1984] distinguished between main and side effects in SIPE and utilized this distinction in simplifying conflict resolution. Yang *et al.* [1996] provided a mechanism for specifying primary effects in their ABTWEAK system.

Researchers have also used primary effects to improve the effectiveness of abstraction problem solving. In particular, Yang and Tenenberg [1990] employed a combination of abstraction and primary effects to improve the efficiency of the ABTWEAK search engine. Knoblock [1994] utilized primary effects in the automatic generation of abstraction hierarchies. We investigated the relationship between primary effects and abstraction, in collaboration with Yang, and developed an algorithm for automatically abstracting effects of operators [Fink and Yang, 1992a].

Despite the importance of primary effects, this notion long remained at an informal level and researchers did not explore the properties of search with primary effects. In particular, they did not characterize appropriate selections of primary effects. The human user remained responsible for identifying important effects, which required experience and familiarity with the specific domain. An unexperienced user could provide an inappropriate selection of primary effects, thus causing incompleteness or inefficiency.

We have studied the use of primary effects, in collaboration with Yang, which has led to a formalization of their role in problem solving, and development of algorithms for the automatic selection of appropriate effects [Fink and Yang, 1997]. This work has revealed that primary effects may exponentially improve the efficiency, but choosing them appropriately is often a difficult task, whereas an improper selection can cause three major problems.

First, it may compromise completeness, that is, cause a failure on a solvable problem. For example, if the fireplace is the only source of light, but illumination is not its primary effect, then we cannot solve the problem of illuminating the room. Second, primary effects may cause generating unnecessarily costly solutions. For example, electric lamps may prove more expensive than firewood. Third, the use of primary effects may increase the search depth, which sometimes results in an exponential increase of the search time, despite the reduction of the branching factor.

The results of the work with Yang has been twofold. First, we have formalized the reasons for incompleteness and generating costly solutions, and derived a condition for avoiding these problems. Second, we have applied this result to design an inductive learning algorithm that automatically selects primary effects. The formal analysis shows that the resulting selection exponentially reduces search and ensures a high probability of completeness. We have used the ABTWEAK system to test the developed techniques, and confirmed analytical predictions with empirical results.

When developing the $\mathcal{S}$HAPER system, we implemented search with primary effects in PRODIGY4 and adapted the ABTWEAK learning techniques for the PRODIGY architecture. The main difference from the work with Yang is the use of primary effects with depth-first search, as opposed to ABTWEAK's breadth-first strategy. We have also extended the learner to the richer domain language of the PRODIGY system.

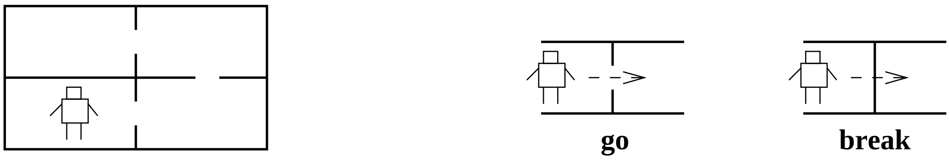
We report the results of the joint work with Yang on primary effects in ABTWEAK, and their extension for the PRODIGY architecture. First, we explain the use of primary effects in goal-directed reasoning (Section 3.1), derive the condition for preserving completeness (Section 3.2), and analyze the resulting search reduction (Section 3.3).

Then, we describe the automatic selection of primary effects, using heuristics and a learning algorithm. A fast heuristic procedure produces an initial selection of primary effects (Section 3.4), and then the inductive learner revises the initial selection, to ensure that it does not compromise completeness and allows the system to find near-optimal solutions (Section 3.5).

Finally, we describe experiments on search with the resulting primary effects, which confirm the analytical predicted search reduction. The empirical results demonstrate that the appropriate choice of primary effects may lead to an exponential efficiency improvement. The first series of experiments is in the ABTWEAK system, which uses breadth-first search (Section 3.6). The second series is on the use of primary effects with the depth-first search of the PRODIGY architecture (Section 3.7).

3.1 Search with primary effects

We describe the use of primary effects in goal-directed reasoning and discuss the related trade-off between efficiency and solution quality. In the work on primary effects, we measure



(a) Example of a connected world. (b) Operators.

Figure 3.1: Simple robot world: The robot may go through doorways and break through walls. If all rooms are connected, then we view change of the robot's position as a side effect of **break**.

the quality of a solution by the total cost of its operators, which is a special case of the quality functions in the generalized analysis of representations (see Section 7.3).

We give motivating examples (Section 3.1.1), formalize the notion of primary effects (Section 3.1.2), and explain their role in backward chaining. The described techniques work for most goal-directed problem solvers, including backward-chaining systems and PRODIGY algorithms; however, they are not applicable to forward-chaining algorithms.

3.1.1 Motivating examples

We have used the fireplace example in explaining the notion of primary effects and now give two more informal examples, which illustrate the two main uses of primary effects: reducing the search time and improving the solution quality. We will later formalize these examples and use them in describing the search with primary effects.

Robot world

First, we describe a simple version of the STRIPS world [Fikes and Nilsson, 1971], which includes a robot and several rooms (see Figure 3.1a). The robot can go between two rooms connected by a door, as well as break through the wall to an adjacent room, thus creating a new doorway (Figure 3.1b).

Suppose that the robot world is connected, that is, it has no regions completely surrounded by walls, and the robot can accomplish every location change by a series of **go** operators (Figure 3.1a). Then, we can view the location change as a side effect of the **break** operator, which means that we use this operator only for the purpose of making new doorways. If the only goal is to move the robot to a certain room, then the problem solver disregards **break** and uses **go** operators. This restriction reduces the branching factor and may improve the efficiency of search (see Section 3.3).

Machine shop

Next, consider a machine shop that allows cutting, drilling, polishing, painting, and other machining operations, and suppose that a problem solver has to generate plans for producing parts of different quality. The production of higher-quality parts requires more expensive operations.

The solver may use expensive operations to make low-quality parts, which sometimes simplifies the search but leads to generating suboptimal solutions. For example, it may choose a high-precision drilling operation instead of normal drilling, which would result in a costly solution.

The use of primary effects enables the system to avoid such situations. For example, we may view making a hole as a side effect of high-quality drilling, and the precise position of the hole as its primary effect. Then, the problem solver chooses high-quality drilling only when precision is important. In Section 3.6.2, we give a formal description of this domain and present experiments on the use of primary effects to select appropriate machining operations.

3.1.2 Main definitions

We extend the robot example, encode it in the PRODIGY domain language, and use this example to illustrate the main notions related to search with primary effects.

The extended domain includes a robot, a ball, and four rooms (see Figure 3.2a). To describe its current state, we have to specify the location of the robot and the ball, and list the pairs of rooms connected by doors. We therefore use three predicates for encoding domain states, (robot-in <room>), (ball-in <room>), and (door <from> <to>) (Figure 3.2c). For example, the literal robot-in room-1 means that the robot is in room-1, ball-in room-4 means that the ball is in room-4, and door room-1 room-2 means that room-1 and room-2 are connected by a doorway.

The robot may go between two rooms connected by a door, carry the ball, throw it through a door into an adjacent room, or break through a wall. We encode these actions by four operators, given in Figure 3.2(d). In addition, the domain includes the inference rule **add-door** (Figure 3.2e), which ensures that every door provides a two-way connection, that is, (door <from> <to>) implies (door <to> <from>).

For example, consider the problem with the initial state shown in Figure 3.2(a) and the goal to get the ball into room-3. The robot can achieve it by breaking through the wall into room-4 and then throwing the ball into room-3: "**break**(room-3,room-4), **throw**(room-4,room-1)" (see Figure 3.4).

The operators in the robot domain have constant costs (Figure 3.2d), which is a special case of using cost functions (see Section 2.3.1). We measure the quality of a solution by the total cost of its operators: the smaller the cost, the better the solution. An *optimal solution* to a given problem is a complete solution that has the lowest cost.

For instance, suppose that the problem solver needs to move the robot from room-1 to room-4, and it achieves this goal using three **go** operators, "**go**(room-1,room-2), **go**(room-2,room-3), **go**(room-3)room-4," with the total cost of $2 + 2 + 2 = 6$. This solution is not optimal, since the same goal can be achieved by the operator **break**(room-1,room-4), with a cost of 4.

If an operator has several effects, we may choose certain important effects among them and use the operator only for achieving these important effects. The chosen effects of the operators are its *primary effects,* and the others are *side effects.* For example, we may view (door <from> <to>) as a primary effect of **break**(<from>,<to>), and (robot-in <to>) as its side effect. Note that we choose primary effects not only among the simple effects of an operator, but also among its conditional effects. If a conditional effect has several actions, we may divide them into primary and side actions (see the syntax of conditional effects in

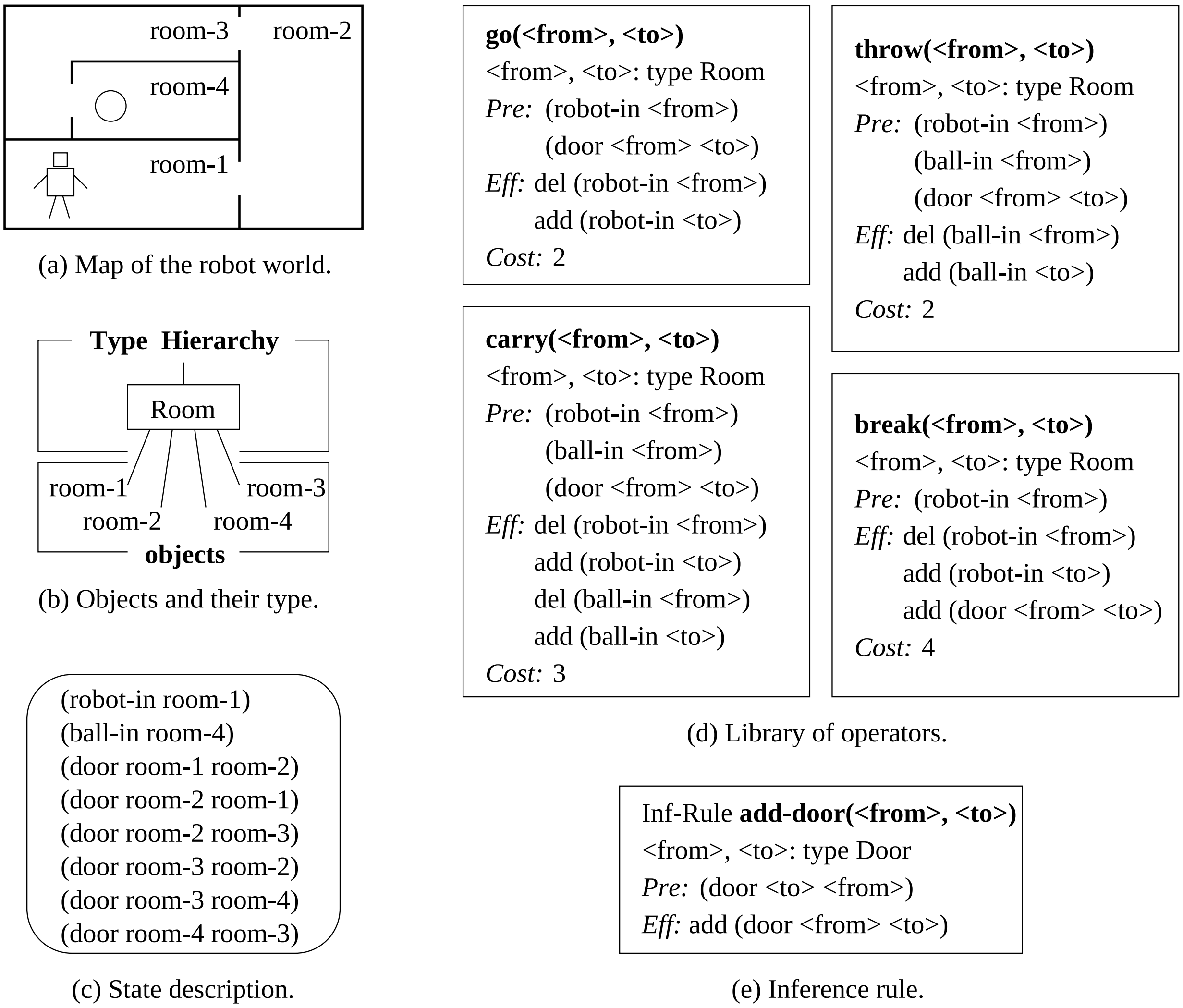


Figure 3.2: Robot Domain. We give an example of a world state (a), the encoding of this state in the PRODIGY language (b,c), and the list of operators and inference rules (d,e). The robot's actions include going between rooms, carrying and throwing a ball, and breaking through walls.

Section 2.2.1).

Inference rules may also have primary and side effects, which limits their use in backward chaining. For example, if the effect of **add-door** is a side effect (see Figure 3.2), then PRODIGY never adds it to the tail. On the other hand, primary effects do *not* affect the forced application of eager inference rules.

Note that, if an eager inference rule has no primary effects, then the *Backward-Chainer* procedure disregards it, but the system applies it in the forced forward chaining. Thus, if the human operator wants to prevent the utilization of eager rules in goal-directed reasoning, she should use only side effects in their encoding. On the other hand, if an operator or lazy rule has no primary effects, then the system completely ignores it.

We now define the notion of *primary-effect justification*, which characterizes solutions constructed with the use of primary effects. It is similar to the definition of justification

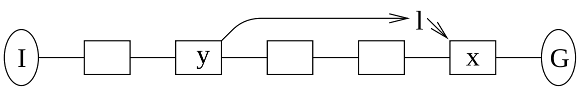
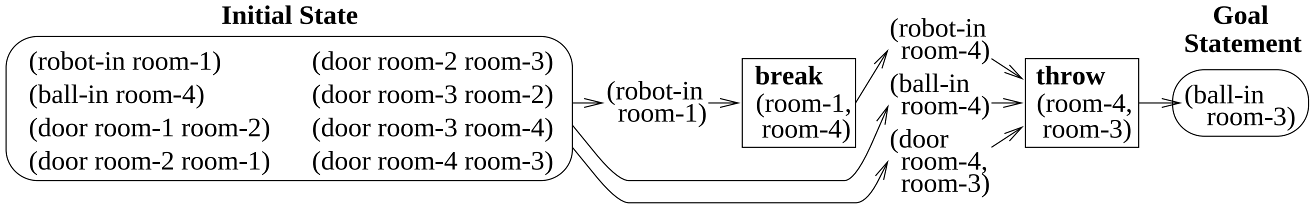
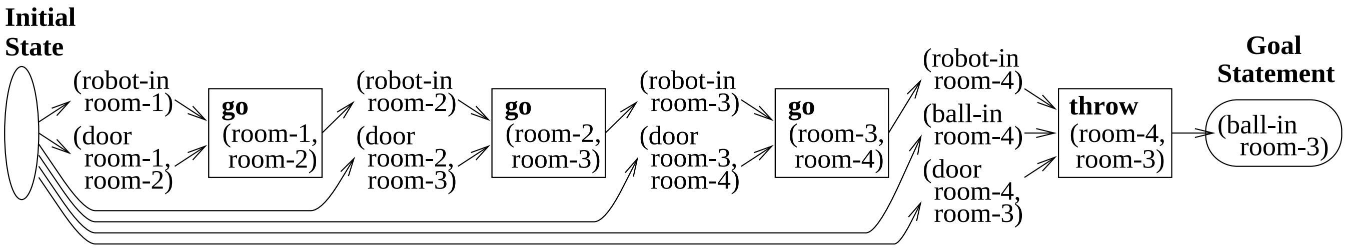


Figure 3.3: Definition of justified effects: If a literal l is a precondition of x and an effect of y, and no operator or inference rule between x and y has an identical effect, then l is a justified effect of x.



(a) Using the **break** operator to get into room-4.



(b) Using a sequence of **go** operators.

Figure 3.4: Two solutions for moving the ball into room-3 (see Figure 3.2a). To reach the ball, the robot may either break the wall between room-1 and room-4 (a), or go around through room-2 and room-3 (b). Arrows show the achievement of the goal literal and operator preconditions.

by Knoblock *et al.* [1991], in their formal model of abstraction. This notion is independent of specific problem solvers, which enables us to identify general properties of search with primary effects and derive conditions for its completeness.

We have already defined a justified effect of an operator or inference rule in a complete solution (see Section 2.5.4). Recall that an effect literal is called *justified* if it is the last effect that achieves a subgoal or negates the conditions of a clobber. Informally, it means that the literal is necessary for the correctness of the solution. We illustrate this notion in Figure 3.3, where l is a precondition of x and a justified effect of y.

Note that an effect literal may be justified even if it holds before the execution of the operator. To repeat the related example from Section 2.5.4, consider some solution $step_1, step_2, ..., step_n$ and suppose that $step_1$, $step_2$, and $step_3$ all achieve some precondition of $step_4$. Then, the corresponding effect literal of $step_3$ is justified, since it is the last among them, whereas the identical effects of $step_1$ and $step_2$ are not justified.

To illustrate justification, we again consider the task of bringing the ball to room-3 and its solution in Figure 3.4(a). The (robot-in room-4) effect of **break** and the (robot-in room-4) effect of **throw** are justified, whereas the other effects are not.

A complete solution is *primary-effect justified* if every operator and every lazy rule has a justified primary effect. Informally, it means that no operator or inference rule is used for its

operators	**primary effects**
go(<from>,<to>)	add (robot-in <to>)
carry(<from>,<to>)	add (ball-in <to>)
throw(<from>,<to>)	add (ball-in <to>)
break(<from>,<to>)	add (door <from> <to>)
add-door(<from>,<to>)	—

(a)

operators	**primary effects**
go(<from>,<to>)	del (robot-in <from>), add (robot-in <to>)
carry(<to>,<from>)	add (ball-in <to>)
throw(<from>,<to>)	del (ball-in <from>), add (ball-in <to>)
break(<from>,<to>)	add (door <from> <to>)
add-door(<from>,<to>)	—

(b)

Table 3.1: Selections of primary effects for the Robot Domain, shown in Figure 3.2. Observe that the first selection does not allow PRODIGY to achieve deletion goals, such as "not (robot-in room-1)."

side effects. We discussed properties of justified solutions in an article on improving solution quality [Fink and Yang, 1992b].

For example, suppose that the primary effects are as shown in Table 3.1(a), in particular, (robot-in <to>) is a primary effect of **go** and a side effect of **break**. Then, the solution in Figure 3.4(a) is not primary-effect justified, because it does not utilize the primary effect of the **break** operator. On the other hand, the solution with **go** operators in Figure 3.4(b) is justified.

3.1.3 Search algorithm

We now explain the role of primary effects in goal-directed reasoning and discuss the corresponding modifications to the PRODIGY search engine. When adding a new operator to the current incomplete plan, a problem solver selects it among the operators whose primary effects match the current subgoal; however, after inserting the selected operator into the solution, the solver may utilize its side effects to achieve other subgoals. If the solver uses inference rules in backward chaining, then the same restrictions apply to the choice of matching rules. This general principle underlies the ABTWEAK and PRODIGY implementations of search with primary effects.

In Figure 3.5, we give a modification of the PRODIGY backward-chainer, restricted to the use of operators with matching primary effects, which differs from the original unrestricted procedure only in Step 4c (see Figure 2.8). We do *not* modify the other two procedures of the search engine, *Base*-PRODIGY and *Operator-Application* (see Figure 2.8).

For example, suppose that the primary effects are as shown in Table 3.1(a) and consider a problem with two goals, (door room-1 room-4) and (ball-in room-3), as shown in Figure 3.6.

The unrestricted procedure would consider two alternatives for achieving the first subgoal: the **break** operator and the **add-door** rule. On the other hand, *Prim-Back-Chainer* uses **break** and disregards the other alternative, because **add-door** has no primary effects.

When the problem solver applies an operator, it uses both primary and side effects in updating the current state, and may later utilize useful side effects. For example, suppose that the solver applies the **break** operator and then adds **throw**(room-4,room-3) to achieve the other goal, (ball-in room-3) (Figure 3.6b). Since **break** achieves the precondition (robot-in room-4) of **throw**, PRODIGY can apply the **throw** operator, thus solving the problem. Thus, the system uses two effects of **break** (see Figure 3.6c), one of which is a side effect.

Since PRODIGY selects an operator for achieving one of its effects, and does not consider other effects until its application, the appropriate utilization of multiple effects is often a difficult problem, which may require advanced heuristics or control rules. This need for control knowledge is a general problem of goal-directed reasoning, unrelated to the use of primary effects.

The ABTWEAK system chooses operators whose primary effects match current subgoals, but it has a different mechanism for utilizing other effects. After inserting an operator into an incomplete solution, the system may unify its effects with matching subgoals and impose the corresponding ordering constraints. For example, it may add the **break** and **throw** operators for achieving the two subgoals, and then unify a side effect of **break** with the corresponding precondition of **throw** (see Figure 3.7). The reader may find a description of this algorithm in the articles on the ABTWEAK system [Yang *et al.*, 1996; Yang and Murray, 1994; Fink and Yang, 1997].

Observe that, if all effects of all operators and inference rules are primary, then their use is identical to the search without primary effects. This rule holds for all problem solvers that support the use of primary effects, including PRODIGY and ABTWEAK.

3.2 Completeness of primary effects

The utility of primary effects depends crucially on an appropriate selection mechanism. We discuss possible problems of search with primary effects and methods for avoiding them (Section 3.2.1). In particular, we derive a condition for ensuring completeness of search (Section 3.2.2), which underlies the algorithm for learning primary effects (see Section 3.5).

3.2.1 Completeness and solution costs

Primary effects reduce the search space and usually lead to pruning some solution branches. An inappropriate selection of primary effects may result in pruning *all* solutions, thus causing a failure on a solvable problem.

For example, consider the robot domain with the primary effects given in Table 3.1(a) and suppose that the robot has to vacate room-1, that is, the goal is "not (robot-in room-1)." The robot may achieve it by going to room-2, or by breaking into room-3 or room-4; however, if a problem solver uses only primary effects, it will not find a solution, because "del (robot-in <room>)" is not a primary effect of *any* operator. To preserve completeness, we have to select additional primary effects, as shown in Table 3.1(b).

Prim-Back-Chainer

1c. Pick a literal l among the current subgoals.
Decision point: Choose one of the subgoal literals.

2c. Pick an operator or inference rule *step*
that achieves l as a primary effect.
Decision point: Choose one of such operators.

3c. Add *step* to the tail and establish a link from *step* to l.

4c. Instantiate the free variables of *step*.
Decision point: Choose an instantiation.

5c. If the effect achieving l has conditions,
then add them to *step*'s preconditions.

Figure 3.5: Backward-chaining procedure that uses primary effects. When adding a new step to the tail, the *Prim-Back-Chainer* algorithm chooses among the operators and inference rules whose primary effects match the current subgoal (see line 2c).

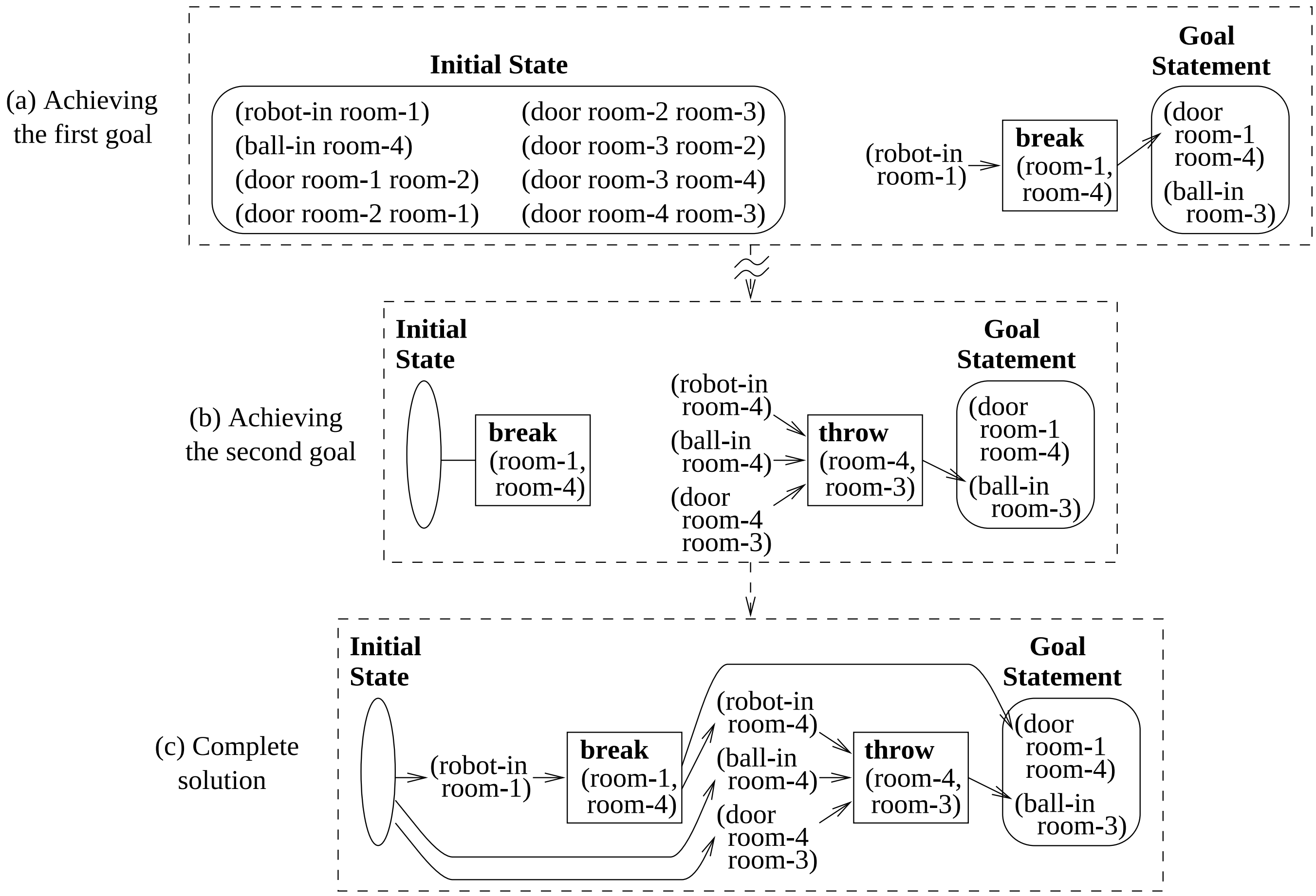


Figure 3.6: PRODIGY search with primary effects. Note that **break** is the only operator that achieves the first goal as a primary effect. Similarly, **throw** is the only match for the second goal.

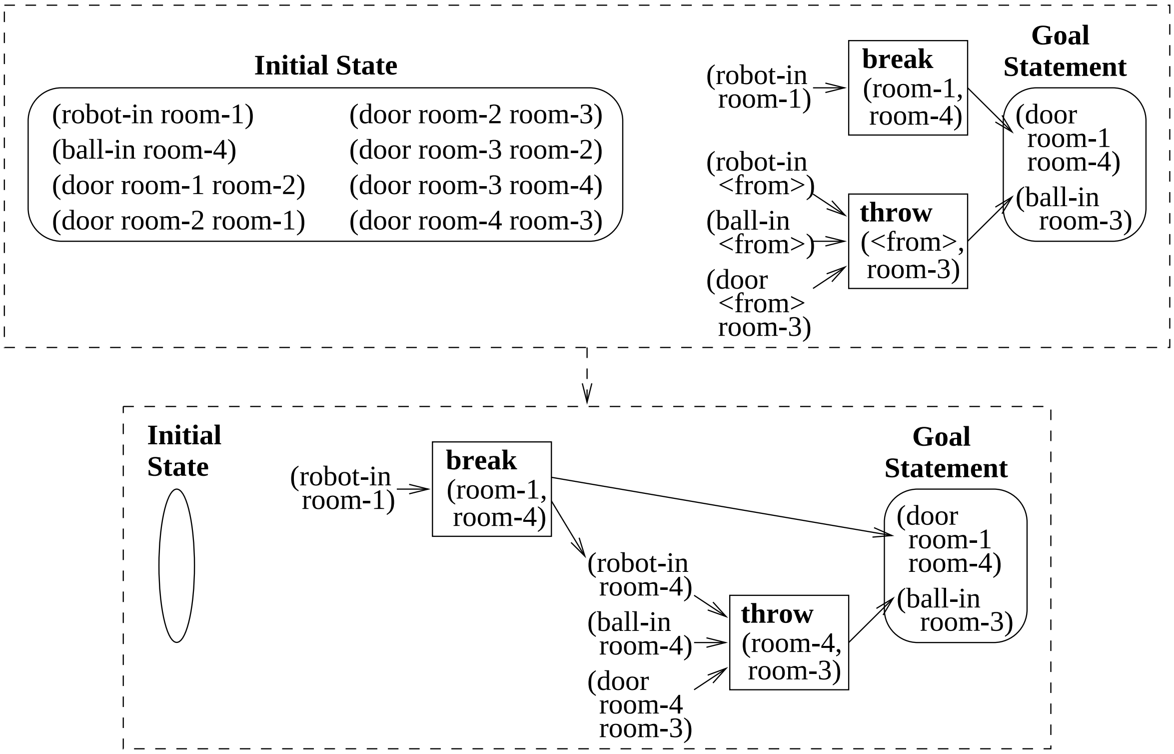


Figure 3.7: ABTWEAK search with primary effects.

We discuss two factors the determine the completeness of search with primary effects: the existence of a primary-effect justified solution and the solver's ability to find it.

Complete selections

If every solvable problem in a domain has a primary-effect justified solution, we say that the selection of primary effects is *complete.* Finding a selection with this property is essential for the overall completeness of problem solving. For example, the choice of primary effects given in Figure 3.1(a) is incomplete, which causes a failure on some problems. On the other hand, the selection in Figure 3.1(b) is complete.

Observe that the definition of a complete selection is independent of a specific planning algorithm. Also note that, if the selection comprises *all* effects of all operators and inference rules, then it is trivially complete, which implies that every domain has at least one complete selection.

Primary-effect complete search

If a problem solver uses primary effects and can solve every problem that has a primary-effect justified solution, then it is called *primary-effect complete.* The PRODIGY search engine does *not* satisfy this condition. On the other hand, the extended algorithm of Section 2.5 is primary-effect complete, if the domain does not include eager inference rules and functional types. The proof is similar to the completeness proof in Section 2.5.4.

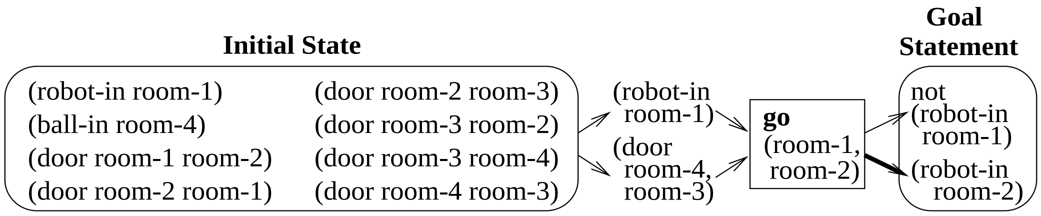


Figure 3.8: Solution that requires backtracking over the choice of a subgoal; the thick arrow marks the justified primary effect.

Note that, if *all* effects are primary, then a solver with this completeness property finds a solution for every solvable problem; hence, primary-effect completeness implies standard completeness. The reverse of this observation does not hold: a complete search algorithm may not be primary-effect complete.

For example, consider the extended solver and suppose that we disable the backtracking over the choice of a subgoal in Line 1c (see Figure 2.32). This modification preserves the standard completeness but compromises primary-effect completeness.

To illustrate the resulting incompleteness, consider the primary effects in Table 3.1(a) and the problem with two goals, "not (robot-in room-1)" and "(robot-in room-2)." This problem has a simple primary-effect justified solution, given in Figure 3.8; however, the modified algorithm may fail to solve it. If the algorithm begins by choosing the first goal, it does not find a matching primary effect and terminates with failure.

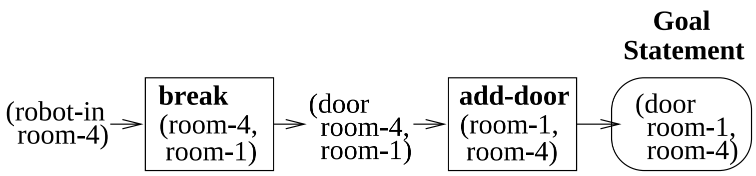
Cost increase

Even if primary effects do not compromise completeness, their use may result in pruning all optimal and near-optimal solutions from the search space, and lead to finding a poor-quality solution.

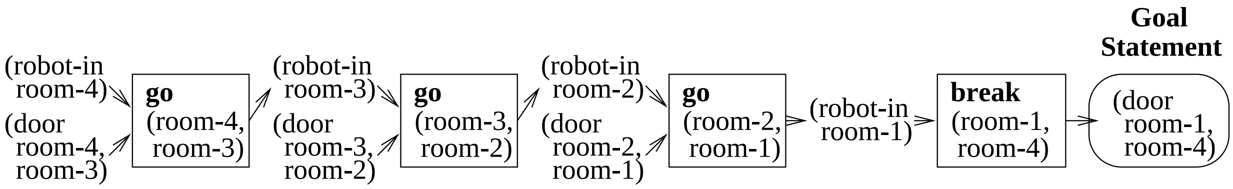
For example, suppose that the robot is initially in room-4 and the goal is (door room-1 room-4). The optimal solution to this problem utilizes the inference rule: "**break**(room-4,room-1), **add-door**(room-1,room-4)." If the problem solver uses **add-door** in backward chaining, it may construct the tail given in Figure 3.9(a), which leads to the optimal solution.

Now suppose that the solver uses the primary effects given in Table 3.1(b). Since the **add-door** rule has no primary effects, the solver chooses **break**(room-1,room-4) for achieving the goal, constructs the tail in Figure 3.9(b), and produces the solution "**go**(room-4,room-3), **go**(room-3,room-2), **go**(room-2,room-1), **break**(room-1,room-4)," which is the best primary-effect justified solution.

In this example, the minimal cost of a primary-effect justified solution is $2+2+2+4 = 10$, whereas the cost of the optimal solution is 4. The ratio of these costs, $10/4 = 2.5$, is called the *cost increase* for the given problem. Intuitively, it measures the deterioration of solution quality due to the use of primary effects.



(a) Backward chaining without primary effects.



(b) Backward chaining using primary effects.

Figure 3.9: Tails that lead to (a) the optimal solution and (b) the best primary-effect justified solution.

3.2.2 Condition for completeness

To ensure overall search completeness, we have to provide a complete selection and a primary-effect complete solver. The first of the two requirements is the main criterion for selecting appropriate primary effects. We now derive a condition for satisfying this requirement, which not only ensures the completeness of a selection, but also limits the cost increase. We will use this condition in designing an algorithm that selects primary effects.

Replacing sequence

Consider some fully instantiated operator or inference rule, which we will call *step*, an initial state I that satisfies the preconditions of *step*, and the goal of achieving all side effects of *step* and preserving all features of the initial state that are not affected by *step*. Formally, this goal is the conjunction of the side effects of *step* and the initial-state literals unaffected by *step*.

We give an algorithm for computing this conjunction in Figure 3.10. Since the resulting goal is a function of *step* and I, we denote it by $G(step, I)$. Note that the goal does not include the primary effects of *step* and does not require preserving the features of the state affected by the primary effects.

If a problem solver applies *step* to the initial state, it achieves all goal literals; however, this solution is *not* primary-effect justified. A *replacing sequence* of operators and inference rules is a primary-effect justified solution that achieves the goal $G(step, I)$ from the same initial state I; that is, it achieves all side effects of *step* and leaves unchanged the other literals of I.

For example, consider the fully instantiated operator **break**(room-1,room-4), with side effects "del (robot-in room-4)" and add (robot-in room-1)," and the state in Figure 3.2, which satisfies the operator's preconditions. For this operator and initial state, we may construct the replacing sequence "**go**(room-1,room-2), **go**(room-2,room-3), **go**(room-3,room-4)," which is

Generate-Goal($step, I$)
The input includes an instantiated operator or inference rule, *step*,
and an initial state I that satisfies the preconditions of *step*.

Create a list of literals, which is initially empty.
For every side-effect literal *side* of *step*:
  If *step* adds this literal,
    then add *side* to the list of literals.
  If *step* deletes this literal,
    then add "not *side*" to the list of literals.
For every literal *init* of the initial state I:
  If *step* does not add or delete this literal,
    then add *side* to the list of literals.
Return the conjunction of all literals in the list.

Figure 3.10: Generating the goal $G(step, I)$ of a replacing sequence. When the algorithm processes the effect literals of *step*, it considers all simple effects and the if-effects whose conditions hold in the state I.

a primary-effect justified solution that achieves both side effects of **break**, and does not add or delete other literals.

The cost of the cheapest replacing sequence is called the *replacing cost* of *step* in the initial state I. For example, the cheapest replacing sequence for **break**(room-1,room-4) consists of three **go** operators; hence, its replacing cost is 6.

Note that the replacing cost may be smaller than the cost of *step*. For example, suppose that the **carry**(<from>,<to>) operator has two primary effects, "del (ball-in <from>)" and "add (ball-in <to>)," and consider its instantiation **carry**(room-1,room-2). The one-operator replacing sequence "**go**(room-1,room-2)" has a cost of 2, which is smaller than **carry**'s cost of 3.

Completeness condition and its proof

We state a completeness condition in terms of replacing sequences:

> **Completeness:** Suppose that, for every fully instantiated operator and inference rule, and every initial state that satisfies its preconditions, there is a primary-effect justified replacing sequence. Then, the selection of primary effects is complete.
>
> **Limited cost increase:** Suppose that there is a positive real value C such that, for every instantiated operator *op* and every initial state, the replacing cost is at most $C \cdot cost(op)$. Suppose further that, for every inference rule and every state, the replacing cost is 0. Then, the cost increases of *all* problems in the domain are at most $\max(1, C)$.

The proof is based on the observation that, given a problem and its optimal solution, we may substitute all operators and inference rules with corresponding replacing sequences. We thus obtain a primary-effect justified solution, whose cost is at most C times larger than that of the optimal solution. To formalize this observation, we consider an arbitrary problem, with an optimal solution "$step_1, step_2, ..., step_n$," and construct its primary-effect justified solution.

Suppose that the completeness condition holds, that is, we can find a replacing sequence for every operator and inference rule. If $step_n$ is not primary-effect justified, we substitute it with the cheapest replacing sequence. If $step_{n-1}$ is not primary-effect justified in the resulting solution, we also substitute it with the cheapest replacing sequence. We repeat this operation for all other operators and inference rules, considering them in the reverse order, from $step_{n-2}$ to $step_1$. Observe that, when we replace some $step_i$, all steps *after* it remain primary-effect justified, which implies that the replacement of all steps in the reverse order leads to a primary-effect justified solution.

Now suppose that the condition for limited cost increase also holds. Then, for every replaced operator and inference rule $step_i$, the cost of the replacing sequence is at most $C \cdot cost(step_i)$, which implies that the total cost of the primary-effect justified solution is at most $\max(1, C) \cdot (cost(step_1) + cost(step_2) + ... + cost(step_n))$.

Use of the completeness condition

The completeness condition suggests a technique for selecting primary effects: we should consider instantiated operators and states that satisfy their preconditions, and ensure that we can always find a replacing sequence. Search through *all* instantiated operators and initial states is usually intractable; however, we can guarantee a *high probability* of completeness by considering a small random selection of operator instantiations and states. We use this probabilistic approach to design a learning algorithm that selects primary effects of operators (Section 3.5).

If the domain has no eager inference rules, and problems may have any initial states and goal statements, then the described condition is not only sufficient but also necessary for completeness. That is, if some operator or inference rule, in some initial state, has no replacing sequence, then the use of primary effects compromises the completeness.

On the other hand, if we restrict the allowed goals, then we may be able to select fewer primary effects without sacrificing completeness. For example, if the robot never has to vacate a room, then "del (robot-in <room>)" need not be a primary effect of any operator. In Section 5.3, we will discuss some methods of selecting primary effects for restricted goals.

Avoiding redundant effects

We are interested in finding a *minimal* selection of primary effects that ensures completeness and a small cost increase. A primary effect is *redundant* if we can demote it to a side effect without compromising completeness or increasing solution costs.

For example, if the operator **carry**(<from>,<to>) has two primary effects, "add (ball-in <to>)" and "del (ball-in <from>)," then the latter effect is redundant. Demoting it to a side effect of **carry** does not affect the costs of primary-effect justified solutions, because the solver may use the cheaper operator **throw** for achieving the same result.

Redundant effects increase the branching factor of search without reducing search depth or improving solution quality, and avoiding them is one of the main goals in designing an algorithm for learning primary effects.

3.3 Analysis of search reduction

We present an analytical comparison of search efficiency with and without the use of primary effects. The purpose of the analysis is to estimate the search reduction for backward-chaining solvers, and identify the factors that determine the utility of primary effects. We show that the use of primary effects reduces the branching factor, but increases the search depth, and that efficiency depends on the trade-off between these two factors.

The comparison is an approximation based on several simplifying assumptions about properties of the search space, similar to the analysis of abstraction search by Korf [1987] and Knoblock [1991]. Even though real domains usually do not satisfy these assumptions, experimental results confirm the analytical predictions (see Sections 3.6 and 3.7).

Simplifying assumptions

When a solver algorithm works on a problem, it expands a tree-structured search space, whose nodes correspond to intermediate incomplete solutions, and the solver's running time is proportional to the number of nodes in the expanded space [Minton *et al.*, 1991]. We estimate the space size for search with and without primary effects, and use these estimates to analyze the utility of primary effects.

For simplicity, we assume that the domain does not include inference rules, which means that *Prim-Back-Chainer* uses only operators in constructing the tail. When adding a new operator to achieve some subgoal, the algorithm may use any operator with a matching primary effect. To estimate the number of matching operators, we denote the number of nonstatic predicates in the domain by N and the total number of primary effects in all operators by PE, which stands for "primary effects;" that is, we count the number of primary effects, $\#(\text{Prim}(op))$, of each operator op and define PE as the sum of these numbers:

$$PE = \sum_{op} \#(\text{Prim}(op)).$$

A nonstatic predicate gives rise to two types of subgoals, *adding* and *deleting* its instantiations; hence, the average number of operators that match a subgoal is $\frac{PE}{2 \cdot N}$. If we assume that the number of matching operators is the same for all additions and deletions of nonstatic predicates, then it is exactly $\frac{PE}{2 \cdot N}$.

After selecting an operator, the problem solver has to pick its instantiation and then choose the next subgoal. In addition, the solver may apply some of the tail operators. These choices lead to expanding multiple new branches of the search space; we denote their number by BF, which stands for "branching factor." Since the full decision cycle of PRODIGY includes

1. The domain has no inference rules.
2. The number of matching operators is the same for all subgoals.
3. The overall branching factor BF of instantiating a new operator, applying tail operators, and choosing a subgoal is constant throughout the search.
4. All operators have the same cost.

Figure 3.11: Simplifying assumptions.

PE	total number of primary effects in all operators
E	total number of all effects in all operators
N	number of nonstatic predicates in the problem domain
BF	overall branching of instantiating a new operator, applying operators, and choosing the next subgoal
C	cost increase of problem solving with primary effects
n	number of operators in the solution found without primary effects

Figure 3.12: Summary of the notation in the search-reduction analysis.

choosing an operator for the current subgoal, instantiating it, applying some tail operators, and choosing the next subgoal (see Figure 2.9), it gives rise to $BF \cdot \frac{PE}{2 \cdot N}$ new branches.

To estimate the branching factor without the use of primary effects, we define E as the total number of all effects in all operators:

$$E = \sum_{op} \#(\text{Prim}(op)) + \sum_{op} \#(\text{Side}(op)).$$

Then, the branching factor of the PRODIGY decision cycle is $BF \cdot \frac{E}{2 \cdot N}$.

Finally, we assume that all operators have the same cost. The main assumptions are listed in Figure 3.11, and the notation used in the analysis is summarized in Figure 3.12 (the last two symbols in Figure 3.12 are introduced in the next few paragraphs).

Exponential efficiency improvement

First, we consider search without primary effects and assume that the resulting solution comprises n operators. Since the branching factor of a decision cycle is $BF \cdot \frac{E}{2 \cdot N}$, we conclude that the number of nodes representing one-operator incomplete solutions is $BF \cdot \frac{E}{2 \cdot N}$, the number of nodes representing two-operator solutions is $(BF \cdot \frac{E}{2 \cdot N})^2$, and so on. The total number of nodes is

$$1 + BF \cdot \frac{E}{2 \cdot N} + (BF \cdot \frac{E}{2 \cdot N})^2 + ... + (BF \cdot \frac{E}{2 \cdot N})^n = \frac{(BF \cdot \frac{E}{2 \cdot N})^{n+1} - 1}{BF \cdot \frac{E}{2 \cdot N} - 1}. \quad (3.1)$$

Now suppose that the problem solver uses primary effects and the cost increase is C, that is, the resulting solution has $C \cdot n$ operators, which translated into the proportional increase of the search depth. The branching factor of adding an operator with a matching primary effect is $BF \cdot \frac{PE}{2 \cdot N}$; hence, the number of nodes is

$$\frac{(BF \cdot \frac{PE}{2 \cdot N})^{C \cdot n+1} - 1}{BF \cdot \frac{PE}{2 \cdot N} - 1}. \tag{3.2}$$

We next determine the ratio of running times of search with and without primary effects. Since the running time is proportional to the number of nodes in the search space, the time ratio is determined by the ratio of search-space sizes:

$$\frac{((BF \cdot \frac{PE}{2 \cdot N})^{C \cdot n+1} - 1)/(BF \cdot \frac{PE}{2 \cdot N} - 1)}{((BF \cdot \frac{E}{2 \cdot N})^{n+1} - 1)/(BF \cdot \frac{E}{2 \cdot N} - 1)} \approx \frac{(BF \cdot \frac{PE}{2 \cdot N})^{C \cdot n}}{(BF \cdot \frac{E}{2 \cdot N})^{n}} = \left(\frac{(BF \cdot \frac{PE}{2 \cdot N})^{C}}{BF \cdot \frac{E}{2 \cdot N}} \right)^{n}. \tag{3.3}$$

If we denote the base of this exponent by r:

$$r = \frac{(BF \cdot \frac{PE}{2 \cdot N})^{C}}{BF \cdot \frac{E}{2 \cdot N}} = \frac{2 \cdot N}{E \cdot BF} \cdot \left(\frac{PE \cdot BF}{2 \cdot N} \right)^{C}, \tag{3.4}$$

then we may rewrite Expression 3.3 for the time ratio as r^n, which shows that the *saving in running time grows exponentially with the solution length* n.

Conditions for efficiency improvement

Primary effects improve the efficiency only if $r < 1$, which means that $\frac{2 \cdot N}{E \cdot BF} \cdot (\frac{PE \cdot BF}{N})^C < 1$. We solve this inequality with respect to C and conclude that primary effects improve performance when

$$C < \frac{\log E + \log BF - \log(2 \cdot N)}{\log PE + \log BF - \log(2 \cdot N)}. \tag{3.5}$$

Observe that $E > PE$, which implies the right-hand side of Inequality 3.5 is larger than 1. Therefore, if primary effects do not cause a cost increase, that is $C = 1$, then their use reduces the running time. This observation, however, does *not* imply that we should minimize the cost increase: primary effects with a significant cost increase sometimes give a greater search reduction.

We can draw some other conclusions from the expression for r (Equation 3.4). The base of the exponent in this expression, $\frac{PE \cdot BF}{N}$, is the branching factor of the search with primary effects and, therefore, it is larger than 1. We conclude that r grows with an increase of C, which implies that *the time savings increase with a reduction of* C. Also, r grows with an increase of PE and, therefore, *the time savings increase with a reduction of the number of primary effects* PE.

The efficiency depends on the tradeoff between the number of primary effects PE and the cost increase C: as we select more primary effects, PE increases. whereas C decreases. To minimize r, we have to strike the right balance between C and PE. Observe that the branching factor of search is proportional to PE, and the search depth is proportional to C. Therefore, the choice of additional primary effects increases branching, but reduces the depth.

We also conclude from Equation 3.4 that redundant effects worsen the performance. Recall that a primary effect is redundant if we can demote it to a side effect without increasing solution costs. The demotion of a redundant effect decreases E without increasing C and, hence, reduces the search space.

Discussion of the assumptions

Even though the derivation is based on several strong assumptions, the main conclusions usually remain valid even when the search space does not have the assumed properties. In particular, the relaxation of Assumptions 1 and 4 in Figure 3.11 does not change the quantitative result (see Expression 3.3), though it requires a more complex definition of the *BF* value. We conjecture that Assumption 2 may also be lifted, if we suppose that the use of primary effects does not change the distribution of subgoals.

The described estimation of the time ratio works not only for PRODIGY search, but also for most backward-chaining solvers, including least-commitment systems. In particular, we have applied it to the search space of TWEAK [Fink and Yang, 1995] and obtained the identical expression for the time ratio (see Expression 3.3), but the *BF* value in that derivation had a different meaning. Since TWEAK creates a new node of the search space by inserting a new operator or imposing a constraint on the order of executing old operators, we have defined *BF* as the number of different ways to impose constraints after inserting an operator.

We have estimated the total number of nodes up to the depth of the solution node. If a problem solver uses breadth-first search, it visits all of them; however, PRODIGY uses depth-first strategy and may explore a much smaller space. Its *effective branching factor* is the average number of alternatives considered in every decision point, which depends on the frequency of backtracking [Nilsson, 1971]. This factor is usually proportional to the overall branching factor of the search space, which leads us to the same time-ratio estimate, with *BF* adjusted to account for the smaller effective branching.

The key assumption is that the effective branching factor of the decision cycle, $BF \cdot \frac{PE}{2 \cdot N}$, is proportional to the number of primary effects, *PE*. If a domain does not satisfy it, then we cannot use Expression 3.3 to estimate the search reduction. Advanced heuristics and control rules may significantly reduce the effective branching of search without primary effects, which invalidates the estimate of the time ratio.

In particular, if primary effects do not reduce the effective branching, then they do not improve the efficiency. For example, if a depth-first solver always finds a solution without backtracking, then primary effects do not reduce its search. On the other hand, it may benefit from the use of primary effects that improve the solution quality (see the experiments in Section 3.7).

To summarize, the analysis has shown that an appropriate choice of primary effects significantly reduces the search, whereas poorly selected effects may exponentially increase the search time. Experiments support this conclusion for most domains, even when the search space does not satisfy the assumptions of the analysis. We will present empirical results for ABTWEAK's breadth-first search (Section 3.6) and for the depth-first PRODIGY algorithm (Section 3.7).

3.4 Automatically selecting primary effects

We describe an algorithm that automatically chooses primary effects of operators and inference rules, by static analysis of the domain description. The selection technique is based on simple heuristics, which usually lead to generating a near-complete selection of primary effects; however, they do *not* guarantee completeness.

If problem solving with the resulting selection reveals its incompleteness, then the system either discards the selection or chooses additional primary effects. In Section 3.5, we will describe a learning algorithm that selects more effects and ensures a high probability of completeness.

First, we present the *Chooser* algorithm, which combines several heuristics for selecting primary effects (Section 3.4.1). Second, we give the *Matcher* algorithm, which generates all possible instantiations of operators and inference rules, and improves the effectiveness of *Chooser* (Section 3.4.2). Both algorithms belong to SHAPER's library of description changers.

3.4.1 Selection heuristics

We present a heuristic algorithm, called *Chooser*, which processes a list of operators, and generates a selection of primary and side effects. The purpose of this algorithm is to construct a selection that reduces the branching factor, without a significant violation of completeness.

The human user has two options for affecting the choice of primary effects. First, she may specify a desirable limit C on the cost increase. The algorithm utilizes this limit in the heuristic selection of primary effects; however, it does *not* guarantee that the cost increase of all problems will be within the specified bound C.

Second, the human operator has an option to pre-select some primary and side effects, before applying the *Chooser* algorithm. If an effect is *not* pre-selected as primary or side, then we call it a *candidate effect.* In particular, if the user does not provide any pre-selection, then *all* effects of operators and inference rules are candidate effects.

The algorithm preserves the initial pre-selection, and chooses additional primary effects among the candidate effects. If some candidate effects do not become primary, then problem solvers treat them in the same way as side effects. In other words, PRODIGY search algorithms do *not* distinguish between candidate and side effects.

We summarize the specification of the *Chooser* description changer in Figure 3.13 and show the algorithm that satisfies this specification in Figure 3.14. We do not give pseudocode for two procedures, *Choose-Delete-Operator* and *Choose-Delete-Inf-Rule*, as they are very similar to *Choose-Add-Operator* and *Choose-Add-Inf-Rule.*

The description changer consists of two parts: *Choose-Initial* and *Choose-Extra.* The first part is the main algorithm, which generates an initial selection of primary and side effects. The second module is an optional procedure for choosing additional primary effects, which may be disabled by the user. Note that these two modules may select primary effects not only among simple effects, but also among actions of if-effects.

Generating initial selection

The *Choose-Initial* algorithm ensures that every nonstatic predicate is a primary effect of some operator or inference rule. This condition is necessary for completeness of problem solving with primary effects: if some predicate were not selected as a primary effect, then the solver algorithm would be unable to achieve it.

For every nonstatic predicate *pred* that is *not* a primary effect in the user's pre-selection, the algorithm looks for some operator with a candidate effect *pred*, and makes *pred* a primary

Type of description change: Selecting primary effects of operators and inference rules.

Purpose of description change: Minimizing the number of primary effects, while ensuring near-completeness and a limited cost increase.

Use of other algorithms: None.

Required input: Description of the operators and inference rules.

Optional input: Preferable cost-increase limit C; pre-selected primary and side effects.

Figure 3.13: Specification of the *Chooser* algorithm.

operators	**primary effects**
(a) Pre-selection by the human user.	
carry(<from>,<to>)	add (ball-in <to>)
(b) *Choose-Initial*: Initial selection of primary effects.	
go(<from>,<to>)	del (robot-in <from>) add (robot-in <to>)
throw(<from>,<to>)	del (ball-in <from>)
break(<from>,<to>)	add (door <from> <to>)
(c) *Choose-Extra*: Heuristic selection of extra effects.	
add-door(<from>,<to>)	add (door <from> <to>)
(d) *Completer*: Learned primary effects.	
throw(<from>,<to>)	add (ball-in <to>)

Table 3.2: Steps of selecting primary effects.

effect of this operator. If the predicate is *not* a candidate effect of any operator, then the algorithm makes it a primary effect of some inference rule.

If the user has specified a desired limit C on the cost increase, then the algorithm takes it into account when choosing an operator for achieving *pred* (see the *Choose-Add-Operator* procedure in Figure 3.14). First, the algorithm finds the cheapest operator that achieves *pred*, whose cost is denoted "*min-cost*" in the pseudocode. Then, it identifies the operators whose cost is within $C \cdot$ *min-cost*, and chooses one of them for achieving *pred* as a primary effect.

Example of an initial selection

Consider the application of the *Choose-Initial* algorithm to the Robot Domain in Figure 3.2. We assume that the desired cost-increase limit is $C = 1.5$, and the user has pre-selected "add (ball-in <to>)" as a primary effect of the **carry** operator (see Table 3.2a).

Chooser(C)
The algorithm optionally inputs a desired cost-increase limit C, whose default value is infinity.
It also accesses the operators and inference rules, with their pre-selected primary and side effects.

Call *Choose-Initial*(C), to generate an initial selection of primary effects.
Optionally call *Choose-Extra*, to select additional primary effects.

Choose-Initial(C) — Ensure that every nonstatic predicate is a primary effect.
For every predicate *pred* in the domain description:
  If some operator adds *pred*,
    then call *Choose-Add-Operator*(*pred*,C).
  If some operator or inference rule adds *pred*,
      and no operator adds it as a primary effect,
    then call *Choose-Add-Inf-Rule*(*pred*).
  If some operator deletes *pred*,
    then call *Choose-Del-Operator*(*pred*,C).
  If some operator or inference rule deletes *pred*,
      and no operator deletes it as a primary effect,
    then call *Choose-Del-Inf-Rule*(*pred*).

Choose-Add-Operator(*pred*, C) — Ensure that *pred* is a primary effect of some operator.
Determine the cheapest operator that adds *pred*; let its cost be *min-cost*.
If some operator, with cost at most $C \cdot$ *min-cost*, adds *pred* as a primary effect,
  then terminate (do not select a new primary effect).
If there are operators, with cost at most $C \cdot$ *min-cost*, that add *pred* as a candidate effect,
  then select one of these operators, make *pred* its primary effect, and terminate.
If some operator, with cost larger than $C \cdot$ *min-cost*, adds *pred* as a primary effect,
  then terminate (do not select a new primary effect).
If there are operators, with cost larger than $C \cdot$ *min-cost*, that add *pred* as a candidate effect,
  then select one of these operators, make *pred* its primary effect, and terminate.

Choose-Add-Inf-Rule(*pred*) — Ensure that *pred* is a primary effect of some inference rule.
If some inference rule adds *pred* as a primary effect,
  then terminate (do not select a new primary effect).
If there are inference rules that add *pred* as a candidate effect,
  then select one of these rules, make *pred* its primary effect, and terminate.

Choose-Extra — Ensure that every operator and inference rule has a primary effect.
For every operator and inference rule in the domain description:
  If it has candidate effects, but no primary effects,
    then select one of its candidate effects as a new primary effect.

Figure 3.14: Heuristic selection of primary effects: *Choose-Initial* generates an initial selection of primary and side effects, and then *Choose-Extra* selects additional primary effects. We do not give pseudocode for the *Choose-Delete-Operator* and *Choose-Delete-Inf-Rule* procedures, which are analogous to *Choose-Add-Operator* and *Choose-Add-Inf-Rule*.

Suppose that the algorithm first selects an operator for achieving the effect "add (robot-in <to>)." The cheapest operator that has this effect is **go**(<from>,<to>), whose cost is 2. Thus, the algorithm will try to find an operator, with a candidate effect "add robot-in," whose cost is at most $1.5 \cdot 2 = 3$. The domain includes two operators that satisfy this condition, **go** and **carry**.

If the user does not provide a heuristic for selecting among available operators, then *Choose-Initial* picks the cheapest operator with a matching candidate effect. Thus, the algorithm selects the **go** operator for adding robot-in as a primary effect. Similarly, it selects the **go** operator for achieving "del robot-in," **throw** for "del ball-in," and **break** for "**add door**." We summarize the resulting selection in Table 3.2(b).

Optional heuristics

When the *Choose-Initial* algorithm picks an operator for achieving *pred* as a primary effect, it may have to select among several matching operators. For example, if the algorithm chooses an operator for "add robot-in," with cost at most 3, then it must choose between **go** and **carry**. We have implemented three optional heuristics for selecting among several operators. The user may include them in the *Choose-Initial* procedure or, alternatively, provide her own heuristics.

Choosing an operator without primary effects: If some operator, with a candidate effect *pred*, has no primary effects, then the algorithm selects this operator and marks *pred* as its primary effect. This heuristic ensures that most operators have primary effects; thus, it is effective for domains that do not include unnecessary operators.

Preferring an operator with weak preconditions: When the *Choose-Initial* algorithm uses this heuristic, it looks for the operator with the weakest preconditions that achieves *pred*. This strategy helps to reduce the search, but it may negatively affect the solution quality. For example, suppose that we employ it for choosing primary effects in the Robot Domain, with the cost-increase limit $C = 2$. Then, the algorithm chooses the **break** operator for adding robot-in as a primary effect, because the preconditions of this operator are weaker than that of **go** and **carry**. The resulting selection forces a problem solver to use **break** for moving the robot between rooms; thus, it reduces the search, but leads to constructing costly plans.

Improving the quality of the abstraction hierarchy: If the system utilizes primary effects in abstraction problem solving, then the abstraction hierarchy may depend on the selected effects. We have implemented a selection heuristic, for the *Choose-Initial* algorithm, that improves the quality of the resulting hierarchy. In Section 5.1, we will describe this heuristic and use it to combine *Chooser* with an abstraction algorithm.

Choosing additional effects

The optional *Choose-Extra* procedure selects additional primary effects, to ensure that every operator and every inference rule has at least one primary effect (see Figure 3.14). If the user believes that the domain description has no unnecessary operators, then she should enable this procedure.

For every operator and rule that has *no* primary effects, *Choose-Extra* algorithm promotes one of its candidate effects to a primary effect. For example, if we apply *Choose-Extra* to the Robot Domain with the initial selection given in Table 3.2(a,b), then it will select the effect "add (door <from> <to>)" of the **add-door** inference rule as a new primary effect (see Table 3.2c).

The *Choose-Extra* procedure includes two heuristics for selecting a primary effect among several candidate effects. First, it chooses effects that add predicates, rather than deletion effects. For example, if the go operator did not have primary effects, then the algorithm would choose "add (robot-in <from>)," rather than "del (robot-in <to>)," as its new primary effect. Second, it prefers predicates achieved by the fewest number of other operators.

If we use *Chooser* in conjunction with an abstraction generator, then we disable these two heuristics and instead employ a selection technique that improves the quality of the resulting abstraction hierarchy (see Section 5.1).

Running time

We next give the time complexity of the *Choose-Initial* and *Choose-Extra* algorithms, which does *not* include the complexity of optional heuristics for selecting among available operators and candidate effects. If the user adds complex heuristics to the *Chooser* algorithm, they may significantly increase the running time.

The running time depends on the total number of all effects, in all operators and inference rules, and on the number of nonstatic predicates in the domain description. We denote the total number of effects by E, and the number of nonstatic predicates by N (see Figure 3.12).

For each predicate *pred*, the *Choose-Initial* algorithm invokes four procedures. First, it calls *Choose-Add-Operator* and *Choose-Del-Operator*, whose running time is proportional to the total number of effects in all operators. Then, it applies *Choose-Add-Inf-Rule* and *Choose-Del-Inf-Rule*, whose time is proportional to the number of inference-rule effects. Thus, the complexity of executing all four procedures is $O(E)$, and the overall time for processing all nonstatic predicates is $O(E \cdot N)$. Finally, the complexity of the *Choose-Extra* procedure is $O(E)$, which implies that the overall running time of the *Chooser* algorithm is $O(E \cdot N)$.

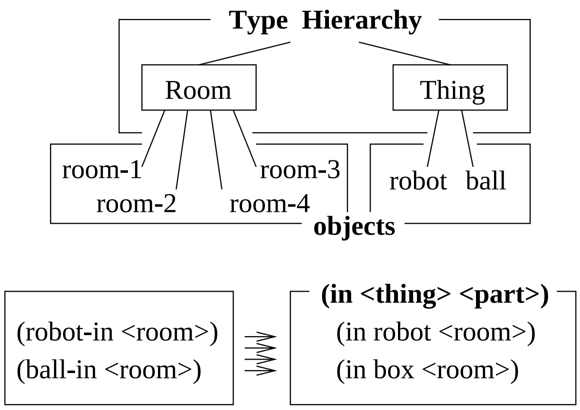
We have implemented *Chooser* in Common Lisp and timed its execution on a Sun 5 machine. The *Choose-Initial* procedure takes about $(E \cdot N) \cdot 10^{-4}$ seconds, whereas the execution time of *Choose-Extra* is approximately $E \cdot 6 \cdot 10^{-4}$ seconds.

3.4.2 Instantiating the operators

We have described an algorithm for selecting primary effects of *uninstantiated* operators and inference rules. We now point out some drawbacks of the selection heuristics and then describe a technique for eliminating these drawbacks.

The main shortcoming of the heuristics for processing uninstantiated operators is their sensitivity to syntactic features of a specific domain encoding. In particular, if we use too general predicates in the description of operators, *Chooser* may produce an insufficient selection of primary effects.

To illustrate this problem, we consider the Robot Domain (see Figure 3.2) and replace the predicates (robot-in <room>) and (ball-in <room>) in the description of operators with a



(a) Replacing two predicates with a more general one.

operators	**primary effects**
Pre-selection by the human user.	
carry(<from>,<to>)	add (in ball <to>)
Choose-Initial: Initial selection of primary effects.	
go(<from>,<to>)	del (in robot <from>)
break(<from>,<to>)	add (door <from> <to>)
Choose-Extra: Heuristic selection of extra effects.	
throw(<from>,<to>)	add (in ball <from>)
add-door(<from>,<to>)	add (door <from> <to>)

(b) Resulting incomplete selection.

Figure 3.15: Generating an incomplete selection of primary effects, due to the use of general predicates. To avoid this problem, we need to apply the *Matcher* algorithm, which generates all instantiations of the operators and inference rules, before running *Chooser*.

general predicate (in <thing> <room>), where <thing> is either robot or ball (see Figure 3.15a). If the user has pre-selected "add (in <thing> <to>)" as a primary effect of **carry**, then *Chooser* constructs the selection given in Figure 3.15(b).

Note that, since we use a general predicate for encoding the effects (in robot <room>) and (in ball <room>), the algorithm does not distinguish between these two effects; hence, it generates an incomplete selection, which does not include primary effects for adding (in robot <room>) and deleting (in ball <room>).

If the costs of some operators in the domain description are not constant, the system needs to estimate their cost ranges and then utilize these ranges in choosing primary effects. The use of general operators may result in wide ranges, which negatively affect the effectiveness of selection heuristics. For example, if the user specifies a generalized operator for going between rooms and breaking through walls (Figure 3.16), with cost range from 2 to 4, then *Chooser* cannot determine whether moving the robot is cheaper than carrying the ball.

We may avoid these problems by constructing all instantiations of the available operators and inference rules, and applying the *Chooser* algorithm to select primary effects of the resulting instantiations. We illustrate the use of this technique in the Robot Domain, discuss its main advantages and shortcomings, and then describe an algorithm that instantiates operators and inference rules.

After *Chooser* has processed the instantiations, we may optionally convert the resulting selection of primary effects to the corresponding selection for uninstantiated operators. In Figure 3.17, we give a simple conversion procedure, which loops through operators and inference rules, and marks their primary effects. We denote the currently considered operator or inference rule by $step_{\mathrm{u}}$, where the subscript "u" stands for "uninstantiated." The procedure identifies the effects of $step_{\mathrm{u}}$ that are primary in at least one instantiation, and marks them all as primary effects of $step_{\mathrm{u}}$ itself.

```
go-or-break(<from>, <to>)
<from>, <to>: type Room
Pre:  (robot-in <from>)
Eff:  del (robot-in <from>)
      add (robot-in <to>)
      (if (not (door <from> <to>))
         add (door <from> <to>))
Cost function:
       If (door <from> <to>),
          then, return 2; else, return 4.
```

Figure 3.16: A general operator for moving the robot; it has a wide cost range, from 2 to 4, which reduces the effectiveness of heuristics for choosing primary effects.

Generalize-Selection
For every uninstantiated operator and inference rule $step_{\mathrm{u}}$:
  For every candidate effect of $step_{\mathrm{u}}$:
    If this effect is primary in some instantiation,
      Then make it a primary effect of $step_{\mathrm{u}}$.

Figure 3.17: Converting a fully instantiated selection of primary effects to the corresponding selection for the uninstantiated operators. We may optionally apply this procedure after choosing primary effects of the operator instantiations. The procedure marks an effect of $step_{\mathrm{u}}$ as primary if and only if it has been chosen as a primary effect in at least one instantiation of $step_{\mathrm{u}}$.

Example of instantiating the operators

We consider a simplified version of the Robot Domain, which does *not* include the **break** operator and **add-door** inference rule, and suppose that the room layout is as shown in Figure 3.2(a). We give static predicates that encode this layout in Figure 3.18 and list all instantiated operators in Figure 3.19.

Observe that the list does not include infeasible instantiations, such as **go**(room-1,room-4) or **throw**(room-1,room-3). Also note that the preconditions of the instantiated operators do not include static literals, which always hold for the given room layout.

Suppose that the user has marked "add (in <thing> <to>)" as a primary effect of **carry** and invoked *Chooser* to select other primary effects. If the algorithm processes the instantiated operators, it generates the selection given in Table 3.3.

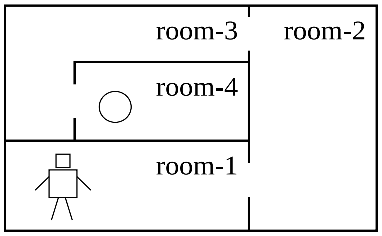
We may then apply the *Generalize-Selection* procedure, which chooses the corresponding primary effects of uninstantiated operators, thus constructing the selection in Table 3.4. Note that the resulting selection is similar to that in Table 3.2, despite the use of a more general predicate in the operator description.

operators	**primary effects**
Pre-selection by the human user.	
carry(room-1,room-2)	add (in robot room-2)
carry(room-2,room-1)	add (in robot room-1)
carry(room-2,room-3)	add (in robot room-3)
carry(room-3,room-2)	add (in robot room-2)
carry(room-3,room-4)	add (in robot room-4)
carry(room-4,room-3)	add (in robot room-3)
Choose-Initial: Initial selection of primary effects.	
go(room-1,room-2)	del (in robot room-1) add (in robot room-2)
go(room-2,room-1)	del (in robot room-2) add (in robot room-1)
go(room-2,room-3)	del (in robot room-2) add (in robot room-3)
go(room-3,room-2)	del (in robot room-3) add (in robot room-2)
go(room-3,room-4)	del (in robot room-3) add (in robot room-4)
go(room-4,room-3)	del (in robot room-4) add (in robot room-3)
throw(room-1,room-2)	del (in ball room-1)
throw(room-2,room-3)	del (in ball room-2)
throw(room-3,room-4)	del (in ball room-3)
throw(room-4,room-3)	del (in ball room-4)
Choose-Extra: Heuristic selection of extra effects.	
NO ADDITIONAL PRIMARY EFFECTS	

Table 3.3: Primary effects of the fully instantiated operators, selected by the *Chooser* algorithm.

operators	**primary effects**
carry(<from>,<to>)	add (in ball <to>)
go(<from>,<to>)	del (in robot <from>) add (in robot <to>)
throw(<from>,<to>)	del (in ball <from>)

Table 3.4: Primary effects of uninstantiated operators in the simplified Robot Domain, which correspond to *Chooser*'s selection for the instantiated domain description (see Table 3.3).



Set of Objects

room-1, room-2, room-3, room-4: type Room
robot, ball: type Thing

Static Literals

(door room-1 room-2)	(door room-2 room-1)
(door room-2 room-3)	(door room-3 room-2)
(door room-3 room-4)	(door room-4 room-3)

Figure 3.18: Encoding of the room layout in the simplified Robot Domain.

Advantages and drawbacks

The described example illustrates both advantages and shortcomings of processing the instantiated operators. The main advantage is the reduction of *Chooser*'s sensitivity to a specific domain encoding. In addition, the system may generate instantiations for a particular class of problems, thus enabling the *Chooser* algorithm to construct a problem-specific selection of primary effects. The utilization of problem-specific information often helps to improve the efficiency of search with primary effects.

For example, consider the Robot Domain and suppose that we need to solve several problems in which there is no ball. To generate an appropriate selection of primary effects, we remove the ball from the set of objects given in Figure 3.18. Then, the system prunes all instantiations of **throw** and **carry**, and *Chooser* does not select primary effects of these two operators.

On the negative side, the instantiation technique has two major drawbacks. First, the human operator must supply a list of the allowed object instances, and the resulting selection may be inappropriate for solving problems with other instances. Thus, the technique works only for domains with a finite set of objects.

Second, the generation of all instantiations may lead to a combinatorial explosion. To reduce the number of instantiated operators, the algorithm exploits information about static features of the domain and prunes most of the infeasible instantiations; however, it does not guarantee the complete elimination of unexecutable operators.

Generating feasible instantiations

We next present the *Matcher* algorithm, which inputs a list of the available objects, and generates instantiated operators and inference rules. If the human operator specifies static literals that hold in the initial states of all problems, the algorithm utilizes them for pruning infeasible instantiations. We say that the specified literals have *known truth values,* whereas the values of the other literals are unknown.

We show the specification of *Matcher* in Figure 3.20 and give pseudocode in Figure 3.21. When *Matcher* processes an operator or inference rule, denoted $step_{\mathrm{u}}$ in the pseudocode, it generates all instantiations of $step_{\mathrm{u}}$ that match the given static literals. In Figure 3.23, we illustrate the processing of the **go** operator in the simplified Robot Domain. Recall that the simplified domain has no **break** operation; hence, the predicate `(door <from> <to>)` is static.

go(room-1, room-2)
Pre: (in robot room-1)
Eff: del (in robot room-1)
add (in robot room-2)

go(room-2, room-3)
Pre: (in robot room-2)
Eff: del (in robot room-2)
add (robot-in room-3)

go(room-3, room-4)
Pre: (in robot room-3)
Eff: del (in robot room-3)
add (in robot room-4)

go(room-2, room-1)
Pre: (in robot room-2)
Eff: del (in robot room-2)
add (in robot room-1)

go(room-3, room-2)
Pre: (in robot room-3)
Eff: del (in robot room-3)
add (in robot room-2)

go(room-4, room-3)
Pre: (in robot room-4)
Eff: del (in robot room-4)
add (in robot room-3)

throw(room-1, room-2)
Pre: (in robot room-1)
(in ball room-1)
Eff: del (in ball room-1)
add (in ball room-2)

throw(room-2, room-3)
Pre: (in robot room-2)
(in ball room-2)
Eff: del (in ball room-2)
add (in ball room-3)

throw(room-3, room-4)
Pre: (in robot room-3)
(in ball room-3)
Eff: del (in ball room-3)
add (in ball room-4)

throw(room-2, room-1)
Pre: (in robot room-2)
(in ball room-2)
Eff: del (in ball room-2)
add (in ball room-1)

throw(room-3, room-2)
Pre: (in robot room-3)
(in ball room-3)
Eff: del (in ball room-3)
add (in ball room-2)

throw(room-4, room-3)
Pre: (in robot room-4)
(in ball room-4)
Eff: del (in ball room-4)
add (in ball room-3)

carry(room-1, room-2)
Pre: (in robot room-1)
(in ball room-1)
Eff: del (in robot room-1)
add (in robot room-2)
del (in ball room-1)
add (in ball room-2)

carry(room-2, room-3)
Pre: (in robot room-2)
(in ball room-2)
Eff: del (in robot room-2)
add (in ball room-3)
del (in ball room-2)
add (in ball room-3)

carry(room-3, room-4)
Pre: (in robot room-3)
(in ball room-3)
Eff: del (in robot room-3)
add (in robot room-4)
del (in ball room-3)
add (in ball room-4)

carry(room-2, room-1)
Pre: (in robot room-2)
(in ball room-2)
Eff: del (in robot room-2)
add (in robot room-1)
del (in ball room-2)
add (in ball room-1)

carry(room-3, room-2)
Pre: (in robot room-3)
(in ball room-3)
Eff: del (in robot room-3)
add (in robot room-2)
del (in ball room-3)
add (in ball room-2)

carry(room-4, room-3)
Pre: (in robot room-4)
(in ball room-4)
Eff: del (in robot room-4)
add (in robot room-3)
del (in ball room-4)
add (in ball room-3)

Figure 3.19: Fully instantiated operators in the simplified Robot Domain, which has no **break** operation and **add-door** inference rule. We show only feasible instantiations, which can be executed in the given room layout (see Figure 3.18).

Type of description change: Generating fully instantiated operators and inference rules.

Purpose of description change: Producing all feasible instantiations, while avoiding the instantiations that can never be executed.

Use of other algorithms: None.

Required input: Description of the operators and inference rules; all possible values of the variables in the domain description.

Optional input: Static literals that hold in the initial states of all problems.

Figure 3.20: Specification of the *Matcher* algorithm.

Matcher
The algorithm inputs a set of the available object instances,
as well as a list of static literals that hold in all problems.
It also accesses the description of operators and inference rules.

For every uninstantiated operator and inference rule $step_u$:
1. Apply the *Remove-Unknown* procedure (see Figure 3.22) to the preconditions of $step_u$.
   (The procedure simplifies the preconditions, by pruning predicates with unknown truth values.)
2. For every if-effect of $step_u$,
   apply *Remove-Unknown* to simplify the conditions of the if-effect.
3. Generate all possible instantiations of the resulting simplified version of $step_u$.
4. Convert them to the corresponding instantiations of the original version of $step_u$.
5. For every instantiation of $step_u$,
   delete literals with known truth values from the preconditions of the instantiation.

Figure 3.21: Generating all feasible instantiations of operators and inference rules.

First, the algorithm simplifies the precondition expression of $step_u$, by removing all predicates with unknown truth values (see Line 1 in Figure 3.21). Similarly, it simplifies the condition expressions of all if-effects in $step_u$ (Line 2). For instance, when the algorithm processes the go operator, it prunes the (in robot <room>) precondition (see Figure 3.23b).

We give a detailed pseudocode for the *Remove-Unknown* function (see Figure 3.22), which deletes predicates with unknown values from a boolean expression. Note that the expression may include conjunctions, disjunctions, negations, and quantifications. The function recursively parses the input expression, identifies predicates with unknown truth values, and replaces them by boolean constants. If a predicate is inside an odd number of negations, then *Remove-Unknown* replaces it by false; otherwise, the predicate is replaced by true. This simplification procedure preserves all feasible instantiations of the original boolean expression. That is, if an instantiation satisfies the initial expression in at least one domain state, then it also satisfies the simplified expression.

Second, the *Matcher* algorithm generates all possible instances of the simplified version of $step_u$ (Line 3). We use a standard instantiation procedure, which recursively parses the

Remove-Unknown(*bool-exp*, *negated*)
The input includes a boolean expression, *bool-exp*, as well as a boolean value *negated*, which is 'true' if *bool-exp* is inside an uneven number of negations. In addition, the algorithm accesses the list of static literals with known truth values.

Determine whether the expression *bool-exp*
  is a predicate, conjunction, disjunction,
  negation, or quantified expression.
Call the appropriate subroutine (see below),
  and return the resulting expression.

Remove-Predicate(*pred*, *negated*)
The function inputs a predicate, denoted *pred*.

If the truth values of *pred* are unknown,
  then return $\neg$*negated* (that is, 'true' or 'false').
Else, return *pred* (that is, the unchanged predicate).

Remove-From-Conjunction(*bool-exp*, *negated*)
The function inputs a conjunctive expression, *bool-exp*; that is, *bool-exp* has the form '(and *sub-exp sub-exp*...).'

New-Exps := $\emptyset$.
For every term *sub-exp* of the conjunction *bool-exp*:
  new-exp := *Remove-Unknown*(*sub-exp*, *negated*).
  If *new-exp* is false, then return false
    (do not process the remaining terms).
  If *new-exp* is not true or false
    (that is, it is an expression with variables),
    then *New-Exps* := *New-Exps* $\cap$ {*new-exp*}.
If *New-Exps* is $\emptyset$, then return true.
Else, return the conjunction of all terms in *New-Exps*.

Remove-From-Disjunction(*bool-exp*, *negated*)
The function inputs a disjunctive expression, *bool-exp*; that is, *bool-exp* has the form '(or *sub-exp sub-exp*...).'

New-Exps := $\emptyset$.
For every term *sub-exp* of the disjunction *bool-exp*:
  new-exp := *Remove-Unknown*(*sub-exp*, *negated*).
  If *new-exp* is true, then return true
    (do not process the remaining terms).
  If *new-exp* is not true or false
    (that is, it is an expression with variables),
    then *New-Exps* := *New-Exps* $\cap$ {*new-exp*}.
If *New-Exps* is $\emptyset$, then return false.
Else, return the disjunction of all terms in *New-Exps*.

Remove-From-Negation(*bool-exp*, *negated*)
The function inputs a negated expression, *bool-exp*; that is, *bool-exp* has the form '(not *sub-exp*).'

Let *sub-exp* be the expression inside the negation.
new-exp := *Remove-Unknown*(*bool-exp*, $\neg$*negated*).
If *new-exp* is true or false, then return $\neg$*new-exp*
  (that is, the function returns 'false' or 'true').
Else, return the negation of *new-exp*.
  (that is, it returns an expression with variables).

Remove-From-Quantification(*bool-exp*, *negated*)
The function inputs a boolean expression, *bool-exp*, with universal or existential quantification.

Let *sub-exp* be the expression inside the quantification.
new-exp := *Remove-Unknown*(*bool-exp*, *negated*).
If *new-exp* is true or false, then return *new-exp*.
Else, return the quantification of *new-exp*.

Figure 3.22: Deletion of the predicates with unknown truth values from a boolean expression. The *Remove-Unknown* procedure inputs an expression, which encodes either operator preconditions or if-effect conditions, and recursively processes its subexpressions. The procedure may return true, false, or an expression with variables. The auxiliary boolean parameter, denoted *negated*, indicates whether the current subexpression is inside an odd number of negations. When the *Matcher* algorithm invokes *Remove-Unknown*, it sets the initial value of *negated* to false.

Set of Objects
room-1, room-2, room-3, room-4: type Room
robot, ball: type Thing

Static Literals
(door room-1 room-2) (door room-2 room-1)
(door room-2 room-3) (door room-3 room-2)
(door room-3 room-4) (door room-4 room-3)

go(<from>, <to>)
Pre: (in robot <from>)
(door <from> <to>)
Eff: del (in robot <from>)
add (in robot <to>)

(a) Input of the *Matcher* algorithm: Available object instances, literals with known truth values, and an operator description.

go(<from>, <to>)
Pre: (door <from> <to>)
Eff: del (in robot <from>)
add (in robot <to>)

(b) Removal of the predicates with unknown truth values from the precondtion expression.

<from>	**<to>**
room-1	room-2
room-2	room-3
room-3	room-4
room-2	room-1
room-3	room-2
room-4	room-3

(c) Generation of all matching instantiations of variables.

go(room-1, room-2)
Pre: (in robot room-1)
(door room-1 room-2)
Eff: del (in robot room-1)
add (in robot room-2)

go(room-1, room-2)
Pre: (in robot room-1)
(door room-1 room-2)
Eff: del (in robot room-1)
add (in robot room-2)

go(room-1, room-2)
Pre: (in robot room-1)
(door room-1 room-2)
Eff: del (in robot room-1)
add (in robot room-2)

go(room-1, room-2)
Pre: (in robot room-1)
(door room-1 room-2)
Eff: del (in robot room-1)
add (in robot room-2)

go(room-1, room-2)
Pre: (in robot room-1)
(door room-1 room-2)
Eff: del (in robot room-1)
add (in robot room-2)

go(room-1, room-2)
Pre: (in robot room-1)
(door room-1 room-2)
Eff: del (in robot room-1)
add (in robot room-2)

(d) Construction of all operator instantiations.

go(room-1, room-2)
Pre: (in robot room-1)
Eff: del (in robot room-1)
add (in robot room-2)

go(room-2, room-3)
Pre: (in robot room-2)
Eff: del (in robot room-2)
add (robot-in room-3)

go(room-3, room-4)
Pre: (in robot room-3)
Eff: del (in robot room-3)
add (in robot room-4)

go(room-2, room-1)
Pre: (in robot room-2)
Eff: del (in robot room-2)
add (in robot room-1)

go(room-3, room-2)
Pre: (in robot room-3)
Eff: del (in robot room-3)
add (in robot room-2)

go(room-4, room-3)
Pre: (in robot room-4)
Eff: del (in robot room-4)
add (in robot room-3)

(e) Deletion of the precondtion literals with known truth values.

Figure 3.23: Constructing the instantiations of the **go** operator in the simplified Robot Domain.

precondition expression and if-effect conditions, and outputs all instances of $step_u$ that match the given static literals. In Figure 3.23(c), we give a table of all variable instantiations for the **go** operator. Then, the algorithm constructs the corresponding instances of the full version of $step_u$ (Line 4), as illustrated in Figure 3.23(d).

Third, *Matcher* removes the static literals with known truth values from the preconditions of the resulting instances of $step_u$ (Line 5). For example, it removes the **break** literals from the instantiated **go** operators, as shown in Figure 3.23(e). Note that the resulting instantiations of **go** are identical to those in Figure 3.19.

Running time

We now determine the time complexity of generating the instantiations and selecting their primary effects. The complexity depends on the total number of effects, in all generated instantiations of operators and inference rules, and the total number of nonstatic literals in the instantiated domain descriptions. We denote the number of all instantiated effects by E_i and the number of nonstatic literals by N_i, where the subscript "i" stands for "instantiated."

The running time of the *Matcher* algorithm is proportional to the total number of instantiated effects. The per-effect time depends on the complexity of precondition expressions; in our experiments, it ranged from 1 to 5 milliseconds. Thus, the total time for instantiating all operators and inference rules is within $E_i \cdot 5 \cdot 10^{-3}$ seconds.

The complexity of applying the *Chooser* algorithm to the instantiated domain description is $O(E_i \cdot N_i)$. Finally, an efficient implementation of *Generalize-Selection* algorithm takes $O(E_i)$ time. Thus, the overall time complexity of generating all instantiations, choosing their primary effects, and constructing the corresponding selection for uninstantiated operators is $O(E_i \cdot N_i)$. In practice, these procedures take up to $(E_i + 60) \cdot N_i \cdot 10^{-4}$ seconds.

3.5 Learning additional primary effects

The *Chooser* algorithm may significantly improve the efficiency of search; however, the selected primary effects sometimes cause two serious problems. First, the resulting selection is not immune to incompleteness and, second, its use may lead to a large cost increase.

For example, suppose that the robot and the ball are in room-1, and the goal is to move the ball to room-2 and keep the robot in room-1. We may achieve it by **throw**(room-1,room-2); however, if the solver uses primary effects selected by *Chooser* (see Table 3.2a–c), it will not consider this solution, because the new position of the ball is not a primary effect of **throw**. Instead, it will find a costlier solution, "**carry**(room-1,room-2), **go**(room-2,room-1)."

To address this problem, we have developed an inductive learning algorithm, called *Completer*, that chooses additional primary effects and ensures a high probability of completeness and limited cost increase. This algorithm also serves as a description changer in the $\mathcal{S}$HAPER system. The main drawback of *Completer* is its significant learning time, which may prove larger than the time for solving individual problems without primary effects; however, we usually amortize it over multiple problem instances.

The learner inputs a selection of primary effects, produced by either the human user or the *Chooser* algorithm, and the user-specified limit C on the cost increase. The input

Type of description change: Selecting primary effects of operators and inference rules.

Purpose of description change: Minimizing the number of primary effects, while ensuring that the cost increase is no larger than the user-specified bound C.

Use of other algorithms: A problem solver (PRODIGY or ABTWEAK), which constructs replacing sequences; a generator of initial states.

Required input: Description of the operators and inference rules; limit C on the allowed cost increase.

Optional input: Pre-selected primary and side effects; set of initial states.

Figure 3.24: Specification of the *Completer* algorithm.

selection usually consists of primary and candidate effects; however, it may also include pre-selected side effects.

The algorithm tests the completeness of the input selection, by solving a collection of simple problems. If the selection proves incomplete or does not satisfy the cost-increase limit, it promotes some candidate effects to primary effects. Recall that problem solvers do *not* differentiate between unpromoted candidate effects and side effects. Thus, all unpromoted effects become side effects upon the completion of learning.

The completeness test is based on the condition given in Section 3.2.2. For every operator and inference rule, the algorithm generates several initial states that satisfy its preconditions and tries to construct primary effect-justified replacing sequences for these states. If *Completer* cannot find replacing sequences within the user-specified cost increase, it concludes that the current selection is incomplete and promotes some candidate effects.

The *Completer* algorithm does *not* choose new primary effects among the pre-selected side effects. Thus, if the user specifies inappropriate side effects, the algorithm may be unable to generate a complete selection. We have provided an option for testing the completeness of the final selection, which helps the user to detect possible problems with the manual pre-selection of side effects. On the other hand, an expert user may purposefully specify side effects that prevent completeness. For example, she may sacrifice completeness for efficiency or generate a selection for a limited subclass of problems.

The learning algorithm has to use some backward-chaining solver, which searches for replacing sequences. We experimented with the use of PRODIGY and TWEAK search algorithms. In addition, *Completer* needs some procedure for generating random initial states or, alternatively, access to a library of previously encountered domain states. We summarize the specification of *Completer* in Figure 3.24.

We use the theory of *probably approximately correct learning* in developing an algorithm that satisfies this specification. First, we present the learning algorithm (Section 3.5.1) and discuss some heuristics for improving its effectiveness (Section 3.5.2). Then, we derive the sample complexity of the algorithm, that is, the number of learning examples required for ensuring a high probability of completeness and limited cost increase.

3.5.1 Inductive learning algorithm

We describe a technique for testing the completeness of the initial selection and learning additional primary effects, which ensure a high probability of completeness. We explain the use of two special parameters for specifying the required probability of learning an appropriate selection, give the learning algorithm, and illustrate its application to the Robot Domain.

Probability of success

The input of the learning algorithm includes not only the initial selection of primary effects and the limit C on the allowed cost increase, but also two probability values, ϵ and δ. These values are standard parameters of the probably approximately correct (PAC) learning [Valiant, 1984], which determine the probability requirements for the success of inductive learning. We briefly describe the meaning of ϵ and δ before presenting the learning algorithm. In Section 3.5.3, we give a detailed explanation of their use in the sample-complexity analysis.

The ϵ value determines the required probability of completeness and limited cost increase. The learning algorithm must ensure that a randomly selected solvable problem has a primary-effect justified solution, within the cost increase C, with probability at least $1 - \epsilon$. In other words, at most ϵ of all solvable problems may become unsolvable due to the use of primary effects.

The δ value determines the likelihood of success of the inductive learning. The probability that at most ϵ of all solvable problems may become unsolvable must be at least $1 - \delta$. To summarize, the learning algorithm ensures with probability at least $1 - \delta$ that a primary-effect complete solver is able to find solutions, within the cost increase C, to $1 - \epsilon$ of all solvable problems.

Learning loop

In Figure 3.25, we give the inductive learning algorithm, which loops through the available operators and inference rules and selects new primary effects among their candidate effects. The selection of additional primary effects is based on the derived completeness condition (see Section 3.2.2). When the algorithm processes $step_{\mathrm{u}}$, it generates multiple learning examples and uses them to identify missing primary effects of $step_{\mathrm{u}}$. A *learning example* consists of an initial state I, satisfying the preconditions of $step_{\mathrm{u}}$, and a matching instantiation of $step_{\mathrm{u}}$, denoted $step_{\mathrm{i}}$ (where "i" is for "instantiated").

The *Learn-Example* procedure of *Completer* (see Figure 3.25) constructs and processes a learning example. The construction of an example consists of two steps. First, the learner invokes a procedure for generating random initial states, which returns a state that satisfied $step_{\mathrm{u}}$'s preconditions (see Step 1b). We will discuss the techniques for producing initial states in Section 3.5.2. Second, *Completer* unifies the preconditions of $step_{\mathrm{u}}$ with the literals of the resulting state and generates a matching operator instantiation (see Step 2b). If multiple instantiations of $step_{\mathrm{u}}$ match the state, then the learner randomly chooses one of them.

After constructing a learning example, the algorithm uses it to test the completeness of the current selection (Steps 3b and 4b). The algorithm calls the *Generate-Goal* procedure (see

Completer(C, ϵ, δ)
The algorithm inputs the allowed cost increase, C, and the success-probability parameters, ϵ and δ.
It also accesses the operators and inference rules, with their pre-selected primary and side effects.

For every uninstantiated operator and inference rule $step_u$:
1a. Determine the required number m of learning examples,
which depends on $step_u$, ϵ, and δ (see Section 3.5.3).
2a. Repeat m times:
If $step_u$ has no candidate effects,
then terminate the processing of $step_u$ and go to the next operator or inference rule.
Else, call **Learn-Example**$(step_u, C)$, to generate and process a learning example.

Learn-Example$(step_u, C)$
1b. Call the initial-state generator, to produce a state I that satisfies the preconditions of $step_u$.
2b. Generate a full instantiation $step_u$, denoted $step_i$, that matches the initial state I.
3b. Call *Generate-Goal*$(step_i, I)$, to produce the goal $G(step_i, I)$ of a replacing sequence.
4b. Call the problem solver, to search for a replacing sequence with a cost at most $C \cdot cost(step_u)$.
5b. If the solver fails, select some candidate effect of $step_u$ as a new primary effect.

Figure 3.25: Inductive learning of additional primary effects.

Figure 3.10) to construct the corresponding goal $G(step_i, I)$ and then invokes the available problem solver, to construct a primary-effect justified replacing sequence. The solver searches for a replacing sequence within the cost limit $C \cdot cost(step_i)$, which bounds the search depth.

If the solver fails to construct a primary-effect justified sequence, then the current selection is incomplete and the learning algorithm chooses some candidate effect of $step_u$ as a new primary effect (Step 5b). In Section 3.5.2, we will discuss some heuristics for selecting among the candidate effects.

When we use *Completer* in the ABTWEAK system, the problem solver performs a best-first search for a replacing sequence, and either finds a primary-effect justified replacement or reaches the cost bound. The explored space is usually small, because of a low cost limit. The time for processing an example on a Sun 1000 computer varied from a few milliseconds to half second, depending on the domain.

When we use a PRODIGY system for the generation of replacing sequences, it performs a heuristic depth-first search. When the solver reaches the cost limit, it backtracks and considers a different branch of the search space. If the learning example has no replacing sequence, the solver continues the search until it explores the entire search space. This exploration sometimes takes significant time, because of a large branching factor.

To avoid this problem, the user may provide a time bound for the processing of a learning example. She may specify a constant bound or, alternatively, a Lisp function that computes a bound for every example. If PRODIGY reaches a time bound, it returns failure, and *Completer* selects a new primary effect. This strategy reduces the learning time, but it may result in selecting redundant primary effects.

The number of learning examples, m, depends on the user-specified parameters ϵ and δ, which determine the required probability of generating a complete selection (see Step 1a of *Completer*). The smaller the values of ϵ and δ, the larger the number of examples. In addition, m depends on the number of candidate effects of $step_{\mathrm{u}}$. We will derive an expression for computing m in Section 3.5.3.

The *Completer* algorithm usually uses the required number m of learning examples in processing $step_{\mathrm{u}}$ (see Step 1b); however, if *Completer* promotes *all* candidate effects of $step_{\mathrm{u}}$ to primary effects, without considering all m examples, then it terminates the processing and moves on to the next operator or inference rule.

Example of learning

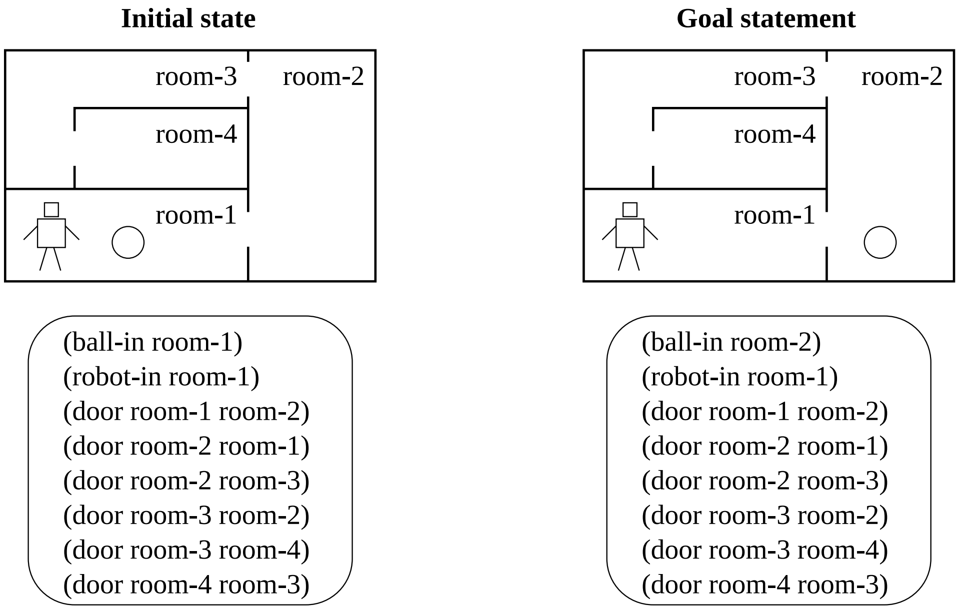
We consider the application of the learning algorithm to the Robot Domain (see Figure 3.2), with the initial selection produced by the *Chooser* algorithm, as shown in Table 3.2(a–c). We assume that the maximal allowed cost increase is $C = 2$, and that the algorithm first considers the **throw** operator, whose candidate effect in the initial selection is the new position of the ball.

Suppose that *Completer* generates the initial state shown in the left of Figure 3.26(a) and the matching operator instantiation **throw**(room-1,room-2). It then invokes the *Generate-Goal* procedure, which produces the goal given in the right of Figure 3.26(a). This goal includes moving the ball to room-2, which is the candidate effect of **throw**, as well as leaving the robot in room-1 and preserving all doorways, which is the part of the initial state that has to remain unchanged.

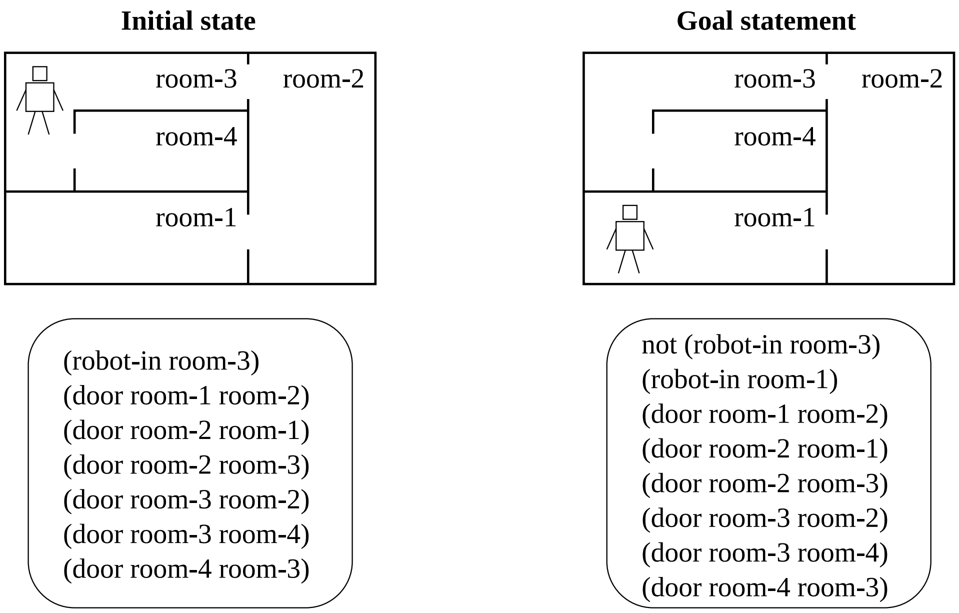
The learning algorithm calls the problem solver to generate a primary-effect justified solution, with a cost at most $C \cdot cost(\textbf{throw}) = 2 \cdot 2 = 4$. The solver fails to find a solution within this cost limit, because the cheapest replacing sequence, which is "**carry**(room-1,room-2), **go**(room-2,room-1)," has a cost of 5. After this failure, *Completer* chooses the candidate effect of the **throw** operator, ball-in, as a new primary effect. If the operator had several candidate effects, the algorithm could choose any of them; however, **throw** has only one candidate effect.

Now suppose that *Completer* considers the **break** operator, chooses an initial state with the robot in room-4, and generates the matching instantiation **break**(room-4,room-1) (see Figure 3.26(b). The candidate effect of this operator is the new position of the robot and, thus, *Completer* considers the problem of moving the robot from room-4 to room-1, within the cost limit $C \cdot cost(\textbf{break}) = 2 \cdot 4 = 8$. The solver algorithm finds a replacing sequence "**go**(room-4,room-3), **go**(room-3,room-2), **go**(room-2,room-1)," with a cost of 6, and the learning algorithm does *not* choose a new primary effect of **break**.

We summarize the steps of choosing primary effects for the Robot Domain in Table 3.2 and show the resulting selection in Table 3.5. This selection is complete and ensures a limited cost increase. For the world map used in the learning examples (see Figure 3.2a), the cost increase is within 1.5.



(a) Example for **throw.**



(b) Example for **break.**

Figure 3.26: Learning example.

operators	**primary effects**
go(<from>,<to>)	del (robot-in <from>), add (robot-in <to>)
carry(<to>,<from>)	add (ball-in <to>)
throw(<from>,<to>)	del (ball-in <from>), add (ball-in <to>)
break(<from>,<to>)	add (door <from> <to>)
add-door(<from>,<to>)	add (door <from> <to>)

Table 3.5: Automatically selected primary effects in the Robot Domain.

3.5.2 Selection heuristics and state generation

We discuss several heuristics used in the *Completer* algorithm and then describe a technique for generating random initial states.

Choice among candidate effects

When *Completer* cannot find a replacing sequence, it often has to select a new primary effect among several candidate effects. An inappropriate choice may increase the redundancy, which affects the efficiency of problem solving. The development of effective heuristics for selecting among available candidate effects is an open problem. We have used two selection techniques, which usually help to avoid redundancy, especially in large-scale domains.

The first technique enables the algorithm to prune some of the inappropriate choices, by analyzing partial solutions to the replacing problem. The underlying idea is to identify the candidate effects that can be achieved by replacing sequences. When the problem solver fails to find a replacing sequence for some learning example $(step_i, I)$, it may construct incomplete solutions that achieve some candidate effects of $step_i$. Then, *Completer* selects a new primary effect among *unachieved* candidate effects.

For example, suppose that the learning algorithm processes the operator **break**(<from>,<to>), with the only primary effect "add (door <from> <to>)," and that the maximal allowed cost increase is $C = 1$. Suppose further that the algorithm constructs the learning example given in Figure 3.26(b) and calls the problem solver to find a replacing sequence, with cost limit 4. The solver fails to find a primary-effect justified replacement; however, it may construct an incomplete solution "**go**(room-4,room-3)," which achieves one of **break**'s candidate effects, "not (robot-in room-4)." Then, the *Completer* algorithm selects the other candidate effect, "add (robot-in <to>)," as a new primary effect.

In Section 5.1, we will describe the second selection technique, which guides the choice among the unachieved candidate effects. This heuristic technique is based on the relationship between primary effects and abstraction in problem solving. It enables *Completer* to choose primary effects that enhance the effectiveness of abstraction search. The resulting selections usually correspond to human intuition. Moreover, if we use an automatic abstraction generator (see Chapter 4), then the selected primary effects improve the quality of the resulting abstractions.

Order of processing operators

The learned selection of primary effects may depend on the order of processing operators and inference rules. We tested several ordering heuristics and decided to process operators in increasing order of their costs. We used this processing order in most experiments, and demonstrated that it usually helps to avoid redundant primary effects. Note that, since inference rules have no cost, the learning algorithm processes them before operators.

An intuitive justification is based on the observation that the problem solver mostly uses cheap operators in the construction of replacing sequences. If we may use some operator or inference rule $step_1$ in a primary-effect justified replacement of $step_2$, then the learning algorithm should process $step_1$ before $step_2$, in order to use the newly selected primary effects of $step_1$ in the generation of replacing sequences for $step_2$. Since the algorithm usually uses cheap operators in replacing sequences for more expensive operators, it should begin by processing cheap operators.

If the cost of an operator depends on an instantiation, then the *Completer* algorithm has to determine the range of possible costs, and use this range in the cost comparison. If the system uses the *Matcher* algorithm in generating the initial selection of primary effects, then it gets all possible instantiations of every operator and determines the exact cost ranges.

Otherwise, *Completer* generates learning examples for all operators *before* ordering the operators, and uses the corresponding instantiations to estimate the cost ranges. For every operator $step_{\mathrm{u}}$, it constructs m learning examples, which include m instantiations of $step_{\mathrm{u}}$ (see Figure 3.25). Then, the algorithm determines the cost range of the resulting instantiations, and uses it as an estimate of $step_{\mathrm{u}}$'s cost range.

After the algorithm determines the cost ranges for all operators, it uses these ranges to determine the processing order. We consider $step_1$ cheaper then $step_2$ only if the maximal cost of $step_1$ is smaller than the minimal cost of $step_2$. This cost comparison determines a partial order of operators in the domain.

If two operators have the same cost, or their relative cost depends on specific instantiations, then we use a different heuristic for their ordering: the learning algorithm processes them in increasing order of the number of their candidate effects. That is, if $step_1$ has fewer candidate effects than $step_2$, then the algorithm processes $step_1$ before $step_2$. We also use this heuristic for ordering inference rules, which have no costs.

This heuristic is based on the observation that, if the learning algorithm selects a new primary effect among multiple candidate effects of an operator or inference rule, it may select a redundant effect. The larger the number of candidate effects, the higher the probability of making an inappropriate choice. If the algorithm first selects primary effects of inf_1 and uses them in the construction of replacing sequences for inf_2, it reduces the number of alternatives in selecting a new primary effect of inf_2, thus lowering the probability of choosing a redundant effect.

Generating initial states

When the learning algorithm processes an operator or inference rule, it uses matching initial states to construct learning examples. Therefore, we have to provide a procedure for the generation of initial states, which inputs an uninstantiated operator or inference rule, $step_{\mathrm{u}}$, and constructs a random initial state, based on some probability distribution over the set of all states that satisfy the preconditions of $step_u$. The distribution should correspond to the probabilities of encountering these states during problem solving.

The development of general techniques for generating consistent domain states is an open research area, closely related to the automatic construction of sample problems for various learning algorithms. We have not investigated this area; however, we have provided a procedure that uses previously generated solutions to construct a library of initial states. If the system has access to a collection of past problem-solving episodes, the user may apply this procedure to generate learning examples.

For every available solution, the procedure determines its initial state and all intermediate states, and stores them in the library. When adding a state to the library, we record the operator or inference rule applied to this state in the solution. We then index all states by the applicable operators and rules, and utilize this indexing for fast retrieval of states that satisfy the preconditions of $step_u$.

If the system does not have a library of past solutions for the current domain, then the user has to design a procedure that generates initial states for this domain. We implemented state generators for several domains and used them in most experiments. Alternatively, the user may hand-code a set of initial states for every operator and inference rule. If she provides fewer then m states for some operator or inference rule, the learning algorithm uses all available states and warns the user that the selection may not satisfy the completeness requirements.

3.5.3 Sample complexity

When the *Completer* algorithm learns primary effects of an operator or inference rule $step_u$, it considers m initial states that satisfy the preconditions of $step_u$. The value of m depends on the success-probability parameters, ϵ and δ, and on the number of candidate effects of $step_u$.

To determine the appropriate m value, we use the framework of *probably approximately correct learning*, which is usually abbreviated as *PAC learning.* This framework is a subfield of computational learning theory [Anthony and Biggs, 1992; Drastal *et al.*, 1994 1997; Haussler, 1988; Natarajan, 1991; Carbonell, 1990], aimed at estimating the required number of learning examples. The dependency between m and the values of ϵ and δ is called the *sample complexity* of the learning algorithm.

We define an *approximately correct* selection of primary effects, which ensures that operators and inference rules almost always have replacing sequences, and derive a dependency between m and the probability of learning an approximately correct selection. This result allows the computation of the m value from the user-specified parameters, ϵ and δ. We then discuss the dependency between sample complexity and learning time.

PAC learning

Suppose that we apply the learning algorithm to select primary effects of some uninstantiated operator or inference rule $step_u$, with j candidate effects. The algorithm randomly chooses m states that satisfy the preconditions of $step_u$, generates instantiations of $step_u$ that match the chosen states, and calls a problem solver to find primary-effect justified replacing sequences.

m	number of learning examples for selecting primary effects of $step_u$
s	number of candidate effects of $step_u$ before learning
ϵ_s	error of the PAC learning for $step_u$'s selection of primary effects
δ_s	probability that the error of $step_u$'s selection is larger than ϵ_s
ϵ	maximal allowed probability of failure to solve a randomly chosen problem
δ	maximal allowed probability of generating an inappropriate selection
n	length of an optimal solution to a randomly chosen problem
n_{max}	maximal possible length of an optimal solution

Figure 3.27: Summary of the notation in the sample-complexity analysis.

The selected initial states and instantiations of $step_u$ are *learning examples*. Formally, every example is a pair $(step_i, I)$, where $step_i$ is a full instantiation of $step_u$ and I is an initial state satisfying the preconditions of I. The algorithm selects learning examples from the set of *all* possible examples, using a certain probability distribution over this set.

We assume that the chance of choosing an instantiation $step_i$ and initial state I during the learning process is equal to the chance of generating the same instantiation of $step_u$ and applying it to the same state I during problem solving. This assumption, called the *stationarity assumption* of PAC learning [Valiant, 1984], is essential for analyzing the probability of completeness.

When the problem solver cannot find a replacing sequence, the learner selects one of the candidate effects of $step_u$ as a new primary effect. The resulting selection of primary effects depends on the chosen initial states and instantiations of $step_u$. Since $step_u$ has j candidate effects, it allows 2^j different selections. Every selection is a *hypothesis* of the PAC learning, and the set of 2^j possible selections is the *hypothesis space*.

If the problem solver can find a replacing plan for an instantiated operator $step_i$ and initial state I, within the cost limit $C \cdot cost(step_i)$, then the current selection of primary effects is *consistent* with the learning example $(step_i, I)$. Note that selecting additional primary effects does not violate consistency; therefore, the learned selection is consistent with all m examples chosen by the algorithm; however, it may not be consistent with other possible examples.

The *error* of the PAC learning for $step_u$ is the probability that the learned selection of $step_u$'s primary effects is not consistent with a randomly selected learning example. The selection is *approximately correct* if the error is no larger than a certain small positive value, ϵ_s, where the index "s" stands for $step_u$.

In Figure 3.27, we summarize the notation used in the analysis of the PAC learning. We have already defined the first three symbols, and will introduce the remaining notation in the following derivation of the sample complexity.

Probability of approximate correctness

The *Completer* algorithm randomly chooses m examples from the set of all learning examples. Since the selection of primary effects is consistent with these m examples, we intuitively expect that it is also consistent with most other examples. If the value of m is sufficiently large, the learned selection is likely to be approximately correct.

Blumer *et al.* [1987] formalized this intuition and derived an upper bound for the probability that the results of learning are *not* approximately correct. This bound depends on the number m of chosen examples and the size of the hypothesis space. For the space of 2^j hypotheses, the probability of learning an incorrect hypothesis is at most

$$2^j \cdot (1 - \epsilon_s)^m.$$

The learning results are *probably approximately correct* if this probability is no larger than a certain small value, denoted δ_s:

$$2^j \cdot (1 - \epsilon_s)^m \leq \delta_s. \tag{3.6}$$

This expression is the classical inequality of PAC learning, which enables us to determine the required number m of learning examples. The following condition on the value of m ensures the satisfaction of Inequality 3.6 [Blumer *et al.*, 1987]:

$$m \geq \frac{1}{\epsilon_s} \cdot \ln \frac{2^j}{\delta_s} = \frac{1}{\epsilon_s} \cdot (\ln \frac{1}{\delta_s} + j \cdot \ln 2). \tag{3.7}$$

Thus, if the m value of the operator or inference rule $step_u$ satisfies this inequality, then the learned selection of $step_u$'s primary effects is probably approximately correct.

Number of initial states

We now relate the required number m of learning examples to the user-specified parameters, ϵ and δ, of the *Completer* algorithm (see Section 3.5.1). Recall that ϵ determines the completeness requirement for an appropriate selection of primary effects: it is the maximal allowed probability of failure to solve a randomly selected problem, within the cost increase C, when using the learned primary effects. The δ value is the allowed probability of generating an inappropriate selection, that is, failing to satisfy the completeness requirement.

First, we assume that the learning algorithm has found an approximately correct selection of primary effects for every operator and inference rule, and estimate the probability of failing to solve a random problem. We express the failure probability through ϵ_s and use this result to determine the dependency between ϵ and ϵ_s.

Suppose that the ϵ_s value is the same for all operators and inference rules, a problem solver uses the learned primary effects to solve some randomly chosen problem, and an optimal solution to this problem consists of n steps. If every operator and inference rule $step_i$ in the optimal solution has a primary-effect justified replacing sequence, with a cost no larger than $C \cdot cost(step_i)$, then the problem has a primary-effect justified solution within the cost increase C (see the proof of the completeness condition in Section 3.2.2).

The probability that every step has a replacing sequence is at least $(1 - \epsilon_s)^n$; hence, the probability that the problem has *no* primary-effect justified solution is at most

$$1 - (1 - \epsilon_s)^n \leq n \cdot \epsilon_s.$$

This expression estimates the probability of failure due to the use of primary effects, which must be no larger than the user-specified parameter ϵ:

$$n \cdot \epsilon_s \leq \epsilon. \tag{3.8}$$

Next, we estimate the probability of learning an inappropriate selection of primary effects and derive the dependency between δ and δ_{s}. We suppose that the δ_{s} value is the same for all operators and inference rules, and denote the total number of operators and rules by s.

If the learning algorithm has not found an approximately correct selection of primary effects for some operator, then the overall selection may not satisfy the completeness requirement. The probability that some operator has inappropriate primary effects is at most

$$1-(1-\delta_{\mathrm{s}})^{s} \leq s \cdot \delta_{\mathrm{s}}.$$

This result is an upper bound for the probability of generating an inappropriate selection, which must be no larger than the δ parameter:

$$s \cdot \delta_{\mathrm{s}} \leq \delta. \tag{3.9}$$

We rewrite Inequality 3.8 as $\frac{1}{\epsilon_{\mathrm{s}}} \geq \frac{n}{\epsilon}$ and Inequality 3.9 as $\frac{1}{\delta_{\mathrm{s}}} \geq \frac{s}{\delta}$, and substitute these lower bounds for $\frac{1}{\epsilon_{\mathrm{s}}}$ and $\frac{1}{\delta_{\mathrm{s}}}$ into Inequality 3.6:

$$m \geq \frac{n}{\epsilon} \cdot (\ln \frac{s}{\delta} + j \cdot \ln 2 = \frac{n}{\epsilon} \cdot (\ln \frac{1}{\delta} + \ln s + j \cdot \ln 2). \tag{3.10}$$

The computation of the m value in the *Completer* algorithm is based on Inequality 3.10. Since the optimal-solution length n depends on a specific problem, we have to use its estimated maximal value in computing m. If this length is at most $n_{\max}$ for all possible problems, then the following value of m satisfies Inequality 3.10:

$$m = \left\lceil \frac{n_{\max}}{\epsilon} \cdot (\ln \frac{1}{\delta} + \ln s + j \cdot \ln 2) \right\rceil. \tag{3.11}$$

The learning algorithm prompts the user to estimate the maximal length of an optimal solution and uses Equality 3.11 to determine the number m of learning examples. If the user does not provide the value of $n_{\max}$, then the algorithm uses the default upper bound, which is currently set to 20.

Thus, the number of examples for $step_{\mathrm{u}}$ depends on the user-specified parameters ϵ and δ, as well as on the number j of $step_{\mathrm{u}}$'s candidate effects, the total number s of operators and inference rules in the domain, and the estimated limit $n_{\max}$ of optional-solution lengths. This dependency of m on the success-probability parameters, ϵ and δ, is called the *sample complexity* of the learning algorithm.

Learning time

The learning time of the *Completer* algorithm is proportional to the total number of learning examples, for all operators and inference rules. If we use Expression 3.11 to determine the m value for each operator and rule, then the total number of examples, M, is roughly proportional to the total number of candidate effects:

$$M = \sum_{op} m_{op} + \sum_{inf} m_{inf}$$

$$= \sum_{op} \left\lceil \frac{n_{\max}}{\epsilon} \cdot (\ln \frac{1}{\delta} + \ln s + j_{op} \cdot \ln 2) \right\rceil + \sum_{op} \left\lceil \frac{n_{\max}}{\epsilon} \cdot (\ln \frac{1}{\delta} + \ln s + j_{inf} \cdot \ln 2) \right\rceil$$

$$\approx \frac{n_{\max}}{\epsilon} \cdot (\ln \frac{1}{\delta} + \ln s) + \frac{n_{\max}}{\epsilon} \cdot (\sum_{op} \ln(j_{op} + 1) + \sum_{inf} \ln(j_{inf} + 1)).$$

For example, consider the automatic selection of primary effects in the Robot Domain. After the use of the *Chooser* algorithm, the total number of remaining candidate effects is 6 (see Table 3.2a–c). If we apply the *Completer* algorithm with $\epsilon = \delta = 0.2$ and $n_{\max} = 5$, then the overall number of learning examples is $M = 185$.

When *Completer* processes a learning example, it invokes a problem solver to search for a replacing sequence, which may take significant time. The processing time depends on the domain and maximal allowed cost increase C, as well as on the specific example. For the Lisp implementation of PRODIGY on a Sun 5, the time for processing an example varies from 0.01 to 0.1 seconds, which means that the overall learning time is usually one or two orders of magnitude larger than *Chooser*'s running time.

The choice between applying the *Completer* algorithm and using *Chooser*'s selection depends on the desired trade-off between speed and accuracy of selecting primary effects. If a domain includes large-scale problems or a large number of small problems, then the learned selection significantly reduces search and justifies the learning time. On the other hand, if the domain does not require extensive search, we may be better off without learning.

We have used a conservative worst-case analysis to derive the sample complexity of *Completer*, which has shown the need for a large number of learning examples. On the other hand, experiments have demonstrated that the algorithm usually needs many fewer examples for generating a complete selection. We conjecture that the average-case sample complexity is smaller than the derived worst-case result, and that we can further reduce it by an intelligent selection of learning examples; however, providing a tighter bound for the complexity is an open problem.

3.6 ABTWEAK experiments

We present a series of experiments on the use of automatically selected primary effects in the ABTWEAK system, which explores the search space in breadth-first order. We performed these experiments in collaboration with Yang, to demonstrate that primary effects exponentially reduce search [Fink and Yang, 1995]. In Section 3.7, we describe similar experiments in the PRODIGY system, which uses depth-first search.

The domain language of ABTWEAK is similar to the basic domain language described in Section 2.2.1, which is less powerful than the full PRODIGY language. It does not allow the use of object types, disjunctive and quantified preconditions, conditional effects, and inference rules. We thus used conjunctive goals and operators with conjunctive preconditions in constructing experimental problems.

We have implemented the algorithm for selecting primary effects and used it with the ABTWEAK system, developed by Yang *et al.* [1996]; both the selection algorithm and ABTWEAK have been coded in Lisp. We have run the experiments using Allegro Common Lisp on a Sun 1000 computer. We first describe experiments with artificial domains (Section 3.6.1), and then give the results for an extended version of the robot world and for a manufacturing domain (Section 3.6.2).

3.6.1 Controlled experiments

We consider a family of artificial domains, similar to the domains developed by Barrett and Weld [1994] for evaluating the efficiency of least-commitment systems. These domains have a number of features that can be varied independently, which enables us to perform controlled experiments.

Artificial domains

We define a problem by $s+1$ initial-state literals, $init_0, init_2, ..., init_s$, and a conjunctive goal statement that includes s literals, $goal_0, goal_2, ..., goal_{s-1}$. The variable name "$s$" stands for the "size of a problem," which is one of the controlled features. The domain contains s operators, $op_0, op_2, ..., op_{s-1}$, where every operator op_i has the single precondition $init_{i+1}$ and $k+1$ effects, which include deleting the initial-state literal $init_i$ and adding the goal literals $goal_i, goal_{i+1}, ..., goal_{i+k-1}$. The goal literals are enumerated modulo s; that is, a more rigorous notation for effects of op_i is $goal_{i \bmod m}, goal_{(i+1) \bmod m}, ..., goal_{(i+k-1) \bmod s}$.

For example, suppose that the goal statement includes six literals, that is, $s=6$. If $k=1$, then every operator op_i adds one goal literal, $goal_i$, and the solution has to contain all six operators: "$op_0\, op_1, op_2, op_3, op_4, op_5$." If $k=3$, then every operator adds three literals; in particular op_0 achieves $goal_0$, $goal_1$, and $goal_2$; and op_3 achieves $goal_3$, $goal_4$, and $goal_5$. In this case, an optimal solution comprises two operators: "op_0, op_3."

Controlled features

The artificial domains allow us to vary the following problem features:

Goal size: The *goal size* is the number of goal literals, s. The length of an optimal solution changes in proportion to the number of goal literals.

Effect overlap: The *effect overlap*, k, is the average number of operators achieving the same literal. In terms of the analysis in Section 3.3, we may express the overlap as the ratio of the total number of operator effects, E, and the number of nonstatic literals, N; that is, $k = E/N$.

Cost Variation: The *cost variation* is the statistical *coefficient of variation* of the operator costs, that is, the ratio of the standard deviation of the costs to their mean. Intuitively, it is a measure of the relative difference among operator costs.

We vary these three features in controlled experiments. Although the domains are artificial, they demonstrate some important characteristics of real-world problems: first, if the goal size increases, the length of the optimal solution also increases; second, if the effect overlap increases, then every operator achieves more goal literals and the solution length decreases.

Varying solution lengths

First, we describe experiments that show the dependency of the search reduction on the optimal solution length, that is, on the number of operators in an optimal solution. We plotted this dependency for different values of the effect overlap and cost increase.

We varied the goal size s from 1 to 20 and constructed conjunctive goal statements by randomly permuting the literals $goal_1, goal_2, ..., goal_s$. We used domains with two different values of the effect overlap, 3 and 5. Note that we did not consider an overlap of 1, because then all effects would have to be primary and, hence, the use of primary effects would not affect the search.

We first applied the ABTWEAK system to all problems without using primary effects, and discarded the problems that took more than sixty seconds. Thus, the experimental results are for the problems solved within one minute. For an effect overlap of 3, the optimal-solution lengths of such problems varied from one to seven operators; for an overlap of 5, the optimal solutions were up to four operators.

We then applied the algorithm for selecting primary effects and used the resulting selections in problem solving. We experimented with two different cost-increase values, $C = 2$ and $C = 5$, and two different values of ϵ and δ, which were $\epsilon = \delta = 0.2$ and $\epsilon = \delta = 0.4$. In addition, we considered cost assignments with three different values of the cost variation: 0, 0.4, and 2.4.

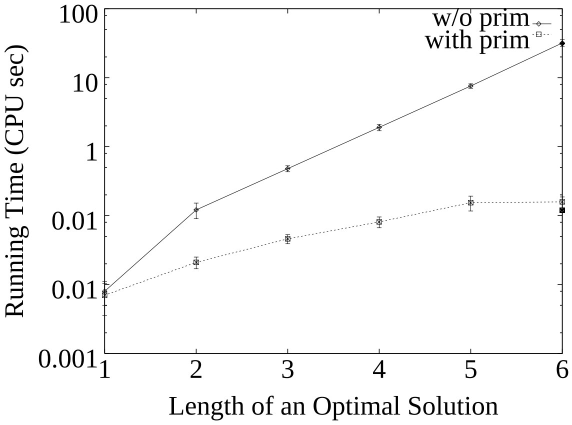
Dependency of the savings on the solution length

The results of the experiments are presented in Figures 3.28 and 3.29: we give the running times of ABTWEAK without primary effects ("w/o prim") and with the use of the learned primary effects ("with prim"). The graphs show the dependency of the search time (vertical axis) on the length of an optimal solution (horizontal axis); note that the time scale is *logarithmic*. Every point in each graph is the average running time for ten different problems, and the vertical bars show the 95% confidence intervals.

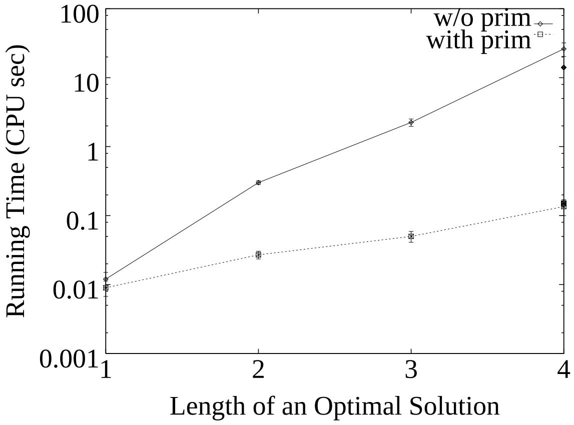
The search with primary effects yielded solutions to *all* problems, within the user-specified cost increase. The time reduction varied depending on the goal size, effect overlap, cost increase, and cost variation; however, it was significant in all cases. The experiments demonstrated that the time savings grow exponentially with an increase in the solution length. These results confirmed the r^n estimate of the search-reduction ratio, where n is the length on an optimal solution (see Section 3.3). The r value in the artificial experiments depended on the effect overlap, varying from 0.3 to 0.6.

Varying effect overlap

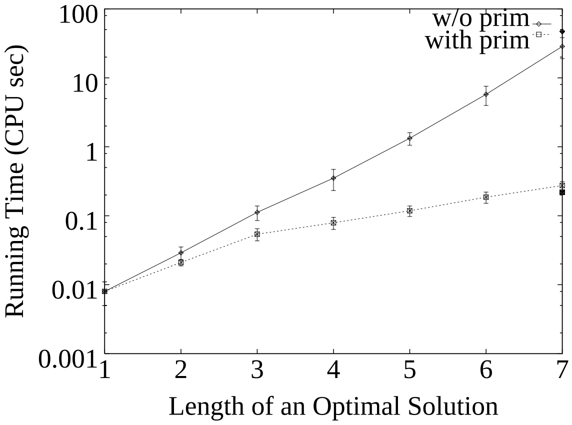
Next, we consider the efficiency improvement for different effect overlaps. Recall that the effect-overlap value is defined as the average number of operators achieving the same literal. We vary this value from 2 to 6, and consider three different cost variations: 0, 0.4, and 2.4. In Figure 3.30, we present the running times of the ABTWEAK system without primary effects ("w/o prim") and with the use of primary effects ("with prim"), for every effect overlap and



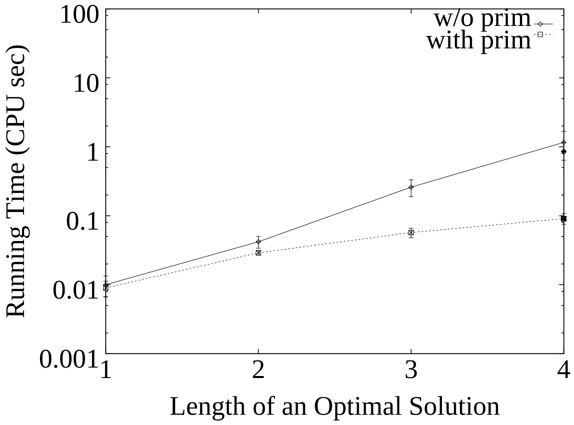
Effect overlap is **3**, cost increase C is **5**, cost variation is **0**, and $\epsilon = \delta = 0.2$.



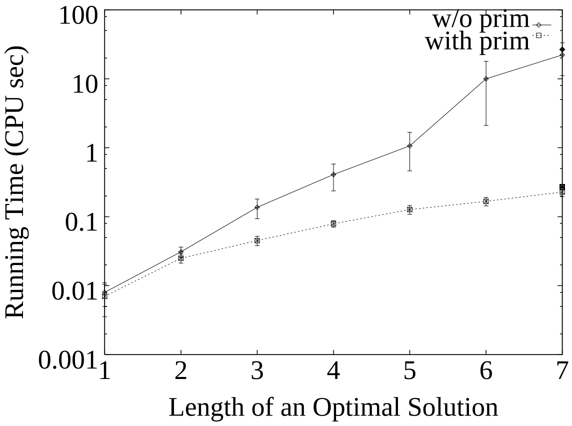
Effect overlap is **5**, cost increase C is **5**, cost variation is **0**, and $\epsilon = \delta = 0.2$.



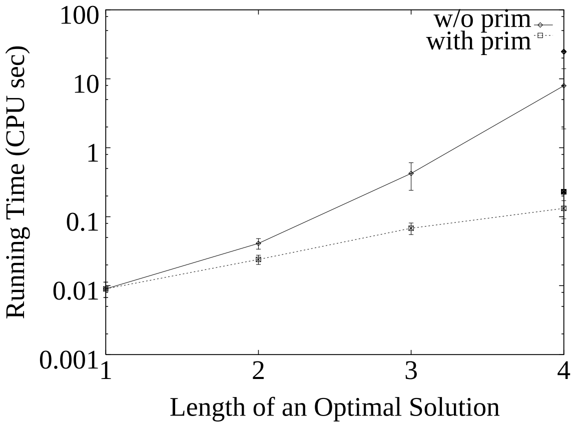
Effect overlap is **3**, cost increase C is **5**, cost variation is **0.4**, and $\epsilon = \delta = 0.2$.



Effect overlap is **5**, cost increase C is **5**, cost variation is **0.4**, and $\epsilon = \delta = 0.2$.

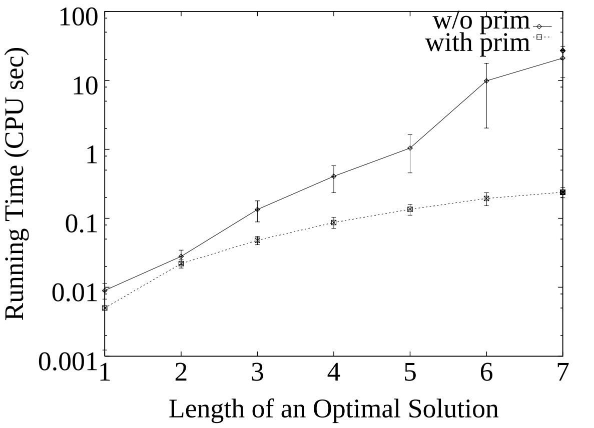


Effect overlap is **3**, cost increase C is **5**, cost variation is **2.4**, and $\epsilon = \delta = 0.2$.

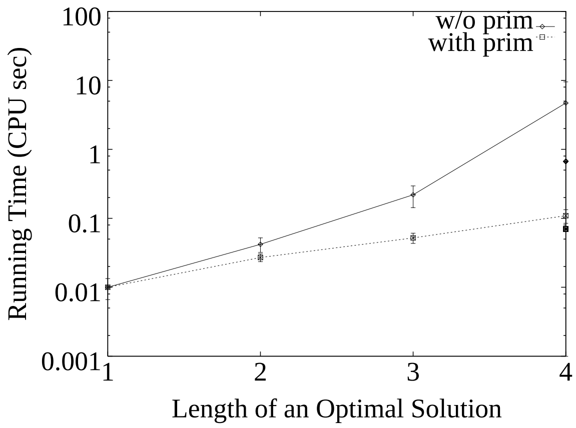


Effect overlap is **5**, cost increase C is **5**, cost variation is **2.4**, and $\epsilon = \delta = 0.2$.

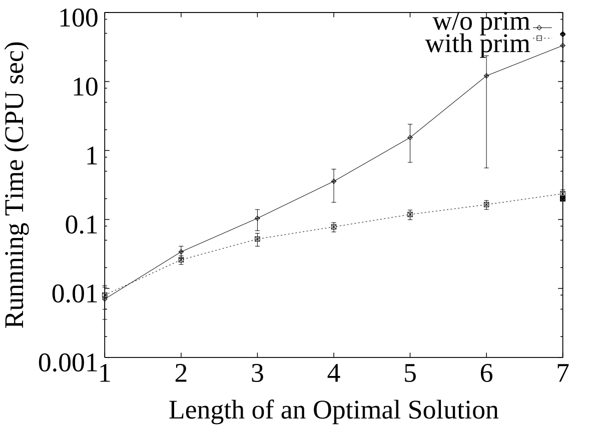
Figure 3.28: Dependency of running time on the length of an optimal solution, for search without and with primary effects in artificial domains. We give results for different effect overlaps (**3** and **5**) and cost variations (**0**, **0.4**, and **2.4**).



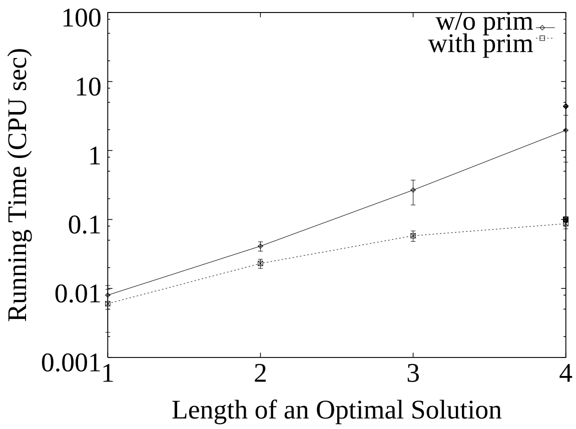
Effect overlap is **3**, cost increase C is **2**, cost variation is **2.4**, and $\epsilon = \delta = 0.2$.



Effect overlap is **5**, cost increase C is **2**, cost variation is **2.4**, and $\epsilon = \delta = 0.2$.



Effect overlap is **3**, cost increase C is **5**, cost variation is **2.4**, and $\epsilon = \delta = 0.4$.



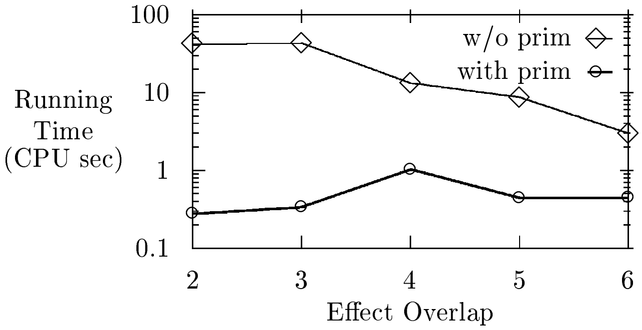
Effect overlap is **5**, cost increase C is **5**, cost variation is **2.4**, and $\epsilon = \delta = 0.4$.

Figure 3.29: Dependency of running time on the optimal-solution length (continued). We consider different values of effect overlap (**3** and **5**) and cost increase C (**2** and **5**).

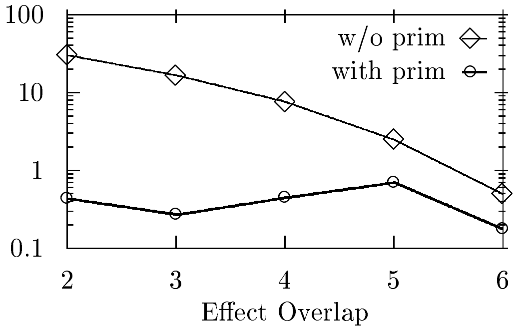
cost variation. The results show that the use of primary effects improves the performance for all effect overlaps, although the time savings are smaller for large overlaps.

3.6.2 Robot world and machine shop

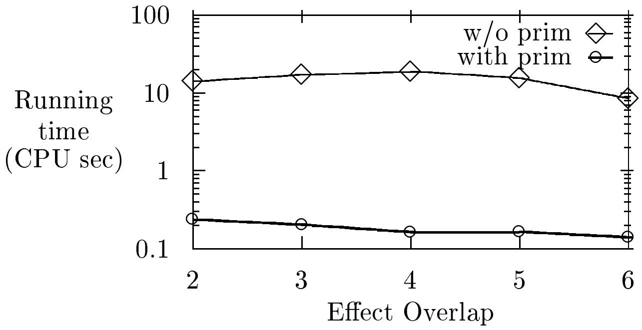
We now illustrate the effectiveness of using primary effects for two other tasks: planning a robot's actions and choosing appropriate operations in a machine shop. The first domain is an extension of the robot-world example. The second domain is a formalization of a motivating example in Section 3.1.1. These experiments also demonstrated exponential improvement, but the r value was closer to 1 than in the artificial domain.



All operators have the same cost;
that is, the cost variation is **0**.



Operators have different costs,
with the cost variation **0.4**.



Operators have different costs,
with the cost variation **2.4**.

Figure 3.30: Dependency of the running time on the effect overlap.

Extended Robot Domain

We consider an extended robot world (see Figure 3.31a), where the robot can move among different locations within a room, go through a door to another room, open and close doors, and climb tables; however, it cannot break through walls.

The domain comprises ten operators, which include two operators for opening and closing doors, four operators for travelling among locations and rooms, with or without a box, and four operators for climbing up and down the table.

In Figure 3.31(b), we give the encoding of two table-climbing operators, where the **climb-up** operator is for getting onto a table without a box, and **carry-up** is for climbing with a box. For convenience, we use the same domain language as in other examples, even though the ABTWEAK syntax for operator description differs from the PRODIGY syntax.

We applied the learning algorithm to chose primary effects, with the cost-increase limit $C = 5$, and $\epsilon = \delta = 0.1$. In Figure 3.31(c), we show the selected primary effects of all operators, as well as their side effects and costs. Observe that the selection matches human intuition: for every box-moving operation, the change in the position of the box is a primary effect, and the change in the robot's position is a side effect.

We applied the problem solver to several problems of different complexity, using the initial state in Figure 3.31(a), with both doors being closed, and randomly generated goal statements. The performance results are given in Table 3.6, which includes the running time and branching factor of search without primary effects ("w/o prim") and with their use ("with prim").

The selected primary effects noticeably improved the efficiency of ABTWEAK. Moreover, search with primary effects yielded optimal solutions to all problems, despite the high cost-increase limit. The r value in these experiments is approximately 0.9, that is, the ratio of search times with and without primary effects is about 0.9^n.

Machining Domain

We next give results for a machining domain, similar to the domain used by Smith and Peot [1992] in their analysis of problem solving with abstraction. The Machining Domain includes a simplified version of cutting, drilling, polishing, and painting operations from the PRODIGY Process-Planning Domain [Gil, 1991].

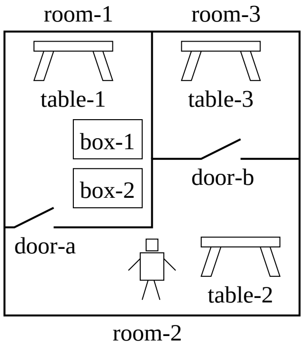
The domain includes eight operators, which encode low-quality and high-quality operations; the production of high-quality parts incurs higher costs. In Figure 3.33, we list the effects and costs of the operators. The order of performing these operations is essential: cutting operators destroy the results of drilling, polishing, and painting; similarly, drilling destroys polishing and painting. The problem solver has to choose between low and high quality, and find the right ordering of operations.

We ran the learning algorithm with $C = 1$ and $\epsilon = \delta = 1$, and it selected the primary effects given in Figure 3.33. The resulting selection prevents the use of costly operations when the quality is not essential, thus forcing the right choice between low and high quality.

Note that the selected primary effects include not only adding but also deleting literals. The deletion effects are redundant, because the problems in this domain never include destructive goals. This example demonstrates that, if we restrict the allowed goals, the completeness condition of Section 3.2.2 may prove too strong. To avoid this problem, the user may pre-select all deletion effects as side effects, thus preventing their promotion to primary.

In Figure 3.32, we summarize the performance results for one hundred randomly generated problems. The graph shows the running times, with and without the use of primary effects, and the 95% confidence intervals. The search with primary effects yielded optimal solutions to all problems and gave a significant time saving, which grew exponentially with the solution length.

The r value for the Machining Domain is about 0.8, that is, the search-time ratio is approximately 0.8^n. Even though r is larger than in the artificial experiments, the search-time ratio is in the same range, from 0.1 to 0.01, because solutions in the Manufacturing Domain are longer than in the artificial domains.



(a) Map of the robot world.

climb-up(<table>)
Pre: (robot-at <table>)
(robot-on-floor)
Eff: del (robot-on-floor)
add (robot-on <table>)
Cost: 1

carry-up(<box>, <table>)
Pre: (robot-at <table>)
(robot-on-floor)
(at <box> <table>)
(on-floor <box>)
Eff: del (robot-on-floor)
add (robot-on <table>)
del (on-floor <box>)
add (on <box> <table>)
Cost: 4

(b) Table-climbing operators.

open(<door>)
Prim: del (closed <door>)
add (open <door>)
Cost: 1

close(<door>)
Prim: del (open <door>)
add (closed <door>)
Cost: 1

climb-up(<table>)
Prim: del (robot-on-floor)
add (robot-on <table>)
Cost: 1

climb-down(<table>)
Prim: del (robot-on <table>)
add (robot-on-floor)
Cost: 1

carry-up(<box>, <table>)
Prim: del (on-floor <box>)
add (on <box> <table>)
Side: del (robot-on-floor)
add (robot-on <table>)
Cost: 4

carry-down(<box>, <table>)
Prim: del (on <box> <table>)
add (on-floor <box>)
Side: del (robot-on <table>)
add (robot-on-floor)
Cost: 4

go-within-room
(<from-loc> <to-loc> <room>)
Prim: del (robot-at <from-loc>)
add (robot-at <to-loc>)
Cost: 1

go-thru-door
(<from-room> <to-room> <door>)
Prim: del (robot-in <from-room>)
add (robot-in <to-room>)
Cost: 2

carry-within-room
(<box>, <from-loc>, <to-loc>, <room>)
Prim: del (at <box> <from-loc>)
add (at <box> <to-loc>)
Side: del (robot-at <from-loc>)
add (robot-at <to-loc>)
Cost: 2

carry-thru-door
(<box>, <from-room>, <to-room>, <door>)
Prim: del (in <box> <from-room>)
add (in <box> <to-room>)
Side: del (robot-in <from-room>)
add (robot-in <to-room>)
Cost: 4

(c) Effects and costs of all operators.

Figure 3.31: Extended Robot Domain.

#	Goal Statement	Optimal Solution		Run Time (CPU sec)		Mean Branching Factor	
		length	cost	w/o prim	with prim	w/o prim	with prim
1	(robot-on table-2)	1	1	0.03	0.02	1.5	1.0
2	(open door-a)	2	2	0.09	0.09	2.0	1.4
3	(robot-in room-1)	3	4	0.15	0.13	1.7	1.3
4	(on box-1 table-1)	5	9	0.55	0.36	2.3	1.3
5	(robot-on table-3)	5	6	0.57	0.37	2.0	1.3
6	(and (robot-on table-1) (closed door-a))	6	7	2.25	1.16	2.1	1.4
7	(and (on box-2 table-1) (open door-b))	7	11	4.65	2.52	2.3	1.4
8	(at box-2 table-2)	7	13	6.20	2.76	2.1	1.3
9	(on box-2 table-2)	8	17	15.09	4.41	2.1	1.4

Table 3.6: Performance of ABTWEAK in the Robot Domain, without and with primary effects.

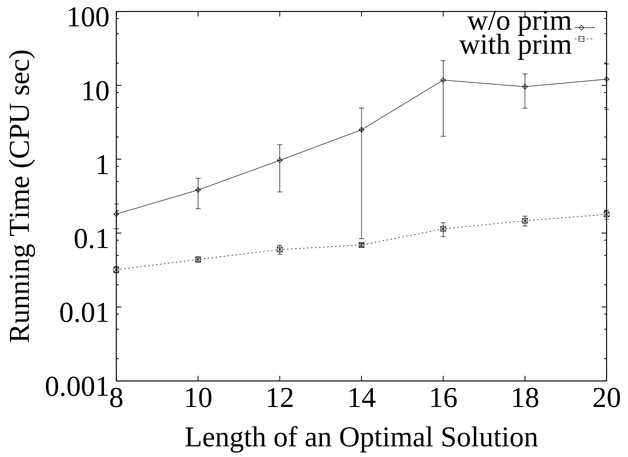


Figure 3.32: ABTWEAK performance in the Machining Domain, without and with primary effects.

3.7 PRODIGY experiments

We next give empirical results on utilizing primary effects, selected by the *Chooser* and *Completer* algorithms, in the PRODIGY system. Unlike ABTWEAK, it performs a depth-first exploration of the available search space, and usually finds suboptimal solutions. We may force the search for a near-optimal solution by setting a cost bound, which prevents the construction of solutions with larger costs (see Section 2.4.2).

Since cost bounds limit the search depth, they affect not only the solution quality but also the time of problem solving. Their effect depends on the domain, as well as on specific problem instances, and may vary from a significant efficiency improvement to an exponential increase in the search time. We give results for problem solving with different cost bounds, as well as without any bound, on a Sun 5 machine.

The results demonstrate that automatically selected primary effects improve the efficiency of the PRODIGY depth-first search, both with and without cost bounds. Moreover, they sometimes guide the system to finding lower-cost solutions. We describe experiments in two domains exported from ABTWEAK (Section 3.7.1) and in two traditional PRODIGY domains (Section 3.7.2), and then conclude with a discussion of the results (Section 3.7.3).

3.7.1 Domains from ABTWEAK

We give results of applying the PRODIGY search engine to problems in two ABTWEAK domains, described in Section 3.6.2, and show that primary effects improve the performance of PRODIGY in both domains.

Extended Robot Domain

We applied PRODIGY to the nine problems given in Table 3.6, without and with primary effects. For every problem, we ran the PRODIGY search algorithm without a cost bound, and then with two different cost bounds. The first cost bound was twice the cost of the

cut-roughly(<part>)
Prim: add (cut <part>)
del (finely-cut <part>)
del (drilled <part>)
Side: del (finely-drilled <part>)
del (polished <part>)
del (finely-polished <part>)
del (painted <part>)
del (finely-painted <part>)
Cost: 1

cut-finely(<part>)
Prim: add (finely-cut <part>)
Side: add (cut <part>)
del (drilled <part>)
del (finely-drilled <part>)
del (polished <part>)
del (finely-polished <part>)
del (painted <part>)
del (finely-painted <part>)
Cost: 2

drill-roughly(<part>)
Prim: add (drilled <part>)
del (finely-drilled <part>)
del (polished <part>)
Side: del (finely-polished <part>)
del (painted <part>)
del (finely-painted <part>)
Cost: 1

drill-finely(<part>)
Prim: add (finely-drilled <part>)
Side: add (drilled <part>)
del (polished <part>)
del (finely-polished <part>)
del (painted <part>)
del (finely-painted <part>)
Cost: 2

polish-roughly(<part>)
Prim: add (polished <part>)
del (finely-polished <part>)
del (painted <part>)
Side: del (finely-painted <part>)
Cost: 1

polish-finely(<part>)
Prim: add (finely-polished <part>)
Side: add (polished <part>)
del (painted <part>)
del (finely-painted <part>)
Cost: 2

paint-roughly(<part>)
Prim: add (painted <part>)
del (finely-painted <part>)
Side: del (polished <part>)
del (finely-polished <part>)
Cost: 1

paint-finely(<part>)
Prim: add (finely-painted <part>)
Side: add (painted <part>)
del (polished <part>)
del (finely-polished <part>)
Cost: 2

Figure 3.33: Effects and costs of operators in the Machining Domain.

#	No Cost Bound		Loose Bound		Tight Bound	
	w/o prim	with prim	w/o prim	with prim	w/o prim	with prim
1	62.71	0.04	0.06	0.04	0.05	0.04
2	142.26	0.06	3.55	0.06	0.23	0.06
3	> 1800.00	0.08	219.43	0.09	1.14	0.09
4	> 1800.00	0.16	> 1800.00	0.17	41.32	0.16
5	> 1800.00	0.15	> 1800.00	0.15	51.98	0.14
6	> 1800.00	6.54	> 1800.00	6.62	173.83	0.39
7	> 1800.00	2.67	> 1800.00	2.49	> 1800.00	3.43
8	> 1800.00	> 1800.00	> 1800.00	> 1800.00	> 1800.00	> 1800.00
9	> 1800.00	> 1800.00	> 1800.00	> 1800.00	> 1800.00	> 1800.00

Table 3.7: Performance of PRODIGY in the Extended Robot Domain. We give running times in seconds, for problem solving without primary effects ("w/o prim") and with their use ("with prim"). For every problem, we first ran the search algorithm without a cost bound, and then with two different bounds. The "loose" cost bound was computed as twice the cost of the optimal solution, whereas the "tight" bound equaled the optimal cost.

optimal solution; we call it a *loose bound.* The second bound, called *tight,* was exactly equal to the optimal cost. We *manually* determined the optimal cost for each problem, and set the appropriate bounds. When the system did not find a solution within 1800 seconds, we interrupted the search.

We give the running times in Table 3.7, which shows that primary effects drastically improve the efficiency of the PRODIGY depth-first search: the time-saving factor in most cases is greater than 500. The r value varies from less than 0.1 in the experiments without a cost bound to approximately 0.4 in problem solving with the optimal-cost bounds.

Search with primary effects yielded optimal solutions to problems 1 through 6, regardless of the cost bound. When solving problem 7 without the tight bound, PRODIGY produced a solution that is one step longer than optimal, with a cost of 12 (versus the optimal cost of 11). The system did not find any solutions to the last two problems, 8 and 9, within the 1800-second time limit.

Machining Domain

We next demonstrate the utility of primary effects for PRODIGY search in the Machining Domain (see Figure 3.33). We have applied the system to a hundred problems, with a 600-second time limit for each problem.

In Figure 3.34, we summarize the results of search without a cost bound. Specifically, we give the running time and solution quality, for problem solving without primary effects (solid lines) and with the use of primary effects (dashed lines), and use vertical bars to mark 95% confidence intervals. The dotted line in Figure 3.34(a) shows the mean costs of optimal solutions.

The system successfully solved all problems within the time limit, and primary effects did *not* improve the efficiency; however, they enabled the search algorithm to find better solutions. The cost-reduction factor ranged from 1.2 to 1.6, with the mean at 1.37.

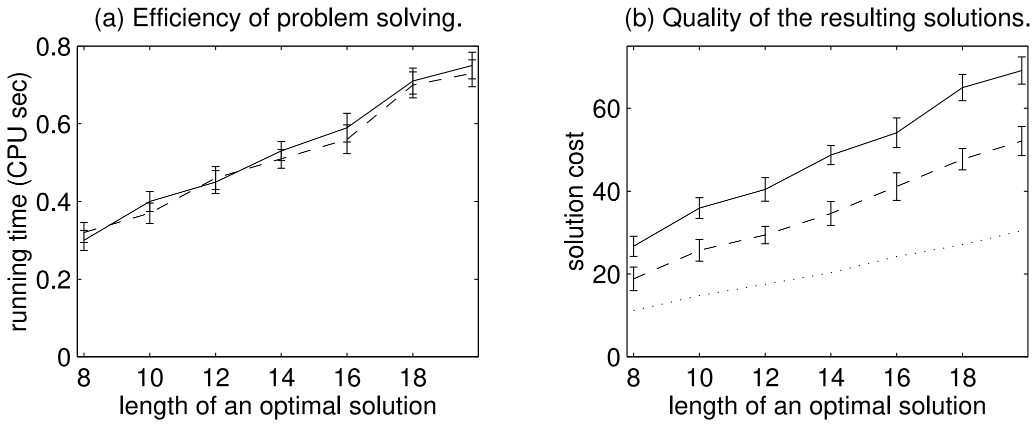


Figure 3.34: PRODIGY performance in the Machining Domain, *without a cost bound.* We show the results of problem solving without primary effects (solid lines) and with their use (dashed lines), as well as the mean cost of optimal solutions (dotted line).

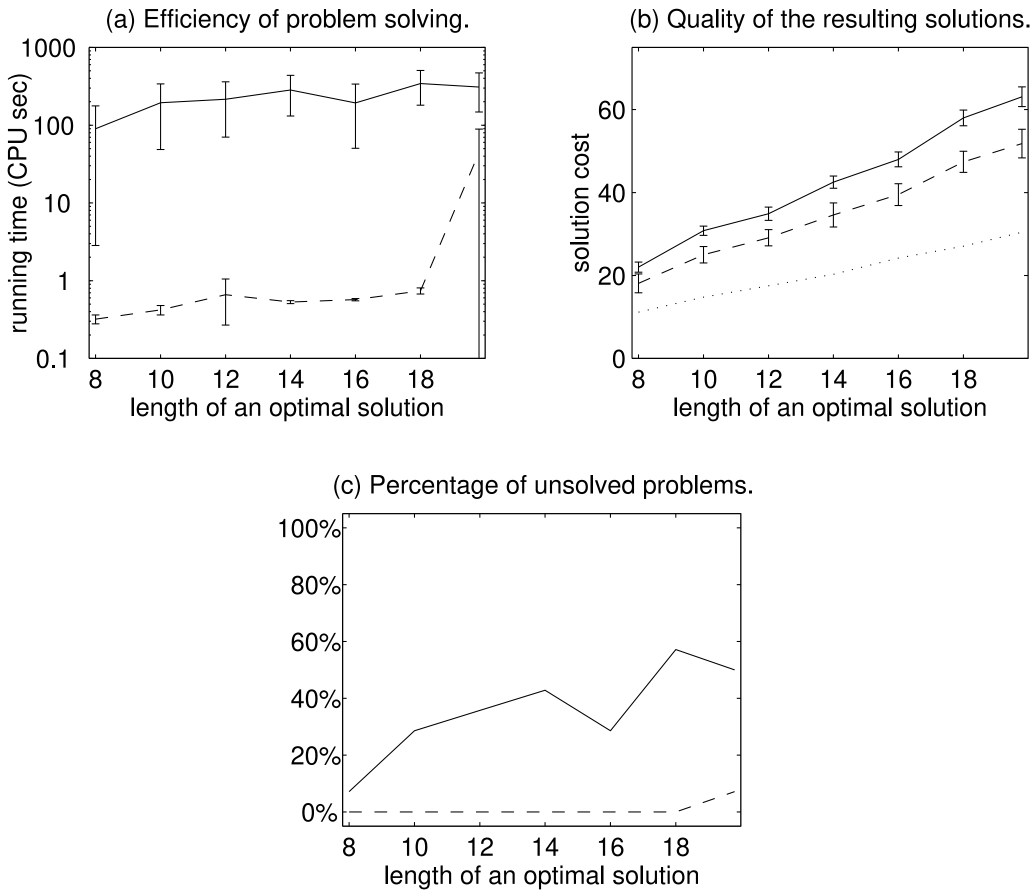


Figure 3.35: PRODIGY performance in the Machining Domain, without primary effects (solid lines) and with the use of primary effects (dashed lines). For every problem, we use a *loose cost bound,* equal to the doubled cost of the optimal solution.

In Figure 3.35, we present the results of problem solving with loose cost bounds. For every problem, the bound was set to the doubled cost of the optimal solution. We computed the optimal costs prior to this experiment, using a domain-specific procedure that exhaustively searched for the optimal solution to each problem. The cost bounds forced PRODIGY to produce better solutions; however, they significantly increased the search complexity, and the system failed to solve some problems within the time bound.

We give the mean running times in Figure 3.35(a), where the time scale is *logarithmic.* We use *all* running times in constructing this graph, including the time spent on the unsolved problems. For most problems, the use of primary effects reduces the search time by more than two orders of magnitude; the r value is about 0.6.

In Figure 3.35(b), we show the mean solutions costs, for the problems solved within the time limit. Primary effects reduce the cost by a factor of 1.1 to 1.3, with the mean at 1.22. Finally, we give the percentage of unsolved problems in Figure 3.35(c).

We also applied the PRODIGY search algorithm with tight cost bounds to the hundred machining problems, thus forcing it to search for optimal solutions; however, the system did not find an optimal solution to any of the problems within the allowed time limit.

3.7.2 Sokoban puzzle and STRIPS world

We next demonstrate the utility of automatically selected primary effects in two other domains, Sokoban puzzle and extended STRIPS world, which have long served as a benchmark for learning and search techniques in the PRODIGY system.

Sokoban domain

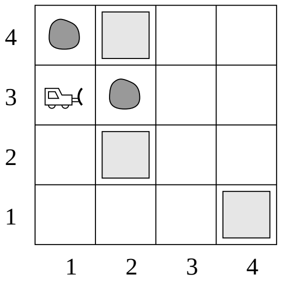
The Sokoban puzzle is a one-player game, apparently invented in 1982 by Hiroyuki Imabayashi, the president of Thinking Rabbit Inc. (Japan). Janghanns and Schaefer [1998; 1999a, 1999b] have recently explored search techniques for Sokoban problems, and implemented a collection of heuristics and learning algorithms designed specifically for this puzzle.

Since PRODIGY is a general-purpose system, which is not fine-tuned for Sokoban, it is much less effective than the specialized algorithms by Janghanns and Schaefer. Even though $\mathcal{S}$HAPER's description changers improve the efficiency of PRODIGY search, they cannot compete with advanced hand-coded heuristics,

The PRODIGY version of Sokoban consists of a rectangular grid, obstacles that block some squares of the grid, a bulldozer that drives among the other squares, and pieces of rock. In Figure 3.36(a), we show an example state of this world and give its encoding in the PRODIGY domain language.

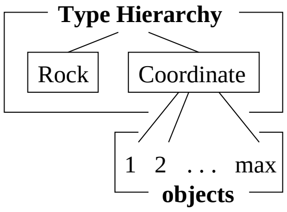
The bulldozer occupies one square of the grid and can drive to any of the adjacent empty squares (see Figure 3.36d); however, it *cannot* enter a square with an obstacle or rock. If the bulldozer is next to a rock, and the square on the other side of the rock is empty, then the dozer may push the rock to this empty square, as shown in Figure 3.36(e). The goal of the puzzle is to deliver the rocks to certain squares.

The Sokoban Domain in the PRODIGY system includes two object types, given in Figure 3.36(b), and eight operators, listed in Figure 3.36(c). The first four operators are for



(blocked 2 3)
(blocked 2 4)
(blocked 4 1)
(dozer-at 1 3)
(at rock-a 1 4)
(at rock-b 2 3)

(a) Example of a world state.

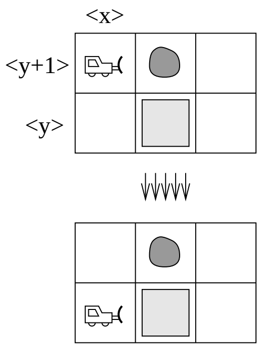


(b) Types of objects.

drive-down(<x>,<y>)
drive-up(<x>,<y>)
drive-left(<x>,<y>)
drive-right(<x>,<y>)

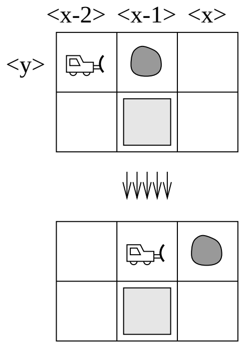
move-down(<rock>,<x>,<y>)
move-up(<rock>,<x>,<y>)
move-left(<rock>,<x>,<y>)
move-right(<rock>,<x>,<y>)

(c) List of operators.



drive-down(<x>,<y>)
<x>: type Coordinate
<y>: type Coordinate
no-rock(<x>,<y>)
<y+1>: type Coordinate
increment(<y>)
Pre: not (blocked <x> <y>)
not (blocked <x> <y+1>)
(dozer-at <x> <y+1>)
Eff: del (dozer-at <x> <y+1>)
add (dozer-at <x> <y>)
Cost: 1

(d) Operator for changing the bulldozer's position.



move-right(<rock>,<x>,<y>)
<x>: type Coordinate
<y>: type Coordinate
no-rock(<x>,<y>)
<x-1>: type Coordinate
decrement(<x>)
<x-2>: type Coordinate
decrement(<x-1>)
Pre: not (blocked <x> <y>)
not (blocked <x-1> <y>)
not (blocked <x-2> <y>)
(ball-at <x-1> <y>)
(dozer-at <x-2> <y>)
Eff: del (at <rock> <x-1> <y>)
add (at <rock> <x> <y>)
del (dozer-at <x-2> <y>)
add (dozer-at <x-1> <y>)
Cost: 2

(e) Operator for moving a rock.

Test Function

no-rock *(<x>,<y>)*
If there is some rock at *(<x>,<y>)*
in the current state of the world,
then return False;
else, return True.

Generator Functions

decrement *(<coordinate>)*
If <*coordinate*> = 0,
then return the empty set;
else, return { <*coordinate*> - 1 }.

increment *(<coordinate>)*
If <*coordinate*> = max,
then return the empty set;
else, return { <*coordinate*> + 1 }.

(f) Functions in the encoding of operators.

Figure 3.36: Sokoban puzzle and its encoding in the PRODIGY domain language.

changing the dozer's position, whereas the other four are for moving rocks. We give the full encoding of two operators, **drive-down** and **move-right**, in Figures 3.36(d) and 3.36(e). Note that the domain includes two generator functions, *decrement* and *increment,* for determining the coordinates of adjacent squares.

The Sokoban puzzle has proved difficult for the PRODIGY system, as well as for other general-purpose problem solvers. If the user does not provide a specialized set of control rules, then PRODIGY can solve only very simple problems in this domain. We demonstrate that primary effects enhance the system's ability to solve Sokoban problems. In Section 4.4, we will show results of utilizing these primary effects in the generation of an abstraction hierarchy for the Sokoban domain.

In Figure 3.37, we give the automatically selected primary effects of all eight operators, as well as their side effects. To test the effectiveness of this selection, we applied PRODIGY to 320 problems, with four different grid sizes: 4×4, 6×6, 8×8, and 10×10.

In Figure 3.38, we give the results of problem solving without primary effects (solid lines) and with their use (dashed lines), for each of the four grid sizes. The graphs show the percentage of problems solved by different time bounds, from 1 to 60 seconds. For example, if we used the 10-second time limit in solving 4×4-grid problems without primary effects, then PRODIGY would solve 8% of them (see the solid line in Figure 3.38a). Similarly, if we used the 10-second bound in solving these problems with primary effects, then the system would solve 68% (see the dashed line in the same graph).

When conducting these experiments, we used a 600-second time limit for every problem; however, we do *not* plot the results for bounds larger that 60 seconds, because further increase of the allowed time has little effect on the percentage of solved problems. In particular, if we raise the time bound from 60 to 600 seconds, then PRODIGY solves only three more problems, out of 320.

The results confirm that the selected primary effects improve the efficiency of solving Sokoban problems. When the system utilizes primary effects, it solves five times more problems. We have also compared the search time with and without primary effects, and found that primary effects significantly reduce search for most problems. The time-saving factor varies from 1 (no time reduction) to more than 100.

We did *not* limit the solution costs in these experiments, and the system produced suboptimal solutions to almost all problems. For most problems, the resulting costs were larger than the doubled costs of the optimal solutions. The use of primary effects neither improved nor hurt the solution quality. We tried to improve the quality by setting cost bounds, but they caused failures on almost all problems, both with and without primary effects.

Extended STRIPS domain

The STRIPS Domain is a large-scale robot world, designed by Fikes and Nilsson [1971; 1993] during their work on the STRIPS system. Later, PRODIGY researchers designed an extended version of this domain and used it for the evaluation of several learning and search techniques [Minton, 1988; Knoblock, 1993; Veloso, 1994].

Even though the STRIPS world and the ABTWEAK Robot Domain (see Figure 3.31) are based on similar settings, the encoding of the Extended STRIPS Domain differs from that

drive-down(<x>,<y>)
Prim: del (dozer-at <x> <y+1>)
add (dozer-at <x> <y>)
Cost: 1

drive-up(<x>,<y>)
Prim: del (dozer-at <x> <y-1>)
add (dozer-at <x> <y>)
Cost: 1

drive-left(<x>,<y>)
Prim: del (dozer-at <x+1> <y>)
add (dozer-at <x> <y>)
Cost: 1

drive-right(<x>,<y>)
Prim: del (dozer-at <x-1> <y>)
add (dozer-at <x> <y>)
Cost: 1

move-down(<rock>,<x>,<y>)
Prim: del (at <rock> <x> <y+1>)
add (at <rock> <x> <y>)
Side: del (dozer-at <x> <y+2>)
add (dozer-at <x> <y+1>)
Cost: 2

move-up(<rock>,<x>,<y>)
Prim: del (at <rock> <x> <y-1>)
add (at <rock> <x> <y>)
Side: del (dozer-at <x> <y-2>)
add (dozer-at <x> <y-1>)
Cost: 2

move-left(<rock>,<x>,<y>)
Prim: del (at <rock> <x+1> <y>)
add (at <rock> <x> <y>)
Side: del (dozer-at <x+2> <y>)
add (dozer-at <x+1> <y>)
Cost: 2

move-right(<rock>,<x>,<y>)
Prim: del (at <rock> <x-1> <y>)
add (at <rock> <x> <y>)
Side: del (dozer-at <x-2> <y>)
add (dozer-at <x-1> <y>)
Cost: 2

Figure 3.37: Effects and costs of operators in the Sokoban Domain.

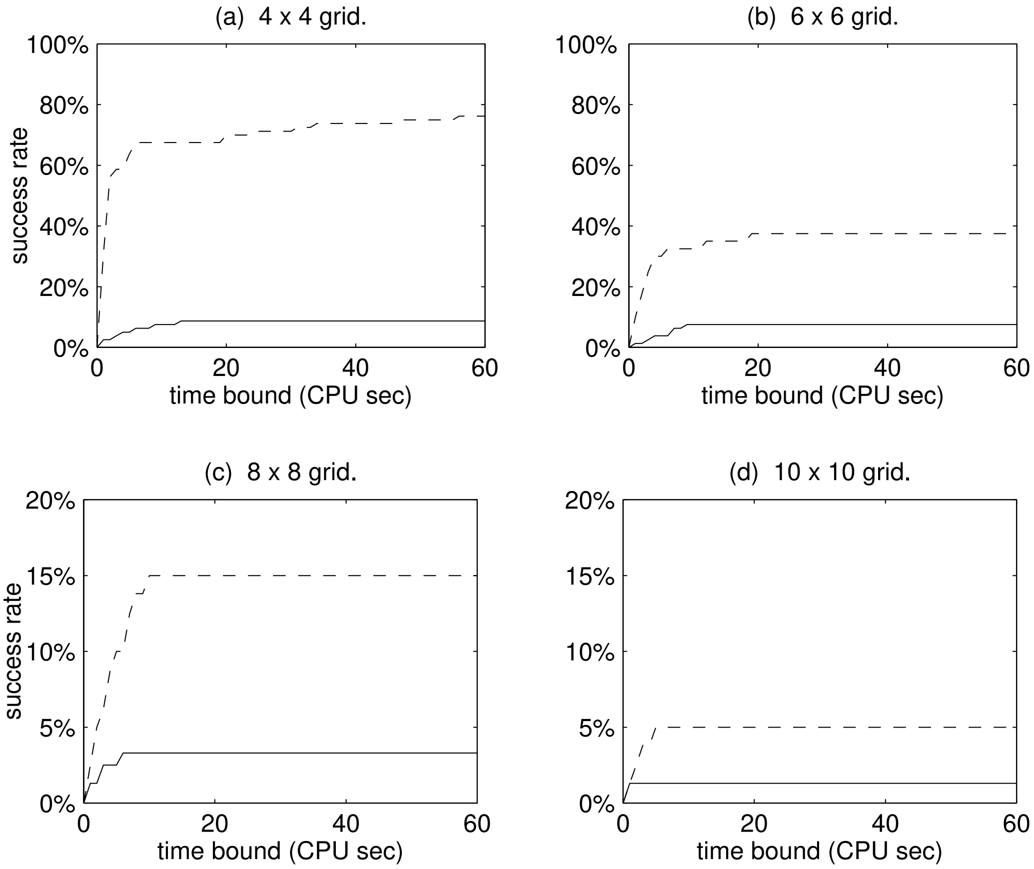


Figure 3.38: PRODIGY performance in the Sokoban Domain, without primary effects (solid lines) and with the use of primary effects (dashed lines). We show the percentage of problems solved by different time bounds, from 1 to 60 seconds. We did not limit solution costs in Sokoban experiments, and the system produced suboptimal solutions to most problems.

of the ABTWEAK domain, which causes different behavior of most learning and search algorithms. In addition, the STRIPS domain describes a richer setting, with more object types and operations, than its ABTWEAK analog.

We give the object types, predicates, and operators of the Extended STRIPS Domain in Figure 3.39, show an example of a domain state in Figure 3.40, and list effects and costs of all operators in Figures 3.41 and 3.42. The domain includes ten types of objects, twelve predicates, and twenty-three operators, some of which have universally quantified effects. The robot may travel among rooms, open and close doors, lock and unlock doors by appropriate keys, pick up and put down small things, and push large things.

We evaluated the system's performance on a hundred problems instainces, which were composed from hand-coded initial states, including the state given in Figure 3.40, and randomly generated goal statements. For every problem, we ran the search algorithm three times: without a cost bound, with the loose bound, and with the tight bound. Recall that the loose cost bound is twice the cost of the optimal solution, whereas the tight bound is equal to the optimal cost; we used a domain-specific procedure to find the optimal costs.

First, we applied PRODIGY to the hundred test problems without primary effects, using the 600-second time limit for every problem. The system solved only two problems in the experiments with the tight cost bounds, and no problems at all without the tight bounds.

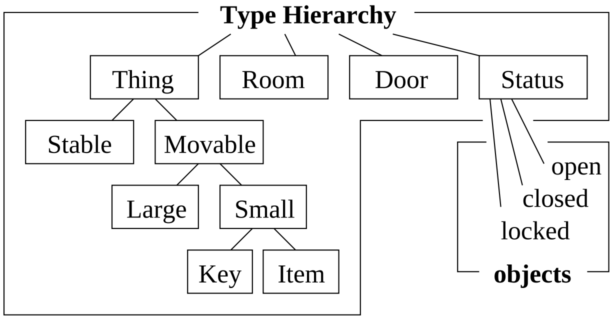
Then, we tested PRODIGY with the automatically selected primary effects. We show the selection of effects in Figures 3.41 and 3.42, and summarize the test results in Figure 3.43. The system solved seventy-eight problems in the experiments without cost bounds (see the solid line in Figure 3.43), seventy-five problems with the loose bounds (dashed line), and forty-one problems with the tight bounds (dotted line). Moreover, most of the solved problems took less than a second; thus, primary effects improved the efficiency by at least three orders of magnitude.

When PRODIGY runs with the tight cost bounds, it produces optimal solutions to all solved problems. On the other hand, search without the tight bounds usually yields costlier solutions. In Figure 3.44, we plot the lengths of the solutions to STRIPS problems, generated without the tight cost bounds, and compare them with the lengths of the optimal solutions. Since the constructed solutions are longer than the optimal ones, the plotted dots are above the diagonal. In Figure 3.45, we give a similar comparison of the solution costs. For most problems, the lengths and costs of the constructed solutions are within a factor of 1.7 from optimal.

Finally, we compare the quality of solutions found without cost bounds and that of solutions constructed with the loose cost bounds (see Figure 3.46). This comparison is based on the seventy-five problems that were solved in both cases. For two of these problems, the search with the loose bounds yielded slightly better solutions. For the other problems, the use of these bounds did not affect the quality.

3.7.3 Summary of experimental results

The empirical results have demonstrated that the automatically selected primary effects improve the performance of backward-chaining algorithms. When we use this speed-up technique with ABTWEAK's breadth-first algorithm, it reduces the branching factor of search



(a) Types of objects.

(fits <key> <door>)	(holding <small>)
(connects <door> <room>)	(arm-empty)
(status <door> <status>)	
	(in <thing> <room>)
(robot-in <room>)	(next-to <thing> <other-thing>)
(robot-at <door>)	(next-to <thing> <door>)
(robot-at <thing>)	(next-to <door> <thing>)

(b) Predicates in the domain encoding.

open(<door>)
close(<door>) } opening and closing a door

lock(<door>)
unlock(<door>) } locking and unlocking a door, with an appropriate key

go-to-door(<door>, <room>) going to a door, within a room

go-to-stable(<stable>, <room>)
go-to-large(<large>, <room>)
go-to-small(<small>, <room>) } going to a stable, large, or small thing, within a room

go-aside(<room>) going to a new location, where there are no things

pick-up(<small>, <room>) picking up a small thing from the floor

put-near-door(<small>, <door>, <room>) putting a small thing near a door

put-near-stable(<small>, <stable>, <room>)
put-near-large(<small>, <large>, <room>)
put-near-small(<small>, <other-small>, <room>) } putting a small thing near a stable, large, or another small thing

put-aside(<small>, <room>) putting a small thing in a new location, away from other things

move-aside(<small>, <room>) moving a small thing away from the robot's current location

push-to-door(<large>, <door>, <room>) pushing a large object to a door, within a room

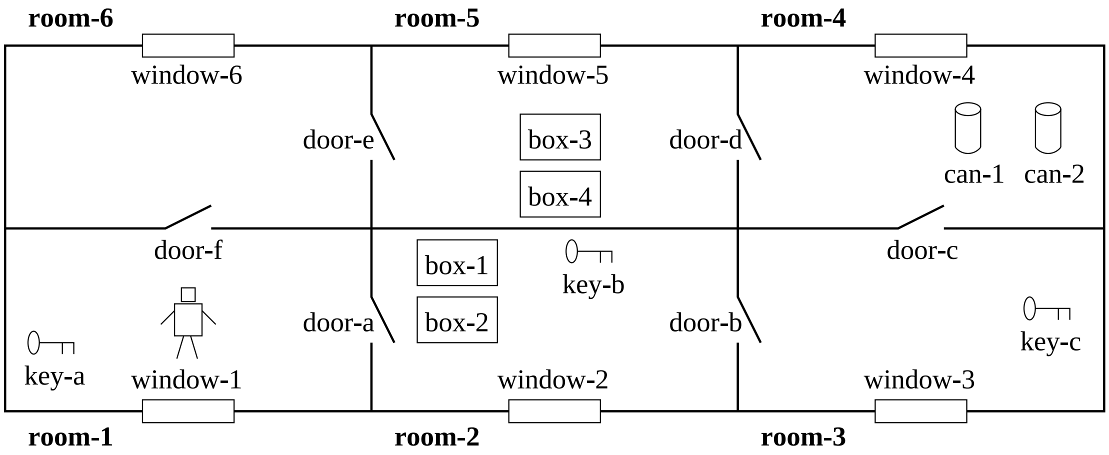
push-to-stable(<large>, <stable>, <room>)
push-to-large(<large>, <other-large>, <room>)
push-to-small(<large>, <small>, <room>) } pushing a large thing to a stable, another large, or small thing, within a room

push-aside(<large>, <room>) pushing a large thing to a new location, away from all other things

go-thru-door(<door>, <from-room>, <to-room>)
push-thru-door(<large>, <door>, <from-room>, <to-room>) } going and pushing a large thing through a door to another room

(c) Operators.

Figure 3.39: Extended STRIPS Domain in the PRODIGY system.



(a) Map of the world.

room-1, room-2, room-3, room-4, room-5, room-6: type Room
door-a, door-b, door-c, door-d, door-e, door-f: type Door
open, closed, locked: type Status
key-a, key-b, key-c: type Key
window-1, wondow-2, wondow-3, window-4, window-5, window-6: type Stable
box-1, box-2, box-3, box-4: type Large
can-1, can-2: type Item

(b) Set of objects.

(connects door-a room-1)	(connects door-d room-4)	(status door-a locked)	(fits key-a room-a)
(connects door-a room-2)	(connects door-d room-5)	(status door-b closed)	(fits key-b room-b)
(connects door-b room-2)	(connects door-e room-5)	(status door-c closed)	(fits key-c room-c)
(connects door-b room-3)	(connects door-e room-6)	(status door-d closed)	(in key-a room-1)
(connects door-c room-3)	(connects door-f room-6)	(status door-e closed)	(in key-b room-2)
(connects door-c room-4)	(connects door-f room-1)	(status door-f closed)	(in key-c room-3)

(in window-1 room-1)	(in box-1 room-2)	(in can-1 room-4)
(in window-2 room-2)	(in box-2 room-2)	(in can-2 room-4)
(in window-3 room-3)	(next-to box-1 box-2)	(next-to can-1 can-2)
(in window-4 room-4)	(next-to box-2 box-1)	(next-to can-2 can-1)
(in window-5 room-5)		
(in window-6 room-6)	(in box-3 room-5)	(robot-in room-1)
	(in box-4 room-5)	(robot-at window-1)
	(next-to box-3 box-4)	(arm-empty)
	(next-to box-4 box-3)	

(c) Encoding of the world state.

Figure 3.40: Example of an initial state in the Extended STRIPS Domain.

open(<door>)
Prim: del (status <door> closed)
add (status <door> open)
Cost: 1

close(<door>)
Prim: del (status <door> open)
add (status <door> closed)
Cost: 1

lock(<door>)
Prim: add (status <door> locked)
Side: del (status <door> closed)
Cost: 2

unlock(<door>)
Prim: del (status <door> locked)
add (status <door> closed)
Cost: 2

go-to-door(<door>, <room>)
Prim: add (robot-at <door>)
Side: (forall <other> of type (or Thing Door)
del (robot-at <other>))
Cost: 2

go-to-stable(<stable>, <room>)
Prim: add (robot-at <stable>)
Side: (forall <other> of type (or Thing Door)
del (robot-at <other>))
Cost: 2

go-to-large(<large>, <room>)
Prim: add (robot-at <large>)
Side: (forall <other> of type (or Thing Door)
del (robot-at <other>))
Cost: 2

go-to-small(<small>, <room>)
Prim: add (robot-at <small>)
Side: (forall <other> of type (or Thing Door)
del (robot-at <other>))
Cost: 2

go-aside(<room>)
Prim: (forall <other> of type (or Thing Door)
del (robot-at <other>))
Cost: 2

go-thru-door
(<door>, <from-room>, <to-room>)
Prim: del (robot-in <from-room>)
add (robot-in <to-room>)
Cost: 3

pick-up(<small>, <room>)
Prim: del (arm-empty)
del (in <small> <room>)
(forall <other> of type (or Thing Door)
del (next-to <other> <small>))
add (holding <small>)
Side: (forall <other> of type (or Thing Door)
del (next-to <small> <other>))
Cost: 1

put-near-door
(<small>, <door>, <room>)
Prim: add (next-to <small> <door>)
add (next-to <door> <small>)
Side: del (holding <thing>)
add (in <thing> <room>)
add (robot-at <thing>)
add (arm-empty)
Cost: 1

put-near-stable
(<small>, <stable>, <room>)
Prim: add (next-to <small> <stable>)
add (next-to <stable> <small>)
Side: del (holding <thing>)
add (in <thing> <room>)
add (robot-at <thing>)
add (arm-empty)
Cost: 1

put-near-large
(<small>, <large>, <room>)
Prim: add (next-to <small> <large>)
add (next-to <large> <small>)
Side: del (holding <thing>)
add (in <thing> <room>)
add (robot-at <thing>)
add (arm-empty)
Cost: 1

put-near-small
(<small>, <other-small>, <room>)
Prim: add (next-to <small> <other-small>)
add (next-to <other-small> <small>)
Side: del (holding <thing>)
add (in <thing> <room>)
add (robot-at <thing>)
add (arm-empty)
Cost: 1

put-aside(<small>, <room>)
Prim: del (holding <small>)
add (in <small> <room>)
add (arm-empty)
Side: add (robot-at <small>)
Cost: 1

move-aside(<small>, <room>)
Prim: (forall <other> of type (or Thing Door)
del (next-to <small> <other>))
Side: (forall <other> of type (or Thing Door)
del (next-to <other> <small>))
Cost: 1

Figure 3.41: Effects and costs of operators in the Extended STRIPS Domain (also see Figure 3.42).

push-to-door(<large>, <door>, <room>)
Prim: (forall <other> of type (or Thing Door) del (next-to <large> <other>))
(forall <other> of type (or Thing Door) del (next-to <other> <large>))
add (next-to <large> <door>)
add (next-to <door> <large>)
Side: (forall <other> of type (or Thing Door) del (robot-at <other>))
add (robot-at <door>)
add (robot-at <large>)
Cost: 4

push-to-stable(<large>, <stable>, <room>)
Prim: add (next-to <large> <stable>)
add (next-to <stable> <large>)
Side: (forall <other> of type (or Thing Door) del (next-to <large> <other>))
(forall <other> of type (or Thing Door) del (next-to <other> <large>))
(forall <other> of type (or Thing Door) del (robot-at <other>))
add (robot-at <stable>)
add (robot-at <large>)
Cost: 4

push-to-large (<large>, <other-large>, <room>)
Prim: add (next-to <large> <other-large>)
add (next-to <other-large> <large>)
Side: (forall <other> of type (or Thing Door) del (next-to <large> <other>))
(forall <other> of type (or Thing Door) del (next-to <other> <large>))
(forall <other> of type (or Thing Door) del (robot-at <other>))
add (robot-at <other-large>)
add (robot-at <large>)
Cost: 4

push-to-small(<large>, <small>, <room>)
Prim: add (next-to <large> <small>)
add (next-to <small> <large>)
Side: (forall <other> of type (or Thing Door) del (next-to <large> <other>))
(forall <other> of type (or Thing Door) del (next-to <other> <large>))
(forall <other> of type (or Thing Door) del (robot-at <other>))
add (robot-at <small>)
add (robot-at <large>)
Cost: 4

push-aside(<large>, <room>)
Prim: (forall <other> of type (or Thing Door) del (robot-at <other>))
Side: (forall <other> of type (or Thing Door) del (next-to <large> <other>))
(forall <other> of type (or Thing Door) del (next-to <other> <large>))
add (robot-at <large>)
Cost: 4

push-thru-door (<large>, <door>, <from-room>, <to-room>)
Prim: del (in <large> <from-room>)
add (in <large> <to-room>)
Side: del (robot-in <from-room>)
add (robot-in <to-room>)
Cost: 6

Figure 3.42: Effects and costs of operators in the Extended STRIPS Domain (also see Figure 3.41).

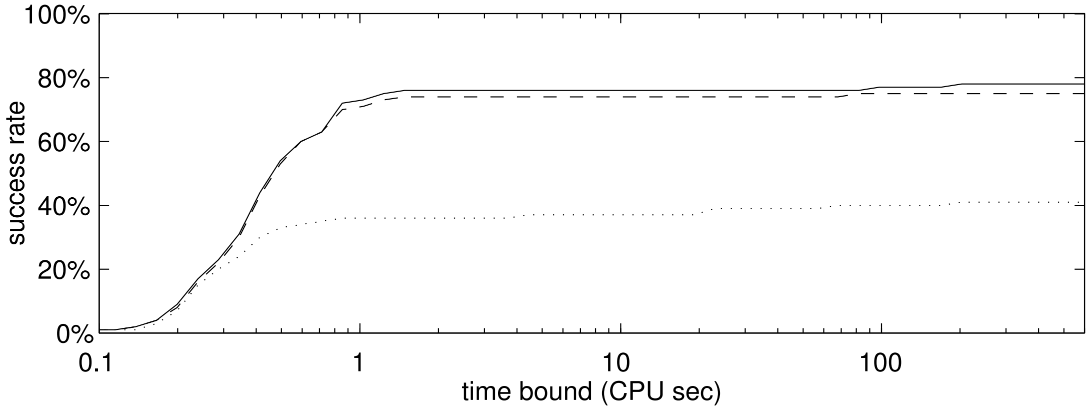


Figure 3.43: PRODIGY performance in the Extended STRIPS Domain, *with primary effects.* We show the percentage of problems solved by different time bounds, for search without cost bounds (solid line), with loose bounds (dashed line), and with tight bounds (dotted lines). If the system does not utilize primary effects, then it almost always fails to find a solution within 600 seconds. Specifically, it solves 2% of the problems if we set tight cost bounds, and no problems at all without tight bounds.

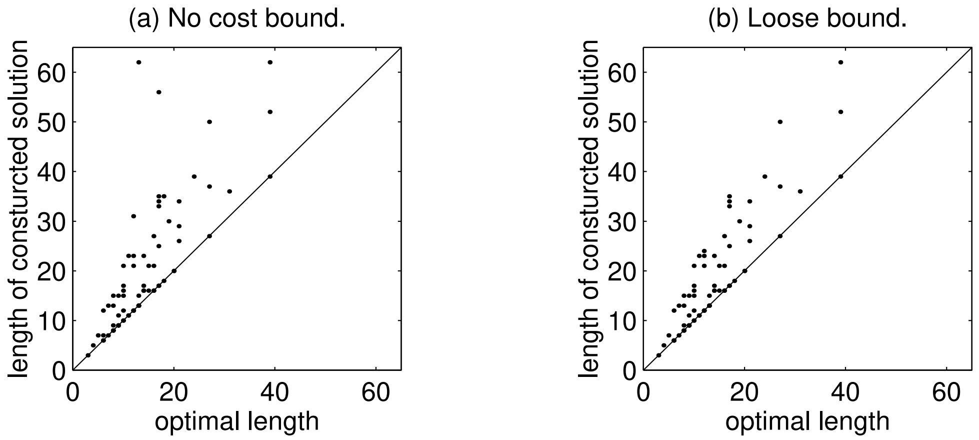


Figure 3.44: Solution lengths in the Extended STRIPS Domain. We give the results of using primary effects without cost bounds and with the loose bounds. The vertical axes show the lengths of the generated solutions, whereas the horizontal axes give the optimal lengths for the same problems.

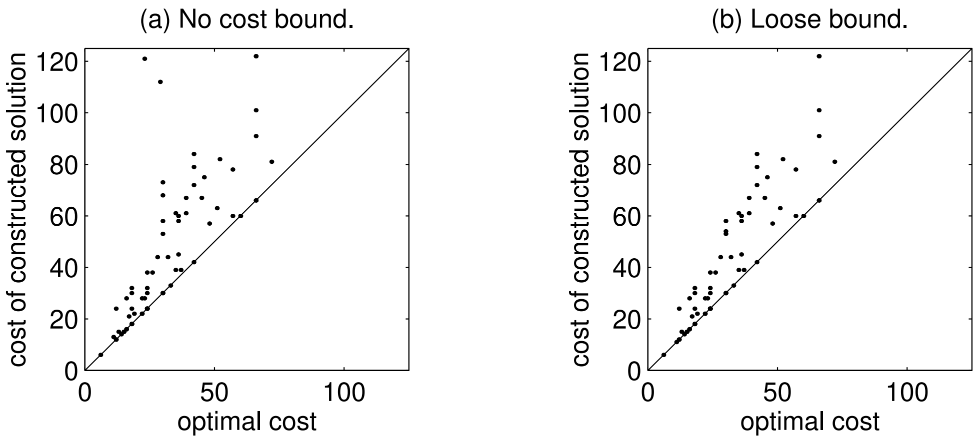


Figure 3.45: Solution costs in the Extended STRIPS Domain.

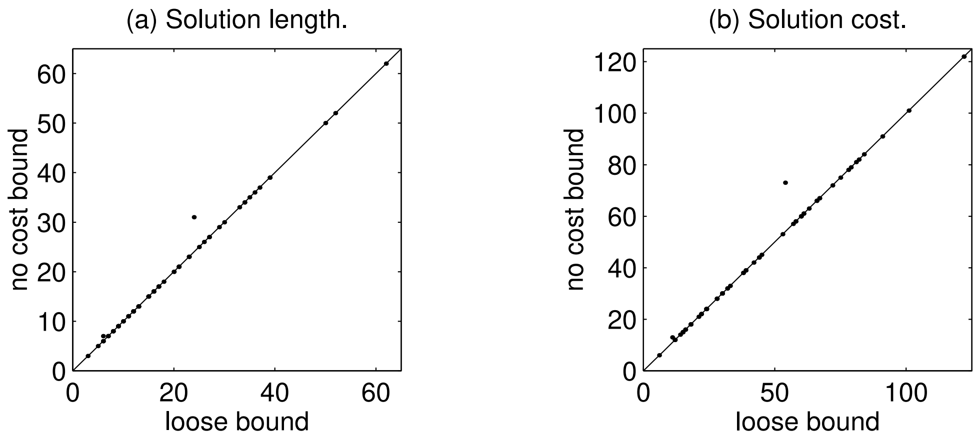


Figure 3.46: Comparison of the solution quality in the experiments without cost bounds and those with the loose cost bounds. The system utilized primary effects in all these experiments. We plot (a) solution lengths and (b) solution costs, for the problems solved in both cases.

and leads to an exponential efficiency improvement. In the PRODIGY system, primary effects not only reduce the search time but also help to improve the solution quality.

In theory, the use of primary effects may *increase* the costs of resulting solutions (see Section 3.2); however, we have not observed the cost increase in any of the ABTWEAK and PRODIGY experiments, and conjecture that it rarely happens in practice.

We have demonstrated that the *Chooser* and *Completer* algorithms accurately identify the important effects of operators, which correspond to human intuition. Note, however, that they improve performance only if some operators do have unimportant effects. Otherwise, *Completer* marks *all* effects as primary and, thus, does not reduce the search space.

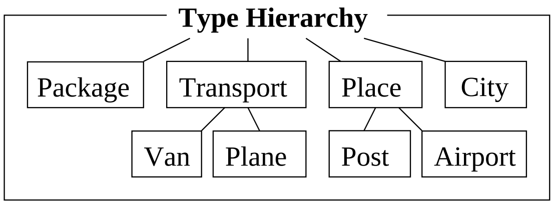
For example, consider the PRODIGY Logistics Domain, constructed by Veloso [1994] for testing her analogical-reasoning system. The domain includes eight types of objects, six operators, and two inference rules; we show the object types and operator effects in Figure 3.47. The problems in this domain require constructing plans for transporting packages among post offices and airports, located in different cities. The system may use vans for the transportation within cities, and planes for carrying packages between airports.

The operators in the Logistics Domain do *not* have unimportant effects. That is, if we mark some effects as side, then the resulting selection causes incompleteness. Thus, the *Completer* algorithm promotes all effects of all operators to primary effects, and the use of this selection does not affect the system's performance.

The Logistics world is not the only PRODIGY domain whose operators have no unimportant effects. The system's collection of domains includes several other simulated worlds with this property, such as the standard Tower-of-Hanoi puzzle (see Section 1.2.1), Briefcase problems [Pednault, 1988b], and Process-Planning Domain [Gil, 1991].

We summarize the results of PRODIGY experiments with primary effects in Table 3.8. The *Chooser* and *Completer* algorithms improved the efficiency of search in the four domains marked by the upward arrows ($\Uparrow$). The time-reduction factor varied from 1 to 200 in the Machining domain, and exceeded 500 in the other three domains.

Primary effects improved the solution quality in the Machining Domain, and did not affect the quality in the Robot world and Sokoban puzzle. When we applied PRODIGY without primary effects to the STRIPS problems, it could not generate solutions in feasible time; thus, we have *no data* on the cost reduction in the Extended STRIPS domain.



fly-plane
(<plane>, <from>, <to>)
<plane>: type Plane
<from>, <to>: type Airport
Eff: del (at <plane> <from>)
add (at <plane> <to>)

load-plane
(<pack>, <plane>, <airport>)
<pack>: type Package
<plane>: type Plane
<airport>: type Airport
Eff: del (at <pack> <airport>)
add (in <pack> <plane>)

unload-plane
(<pack>, <plane>, <airport>)
<pack>: type Package
<plane>: type Plane
<airport>: type Airport
Eff: del (in <pack> <plane>)
add (at <pack> <airport>)

drive-van
(<van>, <from>, <to>)
<van>: type Van
<from>, <to>: type Place
Eff: del (at <van> <from>)
add (at <van> <to>)

load-van
(<pack>, <van>, <place>)
<pack>: type Package
<van>: type Van
<place>: type Place
Eff: del (at <pack> <place>)
add (in <pack> <van>)

unload-van
(<pack>, <van>, <place>)
<pack>: type Package
<van>: type Van
<place>: type Place
Eff: del (in <pack> <van>)
add (at <pack> <place>)

Figure 3.47: Effects of operators in the Logistics Domain. If we apply the *Chooser* and *Completer* algorithms to this domain, then they promote *all* effects to primary. Thus, the resulting selection has no side effects and, hence, it does not improve performance.

Domain	Overall Result	Search-Time Reduction	Solution-Cost Reduction
Extended Robot	$\Uparrow$	> 500	none
Machining	$\Uparrow$	1–200	1.2–1.4
Sokoban	$\Uparrow$	> 500	none
Extended STRIPS	$\Uparrow$	> 500	—
Logistics	–	—	—

Table 3.8: Results of testing the PRODIGY search engine with primary effects. The automatically selected effects improved performance in the first four domains, marked by the upward arrow ($\Uparrow$), and did not affect search in the Logistics Domain. For every domain, we give the time-reduction factor, that is, the ratio of search time without primary effects to that with primary effects. For the Machining Domain, we also give the cost-reduction factor. Note that we cannot evaluate the cost reduction in the STRIPS domain, since PRODIGY was unable to solve STRIPS problems without primary effects.

Chapter 4

Abstraction

Abstraction is the oldest and best-studied approach to changing representations, whose inception dates back to the early days of artificial intelligence. Researchers have used it in a wide variety of AI architectures, as well as in many systems outside of AI.

The central idea of this approach is to identify important aspects of a given problem, construct an outline of a solution, which ignores less significant aspects, and then use it to guide the search for a complete solution. We may define multiple levels of abstraction and move from level to level, constructing more and more detailed outlines.

The applications of this approach take different shapes, from general techniques for simplifying search spaces to highly specialized abstractions in domain-specific systems, and their performance varies widely across systems and domains. The proposed analytical models give disparate efficiency predictions for specific abstraction techniques; however, researchers agree that a *proper* use of abstraction may enhance almost all search and reasoning systems.

In the early Seventies, Sacerdoti experimented with abstraction in backward chaining. He hand-coded relative importance of operator preconditions and used it to automate the construction of solution outlines. This approach proved effective, and AI researchers applied similar techniques in later systems.

In particular, Knoblock adapted it for use in an early version of PRODIGY. He then developed an algorithm for automatic assignment of importance levels to preconditions. The algorithm reduced the system's dependency on the human user; however, it was sensitive to syntactic features of the domain encoding and often required the user to adjust the encoding.

We extended Knoblock's technique and used it in the $\mathcal{S}$HAPER system. In particular, we adapted the abstraction generator to the richer domain language of PRODIGY4 and made it less sensitive to the domain encoding. We explain the use of abstraction in backward-chaining systems (Section 4.1), describe abstraction algorithms in $\mathcal{S}$HAPER (Sections 4.2 and 4.3), and present experiments on their performance (Section 4.4).

4.1 Abstraction in problem solving

We describe abstraction search and a method for automatic generation of abstractions. First, we overview the history of related research (Section 4.1.1). Then, we explain the use of abstraction in classical problem solvers (Section 4.1.2) and discuss its advantages and drawbacks (Section 4.1.3). Finally, we outline Knoblock's technique for abstracting operator preconditions (Section 4.1.5).

4.1.1 History of abstraction

Researchers have developed multiple abstraction techniques and used them in a number of systems. We briefly review the work on abstraction in classical problem solving, which is closely related to our results. The reader may find a more extensive description of past results in the works of Knoblock [1993], Giunchiglia and Walsh [1992], and Yang [1997], as well as in the textbook by Russell and Norvig [1995]. A good source of recent results is the *Proceedings of the Symposium on Abstraction, Reformulation and Approximation* [Lowry, 1992; Van Baalen, 1994; Levy and Nayak, 1995; Ellman and Giunchiglia, 1998].

Newell and Simon [1961; 1972] introduced abstraction in their work on GPS, the first AI problem solver. Their system constructed a high-level plan for solving a problem and used it to guide the search. They tested it in the automated construction of propositional proofs and showed that abstraction reduced the search.

Sacerdoti [1974] extended their technique and used it to develop ABSTRIPS, which combined abstraction with the STRIPS planner [Fikes and Nilsson, 1971]. Given a problem, the system constructed an abstract solution by achieving "important" goals and operator preconditions. Then, it refined the solution by inserting operators for less important subgoals. Sacerdoti partially automated the assignment of importance levels to operator preconditions; however, his technique often produced inappropriate assignments [Knoblock, 1992] and the system depended on the user for finding an effective abstraction.

Following the lead of GPS and ABSTRIPS, researchers used abstraction in other systems, including NONLIN [Tate, 1976; Tate, 1977], NOAH [Sacerdoti, 1977], MOLGEN [Stefik, 1981], and SIPE [Wilkins, 1984; Wilkins, 1988; Wilkins *et al.*, 1995]. All these systems required the user to provide an appropriate abstraction.

Goldstein designed a procedure that automatically generated abstractions for GPS, and demonstrated its utility for several puzzles, including Fool's Disk, Tower of Hanoi, and Monkey and Bananas [Goldstein, 1978; Ernst and Goldstein, 1982]. His procedure constructed a table of transitions among main types of world states, and used it to subdivided the exploration of the state space into several levels.

Christensen [1990] developed a more general technique for automatic abstraction. His algorithm, called PABLO, determined the length of operator sequences for achieving potential subgoals and assigned greater importance to the subgoals that required longer sequences. Unruh and Rosenbloom [1989; 1990] devised a different method of generating abstractions, for look-ahead search in the Soar architecture [Laird *et al.*, 1987].

Knoblock [1993] added abstraction search to the PRODIGY2 system [Minton *et al.*, 1989b]. He then developed the ALPINE algorithm, which automatically abstracted preconditions of operators. Blythe implemented a similar technique in a later version, PRODIGY4; however, he did not extend ALPINE to the richer domain language of PRODIGY4.

Yang *et al.* [1990; 1996] developed the ABTWEAK planner, an abstraction version of TWEAK [Chapman, 1987], and used ALPINE to generate importance assignments for their planner. Then, Bacchus and Yang [1991; 1994] implemented the HIGHPOINT algorithm, an extension to ALPINE that produced abstractions with stronger properties.

Smith and Peot [1992] analyzed the use of ALPINE with the SNLP planner [McAllester and Rosenblitt, 1991] and argued that Knoblock's abstraction is not effective for least-commitment search; however, their argument did not account for positive results of ABTWEAK experiments.

Holte *et al.* [1994; 1996a; 1996b] considered a different approach to generating abstractions. Their system expanded the full search space for a given domain and represented it as a graph. It then used the resulting graph to construct an abstract space. The analysis of the explicit space gave an advantage over other systems and led to the generation of better abstractions. Moreover, it was applicable to a wider range of search algorithms. On the negative side, it worked only for small search spaces.

Several researchers investigated theoretical properties of abstraction and developed formal models of its use in problem solving [Korf, 1980; Giunchiglia and Walsh, 1992]. The proposed models differ in underlying assumptions, and the estimates of efficiency vary from very optimistic [Korf, 1987; Knoblock, 1991] to relatively pessimistic [Bäckström and Jonsson, 1995]; however, the researchers agree on two qualitative conclusions. First, an appropriate abstraction almost always reduces search. Second, a poor choice of abstraction may drastically impair efficiency.

4.1.2 Hierarchical problem solving

We may define abstraction by assigning importance levels to literals in the domain description. This method works for all classical problem solvers. To illustrate it, we describe abstraction for the Drilling Domain in Figure 4.1.

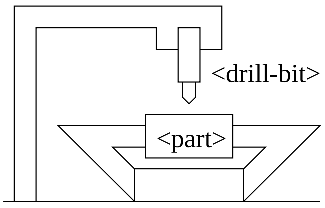
This domain is a simplified version of drilling and painting operations in the PRODIGY Process-Planning Domain [Gil, 1991; Gil and Pérez, 1994]. The drill press uses two types of drill bits, a *spot drill* and a *twist drill.* A spot drill makes a small spot on the surface of a part. This spot guides the movement of a twist drill, which makes a deeper hole.

When painting a part, we have to remove it from the drill press. If a part has been painted before drilling, then the drill press destroys the paint. On the other hand, if we make a surface spot and then paint the part, then the spot disappears; however, painting does not remove deeper holes. Thus, we should paint after the completion of drilling.

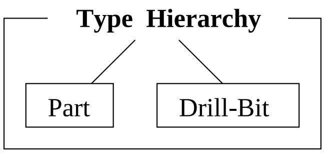
We use natural numbers to encode the importance of literals in a domain, and partition literals into groups by their importance, as shown in Figure 4.2. The resulting partition is called an *abstraction hierarchy;* each group of equally important literals is a *level* of the hierarchy. The topmost level usually comprises the static literals, which encode the unchangeable features of the state. Recall that a literal is static if no operator adds or deletes it.

A hierarchical problem solver begins by finding a solution at the highest nonstatic level, ignoring the subgoals below it. For example, consider the problem in Figure 4.3, which requires drilling and painting a given part. The solver may start by constructing the abstract solution in Figure 4.4(a). In this example, it achieves all subgoals at level 2 and ignores lower-level subgoals.

After constructing an abstract solution, the problem solver *refines* it at the next lower



(a) Drill press.



(b) Types of objects.

put-part(<part>)
<part>: type Part
Pre: (no-part)
Prim: add (holds-part <part>)
Side: del (no-part)

remove-part(<part>)
<part>: type Part
Pre: (holds-part <part>)
Prim: add (no-part)
Side: del (holds-part <part>)

put-drill(<drill-bit>)
<drill-bit>: type Drill-Bit
Pre: (no-drill)
Prim: add (holds-drill <drill-bit>)
Side: del (no-drill)

remove-drill(<drill-bit>)
<drill-bit>: type Drill-Bit
Pre: (holds-drill <drill-bit>)
Prim: add (no-drill)
Side: del (holds-drill <drill-bit>)

drill-spot(<part>, <drill-bit>)
<part>: type Part
<drill-bit>: type Drill-Bit
Pre: (spot-drill <drill-bit>)
(holds-drill <drill-bit>)
(holds-part <part>)
Prim: add (has-spot <part>)
Side: del (painted <part>)

drill-hole(<part>, <drill-bit>)
<part>: type Part
<drill-bit>: type Drill-Bit
Pre: (twist-drill <drill-bit>)
(has-spot <part>)
(holds-drill <drill-bit>)
(holds-part <part>)
Prim: add (has-hole <part>)
Side: del (has-spot <part>)
del (painted <part>)

paint-part(<part>)
<part>: type Part
Pre: not (holds-part <part>)
Prim: (painted <part>)
Side: del (has-spot <part>)

(c) Library of operators.

Figure 4.1: Drilling Domain.

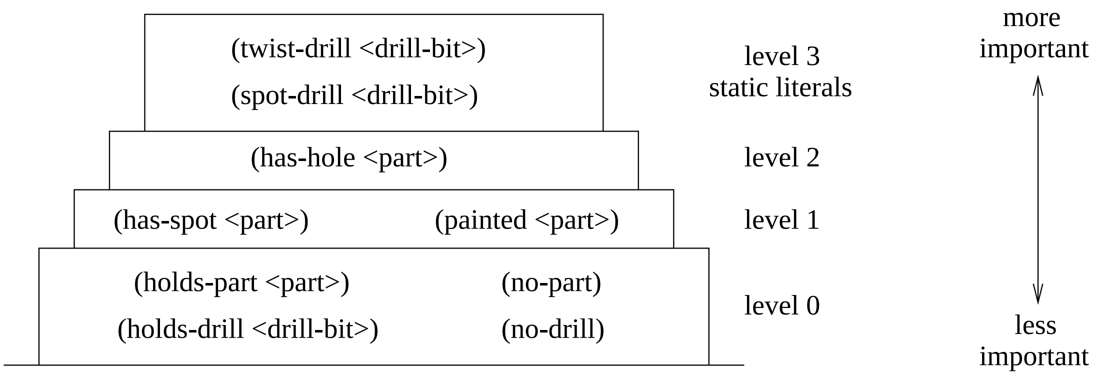


Figure 4.2: Abstraction hierarchy in the Drilling Domain.

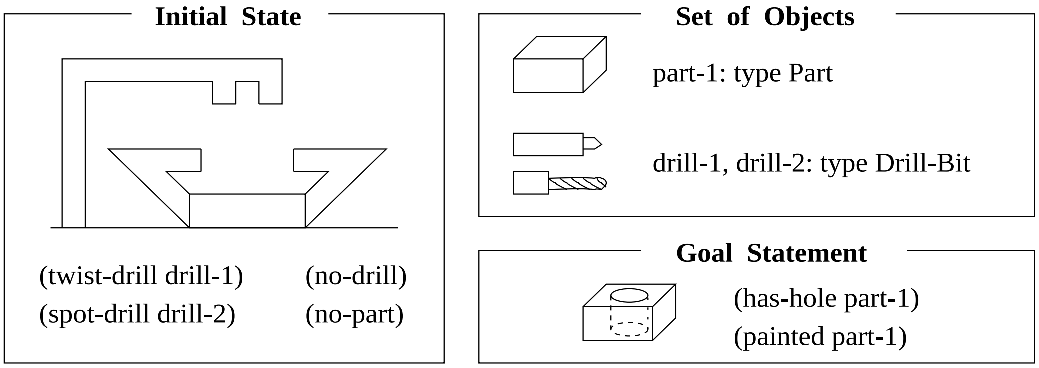


Figure 4.3: Problem in the Drilling Domain.

level, that is, inserts operators for achieving lower-level subgoals. In Figure 4.4(b), we show a refinement of the example solution at level 1. The solver continues to move down the hierarchy, refining the solution at lower and lower levels, until it finally achieves all subgoals at level 0. In Figure 4.4(c), we show a low-level solution for the example problem.

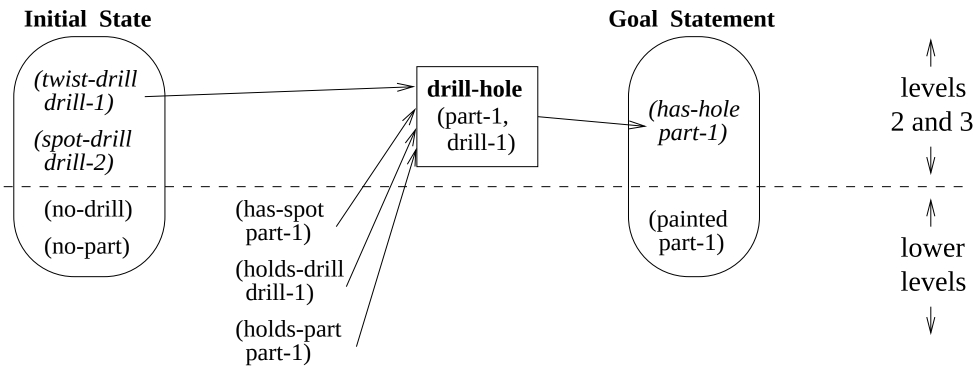
The refinement process preserves the operators of the abstract solution, as well as their relative order. The process may involve significant search, since the inserted operators introduce new subgoals and the solver has to achieve them. For example, consider the operator **put-drill**(drill-1) in Figure 4.4(c), inserted in the process of the low-level refinement. Its precondition (no-drill) becomes a new low-level subgoal and the solver achieves it by adding **remove-drill**(drill-2).

If the problem solver fails to find a refinement, it backtracks to the previous level and constructs a different abstract solution. Observe that, if a problem has a solution at level 0, then there is an abstract solution and sequence of refinements that leads to it. Therefore, *the use of abstraction does not compromise the completeness of a problem solver.*

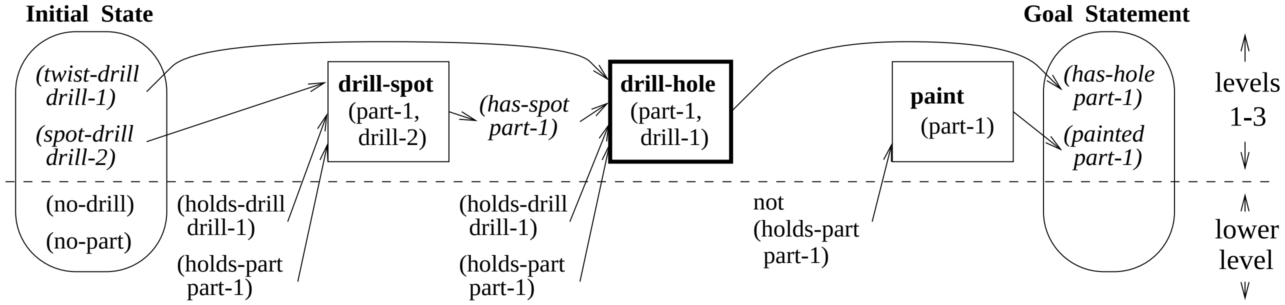
4.1.3 Efficiency and possible problems

Korf [1987] described a simple model of search, which shows why abstraction improves efficiency. His analysis implies that the improvement should be exponential in the number of levels. Knoblock [1991; 1993] developed an alternative model, which gives the same efficiency estimate. Specifically, he showed that abstraction linearly reduces the search depth; since the search time is usually exponential in depth, it results in exponential time reduction.

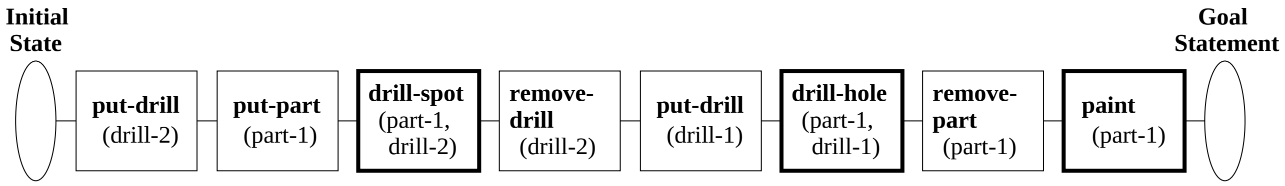
Both models use optimistic assumptions and overestimate the time reduction; however, multiple experiments have confirmed that abstraction reduces search and the reduction grows with the number of levels [Sacerdoti, 1974; Knoblock, 1993; Yang *et al.*, 1996; Tenenberg, 1988]. Experiments also demonstrated that efficiency crucially depends on the properties of a specific hierarchy, and that an inappropriate abstraction may result in gross inefficiency [Bacchus and Yang, 1992; Smith and Peot, 1992]. We now outline the main causes of such inefficiency.



(a) Level 2: Abstract solution.



(b) Level 1: Intermediate refinement.



(c) Level 0: Complete solution.

Figure 4.4: Abstract solution and its refinements. We italicize subgoals at the current level of abstraction and show lower-level subgoals by the usual font. Thick rectangles mark operators inherited from higher levels, whereas thin rectangles show operators inserted at the current level.

Backtracking across levels

A hierarchical problem solver may construct an abstract solution that has no refinement. For instance, if both drill bits in the example problem were twist drills, the solver could still produce the abstract solution in Figure 4.4(a), but it would fail to refine this solution. After failing to find a refinement, the problem solver backtracks to a higher level and constructs a different abstract solution. For example, if the Drilling Domain had a means for making holes without a spot drill, the solver would eventually find the corresponding abstract solution.

Bacchus and Yang [1992; 1994] analyzed backtracking across abstraction levels and demonstrated that it causes an exponential increase in search time. Frequent failures to find a refinement can make hierarchical problem solving less efficient than search without abstraction.

Intermixing abstraction levels

When a problem solver constructs a refinement, it may insert an operator that adds or deletes a high-level literal, thus invalidating the abstract solution. The solver then has to insert operators that restore correctness at the high level. Thus, it has to intermix high-level and low-level search, which significantly reduces the effectiveness of abstraction. This situation differs from backtracking to a higher level: the solver inserts additional operators into the abstract solution, rather than abandoning it and producing an alternative solution.

To avoid this intermixing, we may prohibit the use of operators with abstract-level effects in constructing refinements; however, this restriction eliminates some refinements and causes more frequent backtracking across levels.

Generating long solutions

Hierarchical problem solving usually yields longer solutions than search without abstraction. In particular, if we use abstraction with an admissible solver, it compromises the admissibility; that is, the solver loses the ability to find optimal solutions.

For instance, consider the drilling problem in Figure 4.5(a). A hierarchical solver may construct the abstract solution given in Figure 4.5(b) and then refine it as shown in Figure 4.5(c). In this example, the problem solver has found a shortest solution at level 1 and its shortest refinement at level 0; however, the resulting low-level solution is not optimal. An admissible search would give a shorter solution, as shown in Figure 4.5(d).

If the utility of problem solving depends on solution quality, then an increase in solution length may result in lower overall performance, despite the reduction in running time. Furthermore, the search depth is usually proportional to the solution length, and the search time grows exponentially with depth (for example, see the analysis by Minton *et al.* [1991; 1994]). In some cases, hierarchical search generates unreasonably long solutions, causing gross inefficiency [Bäckström and Jonsson, 1995].

4.1.4 Avoiding the problems

We have discussed the three main problems with abstraction search. The increase in the number of levels may exponentially reduce the search at each level, but it may also aggravate the problems. We illustrate this trade-off in Figure 4.6.

Knoblock [1990; 1993] developed an algorithm, called ALPINE, for generating hierarchies that never cause intermixing of levels. This property was named *ordered monotonicity.* Abstraction hierarchies that satisfies it are called *ordered hierarchies.* Experiments have confirmed that ordered monotonicity improves the efficiency of PRODIGY [Knoblock, 1993; Knoblock *et al.*, 1991a; Carbonell *et al.*, 1992], ABTWEAK [Yang *et al.*, 1996], and other classical systems; however, its enforcement reduces the number of levels and often leads to "collapsing" all literals into a single level.

Bacchus and Yang [1991; 1994] designed the HIGHPOINT system, which generates abstractions with a stronger property. It ensures that the hierarchy satisfies ordered monotonicity, and that every abstract solution has a refinement; thus, the solver never backtracks across

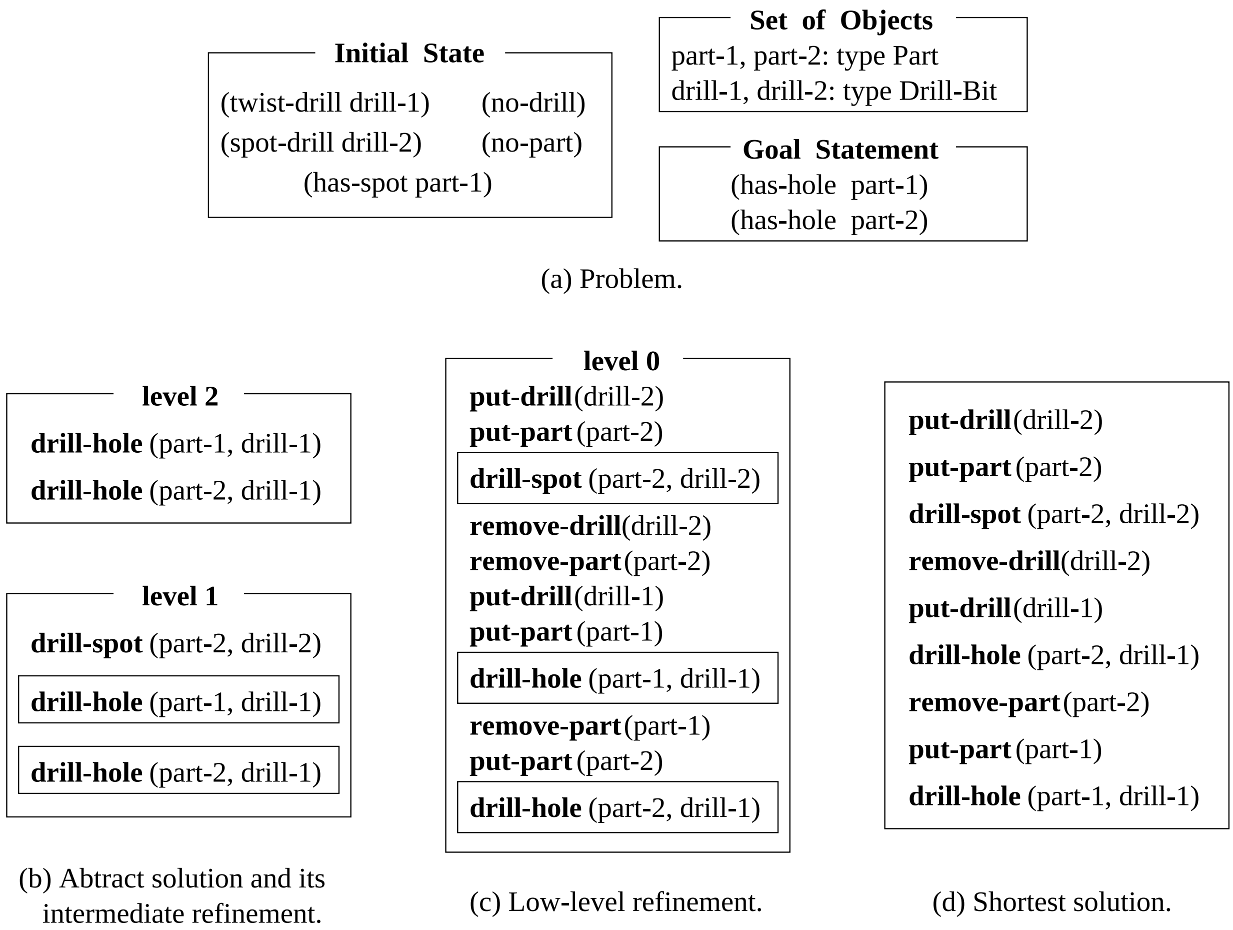


Figure 4.5: Example of compromising admissibility: Hierarchical search yields an eleven-operator solution (c), whereas a shortest solution has nine operators (d). Rectangles in the refined solutions (b, c) mark operators inherited from the higher levels of abstraction.

levels. To enforce these properties, the system imposes several rigid constraints on the abstraction levels of literals, and often fails to find a hierarchy that satisfies them; however, when HIGHPOINT does find a hierarchy, it usually gives better results than ALPINE's abstraction.

To our knowledge, nobody has investigated admissible abstractions in classical problem solving and there is no technique to generate hierarchies that allow finding near-optimal solutions. If we use ALPINE or HIGHPOINT, the resulting solutions may be exponentially longer than optimal [Bäckström and Jonsson, 1995].

In Figure 4.6, we illustrate the relative properties of ABSTRIPS [Sacerdoti, 1974], ALPINE,

Figure 4.6: Trade-off in the construction of hierarchies.

and HIGHPOINT. We have followed Knoblock's approach to resolving this trade-off: the $\mathcal{S}$HAPER system uses ordered hierarchies and maximizes the number of levels within the constraints of ordered monotonicity.

4.1.5 Ordered monotonicity

We review Knoblock's technique for generating ordered hierarchies. It is based on the following constraints, which determine the relative importance of preconditions and effects in fully instantiated operators:

For every instantiated operator *op*:

- if $prim_1$ and $prim_2$ are primary-effect literals of *op*,
  then $level(prim_1) = level(prim_2)$
- if *prim* is a primary-effect literal and *side* is a side-effect literal,
  then $level(prim) \geq level(side)$
- if *prim* is a primary-effect literal and *prec* is a nonstatic precondition literal,
  then $level(prim) \geq level(prec)$

For example, the hierarchy in Figure 4.2 satisfies these constraints. Note that all three inequalities involve primary effects. If an operator has no primary effects, then it requires no constraints, as the system never uses it in problem solving.

Knoblock *et al.* [1991] demonstrated that these constraints guarantee ordered monotonicity for backward-chaining algorithms, as well as for PRODIGY search; however, this result is *not* applicable to forward chainers. We next give an outline of their proof.

When a problem solver refines an abstract solution, it inserts operators for achieving low-level literals. The new operators may affect higher levels in two ways. First, they may add or delete high-level literals and invalidate the abstract solution. Second, they may have high-level preconditions that become new abstract subgoals.

If the hierarchy satisfies the constraints, then neither situation can arise. The first two inequalities ensure that the new operators have no high-level effects, and the last inequality implies that they do not introduce high-level subgoals. Thus, the problem solver does not intermix the refinement process with the abstract-level search.

Knoblock observed that the inequalities do *not* provide a necessary condition for ordered monotonicity. In some domains, they overconstrain the hierarchy and lead to an unnecessary reduction in the number of levels. He described several heuristics for relaxing the constraints. We found a technique for further relaxation, and identified necessary and sufficient conditions for constructing ordered hierarchies [Fink and Yang, 1993]. The relaxation improves the abstraction in some domains, but the percentage of such domains is low. We did not implement this technique in $\mathcal{S}$HAPER.

We have described constraints for *instantiated* operators. If the system uses them, it has to generate all instantiations, which may cause a combinatorial explosion. To avoid this problem, Knoblock implemented an algorithm that constructs a hierarchy of predicates rather than literals, by imposing constraints on the preconditions and effects of operators.

Then, the system determines the level of a literal by its predicate name; thus, if literals have the same name, they belong to the same level.

Note that the precondition predicates of an operator include *all* predicates from the precondition expression, in particular, those from disjunctive and universally quantified subexpressions. The system does *not* impose constraints on the static predicates. Recall that we view a predicate as static if all corresponding literals are static.

The resulting constraints are stronger than the constraints for instantiated operators; hence, they also ensure ordered monotonicity. On the negative side, they may cause overconstraining and collapse of the hierarchy. The burden of avoiding such situations is on the user, who has to select proper predicates for encoding the domain.

To illustrate overconstraining, we consider the Drilling Domain (Figure 4.1) and replace the predicates (has-spot <part>), (has-hole <part>), and (painted <part>) with a more general predicate (has <feature> <part>), where <feature> can be spot, hole, or paint (see Figure 4.7a). Then, we cannot separate levels 1 and 2 of the hierarchy in Figure 4.2 and have to use fewer levels (Figure 4.7b).

As a more extreme example, we can encode the domain using two general predicates: (pred-0 <name-0>) replaces the predicates with no arguments and (pred-1 <name-1> <thing>) replaces the one-argument predicates (see Figure 4.7c). The <name-0> and <name-1> variables range over the names of the original predicates, whereas the values of <thing> are drill bits and parts. Then, the algorithm cannot separate predicates into multiple abstraction levels; hence, it fails to generate any hierarchy.

We have reviewed a technique for constructing problem-independent hierarchies. Knoblock also designed a variation of his algorithm that generates abstractions for specific problems. We will consider the use of problem-specific hierarchies in Section 5.3.

4.2 Hierarchies for the PRODIGY domain language

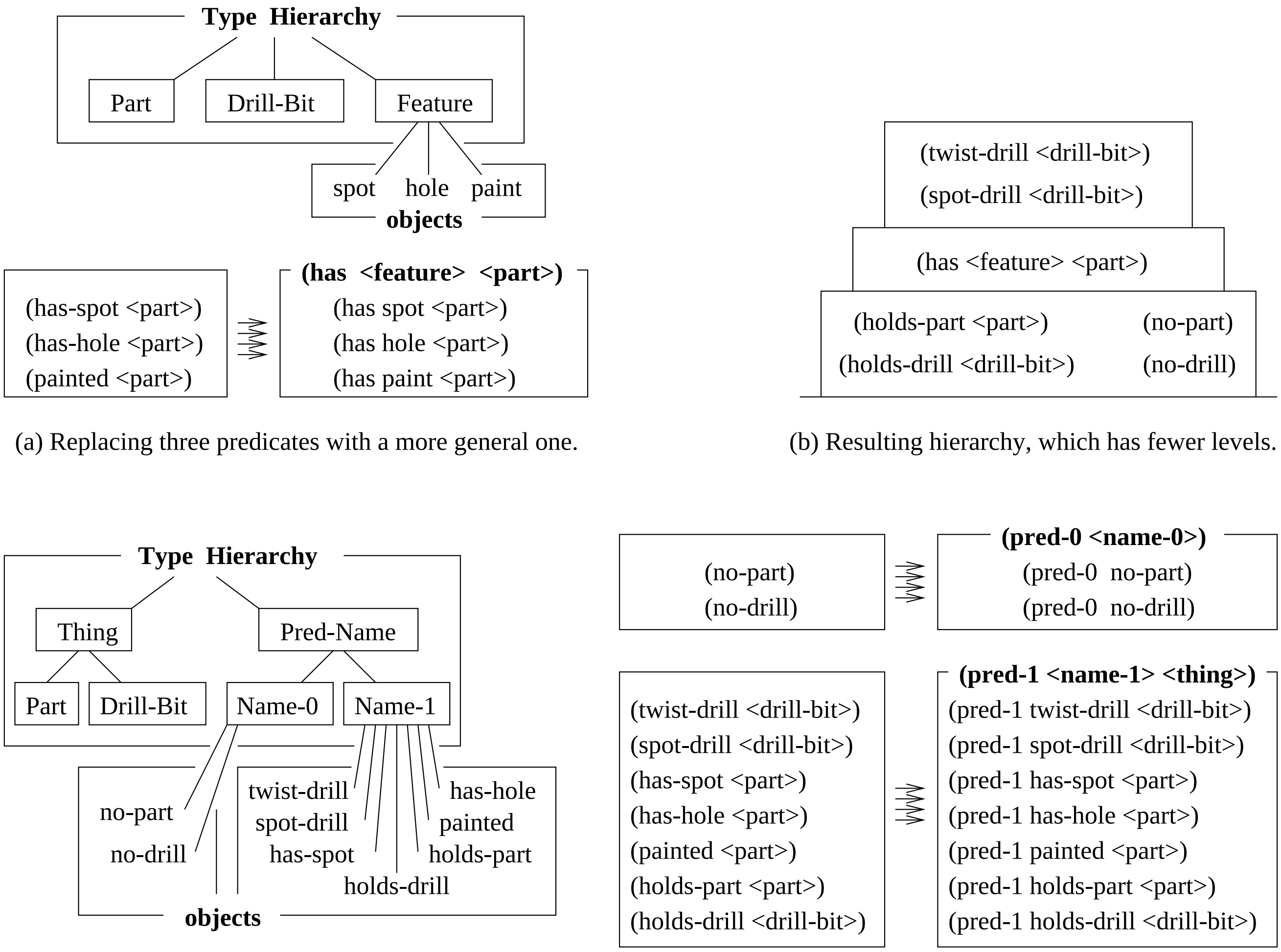
ALPINE was designed for a limited sublanguage of the PRODIGY architecture. In particular, it did not handle if-effects and inference rules. We present an extended set of constraints, which accounts for all features of the PRODIGY language (Section 4.2.1), and give an algorithm for generating hierarchies (Section 4.2.2).

4.2.1 Additional constraints

We describe constraints for if-effects and then discuss the use of eager and lazy inference rules in hierarchical problem solving.

If-effects

Since PRODIGY uses the actions of if-effects in the same way as simple effects (see Sections 2.2.4 and 3.1.3), they require the same constraints; however, we also have to constrain the levels of if-effect conditions.



(a) Replacing three predicates with a more general one.

(b) Resulting hierarchy, which has fewer levels.

(c) Describing the domain with two general predicates, which results in the complete collapse of the hierarchy.

Figure 4.7: Collapse of an ordered hierarchy due to the use of general predicates.

If the system uses an operator for achieving a primary action of some if-effect, then it adds the conditions of this if-effect to the operator's preconditions and views them as subgoals. Therefore, nonstatic conditions require the same constraints as preconditions:

- if $prim$ is a primary action and cnd is a nonstatic condition of an if-effect,
  then $level(prim) \geq level(cnd)$

We summarize all constraints in Figure 4.8(a), where the term "effect" refers to both simple effects and actions of if-effects. To illustrate their use, we consider an extended version of painting in the Drilling Domain, which allows the choice of a color. We encode it by two operators, **pick-paint** and **paint-part**, given in Figure 4.9(a). The first operator does not add any constraints, because it contains only one predicate. The second operator requires the following constraints:

For primary effects:
$level(\mathsf{painted}) = level(\mathsf{part\text{-}color})$

For side effects:
$level(\mathsf{painted}) \geq level(\mathsf{has\text{-}spot})$
$level(\mathsf{painted}) \geq level(\mathsf{part\text{-}color})$

(a) Constraints for an operator:

For every two primary effects, $prim_1$ and $prim_2$:
$level(prim_1) = level(prim_2)$.

For every primary effect *prim* and side effect *side*:
$level(prim) \geq level(side)$.

For every primary effect *prim* and nonstatic precondition *prec*:
$level(prim) \geq level(prec)$.

For every primary action *prim* and nonstatic condition *cnd* of an if-effect:
$level(prim) \geq level(cnd)$.

(b) Constraints for an inference rule:

For every two primary effects, $prim_1$ and $prim_2$:
$level(prim_1) = level(prim_2)$.

For every primary effect *prim* and side effect *side*:
$level(prim) \geq level(side)$.

For every primary effect *prim* and nonstatic precondition *prec*:
$level(prim) = level(prec)$.

For every side effect *side* and nonstatic precondition *prec*:
$level(side) \leq level(prec)$.

For every primary action *prim* and nonstatic condition *cnd* of an if-effect:
$level(prim) = level(cnd)$.

For every side action *side* and nonstatic condition *cnd* of an if-effect:
$level(side) \leq level(cnd)$.

Figure 4.8: Constraints on the literal levels in an ordered abstraction hierarchy.

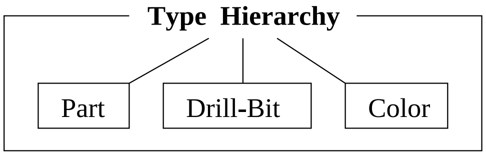
For preconditions:
$level(\textsf{painted}) \geq level(\textsf{holds-part})$
$level(\textsf{part-color}) \geq level(\textsf{holds-part})$

For if-effect conditions:
$level(\textsf{part-color}) \geq level(\textsf{paint-color})$

Eager inference rules

PRODIGY may use eager inference rules in backward chaining, in the same way as operators (see Section 2.3.2); thus, they inherit all operator constraints. In addition, it uses these rules in forward chaining from the current state, which poses the need for additional constraints.

When the system applies an eager rule to the current state, it must not add or delete



pick-paint(<new-color>)
<new-color>, <old-color>: type Color
Prim: add (paint-color <new-color>)
Side: (if (paint-color <old-color>)
del (paint-color <old-color>))

paint-part(<part>)
<part>: type Part
<new-color>, <old-color>: type Color
Pre: not (holds-part <part>)
Prim: (painted <part>)
(if (paint-color <new-color>)
add (part-color <part> <color>))
Side: del (has-spot <part>)
(if (part-color <part> <old-color>)
del (part-color <part> <old-color>))

(a) Painting operators.

Inf-Rule **add-idle**
Pre: (no-drill)
(no-part)
Prim: add (idle)

Inf-Rule **del-color(<part>, <old-color>)**
<part>: type Part
<old-color>: type Color
Pre: not (painted <part>)
(part-color <part> <old-color>)
Side: del (part-color <part> <old-color>)

(b) Inference rules.

Figure 4.9: Extensions to the Drilling Domain.

any literals above the current level of abstraction. Therefore, the effects of the rule must be no higher in the hierarchy than its nonstatic preconditions. Similarly, the actions of each if-effect must be no higher than its conditions:

For every inference rule *inf*:

- if *eff* is an effect and *prec* is a nonstatic precondition of *inf*,
  then $level(eff) \le level(prec)$
- if *eff* is an action and *cnd* is a nonstatic condition of an if-effect,
  then $level(eff) \le level(cnd)$

We combine these inequalities with the operator constraints (see Figure 4.8a) and obtain the constraint set in Figure 4.8(b). For example, consider the inference rules in Figure 4.9(b). The first rule says that, if the drill press has neither a drill bit nor a part, then it is idle. The second rule ensures that unpainted parts have no color; it fires when drilling destroys the paint. These rules give rise to the following constraints:

Inf-Rule **add-idle**:
$level(\textsf{idle}) = level(\textsf{no-drill})$
$level(\textsf{idle}) = level(\textsf{no-part})$

Inf-Rule **del-color**:
$level(\textsf{part-color}) \le level(\textsf{part-color})$

Type of description change: Generating an abstraction hierarchy.

Purpose of description change: Maximizing the number of levels, while ensuring ordered monotonicity.

Use of other algorithms: None.

Required input: Description of the operators and inference rules.

Optional input: Selection of primary effects.

Figure 4.10: Specification of the *Abstractor* algorithm.

Lazy inference rules

Since PRODIGY uses lazy rules in backward chaining, they also inherit the operator constraints; however, the treatment of their effects differs from that of operator effects and requires additional constraints.

If the system moves a lazy rule from the tail to the head and *later* applies an operator that invalidates some preconditions of the rule, then it cancels the rule's effects; that is, it removes all effects of the rule from the current state (see Section 2.3.2). This removal must not affect higher levels of abstraction; therefore, the effects of a lazy rule must be no higher in the hierarchy than its preconditions. Similarly, if an operator application invalidates conditions of some if-effect of a rule, then the system cancels the actions of this if-effect. Thus, the actions of an if-effect must be no higher than its conditions. We conclude that lazy inference rules require the same constraints as eager rules (see Figure 4.8b).

4.2.2 Abstraction graph

We now describe the *Abstractor* algorithm, an extension to ALPINE that generates ordered hierarchies for the full domain language of PRODIGY. It is one of the description changers in the $\mathcal{S}$HAPER system.

The purpose of *Abstractor* is to generate a hierarchy that satisfies the constraints in Figure 4.8 and has as many levels as possible. It constructs a problem-independent hierarchy; that is, the resulting abstraction works for all problem instances in the domain.

The algorithm builds a hierarchy for a specific selection of primary effects. If it has no information about primary effects, then it assumes that all effects are primary. This assumption ensures ordered monotonicity, but may result in overconstraining the hierarchy. We summarize the specification of *Abstractor* in Figure 4.10.

The algorithm generates a *hierarchy of predicates;* that is, it places all literals with a common predicate name on the same level. We have also constructed a problem-specific version that operates with instantiated operators and generates a hierarchy of literals, but have not included it in the $\mathcal{S}$HAPER system. This version uses the *Matcher* algorithm (Section 3.4.2), which produces all instantiated operators.

Add-Operator(*op*)
Pick a primary effect *prim* of *op*.
For every other primary effect *other-prim* of *op*:
  Add an edge from *prim* to *other-prim*.
  Add an edge from *other-prim* to *prim*.
For every side effect *side* of *op*:
  Add an edge from *prim* to *side*.
For every nonstatic precondition *prec* of *op*:
  Add an edge from *prim* to *prec*.
For every if-effect *if-eff* of *op*:
  If *if-eff* has some primary action:
    For every nonstatic condition *cnd* of *if-eff*:
      Add an edge from *prim* to *cnd*.

Add-Side-Rule(*inf*)
For every effect *side* of *inf*:
  For every nonstatic precondition *prec* of *inf*:
    Add an edge from *prec* to *side*.
For every if-effect *if-eff* of *inf*:
  For every action *side* of *if-eff*:
    For every nonstatic condition *cnd* of *if-eff*:
      Add an edge from *cnd* to *side*.

Add-Prim-Rule(*inf*)
Pick a primary effect *prim* of *inf*.
For every other primary effect *other-prim* of *inf*:
  Add an edge from *prim* to *other-prim*.
  Add an edge from *other-prim* to *prim*.
For every side effect *side* of *inf*:
  Add an edge from *prim* to *side*.
For every nonstatic precondition *prec* of *inf*:
  Add an edge from *prim* to *prec*.
  Add an edge from *prec* to *prim*.
For every if-effect *if-eff* of *inf*:
  If *if-eff* has some primary action:
    For every nonstatic condition *cnd* of *if-eff*:
      Add an edge from *prim* to *cnd*.
      Add an edge from *cnd* to *prim*.
  If *if-eff* has no primary actions:
    For every action *side* of *if-eff*:
      For every nonstatic condition *cnd* of *if-eff*:
        Add an edge from *cnd* to *side*.

Figure 4.11: Adding constraint edges.

Encoding of constraints

We encode the constraint set by a directed graph [Knoblock, 1993], as illustrated in Figure 4.14(a). The nonstatic predicates are nodes of this graph, and the constraints are its edges. If the level of some predicate $pred_1$ must be no smaller than that of $pred_2$, then the algorithm adds an edge from $pred_1$ to $pred_2$. If the two predicates must be on the same level, then it adds two edges, from $pred_1$ to $pred_2$ and from $pred_2$ to $pred_1$.

This encoding allows efficient grouping of nodes by their levels, as well as finding the order of the resulting groups (for example, see the algorithms book by Aho *et al.* [1974] or the textbook by Cormen *et al.* [1990]).

Constraint edges for an operator

If an operator has no primary effects, it requires no constraints. For an operator with primary effects, we apply the *Add-Operator* algorithm in Figure 4.11, which imposes the constraints shown in Figure 4.12(a). It picks one of the primary-effect predicates and uses this predicate as a "pivot" node for adding constraints; in the picture, the pivot effect is shown by an oval.

The algorithm adds edges from the pivot predicate to every other primary effect, as well as opposite edges from other primary effects to the pivot. In Figure 4.12, we mark primary effects by thick circumferences; note that they include both simple effects and

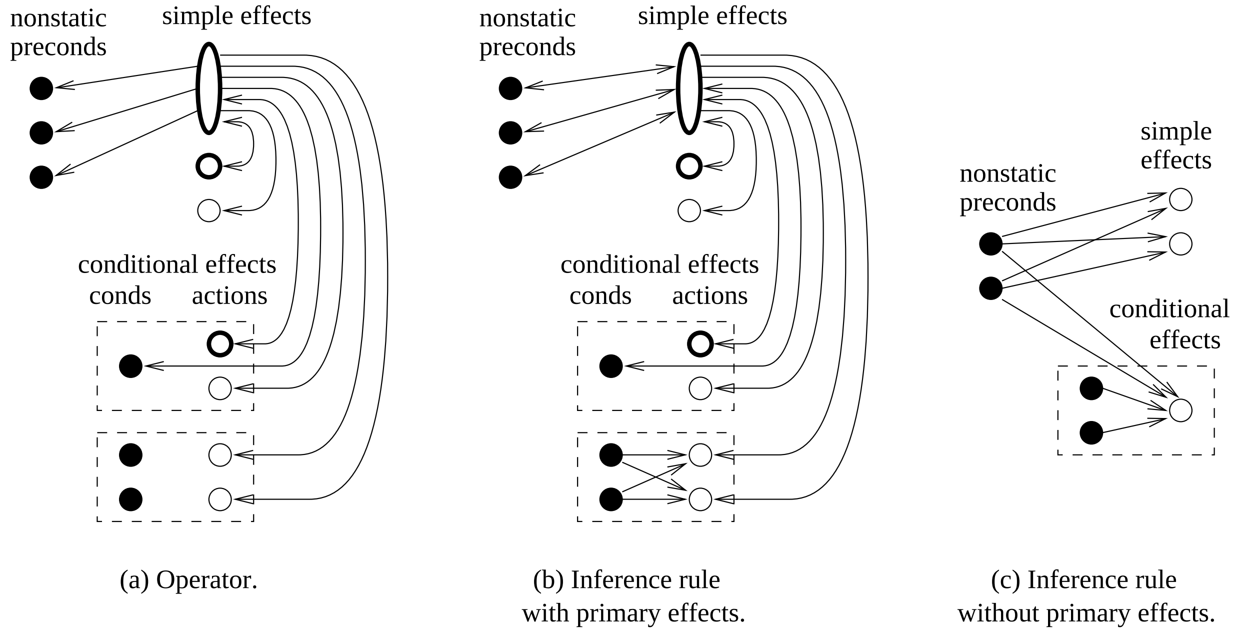


Figure 4.12: Constraint edges in the abstraction graph. We show preconditions and if-effect conditions by black circles, primary effects by thick circumferences, and side effects by thin circumferences. An oval marks the pivot effect, selected arbitrarily among the primary effects.

actions of if-effects. Then, the algorithm adds edges from the pivot to all side effects (thin circumferences) and to all nonstatic preconditions (black circles). Finally, for every if-effect with primary actions, it adds edges from the pivot to the nonstatic conditions of the if-effect (also black circles). The resulting set of constraints is equivalent to the system of inequalities in Figure 4.8(a). We show the constraint edges for the **paint-part** operator in Figure 4.13(a).

Edges for an inference rule

If an inference rule has primary effects, we impose constraints using the *Add-Prim-Rule* algorithm in Figure 4.11; the resulting edges are given in Figure 4.12(b). The algorithm picks a pivot predicate among the primary effects, and adds edges from it to all other effects and all nonstatic preconditions, as well as opposite edges from primary effects and nonstatic preconditions to the pivot.

For every if-effect with primary actions, the algorithm connects its nonstatic conditions with the pivot by "two-way" edges. For an if-effect without primary actions, it adds edges from every nonstatic condition to every action. The resulting constraints are equivalent to the inequalities in Figure 4.8(b). We illustrate them for the **add-idle** rule in Figure 4.13(b).

If a *lazy* inference rule has no primary effects, then the problem solver never uses it, which means that the rule requires no constraints. For an *eager* inference rule without primary effects, we apply the *Add-Side-Rule* algorithm (Figure 4.11). First, it inserts constraint edges from every nonstatic precondition to every effect. Then, for each if-effect, it adds edges from every nonstatic condition to every action. We show these edges in Figure 4.8(c) and illustrate them for the **del-paint** rule in Figure 4.13(c).

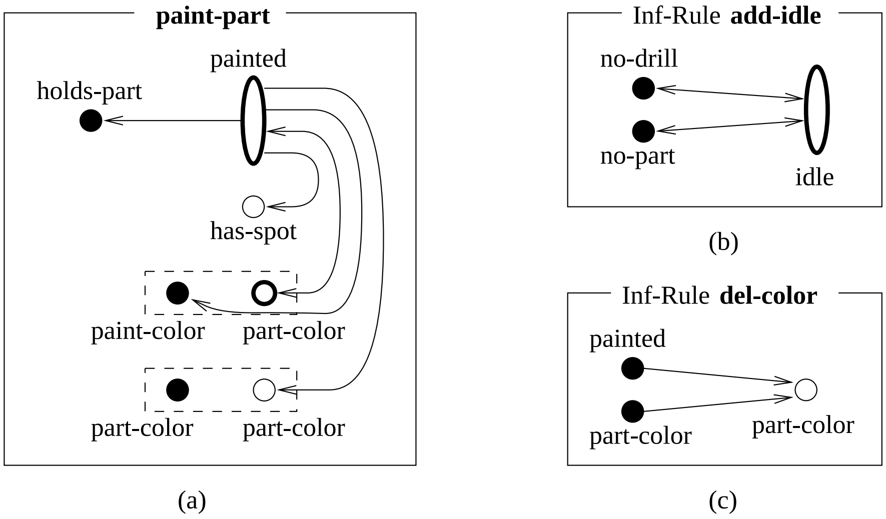


Figure 4.13: Some constraint edges in the Drilling Domain.

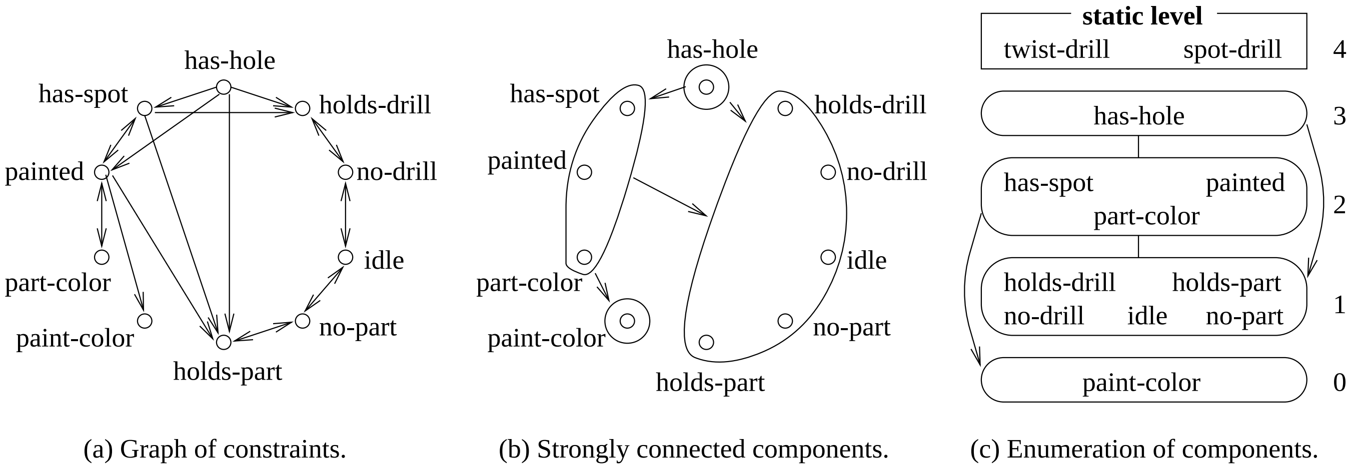


Figure 4.14: Generating an abstraction graph and ordered hierarchy for the Drilling Domain.

Construction of the hierarchy

After adding edges for all operators and inference rules, we obtain a graph that encodes all constraints, as shown in Figure 4.14(a). For every two predicates $pred_1$ and $pred_2$, the graph contains a path from $pred_1$ to $pred_2$ if and only if $level(pred_1) \geq level(pred_2)$. In particular, if there is a path from $pred_1$ to $pred_2$ and back from $pred_2$ to $pred_1$, then they are on the same level. Therefore, the strongly connected components of the graph correspond to the levels of the hierarchy [Knoblock, 1994].

The system identifies the components of the graph, thus grouping the predicates by levels, as illustrated in Figure 4.14(b). The resulting encoding of the hierarchy is called an *abstraction graph*. We use it for the storage of abstraction hierarchies, as well as for the comparison of alternative hierarchies, described in Section 7.1.1.

Before using the hierarchy in problem solving, the system enumerates its levels and adds

Abstractor
Create a graph whose nodes are nonstatic predicates, with no edges.
For every operator *op* in the domain:
  If *op* has some primary effect,
    then call *Add-Operator*(*op*).
For every inference rule *inf* in the domain:
  If *inf* has some primary effect,
    then call *Add-Prim-Rule*(*inf*).
  If *inf* is an eager rule without primary effects,
    then call *Add-Side-Rule*(*inf*).
Identify strongly connected components of the graph.
Topologically sort the components; enumerate them accordingly.

Figure 4.15: Constructing the abstraction graph.

the static level (see Figure 4.14c). The enumeration must be consistent with edges between components: if there is an edge from $level_1$ to $level_2$, then $level_1 > level_2$. If the graph allows several enumerations, we may use any of them. The system applies topological sorting to the components of the abstraction graph and numbers them in the resulting order.

In Figure 4.15, we summarize the algorithm for constructing an ordered hierarchy, called *Abstractor*. It adds constraint edges for all operators and inference rules, identifies abstraction levels, and orders them by topological sorting.

Running time

To analyze the time complexity of *Abstractor*, we denote the number of effect predicates in an operator or inference rule by e, and the number of nonstatic preconditions, along with if-effect conditions, by p. The complexity of adding constraint edges for an operator is linear, $O(e+p)$. If an inference rule has primary effects, and all its if-effects have primary actions, then the complexity of adding the corresponding constraints is also $O(e+p)$. Otherwise, the time for processing the rule is superlinear, $O(e \cdot p)$.

We define E as the total number of effects in all operators and inference rules, and P as the total number of preconditions and if-effect conditions:

$$\begin{aligned} E &= \sum_{op} e_{op} + \sum_{inf} e_{inf}; \\ P &= \sum_{op} p_{op} + \sum_{inf} p_{inf}. \end{aligned}$$

If all inference rules have primary effects, and all their if-effects have primary actions, then the complexity of adding all edges is $O(E+P)$. If not, the complexity is superlinear; however, such situations rarely occur in practice and do not result in significant deviations from linearity.

Finally, we denote the number of nonstatic predicates in the domain by N. The abstraction graph contains N nodes; hence, the time for identifying and sorting its components is $O(N^2)$ (see the text by Cormen *et al.* [1990]). Thus, the overall running time is close to $O(E + P + N^2)$.

We have implemented the *Abstractor* algorithm in Common Lisp and tested it on a Sun 5 computer. Its execution time is approximately $(11 \cdot E + 11 \cdot P + 6 \cdot N^2) \cdot 10^{-4}$ seconds, which is negligibly small in comparison with problem-solving time.

4.3 Partial instantiation of predicates

The effectiveness of *Abstractor* depends on the user's choice of predicates for the domain encoding, and an inappropriate choice may cause a collapse of the hierarchy. We illustrated this problem for the Drilling Domain (Figures 4.1 and 4.2), where the use of general predicates may reduce the number of abstraction levels (Figure 4.7a,b) or even lead to complete collapse (Figure 4.7c).

We say that the hierarchy in Figures 4.2 is *finer-grained* than that in Figure 4.7(b), which means that we may obtain the second hierarchy from the first one by merging some of the abstraction levels. In other words, every level of the hierarchy in Figures 4.2 is a subset of some level in Figure 4.7(b).

Knoblock [1993] showed that fine granularity is a desirable property, since it improves the efficiency of abstraction search, as long as the increase in the number of levels does not compromise ordered monotonicity. Yang *et al.* [1996] came to the same conclusion during their work on the ABTWEAK system. Experiments with SHAPER have also re-confirmed the utility of fine-grained ordered hierarchies.

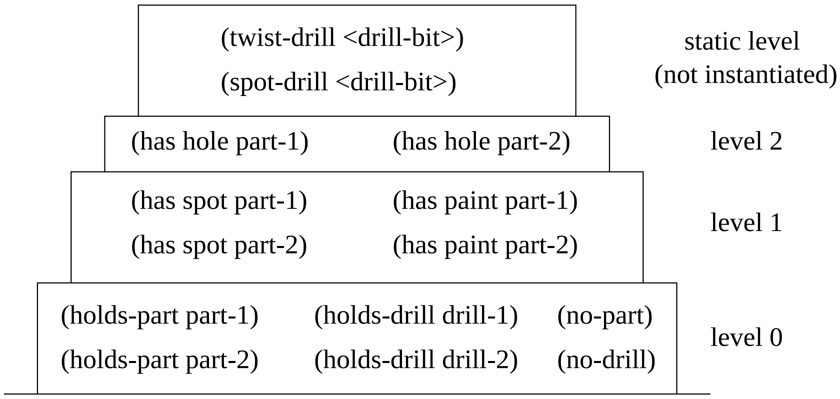
To reduce the system's dependency on the human user, we have developed a description changer that improves the granularity of *Abstractor*'s hierarchy. It identifies predicates that cause unnecessary constraints, and replaces them with more specific predicates. We give an informal overview of this technique (Section 4.3.1), describe the data structures and algorithms for improving the quality of ordered hierarchies (Sections 4.3.1–4.3.4), and then present a procedure for determining the levels of given literals in the improved hierarchy (Section 4.3.5).

4.3.1 Improving the granularity

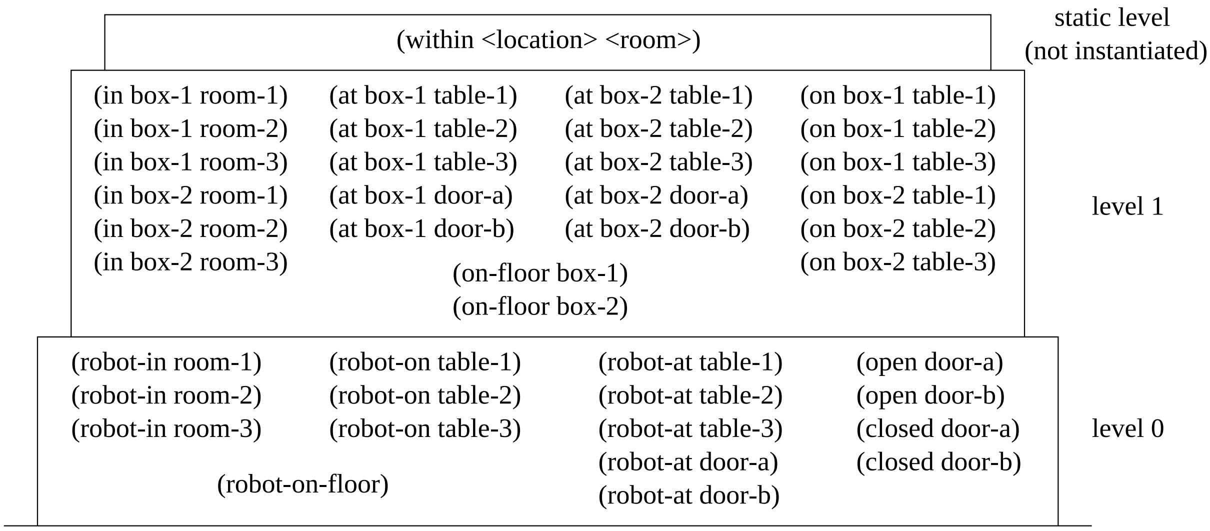
We begin with a review of previous techniques for increasing the number of abstraction levels, which were used with the ALPINE algorithm. Then, we introduce the new technique, implemented in the SHAPER system.

Full and partial instantiation

After Knoblock noted that general predicates may cause a collapse of ALPINE's hierarchy, he found two alternatives for preventing this problem. The first approach is based on generating *all* possible instances of nonstatic predicates. The system invokes an instantiation procedure, similar to the *Matcher* algorithm (see Section 3.4.2), and then builds a hierarchy of the resulting literals.



(a) Abstraction for the Drilling Domain.



(b) Abstraction for the Extended Robot world.

Figure 4.16: Examples of fully instantiated abstraction hierarchies. The first hierarchy is for Drilling problems with two drill bits and two parts. The other is for the Robot world in Figure 3.31(a) (page 131), which includes two boxes, three rooms, two doors, and three tables.

For example, if we apply this procedure to the Drilling Domain with the general predicate has (see Figure 4.7a), it may generate the hierarchy shown in Figure 4.16(a). As another example, its application to the Robot world in Figure 3.31 (page 131) would lead to a three-level hierarchy, given in Figure 4.16(b).

The instantiation technique completely eliminates unnecessary constraints caused by general predicates; however, it has two drawbacks that may lead to a major inefficiency in large domains. First, the system faces a combinatorial explosion in the number of literals, which may result in prohibitive time and space requirements for processing an abstraction graph. Second, it cannot produce a problem-independent abstraction, because object instances vary across problems. For example, the hierarchy in Figure 4.16(b) is not suitable for solving problems with three boxes.

The alternative approach is to "instantiate" predicates with low-level types, that is, with leaves of the type hierarchy. For instance, we may replace the nonstatic predicates at and robot-at in the Robot Domain with four more specific predicates (see Figure 4.17b), and then build the abstraction hierarchy given in Figure 4.17(c). We call this technique a *partial instantiation* of general predicates.

To give another illustration of this technique, suppose that we apply it to the Logistics Domain, introduced in Section 3.7.3 (see Figure 3.47 on page 147). Recall that problems in this domain involve the delivery of packages among locations in different cities. The system will replace the two nonstatic predicates, at and in, with eight more specific predicates (Figure 4.18b), and then generate a four-level hierarchy (Figure 4.18c).

This partial-instantiation procedure takes much less time and space than the construction of a literal hierarchy, but it is *not* immune to explosion in the number of predicates. Furthermore, the instantiation of predicates with leaf types does not always prevent a collapse of abstraction levels. For example, if we apply it to the Drilling Domain with the predicate (has <feature> <part>), then *Abstractor* will produce the three-level hierarchy in Figure 4.7(b). To build a four-level hierarchy, the system would have to replace <feature> with specific instances.

Minimal partial instantiation

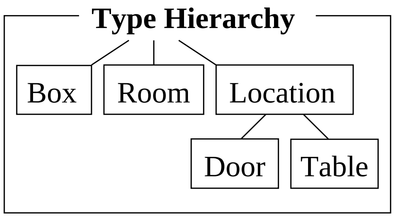
We have illustrated a major trade-off in constructing ordered hierarchies: general predicates allow fast generation and compact storage of an abstraction graph, whereas a full or partial instantiation helps to produce a finer-grained hierarchy. The SHAPER system includes an algorithm for finding the right partial instantiation, which prevents unnecessary constraints without causing a combinatorial explosion (see the specification in Figure 4.20). We named this algorithm *Refiner*, for its ability to refine the generality of predicates in a domain description.

The algorithm constructs a hierarchy of literals, but it does *not* represent literals explicitly. The encoding of the abstraction graph comprises sets of literals, subset relationships between them, and constraints on their abstraction levels (see Figure 4.21); we call it an *instantiation graph*. The encoding size is proportional to the total number of preconditions and effects of all operators in the domain.

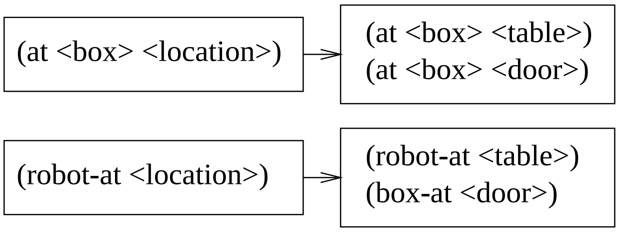
The system analyzes the abstraction graph and generates the minimal partial instantiation of predicates that prevents over-constraining of the hierarchy. For example, the application of this technique to the Drilling, Robot, and Logistics Domains leads to constructing the hierarchies in Figure 4.19. The implemented procedure is suitable not only for generating problem-independent hierarchies, but also for problem-specific abstraction (see Section 5.3). We describe the encoding of an instantiation graph, basic operations for modifying the graph, and their use in constructing a hierarchy of literals.

4.3.2 Instantiation graph

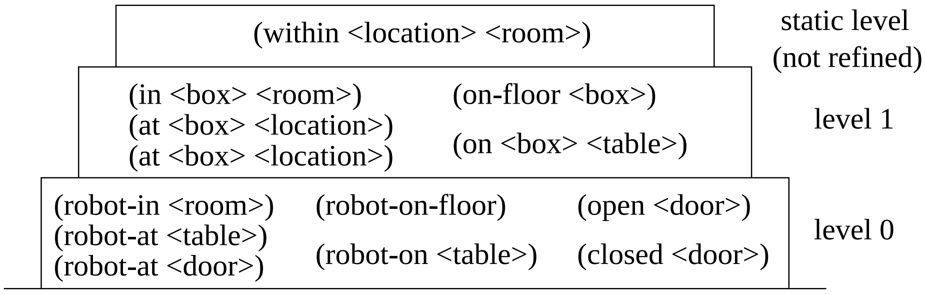
The structure of instantiation graphs is similar to constraint graphs described in Section 4.2.2 (see the example in Figure 4.14). The nodes of a graph are sets of nonstatic literals, and the edges are constraints on their relative importance, which determine abstraction levels. The



(a) Types of objects.

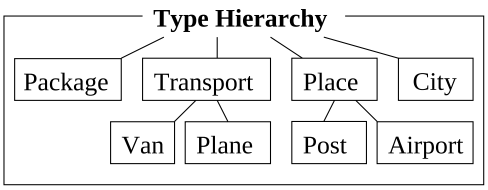


(b) Refinement of nonstatic predicates.

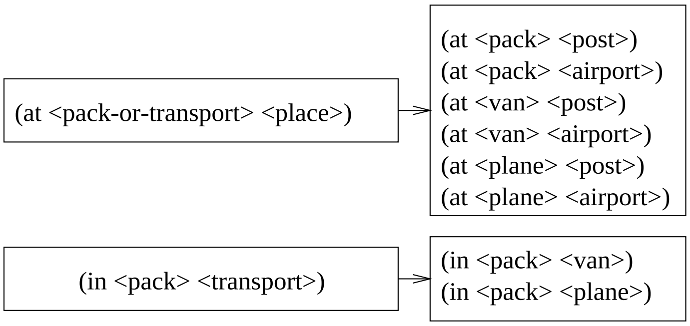


(c) Herarchy of refined predicates.

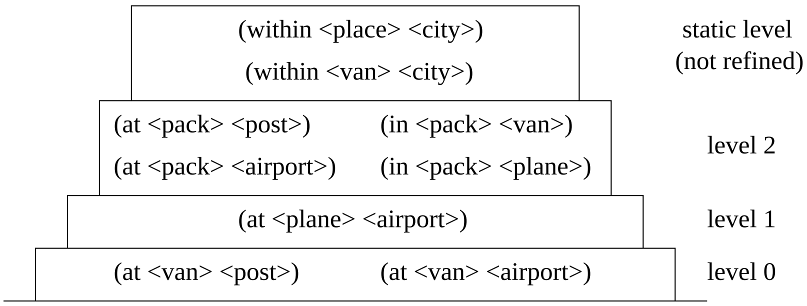
Figure 4.17: Partial instantiation in the Extended Robot Domain (Figure 3.31): The system "instantiates" nonstatic predicates with leaf types, and then arranges them into a hierarchy.



(a) Types of objects.

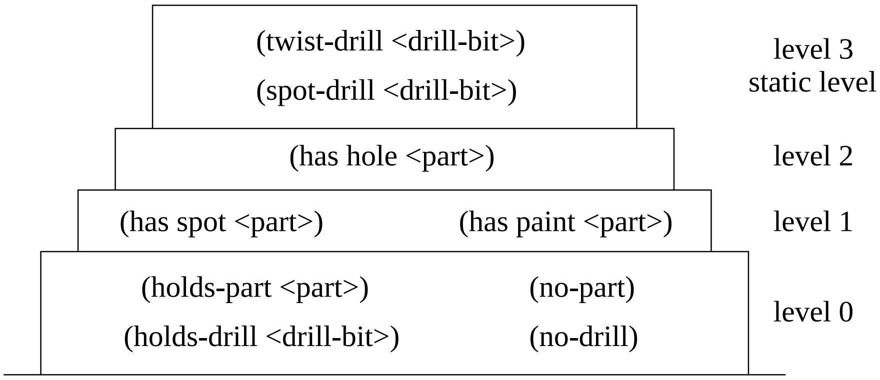


(b) Refinement of predicates.

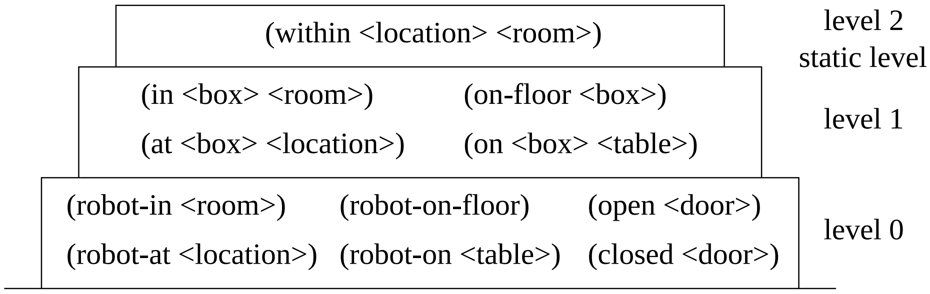


(c) Abstraction hierarchy.

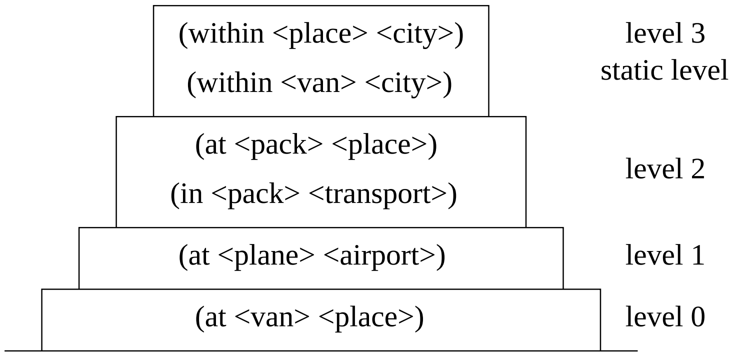
Figure 4.18: Partial instantiation in the Logistics Domain, given in Figure 3.47 (page 147).



(a) Drilling abstraction.



(b) Robot abstraction.



(c) Logistics abstraction.

Figure 4.19: Results of applying the *Refiner* algorithm to (a) Drilling Domain with the predicate `(has <feature> <part>)`, (b) Extended Robot world, and (c) Logistics Domain. The system generates the minimal instantiation of predicates that prevents a collapse of abstraction levels.

Type of description change: Generating more specific predicates.

Purpose of description change: Maximizing the number of levels in *Abstractor*'s ordered hierarchy, while avoiding too specific instantiations.

Use of other algorithms: The *Abstractor* algorithm, which generates abstraction hierarchies based on partially instantiated predicates.

Required input: Description of the operators and inference rules.

Optional input: Selection of primary effects; instance lists for some variables.

Figure 4.20: Specification of the *Refiner* algorithm.

main difference is that literal sets in an instantiation graph may not be disjoint, that is, a given literal may belong to several nodes. The graph encoding consists of two main parts (see Figure 4.21): a collection of literal sets, with links denoting subset relationships, and a graph of strongly connected components, with constraint edges between them.

Sets of literals

The nodes of an instantiation graph are *typed predicates*, which represent literal sets (see Figure 4.21a). A node is defined by a predicate name and list of argument types, where each element of the list is a simple type, disjunctive type, or specific object instance. Recall that a *disjunctive type* is a union of several simple types, which encompasses the instances of all these simple types (see Section 2.3.3). For example, the first argument of the predicate (at <pack-or-transport> <place>) is the disjunctive type (or Package Transport).

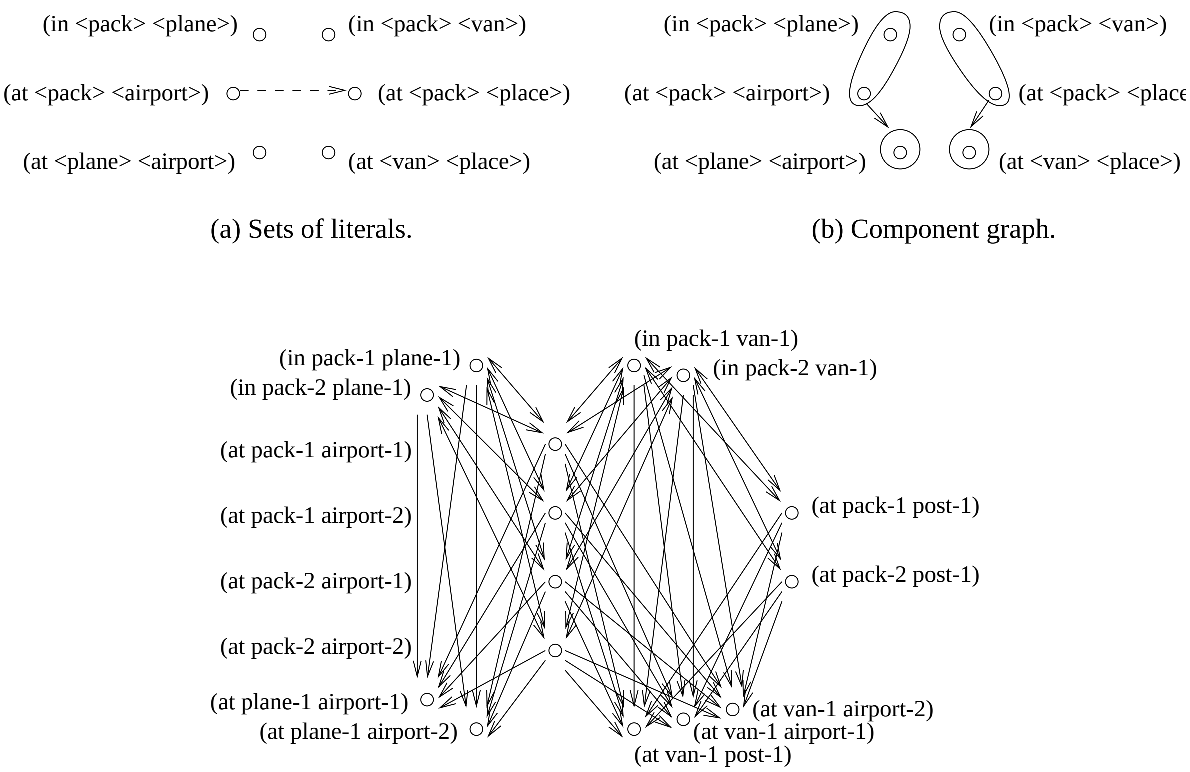
We view a typed predicate as the set of its full instantiations, and define a subset relationship and intersection of predicates by analogy with sets. Specifically, a predicate is a subset of another one if every possible instantiation of the first predicate is also an instantiation of the second. For example, (has hole <part>) is a subset of (has <feature> <part>) (see the Drilling Domain), and (at <pack> <airport>) is a subset of (at <pack> <place>) (Logistics Domain). Similarly, two predicates intersect if they have common instantiations, that is, some literals match both of them. For example, the predicate (at <transport> <airport>) intersects with (at <van> <place>).

When *Refiner* generates a graph, it identifies all subset relationships among nodes and marks them by directed links. For example, the graph in Figure 4.21(a) includes a subset link from (at <park> <airport>) to (at <park> <place>), shown by a dashed arrow.

Components and constraints

The edges of the graph determine relative positions of typed predicates in the abstraction hierarchy, and strongly connected components correspond to abstraction levels. The system constructs the hierarchy by a sequence of modifications to a *component graph* (see Figure 4.21b), which consists of connected components and constraint edges between them.

The *Refiner* algorithm initially creates a separate component for every node of the instantiation graph, and then searches for cycles of constraint edges. When the algorithm



(a) Sets of literals.

(b) Component graph.

(c) Example of an implicit literal graph.

Figure 4.21: Instantiation graph for the Logistics Domain. The encoding of the graph includes (a) nonstatic typed predicates with subset links between them, and (b) a graph of components with constraint edges. It corresponds to (c) an implicit graph of constraints among fully instantiated predicates. The dashed line shows a subset link, whereas solid lines are constraint edges.

identifies a cycle, it merges all components that form this cycle, that is, replaces them with a single large component. We will further describe the procedure for combining components in Section 4.3.4.

We view the component graph as a compact representation of an implicit *literal graph*, whose nodes include all instances of nonstatic predicates. This implicit graph of literals has an edge from l_1 to l_2 if the instantiation graph includes an edge from one of the components that contain l_1 to some component containing l_2. For example, suppose that the Logistics world consists of two airports, one postal office, one airplane, one van, and two packages. Then, the component graph in Figure 4.21(b) corresponds to the literal graph in Figure 4.21(c).

If some typed predicate is in a cycle of constraint edges, then the literal graph includes paths from every instance of this predicate to every other instance; hence, all its instances must be on the same abstraction level. The system detects predicates that belong to constraint cycles, called *looped predicates,* and uses this information to simplify the component graph. On the other hand, if a predicate is *not* looped, then *Refiner* may distribute its instances among several levels of abstraction.

Encoding of directed graphs

The described data structure includes two directed graphs: a graph of subset links, which is acyclic and transitively closed, and a component graph, whose edges are ordered-monotonicity constraints. We represent each directed graph by both adjacency matrix and adjacency lists; the reader may find an explanation of these structures in the text by Cormen *et al.* [1990]. Moreover, for each node we store two adjacency lists, which point to its incoming and outgoing edges.

The adjacency matrix enables *Refiner* to check the presence of an edge between two given nodes in constant time, whereas the lists support a fast identification of all nodes that are adjacent to a specified node. The size of this data structure is proportional to the squared number of typed predicates in the instantiation graph.

4.3.3 Basic operations

We next describe three basic procedures on a graph of predicates: verifying a subset relationship between two typed predicates, checking whether given predicates intersect, and propagating constraints from a predicate to its subset. The *Refiner* algorithm invokes the first two procedures to initialize an instantiation graph, and applies the third operation during the following simplification of the graph.

To simplify the description of these algorithms, we assume that all argument types are disjunctive, that is, we consider simple types to be one-element disjunctions. Furthermore, if a type disjunction includes specific object instances along with simple types, then we view each instance as a simple type, which can be instantiated with only one object. In Figure 4.22, we summarize the notation for main elements of the instantiation graph.

pred	typed predicate, which serves as a node of the instantiation graph
disj-type	disjunctive type, which is an argument type in some predicate
simple-type	simple type or object instance, which is part of a disjunctive type
compt	strongly connected component of the instantiation graph
compt[*pred*]	component that contains a given predicate *pred*
$(compt_1, compt_2)$	constraint edge from the first to the second component

Figure 4.22: Notation in the pseudocode of the *Refiner* algorithm, in Figures 4.23, 4.25, and 4.26.

Subset and intersection tests

We summarize the procedures for checking the subset relationship and intersection of two typed predicates in Figure 4.23. The first procedure is called *Predicate-Subset* (see the left column in Figure 4.23), and the other is *Predicate-Intersection* (see the first two functions in the right column).

The subset-test algorithm verifies that given predicates have the same name and equal number of arguments, and that every argument type in the first predicate is a subset of the corresponding type in the second predicate. The intersection test also compares argument types; specifically, it verifies that every type in the first predicate intersects with the corresponding type in the second one.

Observe that both algorithms have to test the subtype relationship between simple types. To speed-up this operation, the system pre-computes the transitive closure of the type hierarchy and stores it as an adjacency matrix, which allows a constant-time subtype test.

The overall running time of *Predicate-Subset* and *Predicate-Intersection* depends on the number a of arguments in the given predicates, as well as on the number of simple types in each disjunction. If the maximal length of a type disjunction is d, then the complexity of both procedures is $O(a \cdot d^2)$.

Inheritance of constraint edges

Consider two typed predicates, $pred_1$ and $pred_2$, that belong to distinct components, and suppose that $pred_1$ is a subset of $pred_2$ (see Figure 4.24a). Then, we may copy all incoming and outgoing edges of $pred_2$'s component to $pred_1$'s component, as shown in Figure 4.24(b), *without affecting the structure of the implicit literal graph.* We say that $pred_1$ *inherits* the constraint edges of $pred_2$.

The insertion of inherited edges helps to simplify the component graph, without over-constraining the abstraction hierarchy. We present a procedure for performing this operation, called *Copy-Edges*, in Figure 4.23. Its running time is proportional to the number of incoming and outgoing edges of $pred_2$'s component.

4.3.4 Construction of a hierarchy

The system builds an abstraction hierarchy in three steps, summarized in Figure 4.25: (1) generating an initial graph, which consists of nonstatic typed predicates and constraints

Predicate-Subset($pred_1, pred_2$)
Return true if $pred_1$ is a subset of $pred_2$, that is, every instantiation of $pred_1$ is also an instantiation of $pred_2$.

If $pred_1$ and $pred_2$ have different names,
  or different number of arguments,
 then return false.

Let a be the number of arguments in $pred_1$ and $pred_2$.
Repeat for i from 1 to a:
 Let $disj\text{-}type_1$ be the ith argument type in $pred_1$,
  and $disj\text{-}type_2$ be the ith type in $pred_2$.
 If not $Disj\text{-}Type\text{-}Subset(disj\text{-}type_1, disj\text{-}type_2)$,
  then return false.
Return true.

Disj-Type-Subset($disj\text{-}type_1, disj\text{-}type_2$)
Check whether $disj\text{-}type_1$ is a subtype of $disj\text{-}type_2$.

For every $simple\text{-}type_1$ in $disj\text{-}type_1$:
 If not $Simple\text{-}Type\text{-}Subset(simple\text{-}type_1, disj\text{-}type_2)$,
  then return false.
Return true.

Simple-Type-Subset($simple\text{-}type_1, disj\text{-}type_2$)
Check whether $simple\text{-}type_1$ is a subtype of $disj\text{-}type_2$.

For every $simple\text{-}type_2$ in $disj\text{-}type_2$:
 If either $simple\text{-}type_1$ and $simple\text{-}type_2$ are identical
   or $simple\text{-}type_1$ is a subtype of $simple\text{-}type_2$,
  then return true.
Return false.

Predicate-Intersection($pred_1, pred_2$)
Return true if $pred_1$ intersects $pred_2$, that is, some instantiation of $pred_1$ is also an instantiation of $pred_2$.

If $pred_1$ and $pred_2$ have different names,
  or different number of arguments,
 then return false.

Let a be the number of arguments in $pred_1$ and $pred_2$.
Repeat for i from 1 to a:
 Let $disj\text{-}type_1$ be the ith argument type in $pred_1$,
  and $disj\text{-}type_2$ be the ith type in $pred_2$.
 If not $Disj\text{-}Type\text{-}Intersection(disj\text{-}type_1, disj\text{-}type_2)$,
  then return false.
Return true.

Disj-Type-Intersection($disj\text{-}type_1, disj\text{-}type_2$)
Check whether $disj\text{-}type_1$ intersects $disj\text{-}type_2$, that is, whether these types have common instances.

For every $simple\text{-}type_1$ in $disj\text{-}type_1$:
 For every $simple\text{-}type_2$ in $disj\text{-}type_2$:
  If $simple\text{-}type_1$ and $simple\text{-}type_2$ are identical,
    or $simple\text{-}type_1$ is a subtype of $simple\text{-}type_2$,
    or $simple\text{-}type_2$ is a subtype of $simple\text{-}type_1$,
   then return true.
Return false.

Copy-Edges($compt_1, compt_2$)
Ensure that the first component inherits all incoming and outgoing edges of the second component.

For every incoming edge ($other\text{-}compt, compt_2$):
 If $other\text{-}compt$ and $compt_1$ are distinct,
   and there is no edge ($other\text{-}compt, compt_1$),
  then add this edge it to the graph.

For every outgoing edge ($compt_2, other\text{-}compt$):
 If $other\text{-}compt$ and $compt_1$ are distinct,
   there is no edge ($compt_1, other\text{-}compt$),
  then add this edge to the graph.

Figure 4.23: Basic operations on the instantiation graph: a subset test for typed predicates, a similar intersection test, and a function for propagating constraints in the component graph.

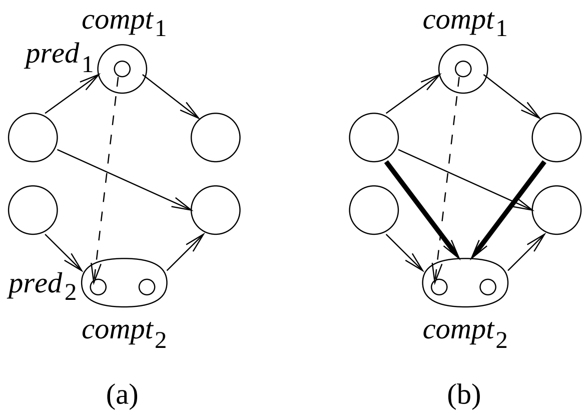


Figure 4.24: Propagation of constraint edges: If a typed predicate $pred_1$ is a subset of a predicate $pred_2$, then the component of $pred_1$ inherits all edges from the component of $pred_2$. We show the inherited constraints by thick lines.

Refiner
Build the initial constraint graph, whose nodes are typed predicates (Figure 4.27a).
Call *Identify-Subsets* (Figure 4.27b) and then *Initialize-Components* (Figure 4.27c).
Repeat the computation until *Add-Subset-Edges* returns false:
  Call *Combine-Components* and then *Add-Subset-Edges* (Figure 4.27d–i).

Figure 4.25: Building a hierarchy of partially instantiated predicates. The algorithm generates a constraint graph of nonstatic predicates, determines subset relationships between predicates, and finds strongly connected components.

on their levels, (2) adding subset links and initializing the component graph, and (3) identifying the strongly connected components. We give pseudocode for the main procedures in Figure 4.26 and illustrate the construction in Figure 4.27.

Initial constraint graph

The first step of *Refiner* is similar to that of *Abstractor* (Section 4.2.2): it creates a graph of typed predicates and adds constraint edges for all operators and inference rules (see the algorithm in Figure 4.11 and the illustration of constraints in Figures 4.12, on pages 162 and 163). The nodes of the initial graph correspond to effects and nonstatic preconditions of operators. For example, the effects of **load-van** in the Logistics Domain (see Figure 3.47) give rise to nodes (at <pack> <place>) and (in <pack> <van>).

The graph may have *cyclic* edges, which point from a node to itself. Note that, if some typed predicate *pred* has a cyclic edge (*pred*, *pred*), then all instances of this predicate must be on the same level of abstraction. In Figure 4.27(a), we show the initial constraint graph for the Logistics Domain.

Identify-Subsets
For every two distinct predicates, $pred_1$ and $pred_2$, in the instantiation graph:
  Call *Predicate-Subset*($pred_1, pred_2$) and *Predicate-Subset*($pred_2, pred_1$),
    to test the subset relationship between these two predicates.
  If both tests return true ($pred_1$ and $pred_2$ are identical):
    Make $pred_1$ inherit all incoming and outgoing edges of $pred_2$.
    If there is an edge between $pred_1$ and $pred_2$, in either direction,
      then add the cyclic edge ($pred_1, pred_1$).
    Remove $pred_2$ from the instantiation graph.
  Else, if $pred_1$ is a subset of $pred_2$:
    Add a subset link from $pred_1$ to $pred_2$.
  Else, if $pred_2$ is a subset of $pred_1$:
    Add a subset link from $pred_2$ to $pred_1$.
  Else, if *Predicate-Intersection*($pred_1, pred_2$):
    Add two constraint edges: from $pred_1$ to $pred_2$ and from $pred_2$ to $pred_1$.
    (These constraints ensure that $pred_1$ and $pred_2$ will be in the same component.)

Initialize-Components
For every predicate *pred* in the graph:
  Create a one-element component containing *pred*.
  If the graph includes the cyclic constraint edge (*pred*, *pred*),
    then mark *pred* as a looped predicate.
For every constraint edge ($pred_1, pred_2$) in the graph:
  If $pred_1$ and $pred_2$ are distinct (it is not a cyclic edge),
    then add the corresponding edge from $compt[pred_1]$ to $compt[pred_2]$.

Combine-Components
Identify strongly connected components of the constraint graph (Figure 4.27d,g),
  and combine old components into the resulting new components (see Figure 4.27e,h).
For every predicate *pred* in the graph:
  If *compt*[*pred*] contains more than one predicate,
    then mark *pred* as looped (it is in some loop of constraint edges).

Add-Subset-Edges
For every predicate *pred* in the graph:
  For every predicate *sub-pred* that is a subset of *pred*:
    If *pred* and *sub-pred* are in the same component:
      Remove *sub-pred* from the graph (Figure 4.27i).
    Else, if *pred* is looped:
      Add the edges (*compt*[*pred*], *compt*[*sub-pred*]) and (*compt*[*sub-pred*], *compt*[*pred*]).
      (Thus, ensure that *pred* and *sub-pred* will be in the same component; see Figure 4.27f).
    Else:
      Call *Copy-Edges*(*compt*[*sub-pred*], *compt*[*pred*]).
      If there is an edge between *compt*[*pred*] and *compt*[*sub-pred*]), in either direction,
        then mark *sub-pred* as a looped predicate.
Return true if the function has added at least one new edge, and false otherwise.

Figure 4.26: Subroutines of the *Refiner* algorithm, given in Figure 4.25.

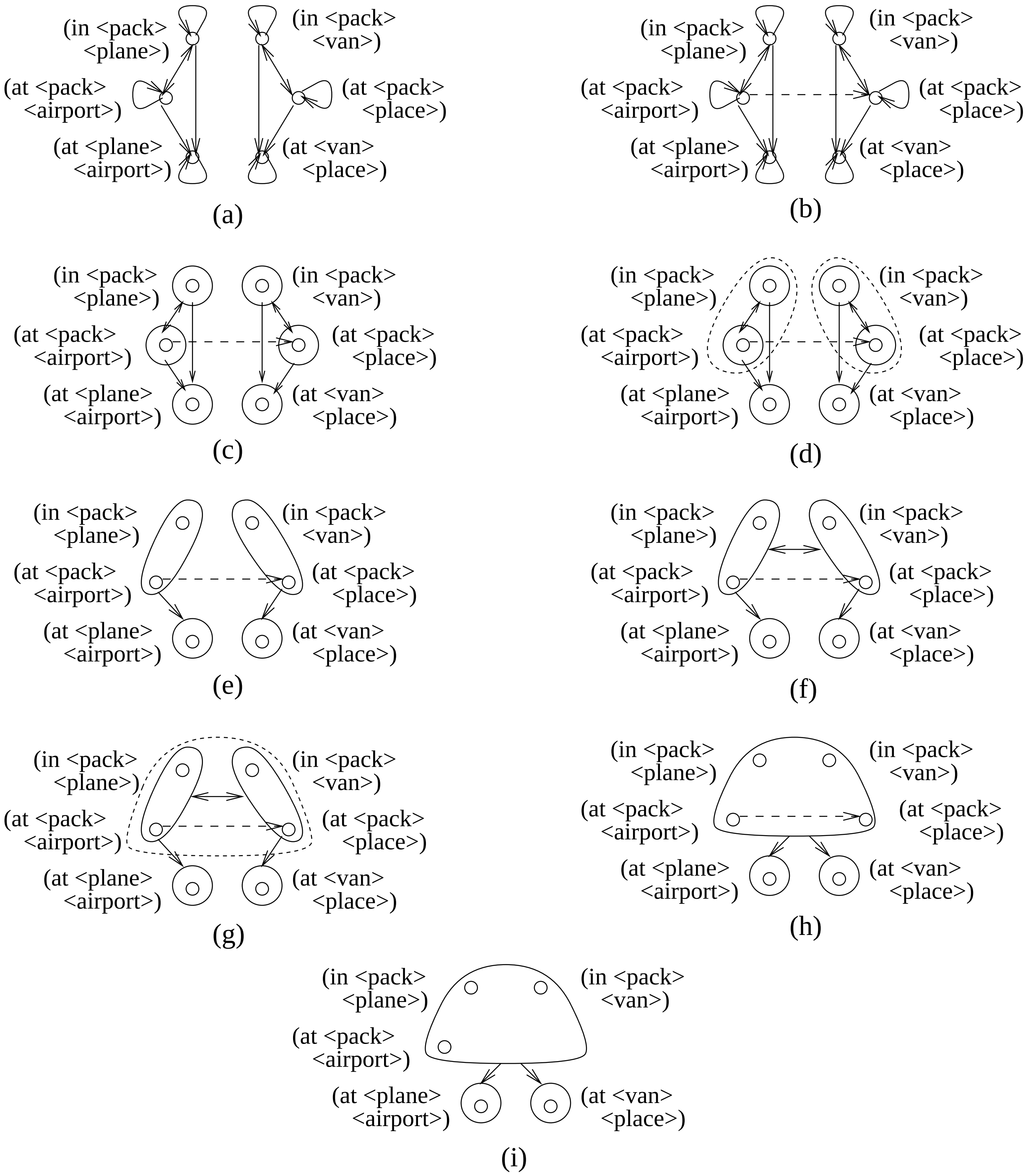


Figure 4.27: Application of *Refiner* to the Logistics Domain; we show the constraint edges by solid lines, and the subset link by dashes. The algorithm (a) generates an initial graph, (b) adds subset links, (c) makes one-element components, (d–h) combines strongly connected components, and (i) removes redundant predicates.

Subsets and initial components

The next step is identification of subsets and intersections among typed predicates (see the *Identify-Subsets* procedure in Figure 4.26). We apply the *Predicate-Subset* and *Predicate-Intersection* test to every pair of nodes in the graph, and add the appropriate subset links. In Figure 4.27(b), we show the link added to the Logistics graph.

If the system finds a pair of intersecting predicates, neither of which is a subset of the other, then it inserts a two-way constraint edge between them, thus ensuring that they are on the same abstraction level. This operation *may* over-constrain the hierarchy, but it rarely causes a collapse of levels. It did *not* induce any unnecessary constraints in the experimental domains.

After completing the subset and intersection tests, *Refiner* initializes the component graph, by creating a separate component for every predicate (see *Initialize-Components* in Figure 4.26). The algorithm does *not* copy cyclic edges to the new graph; instead, it marks the nodes with these edges as looped predicates. In Figure 4.27(c), we give the initial component graph for the Logistics domain, where all predicates are looped.

Strongly connected components

The final stage of the construction is finding the connected components of the instantiation graph (see Figure 4.26d–i). The system repetitively applies two procedures, *Combine-Components* and *Identify-Subsets*, summarized in Figure 4.26. The first procedure identifies the strongly connected components, formed by constraint edges (Figure 4.27d), and modifies the component graph (Figure 4.27e).

The other procedure iterates trough the typed predicates and propagates constraints to their subsets. If *pred* is a looped predicate, then the algorithm adds two-way constraints between *pred* and its subsets, thus ensuring that they are on the same level (see Figure 4.26f). Otherwise, it calls *Copy-Edges* to add the appropriate constraints for *pred*'s subsets.

Running time

The construction of an initial graph (Step 1 in Figure 4.27) has the same complexity as the similar step of *Abstractor*. Its running time is usually $O(E+P)$, where E is the total number of effects in all operators and inference rules, and P is the total number of preconditions and if-effect conditions; however, if some inference rules have no primary effects, then the time complexity may be superlinear (see Section 4.2.1 for more details). When we run the implemented algorithm on a Sun 5 machine, the execution of Step 1 takes about $(P+E) \cdot 14 \cdot 10^{-4}$ seconds.

The time of Step 2 depends on three factors: the number N of typed predicates in the initial graph, the maximal number a of arguments in a predicate, and the maximal length d of a type disjunction. The complexity of the *Identify-Subsets* procedure is $O(a \cdot d^2 \cdot N^2)$, and that of *Initialize-Components* is $O(N^2)$. In practice, these two procedures together take up to $O(2 \cdot a \cdot d^2 + 5) \cdot N^2 \cdot 10^{-4}$ seconds.

Finally, we observe that the algorithm makes at most N iterations at Step 3, because every iteration reduces the number of components. The running time of *Combine-Components* is

$O(N^2)$, and the time of *Add-Subset-Edges* is $O(N^3)$; hence, the worst-case complexity of Step 3 is $O(N^4)$. We conjecture that its complexity can be improved, by a more efficient propagation of constraint edges. We have not investigated this problem because empirical results have proved better than the worst-case estimate. The execution time of Step 3 is usually proportional to N^2, specifically, it varies from $N^2 \cdot 10^{-3}$ to $N \cdot 3 \cdot 10^{-3}$ seconds.

4.3.5 Level of a given literal

When PRODIGY performs a hierarchical search, it uses the component graph to determine the levels of instantiated preconditions of operators and inference rules. For every precondition literal, the system finds a matching typed predicate in the graph, and looks up the level of its component.

If a literal matches several typed predicates, the system selects the *most specific* predicate among them, which is a subset of all other matching predicates. For example, if (at <pack> <airport>) and (at <pack> <place>) were in different components, then it would determine the level of the literal (at pack-1 airport-1) by the component of (at <pack> <airport>).

We present a data structure for a fast computation of literal levels, illustrated in Figure 4.28, and outline procedures for constructing and using this structure.

Sorting the typed predicates

The system constructs a sorted array of predicate names, where each entry points to a list of all typed predicates with the corresponding name. For example, the at entry in Figure 4.28 points to a list of two predicates, (at <plane> <airport>) and (at <van> <place>).

If all predicates in a list belong to the same component, the system replaces them with a more general predicate, defined by name without argument types. For example, it replaces (in <pack> <plane>) and (in <pack> <van>) in the Logistics graph with the generalized predicate in. On the other hand, if predicates with a common name are in several different components, the system topologically sorts their list according to the subset relationship; that is, if $pred_1$ is a subset of $pred_2$, then $pred_1$ precedes $pred_2$ in the sorted list.

In Figure 4.29, we give the procedure for generating an array of predicate lists, called *Sort-Predicates*. Its running time depends on the number N of nonstatic typed predicates, and on the number S of subset links. The worst-case complexity of *Sort-Predicates* is $O(N \cdot \lg N + S)$, and its execution time on a Sun 5 is up to $(2 \cdot N + 3 \cdot S) \cdot 10^{-5}$ seconds.

After applying *Sort-Predicates*, the system sorts and enumerates the components of the instantiation graph, which gives rise to an abstraction hierarchy (see Figure 4.28). Finally, it adds the static level and inserts static predicates into the array.

Finding and hashing components

The *Find-Component* procedure, summarized in Figure 4.29, looks up the component of a given literal. If the graph includes only one predicate with the matching name, the look-up time is $O(\lg N)$; in practice, it takes up to $3 \cdot 10^{-5}$ seconds.

On the other hand, if the procedure has to search through a list of several predicates with this name, then it incurs additional cost, which depends on the length L of the list, as

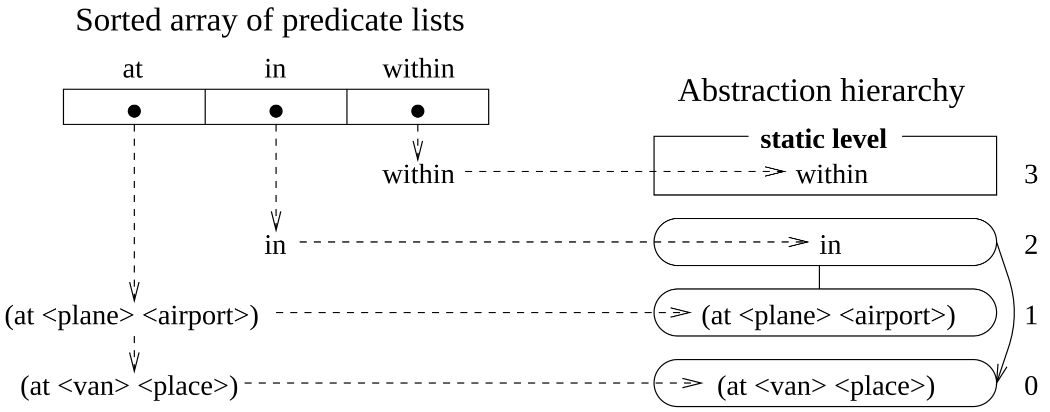


Figure 4.28: Data structure for determining the abstraction levels of literals. The system constructs a sorted array of predicate names, where each entry contains a list of typed predicates with a common name. Every predicate points to its component in the instantiation graph, and the component's number indicates the abstraction level.

Sort-Predicates
For every predicate name in the instantiation graph:
  If all predicates with this name are in the same component:
    Replace them with a single generalized predicate,
      which is defined by name without argument types.
    Create a one-element list containing the generalized predicate.
  Else (these predicates are in several different components):
    Topologically sort them according to the subset relationship.
    Store the resulting list of sorted predicates
      (for every two predicates, if $pred_1$ is a subset of $pred_2$,
      then $pred_1$ is before $pred_2$ in the sorted list).
Create an array of the generated lists
  (each list contains all predicates with a certain name).
Sort the lists into the alphabetical order of names (see Figure 4.28).

Find-Component(l)
The literal l must be an instance of some predicate in the graph.

Identify the list of predicates with the matching name.
If it contains a single predicate *pred*, then return *compt*[*pred*].
For every *pred* from the list, in the subset-to-superset order:
  If *Predicate-Subset*(l, *pred*), then return *compt*[*pred*].

Figure 4.29: Identifying the components of given literals. The first procedure constructs an array of predicates, illustrated in Figure 4.29, and the other uses the array to look up a literal's component.

well as on the number a of the literal's arguments, and on the maximal length d of a type disjunction in the predicate list. The complexity of this additional computation is $O(a \cdot d \cdot L)$, and its empirical time is within $a \cdot d \cdot L \cdot 5 \cdot 10^{-5}$ seconds.

We have also implemented a procedure for hashing the output of *Find-Component*, which improves the efficiency of repeated look-ups. The system creates a separate hash table for every predicate list that has more than one element. For example, it will create a table for the list of `at` predicates in the Logistics Domain. After finding the component of a given literal, the system stores it in the corresponding table, using the literal as a key. The time for adding or retrieving a hash entry varies from 10^{-5} to $3 \cdot 10^{-5}$ seconds.

4.4 Performance of the abstraction search

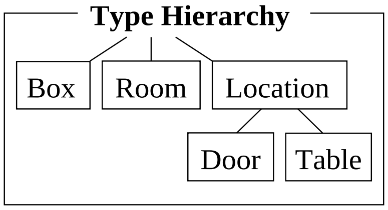
We have tested the efficiency of abstraction search in the PRODIGY system, using the same five domains as in the experiments with primary effects (see Section 3.7). The *Abstractor* algorithm constructed multi-level hierarchies for the Robot world, Sokoban puzzle, and Logistics world; however, it failed in the other two domains. The resulting abstraction improved the performance in the first two domains, and gave mixed results in the Logistics Domain.

Extended Robot Domain

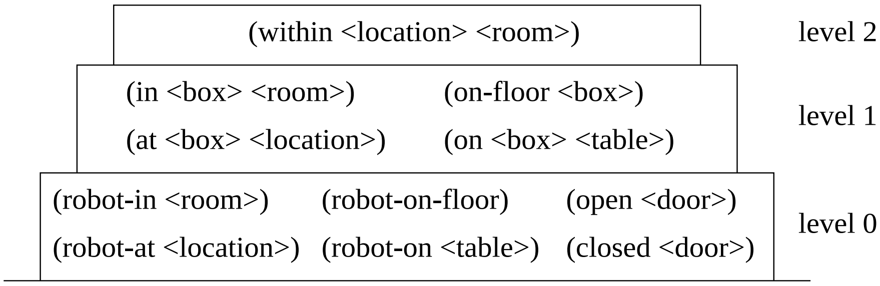
We first describe results in the Extended Robot world, exported from the ABTWEAK system (see Figure 3.31 in page 131). If the *Abstractor* algorithm does not use primary effects, then it fails to construct a multi-level hierarchy for this domain. On the other hand, if the algorithm utilizes the automatically selected effects, then it generates the abstraction hierarchy given in Figure 4.30. The top level of this hierarchy comprises the static information, the middle level contains the predicates that specify the locations of boxes, and the low level includes the robot's position and the state of the doors.

We tested PRODIGY on the nine problems listed in Table 3.7 (page 134). For every problem, we first ran the search algorithm without a cost bound, and then with two different bounds. The loose cost bound was twice the optimal solution cost, whereas the tight bound equaled the optimal cost. The abstraction improved the efficiency of solving the four problems that require moving boxes; it did *not* affect the search on the other five problems. In Table 4.1, we give the times for solving the box-moving problems, without and with abstraction; note that the search algorithm utilized primary effects in both cases. The time-reduction factor in these experiments varied from 1 to almost 10000.

The use of the abstraction hierarchy did *not* affect solution quality. In all cases, the system generated the same solutions without and with abstraction. When the system ran without time bounds, it produced optimal solutions to problems 1–6 and near-optimal solutions to the other three problems. We show the lengths of the resulting solutions in Figure 4.31(a), and the solution costs in Figure 4.31(b). The loose cost bounds did not improve the quality: when the system ran with these bounds, it generated the same solutions to all problems. Finally, search with the tight bounds yielded optimal solutions to problems 1–8 and failed on problem 9.



(a) Types of objects.



(b) Abstraction hierarchy.

Figure 4.30: Abstraction hierarchy in the Extended Robot Domain. The *Abstractor* algorithm generated this hierarchy for problem solving with primary effects, given in Figure 3.31.

#	No Cost Bound		Loose Bound		Tight Bound	
	w/o abs	with abs	w/o abs	with abs	w/o abs	with abs
4	0.16	0.13	0.17	0.14	0.16	0.15
7	2.67	1.26	2.49	1.31	3.43	1.56
8	> 1800.00	0.19	> 1800.00	0.21	> 1800.00	1660.54
9	> 1800.00	0.26	> 1800.00	0.26	> 1800.00	> 1800.00

Table 4.1: Performance in the Extended Robot Domain, on the four problems that involve the moving of boxes (see Table 3.7 on page 134 for the list of problems). We give running times in seconds, for search without abstraction ("w/o abs") and with the automatically generated abstraction ("with abs"). For every problem, we first ran PRODIGY without a cost bound, and then with two different bounds.

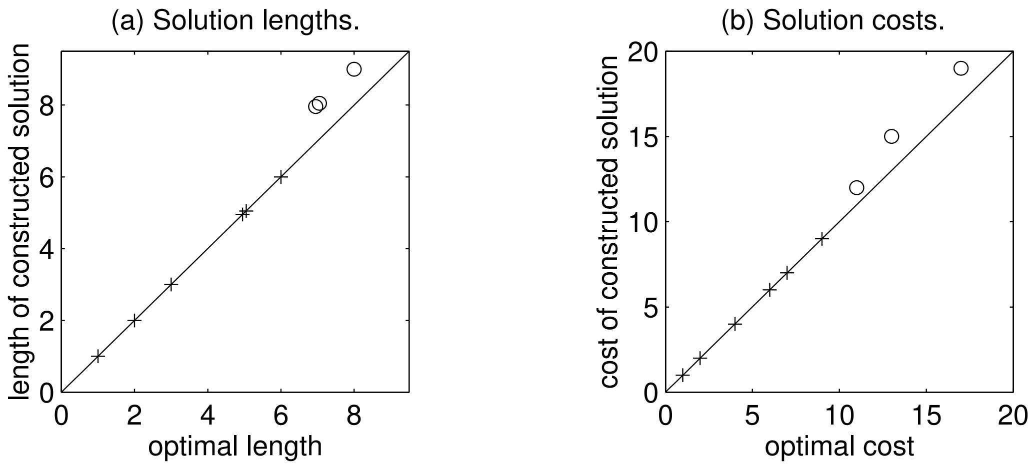


Figure 4.31: Solution quality in the Extended Robot Domain. The system generated optimal solutions to six problems, denoted by pluses (+), and constructed near-optimal solutions to the other three problems, marked by circles (o).

Sokoban Domain

The Sokoban puzzle has been described in Section 3.7.2 (see Figures 3.36 and 3.37). We now give the results of abstraction search in this domain. The application of the *Abstractor* algorithm yielded a three-level hierarchy, given in Figure 4.32. Note that *Abstractor* has to utilize primary effects for generating this abstraction. If the algorithm has no information about primary effects, then it fails to construct a hierarchy.

We applied the system to the same 320 problems as in the experiments of Section 3.7.2, with no cost bounds. The results of abstraction search are summarized in Figure 4.32 (see the dashed lines). We also show the system's performance with primary effects and *no* abstraction (solid lines), as well as the results of problem solving without primary effects (dotted lines).

The use of abstraction increased the percentage of solved problems, especially for larger grids, and did not affect the solution quality. The time-reduction factor varied from 1 to greater than 100. We also experimented with 600-second time bounds, but this ten-fold increase in search time had little effect on the percentage of solved problems. Specifically, the system solved five more problems with abstraction, and three more problems without abstraction.

Logistics Domain

We next give results in the Logistics Domain (see Figure 3.47 on page 147), designed by Veloso [1994] during her work on the analogical reasoning in the PRODIGY system. Logistics problems have proved difficult for general-purpose problem solvers, and the domain has become a traditional benchmark for testing PRODIGY and other AI systems.

The operators in this domain do *not* have unimportant effects; hence, the *Chooser* and *Completer* algorithms do not improve the system's performance. On the other hand, *Abstractor* constructs a four-level hierarchy (see Figure 4.34) and reduces the search complexity.

We have tested the resulting abstraction on five hundred problems of various complexity, with different numbers of cities, vans, airplanes, and packages. We first experimented without cost bounds and then with heuristically computed bounds. The computation of cost bounds was based on the number of packages and cities, and the resulting bounds were near-tight: they ranged from the tight bounds to 50% greater than tight.

The results of problem solving without cost bounds are summarized Figure 4.35, and those with the use of heuristic bounds are given in Figure 4.36. We show the percentage of problems solved by different time bounds, without an abstraction hierarchy (solid lines) and with the four-level hierarchy (dashed lines). In Figure 4.37, we give two different graphs for the same results, which show the dependency between the problem size and the percentage of unsolved problems.

When we did not limit solution costs, the abstraction noticeably increased the percentage of solved problems; the time-reduction factor varied across problems, ranging from 1 to greater than 100. On the other hand, hierarchical search gave very little advantage in the experiments with cost bounds: it reduced the running time in some cases and increased it in other cases; the percentage of problems solved without abstraction was close to that with abstraction.

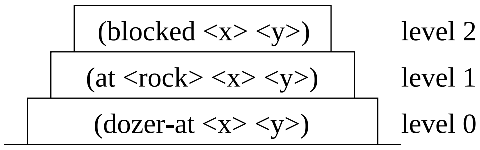


Figure 4.32: Abstraction in the Sokoban Domain, for the primary effects given in Figure 3.37.

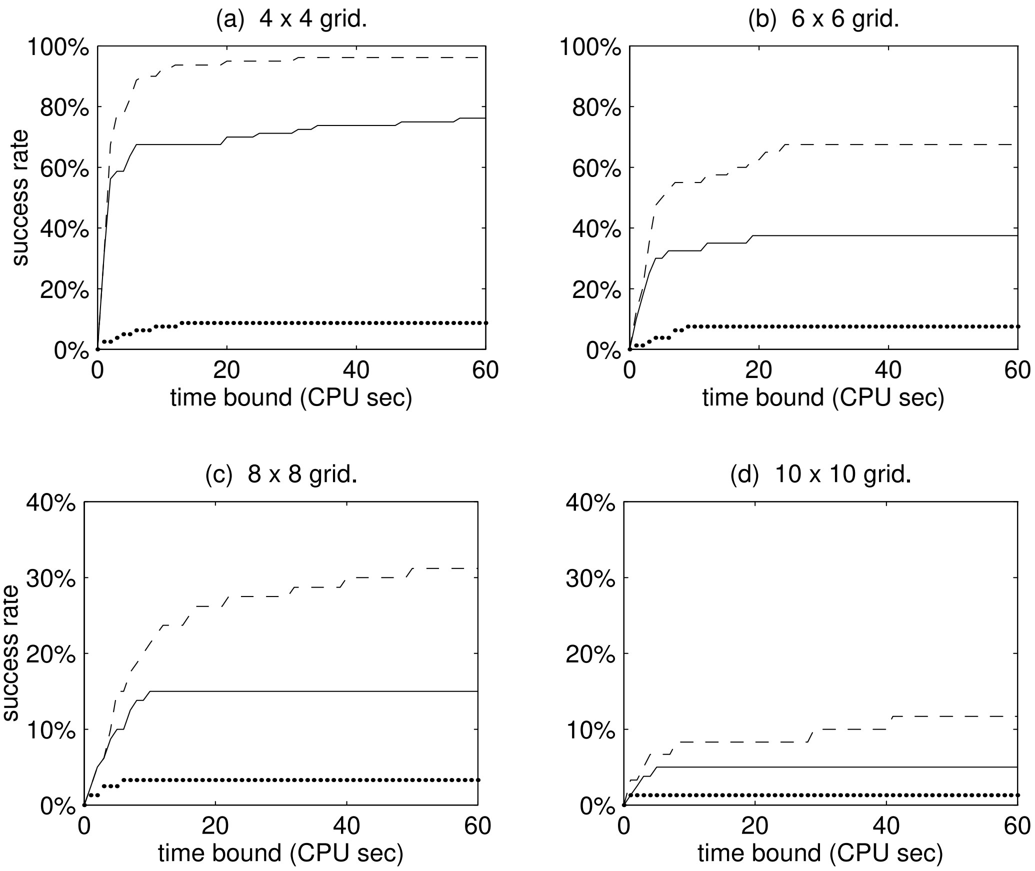
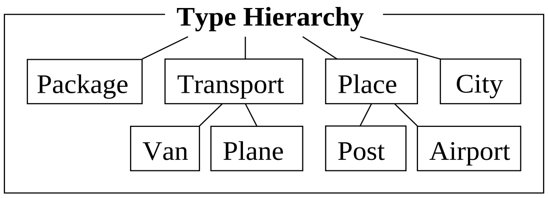
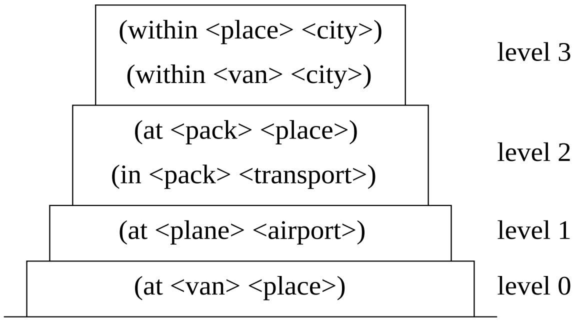


Figure 4.33: Performance in the Sokoban Domain. We plot the percentage of problems solved by different time bound, in three series of experiments: with primary effects and abstraction hierarchy (dashed lines); with primary effects and *no* abstraction (solid lines); and without primary effects (dotted lines).



(a) Types of objects.



(b) Abstraction hierarchy.

Figure 4.34: Abstraction hierarchy in the Logistics Domain.

The use of the abstraction hierarchy had *no* significant effect on solution quality. We give the comparison of solution lengths and costs in Figures 4.38 and 4.39, where solid lines show the solution quality without abstraction, and the dashed lines give the results of hierarchical search. Note that this comparison involves only the problems that were solved both with and without abstraction.

Summary

The empirical results have confirmed that ordered abstraction hierarchies are an effective tool for reducing search. Thus, if the *Abstractor* algorithm constructs a hierarchy, it usually improves efficiency. On the negative side, the algorithm often fails to generate a multi-level hierarchy. The efficiency improvement in our experiments is similar to that in Knoblock's [1994] experiments with ALPINE, and to the results of using abstraction in the ABTWEAK system [Yang *et al.*, 1996].

We summarize the experimental results in Table 4.2, where the upward arrows ($\Uparrow$) denote efficiency improvement, the two-way arrow ($\Updownarrow$) indicates mixed results, and dashes (–) mark the domains that do not allow the construction of ordered hierarchies.

The automatically generated hierarchies improved the system's performance in the Extended Robot Domain and Sokoban Domain. When we used abstraction for solving Logistics problems without cost bounds, it also improved the efficiency. On the other hand, when we used it with near-tight bounds, the results varied from a hundred-fold reduction to a hundred-fold increase of the search time.

Even though abstraction *may* increase solution costs, it did *not* have this effect in any of the test domains. On the other hand, it did not improve the quality either. This result is different from Knoblock's [1993] experiments, which showed a moderate quality improvement in most domains.

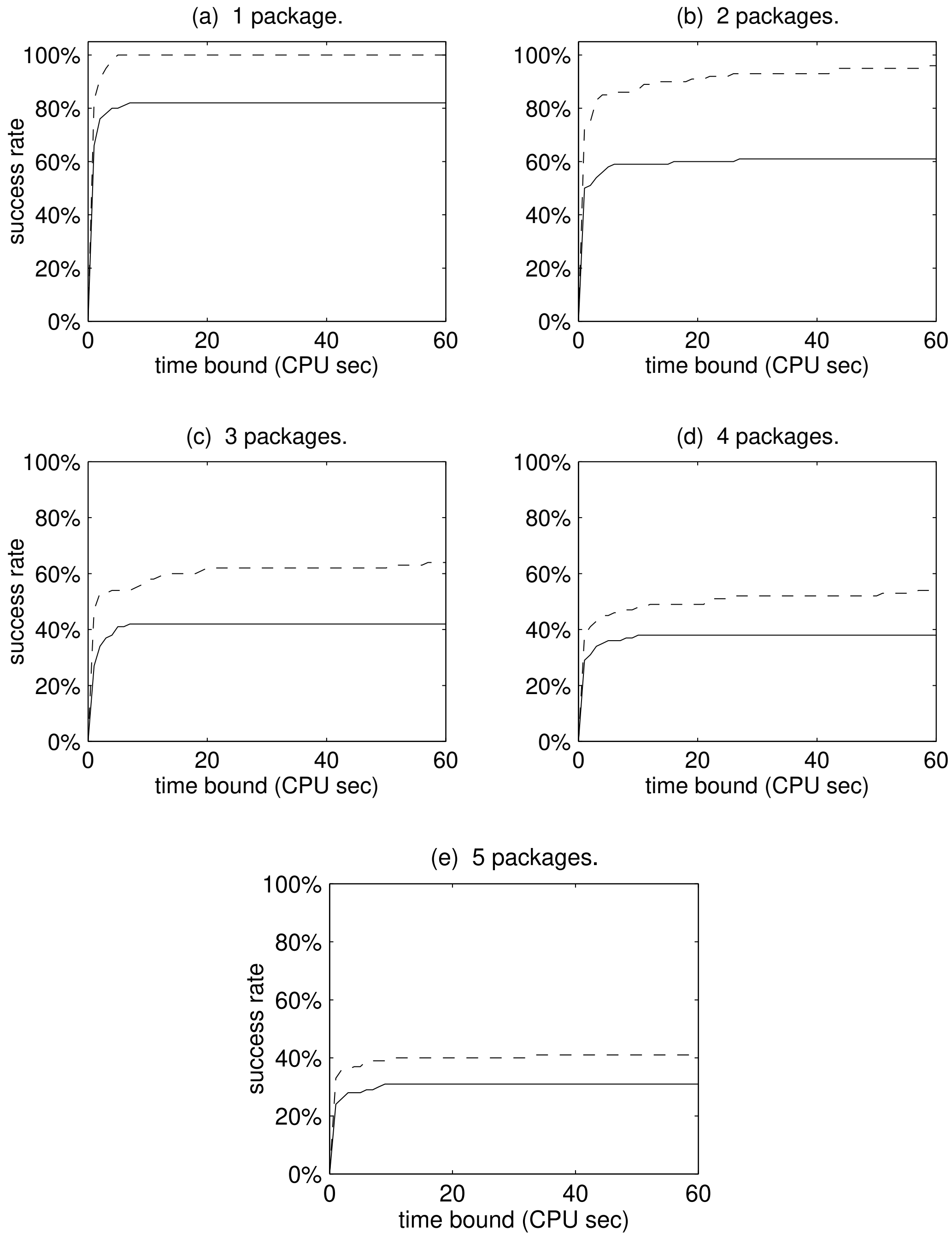


Figure 4.35: Performance in the Logistics Domain, *without cost bounds.* We show the percentage of problems solved by different time bounds, without an abstraction hierarchy (solid lines) and with the automatically generated abstraction (dashed lines).

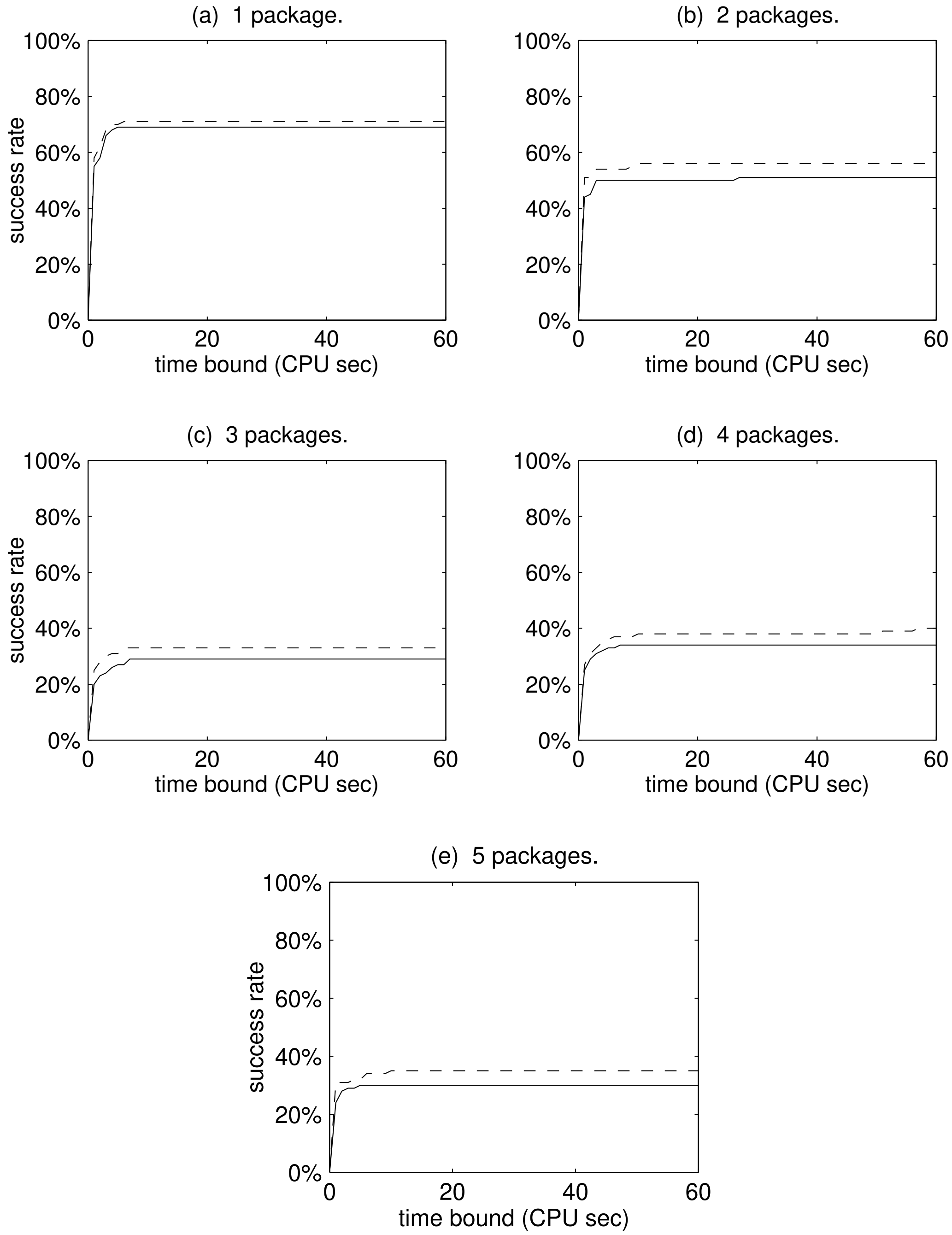


Figure 4.36: Performance in the Logistics Domain, *with heuristically computed cost bounds.* We give results without abstraction (solid lines) and with the use of abstraction (dashed lines).

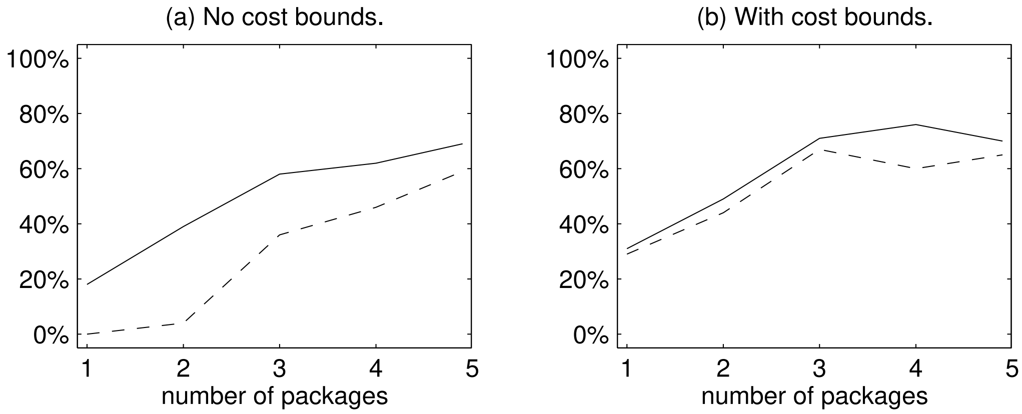


Figure 4.37: Percentage of Logistics problems that were *not* solved within 60 seconds. We show this percentage for search without abstraction (solid lines) and with the use of the automatically generated hierarchy (dashed lines).

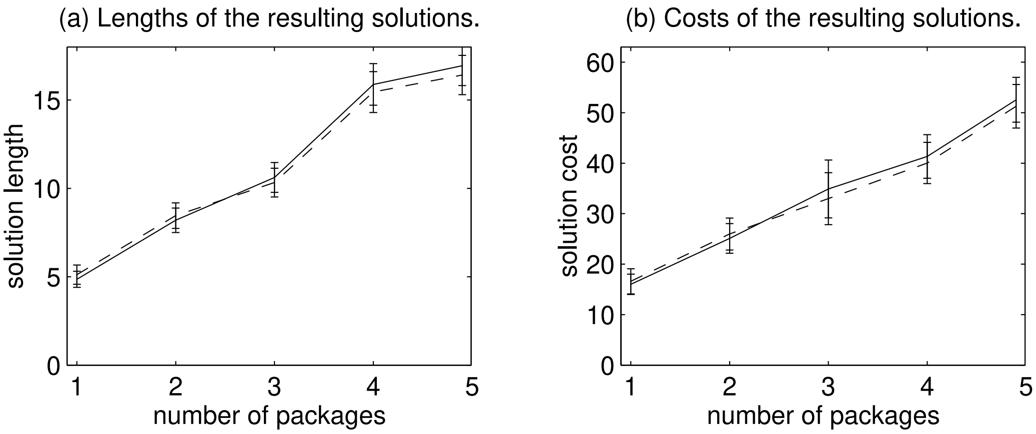


Figure 4.38: Quality of Logistics solutions generated *without cost bounds.* We plot the dependency of solution lengths and costs on the number of delivered packages, for problem solving without abstraction (solid lines) and with the use of abstraction (dashed lines).

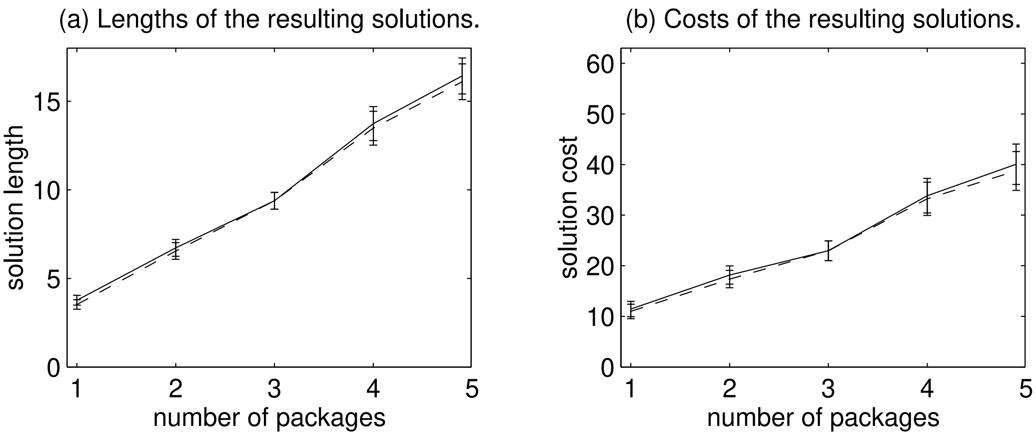


Figure 4.39: Solution quality in the Logistics experiments *with cost bounds.*

Domain	Overall Result	Search-Time Reduction	Solution-Length Reduction	Solution-Cost Reduction
without primary effects				
Extended Robot	–	—	—	—
Machining	–	—	—	—
Sokoban	–	—	—	—
Extended Strips	–	—	—	—
Logistics	$\Updownarrow$	0.01–100	none	none
using primary effects				
Extended Robot	$\Uparrow$	1–10000	none	none
Machining	–	—	—	—
Sokoban	$\Uparrow$	1–100	none	none
Extended STRIPS	–	—	—	—

Table 4.2: Summary of problem solving with abstraction hierarchies. The upward arrow ($\Uparrow$) indicates that abstraction improved the performance, whereas the two-way arrow ($\Updownarrow$) marks the domain with mixed results. When we used the *Abstractor* algorithm without primary effects, it constructed a multi-level hierarchy only in the Logistics Domain. The utility of this hierarchy varied across problems: it reduced the search time in some cases, and increased it in other cases. When we applied *Abstractor* with the automatically selected primary effects, it generated hierarchies for two other domains, and significantly improved the performance in both domains.

Chapter 5

Other enhancements

We describe two procedures for enhancing the power of abstraction and give empirical evidence of their utility. The first technique is aimed at choosing primary effects that improve the quality of an abstraction hierarchy. We outline a heuristic for selecting appropriate effects, which is based on a synergy of *Chooser* with *Abstractor* (Section 5.1), and show that it reduces search in the Machining Domain and Extended STRIPS world (Section 5.2).

The second technique enables the system to construct specialized domain descriptions for a given problem, which are often more effective than general-purpose descriptions (Section 5.3). We show that specialized primary effects and abstraction improve the efficiency in three out of the five test domains (Section 5.4).

5.1 Abstracting the effects of operators

The procedure for abstracting effects is a result of joint work with Yang, aimed at improving performance of the ABTWEAK system. The underlying idea is to increase the number of abstraction levels, by choosing appropriate primary effects [Fink and Yang, 1992a]. We have extended it to the domain language of PRODIGY4, and implemented a heuristic for selecting primary effects, called *Margie*, which has been one of $\mathcal{S}$HAPER's description changers.

Recall that, if a domain encoding has *no* primary effects, then the *Abstractor* algorithm assigns the same importance level to all effects of an operator or inference rule (see Section 4.1.5). Intuitively, the algorithm may abstract some preconditions of an operator, but does not abstract its effects. If the system first invokes *Chooser* and *Completer*, it may be able to abstract side effects of operators and inference rules; however, the resulting selection of primary effects does not always support the construction of a multi-level hierarchy. For example, if we apply *Chooser* to the Machining Domain, it generates the selection in Figure 5.5(a), which leads to the collapse of a hierarchy.

We may often remedy the situation by modifying the choice of primary effects, without compromising the completeness of search. For instance, if *Abstractor* utilizes the modified selection in Figure 5.5(b), then it generates a two-level hierarchy (see Figure 5.5c). As another example, the application of *Chooser* and *Completer* in the Extended STRIPS Domain leads to the primary effects in Figures 3.41 and 3.42 (pages 143 and 144), which do not allow ordered abstraction. On the other hand, the modification illustrated in Figure 5.10 gives rise

Type of description change: Selecting primary effects of operators and inference rules.

Purpose of description change: Maximizing the number of levels in *Abstractor*'s ordered hierarchy, while ensuring near-completeness and a limited cost increase.

Use of other algorithms: The *Abstractor* algorithm, which builds hierarchies for the selected primary effects.

Required input: Description of the operators and inference rules.

Optional input: Preferable cost-increase limit C; pre-selected primary and side effects.

Figure 5.1: Specification of the *Margie* algorithm.

to the hierarchy in Figure 5.11.

The purpose of the *Margie* algorithm is to choose primary effects that improve the quality of abstraction. It inputs the encoding of operators and inference rules, and may optionally utilize pre-selected primary effects and a desirable cost increase (see the specification in Figure 5.1). Its implementation is based on the *Chooser* algorithm, combined with a version of *Abstractor* that produces hierarchies for intermediate selections of primary effects.

Recall that, when *Chooser* picks an operator for achieving some predicate, it may have to select among several matching operators (see the *Choose-Initial* procedure in Figure 3.14, page 101). Similarly, when choosing an additional primary effect of some operator, it may need to select among several candidate effects (see *Choose-Extra* in Figure 3.14).

The *Margie* algorithm prefers the choices that maximize the number of abstraction levels. When selecting among matching operators or among candidate effects, it generates an abstraction graph for each alternative, and makes the choice that results in the maximal number of components. If several choices lead to the same number, the system prefers the graph with fewer constraint edges, thus reducing the chances of a collapse after selecting more effects.

To sum up, *Margie* consists of the *Chooser* algorithm and the two procedures in Figure 5.2, *Select-Step* and *Select-Effect*, which serve as *Chooser*'s selection heuristics. Note that it does not ensure the completeness of search with primary effects; hence, we need to call *Completer* after the execution of *Margie*.

For example, consider the application of the *Margie* algorithm to the extended Tower-of-Hanoi Domain, which comprises the six operators given in Figures 1.5(b) and 1.9(a) (see pages 15 and 20). First, the algorithm invokes the *Choose-Initial* procedure, which marks all effects of **move-small**, **move-medium**, and **move-large** as primary. After *Margie* makes these choices, the abstraction graph is as shown in Figure 5.3(a). Second, the *Choose-Extra* subroutine picks primary effects of the other three operators: **move-sml-mdm**, **move-sml-lrg**, and **move-med-lrg** (see Figures 1.9a).

When *Choose-Extra* processes **move-sml-mdm**, the choice is between "add small" and "add medium." If "add small" is selected as a primary effect, then *Abstractor* must ensure that $level(\text{small}) \geq level(\text{medium})$; hence, it adds a new constraint edge (see Figure 5.3b), which reduces the number of components. On the other hand, if "add medium" is primary, then *Abstractor* does not add new constraints to the graph. Since the goal is to maximize the

Select-Step(*Steps*, *pred*)
The procedure inputs a list of operators and inference rules, denoted *Steps*, with a common candidate effect *pred*. It chooses an operator or rule that should achieve *pred* as a primary effect.

$step_{\text{best}}$:= none (selected operator or inference rule)
$n\text{-}compts_{\text{best}} := 0$ (corresponding number of components in the abstraction graph)
$n\text{-}edges_{\text{best}} := 0$ (corresponding number of constraint edges between components)
For every operator and inference rule *step* in *Steps*:
  Call *Evaluate-Choice*(*step*, *pred*) to determine corresponding *n-compts* and *n-edges*
  If *Better-Choice*($n\text{-}compts, n\text{-}edges, n\text{-}compts_{\text{best}}, n\text{-}edges_{\text{best}}$)
    then $step_{\text{best}} := step$
      $n\text{-}compts_{\text{best}} := n\text{-}compts$
      $n\text{-}edges_{\text{best}} := n\text{-}edges$
Return $step_{\text{best}}$

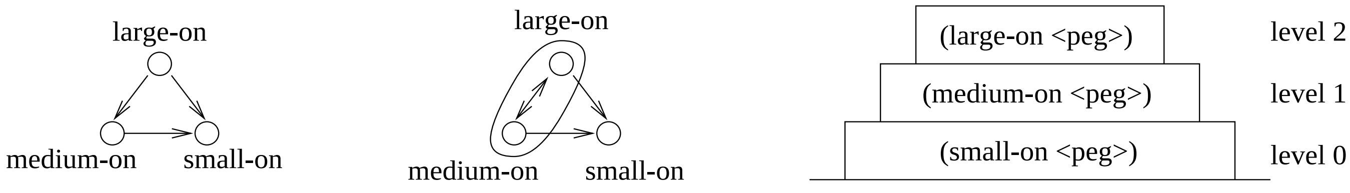
Select-Effect(*step*, *Preds*)
The procedure inputs an operator or inference rule, *step*, and a list of its candidate effects, *Preds*. It chooses an effect that should be primary.

$pred_{\text{best}}$:= none (selected effect)
$n\text{-}compts_{\text{best}} := 0$
$n\text{-}edges_{\text{best}} := 0$
For every effect *pred* in *Preds*:
  Call *Evaluate-Choice*(*step*, *pred*) to determine corresponding *n-compts* and *n-edges*
  If *Better-Choice*($n\text{-}compts, n\text{-}edges, n\text{-}compts_{\text{best}}, n\text{-}edges_{\text{best}}$)
    then $pred_{\text{best}} := pred$
      $n\text{-}compts_{\text{best}} := n\text{-}compts$
      $n\text{-}edges_{\text{best}} := n\text{-}edges$
Return $pred_{\text{best}}$

Evaluate-Choice(*step*, *pred*)
Temporarily promote *pred* to a primary effect of *step*.
Call *Abstractor* to generate a hierarchy for the current selection
  (it does not distinguish between candidate and side effects).
Let *n-compts* be the number of components in the resulting graph,
  and *n-edges* be the number of constraint edges between components.
Demote *pred* back to a candidate effect.
Return *n-compts* and *n-edges*.

Better-Choice($n\text{-}compts, n\text{-}edges, n\text{-}compts_{\text{best}}, n\text{-}edges_{\text{best}}$)
If either $n\text{-}compts > n\text{-}compts_{\text{best}}$,
    or $n\text{-}compts = n\text{-}compts_{\text{best}}$ and $n\text{-}edges < n\text{-}edges_{\text{best}}$,
  then return true.
Else, return false.

Figure 5.2: Selection heuristics for *Margie*, aimed to maximize the number of abstraction levels. Note that, since the *Margie* procedure invokes *Abstractor* to generate hierarchies for the selected primary effects, it produces an *ordered* abstraction hierarchy (see Section 4.1.5).



(a) Optimal graph. (b) Suboptimal graph. (c) Abstraction hierarchy.

Figure 5.3: Intermediate abstraction graphs (a,b) and final hierarchy (c) in the extended Tower-of-Hanoi Domain. The *Margie* algorithm uses intermediate graphs to evaluate alternative choices of primary effects, and prefers the choices that result in more levels.

number of components, *Margie* prefers the "add medium" effect. Then, it applies the same technique to **move-sml-lrg** and **move-med-lrg**, and chooses "add large" as a primary effect of both operators.

After running the *Margie* algorithm, we apply *Completer* to choose additional effects, and obtain the selection shown in Figure 1.9(c). Finally, we invoke *Abstractor*, which utilizes the resulting selection to construct a three-level hierarchy, given in Figure 5.3(c).

5.2 Evaluation of the enhanced abstraction

We have empirically evaluated the effectiveness of *Margie*, using the same simulated worlds as in the experiments with primary effects (Section 3.7) and abstraction (Section 4.4). For every world, we have compared the utility of three different domain descriptions, illustrated in Figure 5.4.

The first of these descriptions is the result of applying *Margie*, *Completer*, and *Abstractor* (see Figure 5.4a). When we apply the *Completer* algorithm for constructing this description, it utilizes *Margie* for choosing among candidate effects. The second description includes primary effects and abstraction generated without the *Margie* algorithm (see Figure 5.4b). Finally, the third one is the original domain encoding, which has no primary effects or abstraction.

The *Margie* algorithm improved performance in the Machining Domain and Extended STRIPS World. On the other hand, when we applied *Margie* in the other three domains, it selected the same primary effects as the *Chooser* algorithm; thus, it did *not* affect performance in these domains.

Machining domain

We first give the results for the Machining Domain, introduced in Section 3.6.2. If the system does not use *Margie* in this domain, then the selected primary effects do *not* allow the generation of an ordered abstraction hierarchy. We have shown this selection of effects in Figure 3.33 (page 133). We partially reproduce it in Figure 5.5(a), which contains the four operators that cause the collapse of the hierarchy.

On the other hand, if we apply the *Margie* algorithm, then it chooses the primary effects given in Figure 5.5(b), which allow the construction of a two-level hierarchy (see Figure 5.5c).

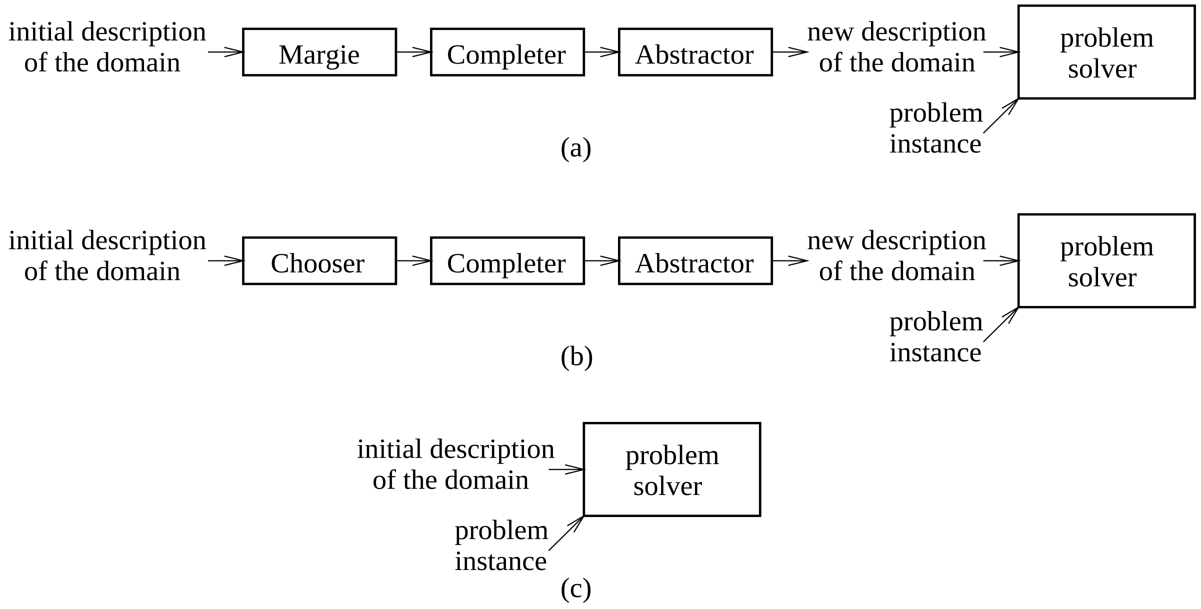


Figure 5.4: Evaluating the utility of the *Margie* algorithm. We compare three problem-solving strategies: (a) search with *Margie*'s abstraction hierarchy; (b) use of abstraction generated without *Margie*; and (c) search without primary effects and abstraction.

Note that the only difference from the previous selection is the choice of an operator for deleting (polished <part>).

In Figure 5.6, we give the results of problem solving without a cost bound. The graphs show the running time and solution quality in three series of experiments: using the two-level abstraction (see the dashed lines), with primary effects and *no* abstraction (solid lines), and without primary effects (dotted lines).

Abstraction improved the running time and enabled the system to find *optimal* solutions to all problems. The time-reduction factor ranged from 1.3 to 1.6, with a mean of 1.39, whereas the factor of solution-cost reduction was between 1.3 and 1.5, with an average of 1.42.

In Figure 5.7, we show the analogous results for search with loose cost bounds. Recall that the loose bound for a problem is equal to the doubled cost of the optimal solution. These bounds did *not* affect performance of abstraction search; however, they increased the problem-solving time in the other two cases (see Figure 5.7(a), where the time scale is *logarithmic*). For most problems, abstraction improved the search time by a factor of 1.3 to 2.0, and reduced the solution cost by a factor of 1.3 to 1.5.

We also ran the solver algorithm with tight cost bounds, thus forcing it to search for optimal solutions. The results of abstraction search were identical to that without cost bounds. On the other hand, when the system ran without abstraction, it did not find an optimal solution to any of the problems within the 600-second time limit.

cut-roughly(<part>)
Prim: add (cut <part>)
del (finely-cut <part>)
del (drilled <part>)
Side: del (finely-drilled <part>)
del (polished <part>)
del (finely-polished <part>)
del (painted <part>)
del (finely-painted <part>)

drill-roughly(<part>)
Prim: add (drilled <part>)
del (finely-drilled <part>)
del (polished <part>)
Side: del (finely-polished <part>)
del (painted <part>)
del (finely-painted <part>)

polish-roughly(<part>)
Prim: add (polished <part>)
del (finely-polished <part>)
del (painted <part>)
Side: del (finely-painted <part>)

paint-roughly(<part>)
Prim: add (painted <part>)
del (finely-painted <part>)
Side: del (polished <part>)
del (finely-polished <part>)

(a) Selection of primary effects without *Margie*.

cut-roughly(<part>)
Prim: add (cut <part>)
del (finely-cut <part>)
del (drilled <part>)
Side: del (finely-drilled <part>)
del (polished <part>)
del (finely-polished <part>)
del (painted <part>)
del (finely-painted <part>)

drill-roughly(<part>)
Prim: add (drilled <part>)
del (finely-drilled <part>)
Side: del (polished <part>)
del (finely-polished <part>)
del (painted <part>)
del (finely-painted <part>)

polish-roughly(<part>)
Prim: add (polished <part>)
del (finely-polished <part>)
del (painted <part>)
Side: del (finely-painted <part>)

paint-roughly(<part>)
Prim: add (painted <part>)
del (finely-painted <part>)
del (polished <part>)
Side: del (finely-polished <part>)

(b) Selection with the use of *Margie*.

(cut <part>)	(finely-cut <part>)	level 1
(drilled <part>)	(finely-drilled <part>)	
(polished <part>)	(finely-polished <part>)	level 0
(painted <part>)	(finely-painted <part>)	

(c) Abstraction hierarchy.

Figure 5.5: Primary effects of the low-quality operations in the Machining Domain. If we do not use *Margie*, then these operations cause the collapse of the ordered abstraction hierarchy. On the other hand, the effects selected by *Margie* allow the construction of a two-level hierarchy.

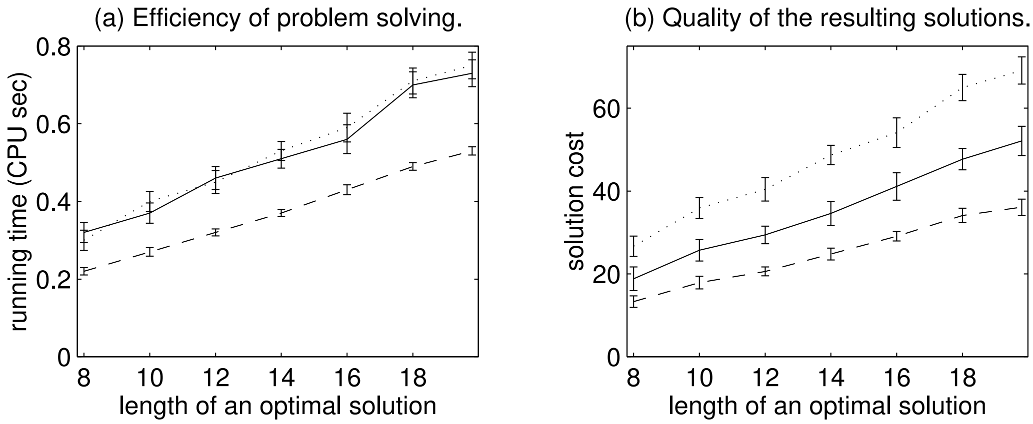


Figure 5.6: Performance in the Machining Domain, *without a cost bound.* We plot the results in three series of experiments: with *Margie*'s primary effects and abstraction hierarchy (dashed lines); with *Chooser*'s effects and *no* abstraction (solid lines); and without primary effects (solid lines). The vertical bars mark the 95% confidence intervals.

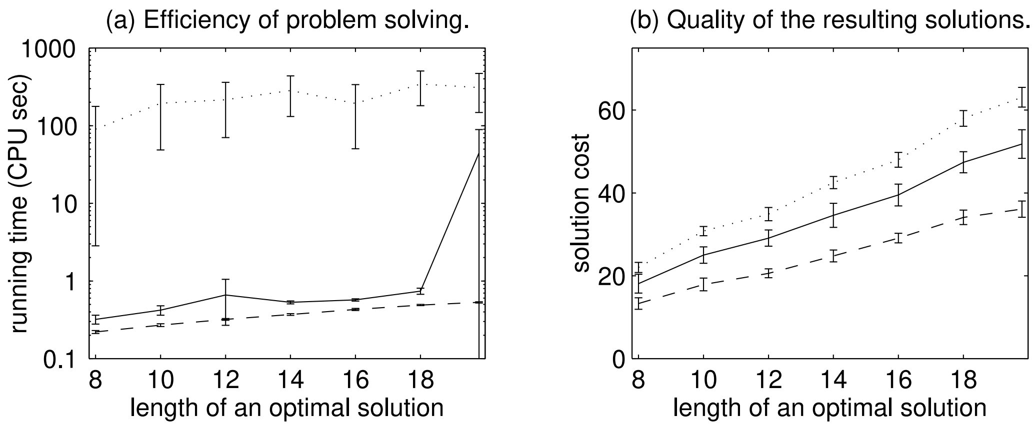


Figure 5.7: Performance in the Machining Domain, with abstraction (dashed lines), without abstraction (solid lines), and without primary effects (dotted lines). For every problem, we use a *loose cost bound,* which is twice the optimal solution cost.

Extended STRIPS domain

We have described the STRIPS world in Section 3.7.2, and demonstrated that the *Chooser* and *Completer* algorithms select appropriate primary effects for this domain; however, the resulting selection does *not* enable *Abstractor* to build a multi-level hierarchy. On the other hand, *Margie* constructs a slightly different selection, which leads to the generation of a four-level abstraction. We show the differences between the two selections in Figure 5.8 and the resulting hierarchy in Figure 5.9(b).

In Figure 5.10, we summarize the system's performance with the four-level hierarchy (dashed lines) and without abstraction (solid lines). The graphs show the results in three series of experiments: without cost bounds, with loose bounds, and with tight bounds. These results show that *Margie*'s hierarchy reduces the search time, regardless of the cost bound.

We give a different graphical summary of the same results in Figure 5.11, where every point denotes a problem. The horizontal axes show the search time with *Margie*'s abstraction, whereas the vertical axes give the search time without abstraction on the same problems. The plot includes not only the solved problems, but also the problems that caused an interrupt upon reaching a 600-second limit. The abstraction search is faster for most problems and, thus, most points are above the diagonal. The ratio of the search time without abstraction to that with abstraction varies from 0.1 to more than 1000.

When the solver ran without tight bounds, it yielded suboptimal solutions to most problems; however, the lengths and costs of the resulting solutions were within a factor of 1.7 from optimal. In Figure 5.12, we show the solution lengths in the experiments with abstraction, and compare them to that without abstraction. This comparison is based on the problems that were solved in both cases. In Figure 5.13, we give a similar plot for the solution costs.

The results show that the application of *Margie* leads to a small improvement in the average solution quality, even though it does not affect the quality for a majority of problems. The length-reduction factor ranges from 0.8 to 1.7, with a mean of 1.08, and the cost-reduction factor is between 0.9 and 1.5, with a mean of 1.06.

If we do *not* apply *Chooser* or *Margie*, and run the system without any selection of primary effects, then its performance is very poor. PRODIGY solves only 2% of the problems when searching with tight cost bounds, and no problems at all without tight bounds.

Summary

The experiments have demonstrated that the *Margie* algorithm enhances the effectiveness of the abstraction generator. Specifically, this algorithm enabled the system to construct hierarchies in the Machining Domain and Extended STRIPS world, which improved performance in both domains (see Table 5.1). On the other hand, *Margie* did *not* affect the system's behavior in the other three domains.

Note that *Margie*'s abstractions not only reduced the search time, but also led to a moderate improvement of solution quality. This result differs from the experiments with the *Abstractor* algorithm (see Section 4.4), which did not affect solution costs. On the other hand, the observed improvement confirms Knoblock's [1993] results, which showed that abstraction may reduce the solution length.

pick-up(<small>, <room>)
Prim: del (arm-empty)
del (in <small> <room>)
(forall <other> of type (or Thing Door)
del (next-to <other> <small>))
add (holding <small>)
Side: (forall <other> of type (or Thing Door)
del (next-to <small> <other>))
Cost: 1

move-aside(<small>, <room>)
Prim: (forall <other> of type (or Thing Door)
del (next-to <small> <other>))
Side: (forall <other> of type (or Thing Door)
del (next-to <other> <small>))
Cost: 1

push-to-door(<large>, <door>, <room>)
Prim: (forall <other> of type (or Thing Door)
del (next-to <large> <other>))
(forall <other> of type (or Thing Door)
del (next-to <other> <large>))
add (next-to <large> <door>)
add (next-to <door> <large>)
Side: (forall <other> of type (or Thing Door)
del (robot-at <other>))
add (robot-at <door>)
add (robot-at <large>)
Cost: 4

push-aside(<large>, <room>)
Prim: (forall <other> of type (or Thing Door)
del (robot-at <other>))
Side: (forall <other> of type (or Thing Door)
del (next-to <large> <other>))
(forall <other> of type (or Thing Door)
del (next-to <other> <large>))
add (robot-at <large>)
Cost: 4

(a) Selection without *Margie.*

pick-up(<small>, <room>)
Prim: del (arm-empty)
del (in <small> <room>)
add (holding <small>)
Side: (forall <other> of type (or Thing Door)
del (next-to <other> <small>))
(forall <other> of type (or Thing Door)
del (next-to <small> <other>))
Cost: 1

move-aside(<small>, <room>)
Prim: (forall <other> of type (or Thing Door)
del (next-to <small> <other>))
(forall <other> of type (or Thing Door)
del (next-to <other> <small>))
Cost: 1

push-to-door(<large>, <door>, <room>)
Prim: add (next-to <large> <door>)
add (next-to <door> <large>)
Side: (forall <other> of type (or Thing Door)
del (next-to <large> <other>))
(forall <other> of type (or Thing Door)
del (next-to <other> <large>))
(forall <other> of type (or Thing Door)
del (robot-at <other>))
add (robot-at <door>)
add (robot-at <large>)
Cost: 4

push-aside(<large>, <room>)
Prim: (forall <other> of type (or Thing Door)
del (next-to <large> <other>))
(forall <other> of type (or Thing Door)
del (next-to <other> <large>))
Side: (forall <other> of type (or Thing Door)
del (robot-at <other>))
add (robot-at <large>)
Cost: 4

(b) Selection with the use of *Margie.*

Figure 5.8: Primary effects of some operators in the Extended STRIPS Domain. (a) If we apply *Chooser* and *Completer*, then the resulting selection leads to the collapse of the hierarchy. (b) The *Margie* algorithm chooses different effects of the illustrated operators, and enables ALPINE to build a four-level hierarchy (see Figure 5.9).

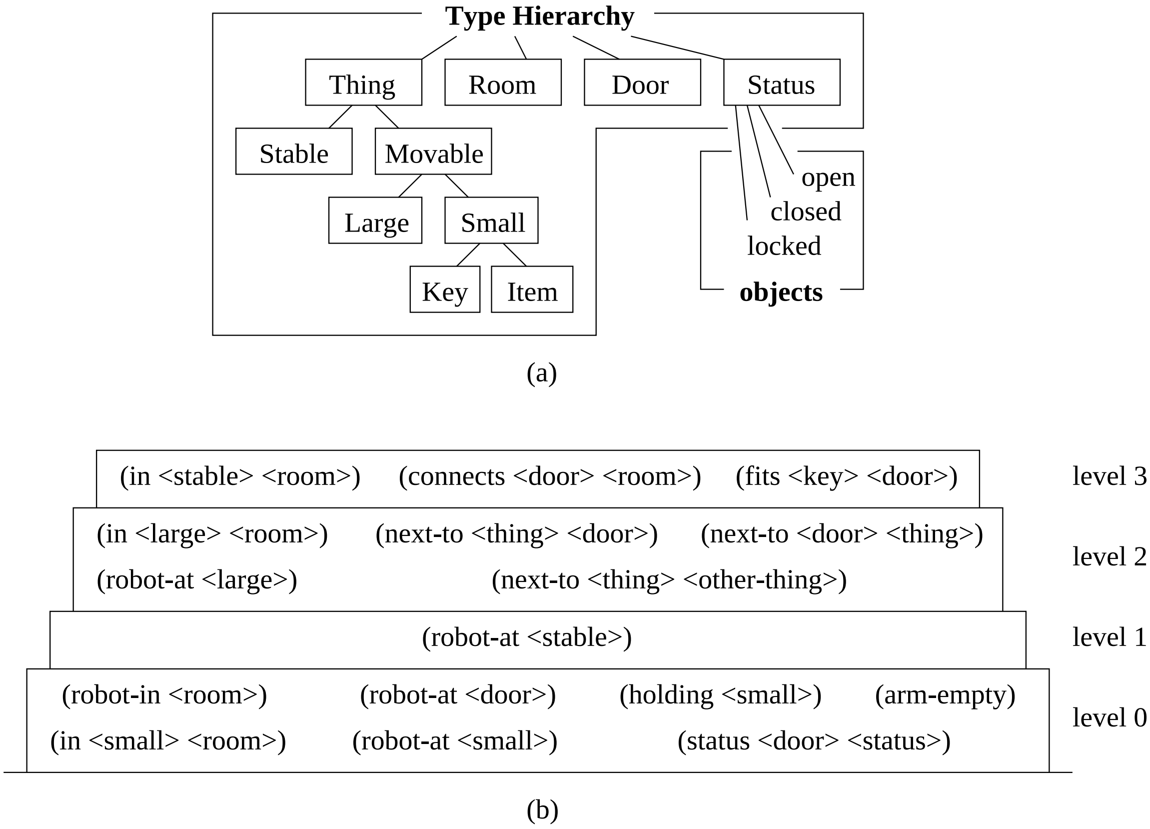


Figure 5.9: Abstraction in the Extended STRIPS domain. We (a) show the object types in this domain, and (b) give the abstraction hierarchy based on *Margie*'s selection of primary effects.

Domain	Overall Result	Search-Time Reduction	Solution-Length Reduction	Solution-Cost Reduction
Extended Robot	–	—	—	—
Machining	⇑	1.3–2.0	1.3–1.5	1.3–1.5
Sokoban	–	—	—	—
Extended STRIPS	⇑	0.1–1000	0.8–1.7	0.9–1.5
Logistics	–	—	—	—

Table 5.1: Results of evaluating the *Margie* algorithm: We compared the search with *Chooser*'s and *Margie*'s primary effects (see Figure 5.4a,b). The application of *Margie* improved performance in two domains, marked by the upward arrow (⇑), and did not affect search in the other domains.

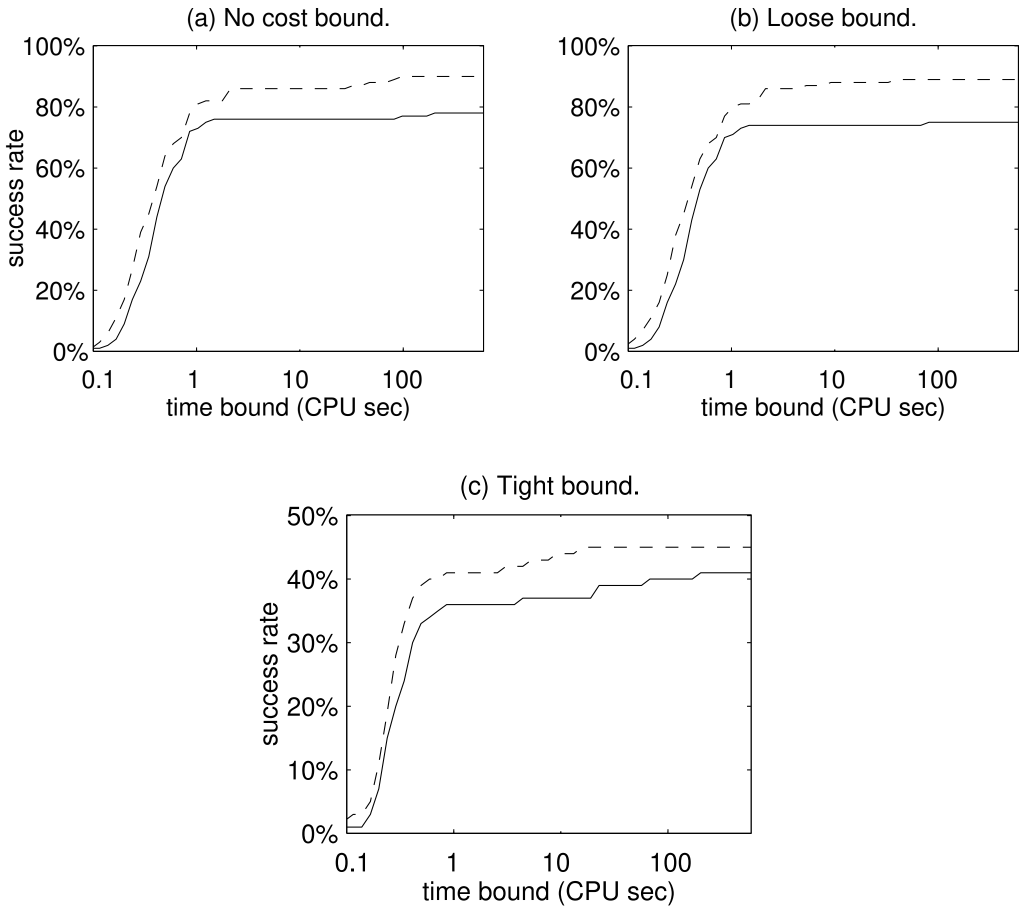


Figure 5.10: Performance in the Extended STRIPS Domain, with *Margie*'s primary effects and abstraction (dashed lines), and with *Chooser*'s effects and *no* abstraction (solid lines). We show the percentage of problems solved by different time bounds, for search (a) without cost bounds, (b) with loose bounds, and (c) with tight bounds.

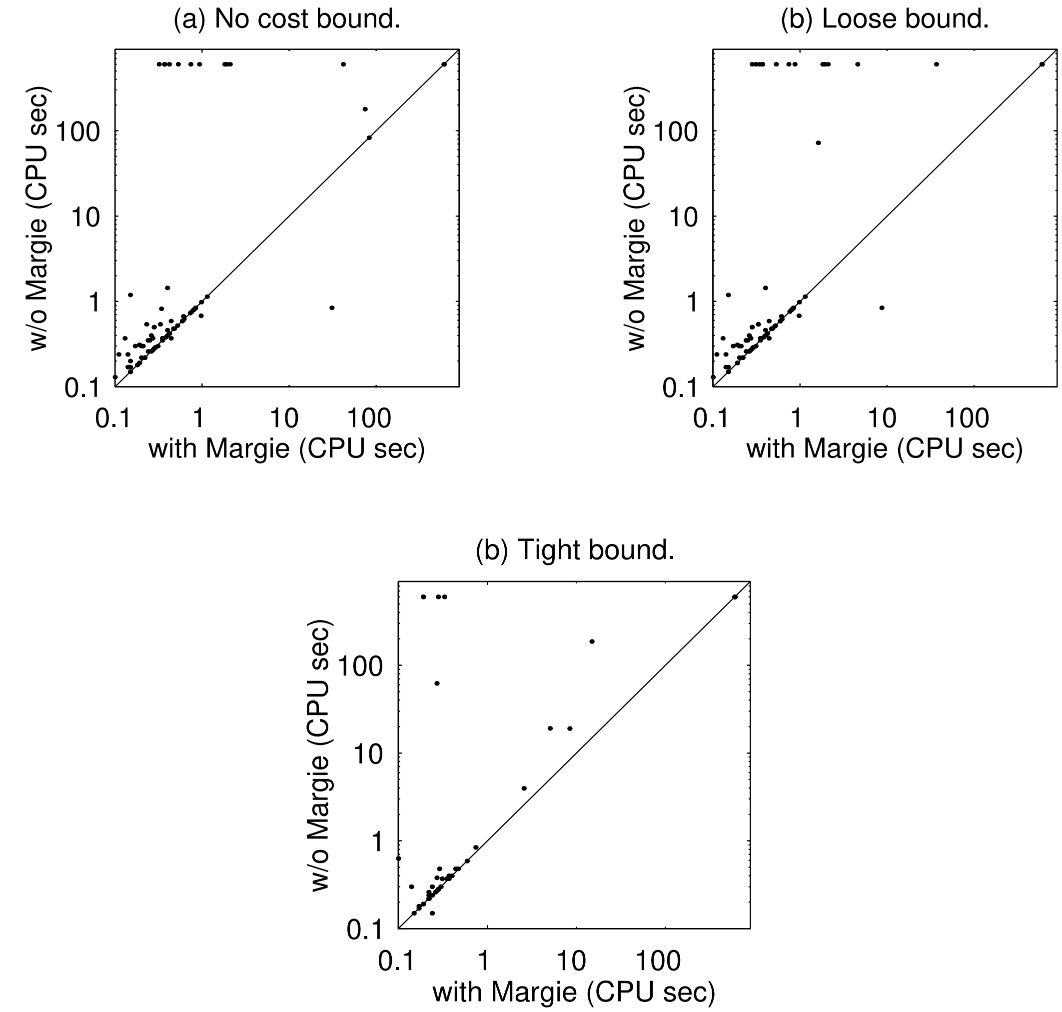


Figure 5.11: Comparison of the search times in the experiments without abstraction and those with *Margie*'s abstraction. Since *Margie* improves performance, most points are above the diagonal.

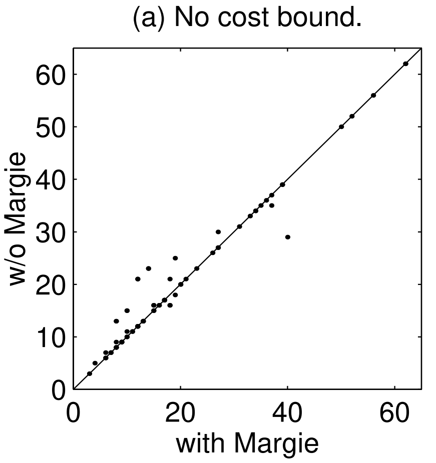


Figure 5.12: Solution lengths in the Extended STRIPS Domain. We give the results of using primary effects without abstraction (vertical axes) and with *Margie*'s abstraction (horizontal axes).

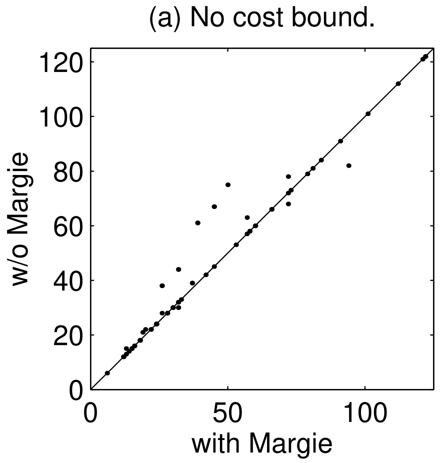


Figure 5.13: Solution costs in the Extended STRIPS Domain.

5.3 Identifying the relevant literals

We have considered description improvers that do *not* utilize information about particular problems. When they produce a new description, PRODIGY may employ it for all problems in the current domain. For example, if we apply *Chooser* and *Completer* to the Machining Domain, they select the primary effects given in Figure 5.18(a), which are suitable for solving any machining problem.

The use of problem-specific knowledge often allows more effective improvements. For example, suppose that we need a domain description for machining tasks that do *not* have destructive goals, such as "not polished" or "not painted." Then, we may modify the selection of primary effects, as shown in Figure 5.18(b), and construct a three-level abstraction (see Figure 5.18c).

An attempt to solve inappropriate problems with a specialized description may lead to inefficiency or gross incompleteness. For example, if we employ the primary effects in Figure 5.18(b) for a problem with destructive goals, PRODIGY will terminate with a failure. Thus, the system may not be able to amortize the cost of a specialized description change over multiple problems; however, the implemented changers usually incur small computational costs and do not require amortization.

Knoblock [1994] explored problem-specific improvements during his work on the ALPINE abstraction generator. He designed a procedure for pinpointing the literals relevant to the goals of a specific problem, and extended ALPINE to utilize this information. The extended version built a hierarchy of relevant literals, and ignored the other literals.

We have applied Knoblock's technique to description changes in the $\mathcal{S}$HAPER system. Specifically, we have implemented an algorithm for identifying relevant literals, called *Relator*, which is an extension to Knoblock's procedure. Furthermore, we enabled *Chooser*, *Completer*, *Abstractor*, and *Margie* to use the resulting relevance data. We give an outline of *Relator*; the reader may find more details in Knoblock's [1993] thesis.

Formally, a literal is *relevant* if it may become a subgoal during search for a solution (see Section 2.2.4). The purpose of *Relator* is to compile a literal set that includes all potential subgoals, while excluding irrelevant literals. For example, suppose that the system needs a specialized description for a drilling problem with the goal (has-spot part-1) (see the domain encoding in Figure 4.1, page 151). Then, the relevance set includes has-spot, spot-drill, holds-drill, holds-part, no-drill, and no-part.

The *Relator* algorithm inputs a list of goals, along with the encoding of operators and inference rules, and identifies all literals that may be relevant to achieving the goals. If the user pre-selects side effects of some operators, the algorithm utilizes these extra data. We give a specification of *Relator* in Figure 5.14 and pseudocode in Figure 5.15.

The algorithm finds all operators that match the goals, and inserts their preconditions into the relevance set. Then, it recursively identifies the preconditions of the operators that achieve the newly added literals. For example, suppose that we call *Relator* for the goal has-spot in the Drilling Domain. First, the algorithm determines that the **drill-hole** operator achieves this goal, and inserts its preconditions, spot-drill, holds-drill, and holds-part, into the relevance set. Then, *Relator* identifies the **put-drill** and **put-part** operators, which match the newly added literals, and inserts their preconditions, no-drill and no-part, into the set.

Type of description change: Identifying relevant literals.

Purpose of description change: Minimizing the set of selected literals, while including all relevant literals.

Use of other algorithms: None.

Required input: Description of the operators and inference rules; list of goal predicates, which may be partially instantiated.

Optional input: Pre-selected side effects.

Figure 5.14: Specification of the *Relator* algorithm.

Relator(*Goals*)
The algorithm inputs a list of goals and returns the set of relevant literals.
It accesses the operators and inference rules, with pre-selected side effects.

New-Literals := *Goals* (newly added literals)
Relevance-Set := *Goals* (set of relevant literals)
Repeat while *New-Literals* is not empty:
  Add-Literals := *Add-Relevant*(*New-Literals*)
  New-Literals := *Add-Literals* − *Relevance-Set*
  Relevance-Set := *Relevance-Set* ∪ *New-Literals*
Return *Relevance-Set*

Add-Relevant(*New-Literals*)
For every literal l in *New-Literals*:
  For every operator and inference rule *step* that achieves l:
    If l is not pre-selected as a side effect of *step*,
      then add the preconditions of *step* to *New-Literals*.
Return *New-Literals*.

Figure 5.15: Identifying the relevant literals for a given list of goals.

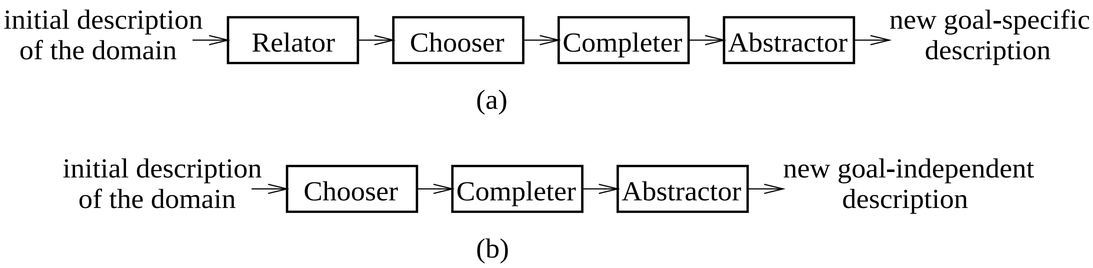


Figure 5.16: Experiments on the effectiveness of the *Relator* algorithm. We test the utility of goal-specific abstraction and primary effects (a), and compare it with the performance of a goal-independent version (b).

5.4 Experiments with goal-specific descriptions

The evaluation of the *Relator* algorithm involved two series of experiments. First, we measured the time of problem solving with goal-specific abstractions and primary effects (see Figure 5.16a). Second, we compared these results with the efficiency of analogous goal-independent descriptions (Figure 5.16b).

The *Relator* algorithm allowed the generation of finer-grained hierarchies for the Machining Domain, Sokoban puzzle, and STRIPS world. On the other hand, it did *not* affect performance in the Robot world and Logistics Domain. The goal-specific descriptions gave mixed results, from a thousand-fold speed-up to a sharp increase of the search time; however, improvements were more frequent then negative results.

Extended Robot Domain

If the system does not utilize the *Relator* algorithm in the Robot Domain, then it selects the primary effects given in Figure 3.31 (page 131), and then generates a three-level abstraction hierarchy (see Figure 4.30 on page 183). On the other hand, the application of *Relator* leads to selecting fewer primary effects and constructing finer-grained hierarchies.

For example, suppose that we need abstraction for problems that have *no* negation goals and, thus, do not require the deletion of literals. The *Relator* algorithm enables the system to select relevant primary effects, shown in Figure 5.17(a), which allow the construction of a six-level hierarchy (Figure 5.17b).

We tested this hierarchy on the nine problems from Table 3.6 (see page 131), and compared it with the goal-independent version (see Table 5.2). The six-level hierarchy reduced the search time on problems 6 and 9, caused an increase in running time on problem 7, and performed identically to goal-independent abstraction in the other six cases.

The *Relator* algorithm did *not* affect the solution quality. In all nine cases, the system generated the same solutions with the goal-specific and goal-independent hierarchy. We showed the lengths and costs of these solutions in Figure 4.31 (see page 183).

open(<door>)
Prim: add (open <door>)
Side: del (closed <door>)

close(<door>)
Prim: add (closed <door>)
Side: del (open <door>)

climb-up(<table>)
Prim: add (robot-on <table>)
Side: del (robot-on-floor)

climb-down(<table>)
Side: del (robot-on <table>)
add (robot-on-floor)

carry-up(<box>, <table>)
Prim: add (on <box> <table>)
Side: del (on-floor <box>)
del (robot-on-floor)
add (robot-on <table>)

carry-down(<box>, <table>)
Side: del (on <box> <table>)
add (on-floor <box>)
del (robot-on <table>)
add (robot-on-floor)

go-within-room
(<from-loc> <to-loc> <room>)
Prim: add (robot-at <to-loc>)
Side: del (robot-at <from-loc>)

go-thru-door
(<from-room> <to-room> <door>)
Prim: add (robot-in <to-room>)
Side: del (robot-in <from-room>)

carry-within-room
(<box>, <from-loc>, <to-loc>, <room>)
Prim: add (at <box> <to-loc>)
Side: del (at <box> <from-loc>)
del (robot-at <from-loc>)
add (robot-at <to-loc>)

carry-thru-door
(<box>, <from-room>, <to-room>, <door>)
Prim: add (in <box> <to-room>)
Side: del (in <box> <from-room>)
del (robot-in <from-room>)
add (robot-in <to-room>)

(a) Example of a problem-specific selection of primary effects.

(within <location> <room>)	level 5
(on <box> <table>)	level 4
(in <box> <room>) (at <box> <location>)	level 3
(robot-on <table>)	level 2
(robot-in <room>) (open <door>) (robot-at <location>) (closed <door>)	level 1
(on-floor <box>) (robot-on-floor)	level 0

(b) Problem-specific abstraction hierarchy.

Figure 5.17: Results of using *Relator* in the Extended Robot Domain. The improved description is suitable for solving all problem that have no deletion goals. If we allowed any goals, the system would generate the three-level hierarchy in Figure 4.30 (page 183).

#	No Cost Bound		Loose Bound		Tight Bound	
	indep	spec	indep	spec	indep	spec
6	2.66	0.91	2.61	0.90	0.27	0.24
7	1.26	3.08	1.31	3.11	1.56	1.68
9	0.26	0.26	0.26	0.26	> 1800.00	571.14

Table 5.2: Performance in the Extended Robot Domain, on problems 6, 7, and 9 from Table 3.6. We give search times for a goal-independent abstraction hierarchy ("indep"), and the corresponding times for goal-specific abstraction ("spec").

Machining Domain

The goal-independent selection of primary effects in the Machining Domain includes deletion effects (see Figure 5.18a), which cause the collapse of the ordered abstraction. The *Relator* algorithm prevents this problem: It helps to construct a selection with fewer effects (Figure 5.18b), which leads to a three-level hierarchy (Figure 5.18c).

In Figure 5.19, we summarize the results of search without cost bounds. Specifically, we give the search time and solution costs, for problem solving with goal-specific abstraction and primary effects (dashed lines), with goal-independent selection of primary effects (solid lines), and without primary effects (dotted lines). In Figure 5.20, we give analogous results for problem solving with loose cost bounds.

The goal-specific abstraction reduced the search time for *all* problems. The reduction factor varied from 1.5 to 2.3, and its mean value was 1.74. Furthermore, when the search algorithm utilized abstraction, it found optimal solutions to all machining problems. The factor of solution-cost reduction ranged from 1.6 to 1.8, with a mean of 1.69.

Extended STRIPS Domain

In Section 3.7.2, we described the goal-independent application of *Chooser* and *Completer* to the STRIPS Domain. The resulting selection of primary effects (see Figures 3.41 and 3.42) significantly reduced the search time, but did *not* allow the construction of an ordered hierarchy. The *Relator* algorithm enables the system to select fewer primary effects and construct an eight-level abstraction hierarchy, given in Figure 5.21(b), which works for problems that have no deletion goals.

In Figure 5.22, we give the results of problem solving with the goal-specific hierarchy (dashed lines) and without abstraction (solid lines). The graphs show the percentage of problems solved by different time bounds. In Figure 5.23, we show an alternative summary of the same results.

When we used abstraction without cost bounds and with tight bounds, it slightly increased the number of problems solved within 600 seconds; however, it had the opposite effect in the experiments with loose bounds. These changes in the percentage of solved problems were *not* statistically significant. The ratio of the search times with and without abstraction varied widely from problem to problem, ranging from less than 0.001 to greater than 1000.

On the other hand, the goal-specific hierarchy gave a statistically significant improvement

cut-roughly(<part>)
Prim: add (cut <part>)
del (finely-cut <part>)
del (drilled <part>)
Side: del (finely-drilled <part>)
del (polished <part>)
del (finely-polished <part>)
del (painted <part>)
del (finely-painted <part>)

drill-roughly(<part>)
Prim: add (drilled <part>)
del (finely-drilled <part>)
del (polished <part>)
Side: del (finely-polished <part>)
del (painted <part>)
del (finely-painted <part>)

polish-roughly(<part>)
Prim: add (polished <part>)
del (finely-polished <part>)
del (painted <part>)
Side: del (finely-painted <part>)

paint-roughly(<part>)
Prim: add (painted <part>)
del (finely-painted <part>)
Side: del (polished <part>)
del (finely-polished <part>)

(a) Problem-independent selection of primary effects.

cut-roughly(<part>)
Prim: add (cut <part>)
Side: del (finely-cut <part>)
del (drilled <part>)
del (finely-drilled <part>)
del (polished <part>)
del (finely-polished <part>)
del (painted <part>)
del (finely-painted <part>)

drill-roughly(<part>)
Prim: add (drilled <part>)
Side: del (finely-drilled <part>)
del (polished <part>)
del (finely-polished <part>)
del (painted <part>)
del (finely-painted <part>)

polish-roughly(<part>)
Prim: add (polished <part>)
Side: del (finely-polished <part>)
del (painted <part>)
del (finely-painted <part>)

paint-roughly(<part>)
Prim: add (painted <part>)
Side: del (finely-painted <part>)
del (polished <part>)
del (finely-polished <part>)

(b) Example of a problem-specific selection.

(cut <part>)	(finely-cut <part>)	level 2
(drilled <part>)	(finely-drilled <part>)	level 1
(polished <part>) (painted <part>)	(finely-polished <part>) (finely-painted <part>)	level 0

(c) Problem-specific abstraction hierarchy.

Figure 5.18: Effects of the low-quality operations in the Machining Domain. When the system generates a goal-independent description, the selected primary effects do not allow abstraction (a). On the other hand, *Relator* enables the system to select fewer primary effects (b) and build a multi-level hierarchy (c).

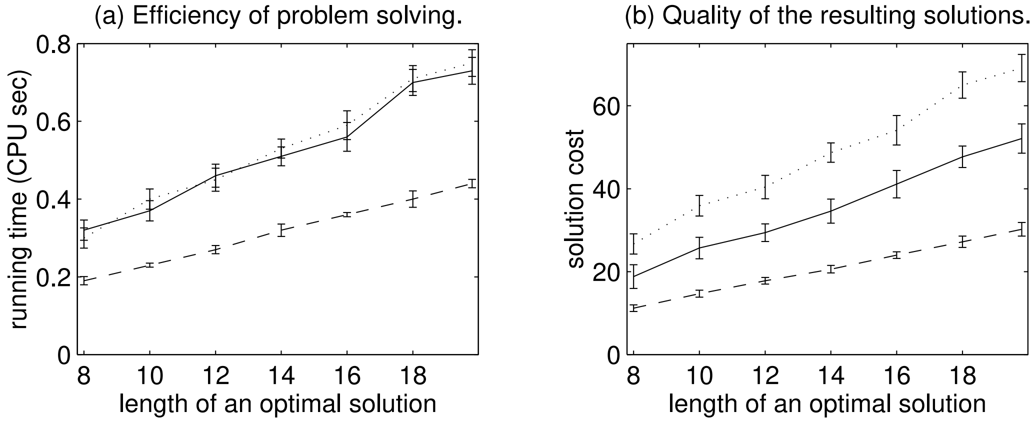


Figure 5.19: Performance in the Machining Domain, *without cost bounds.* We give the results of problem-specific description improvements (dashed lines) and analogous problem-independent improvements (solid line), as well as the results of search with the initial description (dotted lines). The vertical bars show the 95% confidence intervals.

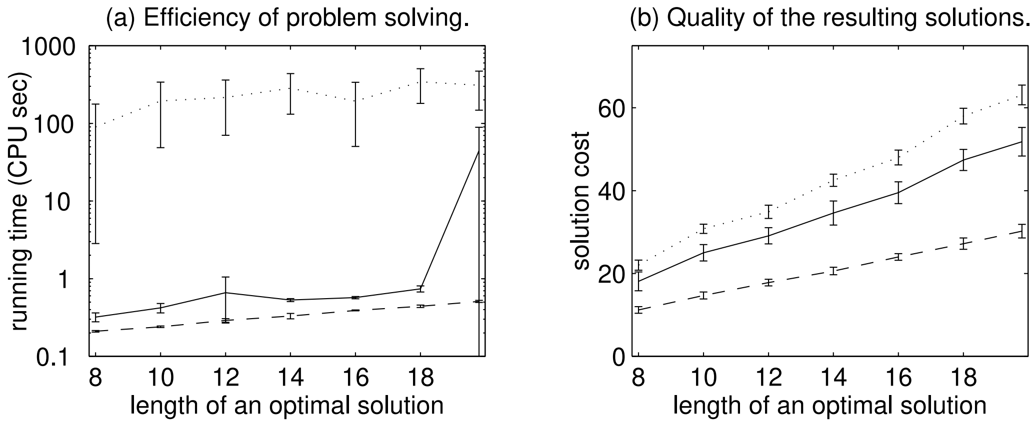
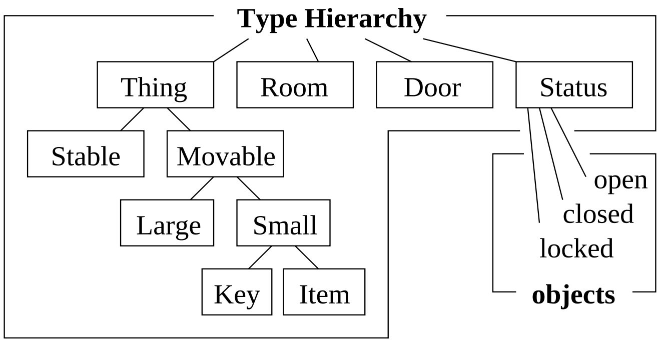
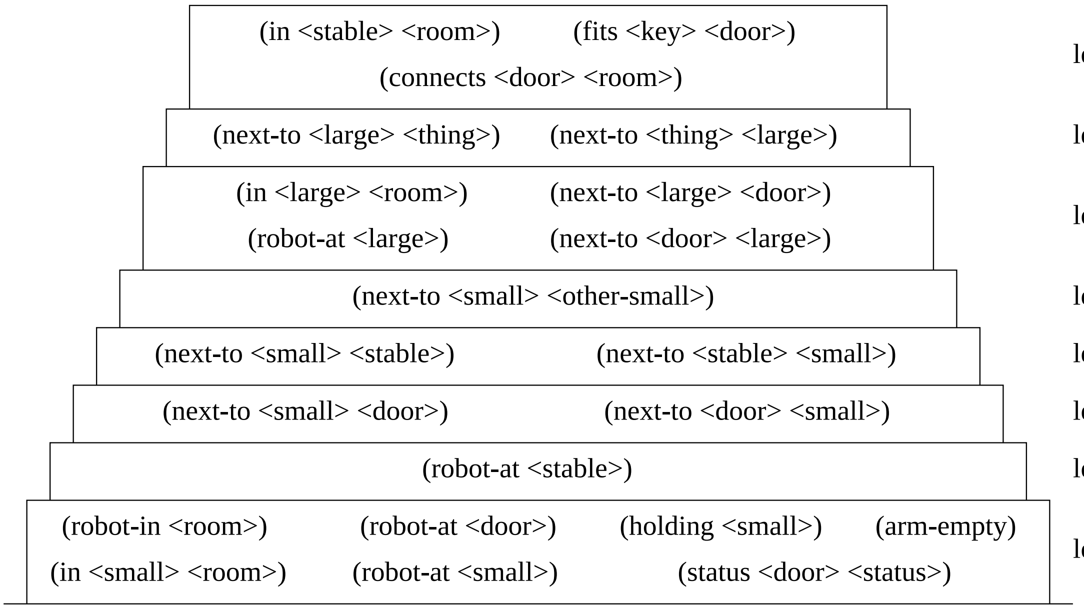


Figure 5.20: Performance in the Machining Domain, with *loose cost bounds.* The legend is the same as in Figure 5.19; however, the running-time scale is logarithmic.



(a)



(b)

Figure 5.21: Goal-specific abstraction in the Extended STRIPS domain. We show the object types in this domain (a), and give an improved abstraction hierarchy (b).

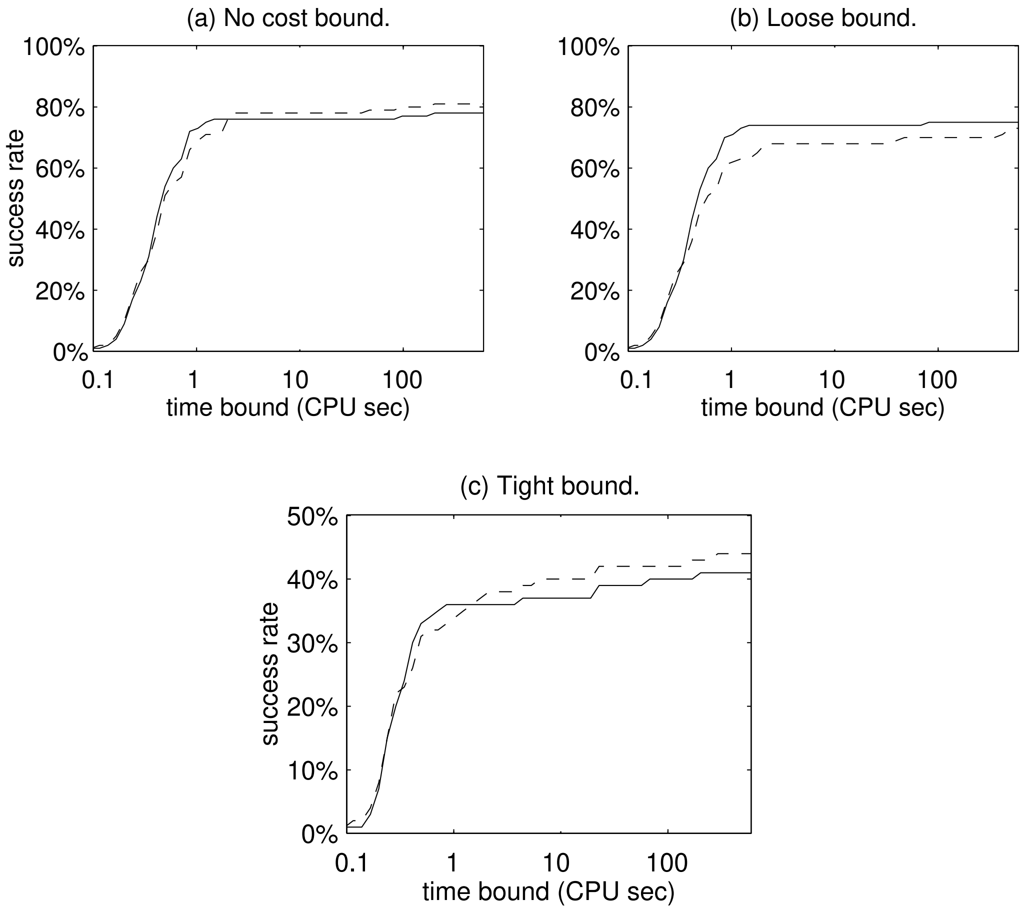


Figure 5.22: Results in the Extended STRIPS Domain, with goal-specific abstraction and primary effects (dashed lines), and with goal-independent effects and *no* abstraction (solid lines). We show the percentage of problems solved by different time bounds, for search without cost bounds (a), with loose bounds (b), and with tight bounds (c).

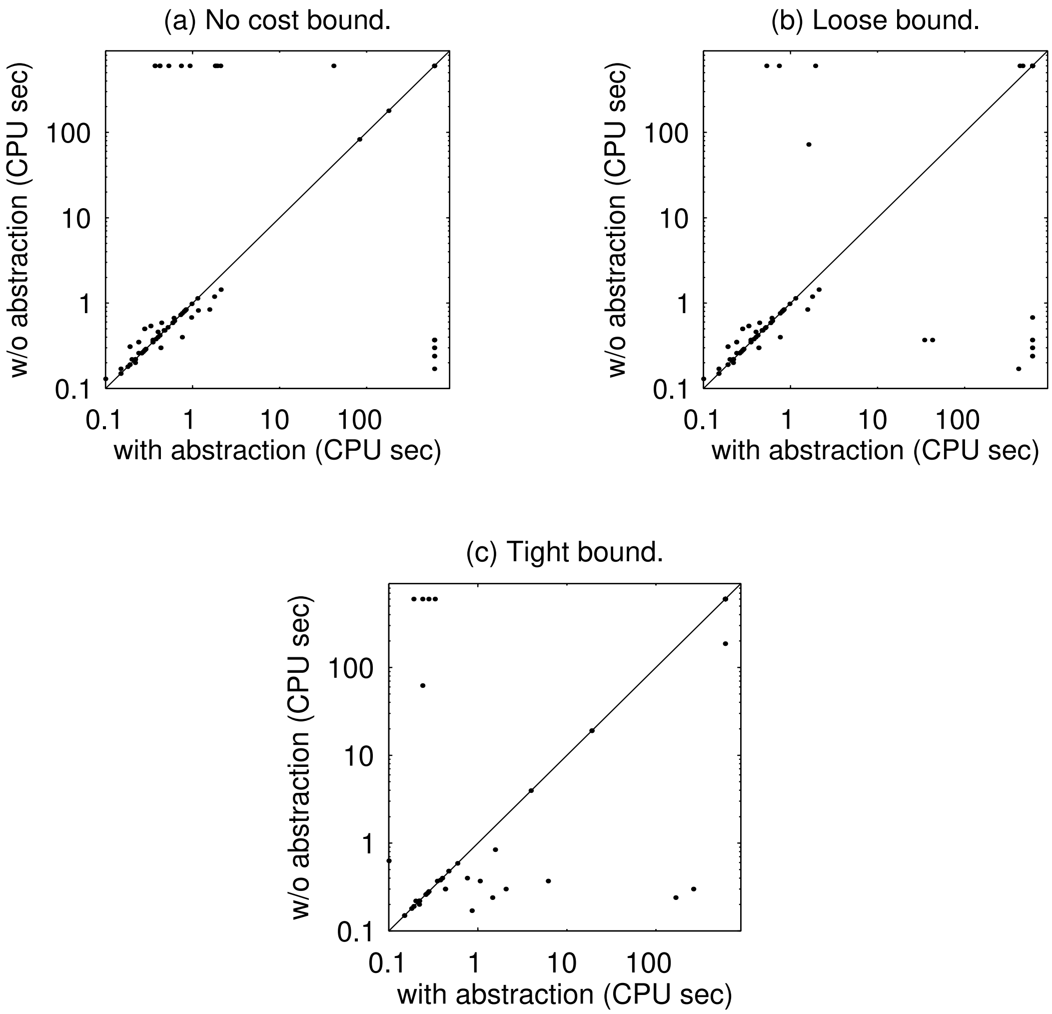


Figure 5.23: Comparison of the search times in the experiments without abstraction (vertical axes) and those with problem-specific abstraction (horizontal axes).

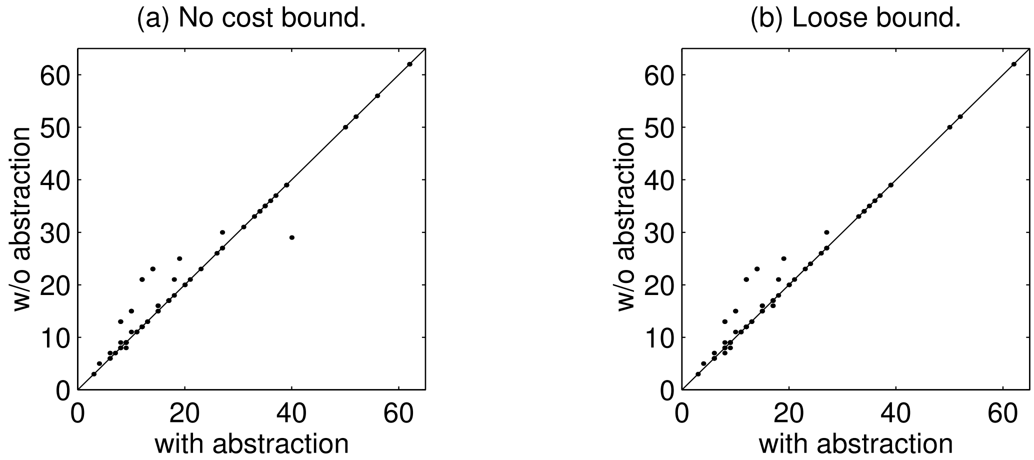


Figure 5.24: Solution lengths in the Extended STRIPS Domain. We give the results for the goal-independent description (vertical axes) and goal-specific descriptions (horizontal axes).

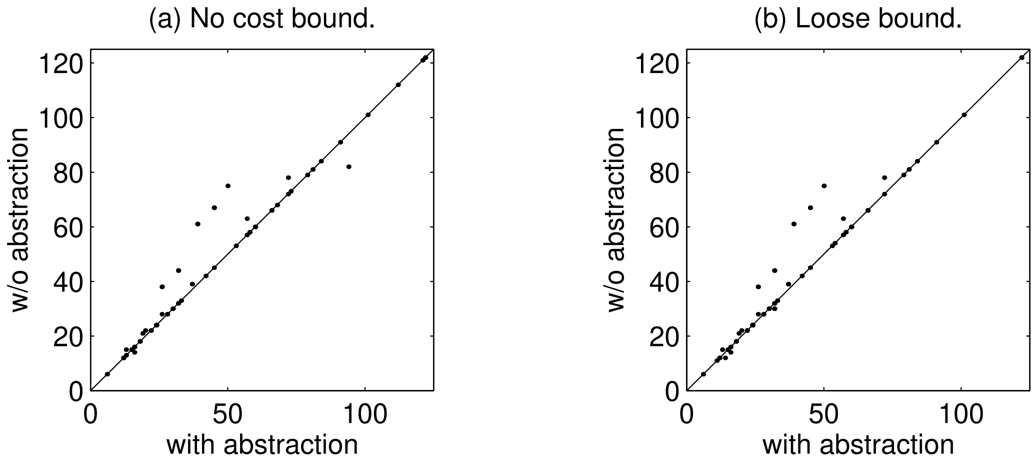


Figure 5.25: Solution costs in the Extended STRIPS Domain.

of solution quality. In Figure 5.24, we compare the solution lengths in the experiments with and without abstraction, using the problems solved in both cases. In Figure 5.25, we give an analogous comparison of solution costs. The length-reduction factor varies from 0.9 to 1.7, and its mean value is 1.07, whereas the cost reduction is between 0.9 and 1.5, with a mean of 1.05.

Summary

The *Relator* algorithm helped to identify relevant primary effects and generate finer-grained hierarchies; however, the resulting efficiency improvements were surprisingly modest. The use of problem-specific abstractions reduced the search times and solution costs in the Machining Domain, but gave mixed results in the simulated robot worlds (see Table 5.3).

The experiments have shown that an increase in the number of abstraction levels does not

Domain	Overall Result	Search-Time Reduction	Solution-Length Reduction	Solution-Cost Reduction
Extended Robot	$\Updownarrow$	0.4–3.1	none	none
Machining	$\Uparrow$	1.5–2.3	1.6–1.8	1.6–1.8
Sokoban	–	—	—	—
Extended STRIPS	$\Updownarrow$	0.001–1000	0.9–1.7	0.9–1.5
Logistics	–	—	—	—

Table 5.3: Summary of experiments with the *Relator* algorithm: We compared the effectiveness of goal-specific and goal-independent description improvements (see Figure 5.16). The application of *Relator* improved efficiency in Machining Domain, gave mixed results in the Robot and STRIPS worlds, and did not affect the performance in Sokoban and Logistics Domains.

always improve efficiency. This observation is different from the results of Knoblock [1993], who reported that finer-grained hierarchies give better performance, and that identification of goal-relevant literals is often essential for efficient abstraction.

Chapter 6

Summary of work on description changers

An immediate research direction is to investigate the integration of these different learning strategies within PRODIGY. *This will entail further work on incorporating other learning methods into the nonlinear planner framework.*
— Manuela M. Veloso [1994], *Planning and Learning by Analogical Reasoning.*

We have described a collection of algorithms that modify PRODIGY domain descriptions and speed up the problem-solving process. The algorithms proved effective in most experimental domains; however, the underlying techniques are heuristic, and they do *not* guarantee an improvement. In the worst case, their use may lead to an exponential increase in search time. In Figure 6.1, we summarize the results of testing these algorithms in five domains.

We now discuss the main results of the work on speed-up techniques and outline some directions for future research. First, we give a list of the implemented algorithms and describe interactions among them (Section 6.1). Then, we summarize the methods for selecting primary effects and generating abstraction hierarchies (Sections 6.2), and discuss some unexplored description changes (Sections 6.3). Finally, we point out some research problems related to the development of new description changers (Section 6.4).

6.1 Library of description changers

We have developed seven algorithms for the automatic improvement of domain descriptions, which form the foundation of the $\mathcal{S}$HAPER system. These algorithms use four main types of description changes: selecting primary effects, building abstraction hierarchies, partially or fully instantiating elements of the domain description, and identifying the literals relevant to the current task.

In Figure 6.1, we give a summary of the description-changing algorithms, divided into two groups. The first group comprises the *main description changers,* which improve the performance of problem-solving algorithms. It includes four algorithms for choosing primary effects and generating abstraction hierarchies.

Domain	Primary Effects (Chapter 3)	Ordered Abstraction (Chapter 4)		Abstraction of Effects (Sec. 5.1 and 5.2)	Identification of Relevant Literals (Sec. 5.3 and 5.4)
		w/o prims	with prims		
Extended Robot	$\Uparrow$	–	$\Uparrow$	–	$\Updownarrow$
Machining	$\Uparrow$	–	–	$\Uparrow$	$\Uparrow$
Sokoban	$\Uparrow$	–	$\Uparrow$	–	–
Extended STRIPS	$\Uparrow$	–	–	$\Uparrow$	$\Updownarrow$
Logistics	–	$\Updownarrow$	$\times$	–	–

Notation:

$\Uparrow$ *positive results:* the technique improved performance on almost all test problems
$\Updownarrow$ *mixed results:* the technique reduced search in some cases, but increased it in other cases
– *no effect on the performance:* the technique did not affect the system's behavior
$\times$ *no experiments:* since operators in the Logistics Domain have no unimportant effects, we cannot test abstraction with primary effects in this domain

Table 6.1: Summary of experiments with the speed-up techniques in five different domains. The table comprises the "Overall Result" columns from Tables 3.8, 4.2, 5.1, and 5.3.

The second group contains the *auxiliary description changers,* which enhance the effectiveness of the main changers. The algorithms of this group analyze the structure of a problem domain and extract information for improving the quality of the primary-effect selections and abstraction hierarchies. The auxiliary algorithms do not directly improve the problem-solving performance and, hence, we always use them in conjunction with the main description changers.

The utility of description-improving algorithms depends on the resulting search reduction and changes in solution quality. We consider an algorithm effective if it reduces search by a significant factor, and the savings grow with problem complexity. If the use of a description changer leads to generating lower-quality solutions, then the time savings should justify the loss of quality. We have described the trade-off between running time and solution quality in problem solving with primary effects and abstraction. In Section 7.3, we will formalize this trade-off and derive a utility function for the evaluation of description changers.

We have demonstrated that the changer algorithms usually improve the performance; however, they do *not* guarantee improvement, and the resulting description changes sometimes impair efficiency or cause an unacceptable decline in solution quality. Therefore, a representation-changing system must have a top-level control module, which selects appropriate description changers, tests the results of their application, and prunes ineffective domain descriptions. We will describe the control module of the SHAPER system in Part III.

We next describe two main types of interactions among the changer algorithms (Section 6.1.1) and review techniques for utilizing problem-specific information in generating new domain descriptions (Section 6.1.2).

Primary effects and abstraction

Chooser: Heuristic selection of primary effects (Section 3.4.1)
The *Chooser* algorithm selects primary effects of operators and inference rules, for improving the efficiency of backward chaining. It uses several simple heuristics, aimed at ensuring near-completeness and limiting the increase in solution costs. The selection algorithm is very fast, but it does not guarantee completeness of search with the chosen primary effects.

Completer: Learning additional primary effects (Section 3.5)
The *Completer* algorithm learns additional primary effects, to ensure a required probability of completeness and limited cost increase. On the negative side, it takes significant time and needs a hand-coded generator of initial states. We use the PAC-learning probability parameters, ϵ and δ, to control the trade-off between completeness probability and learning time.

Abstractor: Building an abstraction hierarchy (Section 4.2)
The *Abstractor* algorithm is an advanced version of the ALPINE abstraction generator, extended to construct hierarchies for the full domain language of the PRODIGY4 system. It imposes constraints on the relative importance of predicates in the description of operators and inference rules, and uses these constraints to abstract some preconditions and side effects.

Margie: Abstracting effects of operators and inference rules (Section 5.1)
The *Margie* algorithm combines the automatic selection of primary effect with the construction of an ordered abstraction hierarchy, thus abstracting unimportant effects. It uses heuristics for selecting primary effects and a dynamic version of *Abstractor*, which efficiently generates hierarchies for multiple alternative selections.

Auxiliary description changers

Matcher: Instantiating operators and inference rules (Section 3.4.2)
The *Matcher* algorithm generates all possible instantiations of operators and inference rules, for given object instances and static features of the initial state. When processing an operator or inference rule, *Matcher* instantiates its variables one by one, and prunes subexpressions that do not affect the overall truth value. We use the resulting instantiations to improve the effectiveness of *Chooser* and *Margie*, which select primary effects.

Refiner: Partial instantiation of predicates (Section 4.3)
The *Refiner* algorithm generates a partial instantiation of predicates, for use in abstraction graphs and relevance sets. When we combine this algorithm with *Abstractor*, it finds a minimal instantiation that does not cause a collapse of abstraction levels. Similarly, when we use *Refiner* in identifying relevant literals, it produces a minimal instantiation that prevents over-generalization.

Relator: Identifying relevant literals (Section 5.3)
The *Relator* module consists of two algorithms for identifying relevant features of a domain. The first algorithm pre-processes the domain description and encodes the relationships among literals as a *relevance graph*. The second algorithm inputs a collection of goals and uses the relevance graph to determine which literals are relevant to achieving these goals. We use the resulting relevance information to enhance the performance of *Chooser*, *Abstractor*, and *Margie*.

Figure 6.1: Summary of description-changing algorithms in the SHAPER system.

Primary effects and abstraction

Chooser: Heuristic selection of primary effects

Completer: Learning additional primary effects

Abstractor: Building an abstraction hierarchy

Margie: Abstracting effects of operators

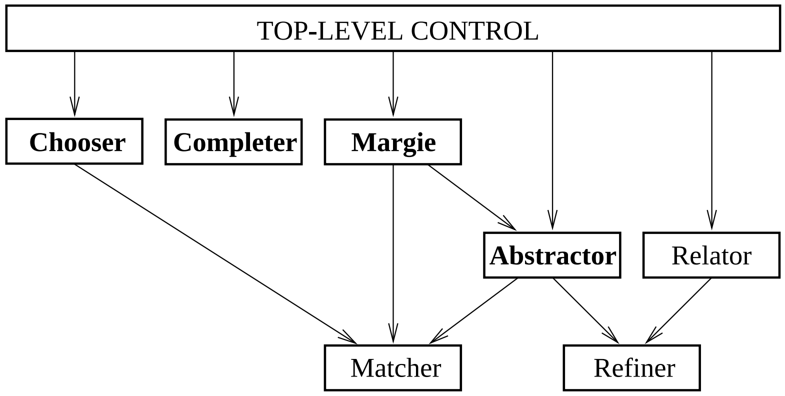
Auxiliary description changes

Matcher: Instantiating operators and inference rules

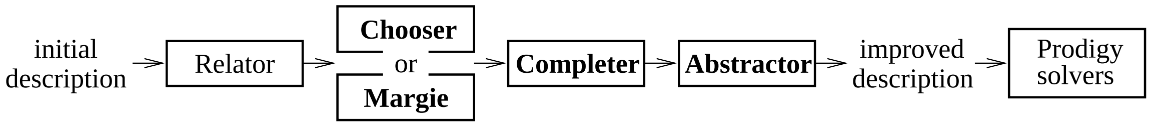
Refiner: Partial instantiation of predicates

Relator: Identifying relevant literals

(a) List of description changers.



(b) Subroutine calls among changers.



(c) Order of applying changers to a domain.

Figure 6.2: Interactions among description-changing algorithms; we boldface the names of the main description changers, and use the standard font for the auxiliary changer algorithms.

6.1.1 Interactions among description changers

A description changer may interact with other changer algorithms in two ways: through subroutine calls and by utilizing the output of previously applied algorithms. In Figure 6.2(b), we show all subroutine calls among the changer algorithms. For example, if we apply the *Matcher* algorithm, then it invokes two other changers, *Matcher* and *Abstractor*. We also show the top-level control module, described in Part III, which selects and invokes appropriate description changers. Note that it does not directly call two auxiliary changers, *Matcher* and *Refiner*.

The second type of interaction is the sequential application of multiple changer algorithms to the same domain description, which allows some algorithms to use the results of previous description changes. For example, if we apply the *Abstractor* algorithm after *Chooser* and *Completer*, then it utilizes the selected primary effects in generating an abstraction hierarchy.

The interactions among description changers determine the appropriate application order.

For instance, if we use *Chooser* and *Abstractor* to improve a domain description, then we first apply the *Chooser* algorithm, which enables *Abstractor* to use the resulting primary effects. As another example, we should not apply *Margie* after *Completer*, since it will cancel the learned primary effects and produce a new selection.

In Figure 6.2(c), we show the appropriate order of applying description changers. When generating a description for a specific goal set, we apply the *Relator* algorithm before all other changers. We may apply either *Chooser* or *Margie* for selecting primary effects, and then *Completer* for improving the selection. When using an abstraction hierarchy, we apply the *Abstractor* algorithm after selecting primary effects.

Note that we do *not* have to use all description changers. We may skip any steps in Figure 6.2(c), but we should not change their order. We use this order in constructing fixed sequences of changers, described in Section 7.1, which serve as operators for expanding the space of alternative domain descriptions.

6.1.2 Description changes for specific problems and problem sets

If the user provides restrictions on the allowed problem instances, then changer algorithms can utilize them to improve the domain descriptions. This option allows the generation of new descriptions for specific problems or groups of problems.

The use of problem-specific information helps to improve performance; however, it limits the use of the resulting descriptions. If some problems do not satisfy the restrictions, then problem solving with these descriptions may lead to incompleteness, gross inefficiency, or even construction of incorrect solutions.

The utility of a problem-specific description depends on the time for generating it, the resulting performance advantage over the problem-independent version, and the number of problems that satisfy the restrictions. In particular, the total time savings on all matching problems should be larger than the time of the description change.

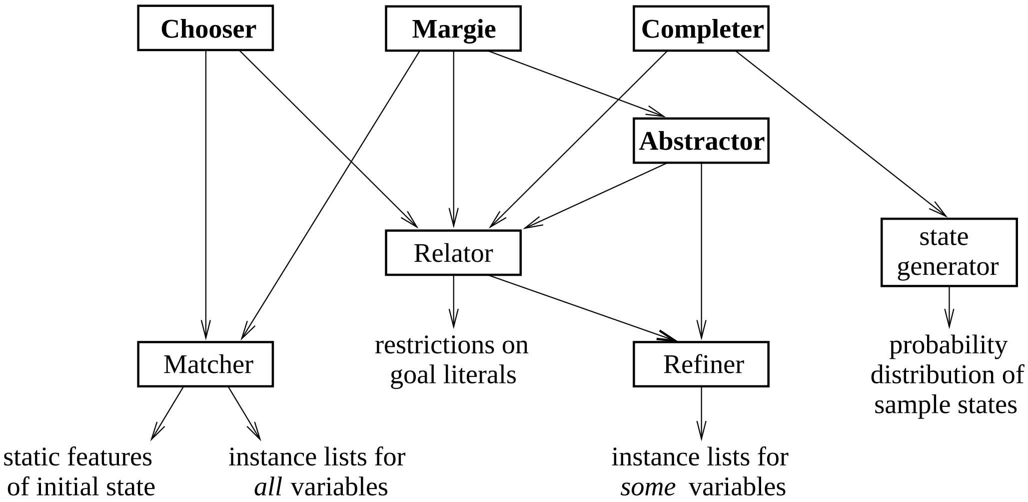
In Figure 6.3(a), we summarize the problem-specific information used in the $\mathcal{S}$HAPER system. The auxiliary description changers access this information directly and use it to pre-process the domain description, whereas the main changers utilize the results of the pre-processing. We illustrate the access to the problem-specific restrictions in Figure 6.3(b), where the boldface font marks the main changer algorithms. Finally, in Figure 6.3(d), we show which changers utilize which restrictions.

For example, the *Matcher* changer uses information about static features of the initial state and available object instances to pre-process the domain description. *Relator* inputs restrictions on goal literals and identifies all other literals relevant to goal achievement. On the other hand, the *Chooser* algorithm does *not* directly access problem-specific restrictions; however, it uses the pre-processing results of *Matcher* and *Relator*, thus utilizing knowledge of static literals, available objects, and allowed goals.

Note that the procedure for generating random initial states is *not* a description changer; however, it also supports the use of problem-specific data. If the user provides a procedure that produces random states for a certain collection of problems, then it allows the selection of primary effects for solving these specific problems.

- Lists of possible instances of variables in the domain description
- Static literals that hold in the initial states of all problems
- Probability distribution over all possible states of the domain
- Restrictions on the literals that can be used in goal statements

(a) Optional problem-specific restrictions.



(b) Access to restrictions on problem instances.

	instance lists	static features	distribution of states	goal literals
	Primary effects and abstraction			
Chooser	+	+		+
Completer	+		+	+
Abstractor	+			+
Margie	+	+		+
	Auxiliary description changers			
Matcher	+	+		
Refiner	+			
Relator	+			+

(c) Utilization of available restrictions.

Figure 6.3: Use of problem-specific information in changing domain descriptions.

6.2 Primary effects and abstraction

The main description-improving techniques in the current version of the $\mathcal{S}$HAPER system include the selection of primary effects and generation of abstraction hierarchies (see Figure 6.1). We summarize the main results on the inductive learning of primary effects (Section 6.2.1 and 6.2.2) and discuss the implemented extensions of Knoblock's abstraction algorithm (Section 6.2.3).

6.2.1 Automatic selection and use of primary effects

We have formalized the use of primary effects in problem solving, described algorithms for the automatic selection of primary effects, and gave an analytical and empirical evaluation of the resulting search reduction.

The evaluation has shown that the use of primary effects leads to a significant efficiency improvement, and that the savings in search time grow exponentially with problem complexity. On the other hand, an improper selection of primary effects may increase search time, compromise completeness, and cause the generation of unnecessarily costly solutions.

We have presented a condition for ensuring the completeness of primary effects and identified the factors that determine the search reduction. The most important factor is the cost increase C, which determines the trade-off between the efficiency of problem solving and the cost of the resulting solutions. If $C = 1$, then primary effects improve efficiency without compromising solution quality. A larger cost increase leads to generating suboptimal solutions, but it may give a more significant efficiency improvement.

We have used the completeness condition to develop an inductive learning algorithm for selecting primary effects of operators, which guarantees a high probability of completeness and limited increase in solution costs. The experiments have confirmed that the algorithm chooses appropriate primary effects, which exponentially reduce the search.

The selection algorithms allow the user to control the trade-off between the time for selecting primary effects and the quality of the resulting selection. The *Chooser* algorithm uses fast heuristics for selecting primary effects, but it may compromise completeness of search. If we use *Chooser*'s selection, the system has to verify its effectiveness in improving the performance of problem solvers. We will describe performance-testing techniques in Part III. If the selection proves ineffective, the system either discards it or invokes *Completer* to learn additional primary effects. We will describe techniques for testing the performance in Part III.

The *Completer* algorithm learns additional primary effects, to ensure completeness and limited cost increase. The learning process takes significant time, which is usually larger than the time for solving individual problems without primary effects; however, we may amortize it over multiple problems in a domain. The user may specify the desired trade-off between completeness probability and learning time, by setting the success-probability parameters of inductive learning, ϵ and δ.

6.2.2 Improvements to the learning algorithm

We next discuss two open problems related to the automatic selection of primary effects: improving the sample complexity of the learning algorithm, and the automatic demotion of redundant primary effects.

Reducing the sample complexity

The main drawback of the *Completer* algorithm is its sample complexity, which results in long learning times. We outline two approaches to reducing the sample complexity, which may become a subject of future work. Note that we did *not* use them in the current implementation.

The first approach is reducing the hypothesis space, that is, the space of allowed selections of primary effects. Recall that the sample complexity of inductive learning linearly depends on the logarithm of the hypothesis-space size (see Inequality 3.7). We have assumed that the learning algorithm searches among *all* possible selections of primary effects. Therefore, if an operator or inference rule $step_{\mathrm{u}}$ has j candidate effects, then the corresponding number of hypotheses is 2^j. This assumption led to Expression 3.11 for the worst-case sample complexity.

To reduce the number of hypotheses, we may enumerate the candidate effects of $step_{\mathrm{u}}$, from eff_1 toeff_j, and force the learning algorithm to promote them in this order. That is, the algorithm selects eff_1 after the first failure to find a replacing sequence, eff_2 after the second failure, and so on. The number of promoted effects is between 0 and j, which means that the algorithm generates one of $j+1$ possible selections.

Thus, the size of the reduced hypothesis space is $j+1$. If the total number of operators and inference rules is s, the estimated maximum of optimal-solution lengths is $n_{\max}$, and the success-probability parameters are ϵ and δ, then the required number of learning examples for $step_{\mathrm{u}}$ is

$$m = \left\lceil \frac{n_{\max}}{\epsilon} \cdot (\ln \frac{1}{\delta} + \ln s + \ln(j+1)) \right\rceil . \qquad (6.1)$$

The resulting reduction in the number of examples depends on the number j of candidate effects. For example, suppose that the number of operators in the domain is $s = 10$, the maximal length of an optimal solution is $n_{\max} = 10$, and the PAC-learning parameters are $\epsilon = \delta = 0.2$. Then, the dependency between j and m is as shown in Figure 6.4(a), where the solid line is for the full hypothesis space, and the dashed line is for the reduced space. The reduction factor grows with increasing j, as shown in Figure 6.4(b).

On the negative side, the fixed order of promotions prevents the use of selection heuristics and often leads to choosing redundant primary effects. We may control the trade-off between the use of heuristics and the complexity reduction, by adjusting the restrictions on the allowed selections of primary effects. Exploring this trade-off and finding the right restrictions is an open problem

The second approach to reducing the learning time is the generation of appropriate examples, which give a steeper learning curve than random examples. For instance, we considered two learning examples for the Robot Domain in the end of Section 3.5.1, and used them to construct a complete selection of primary effects. In this case, the right choice

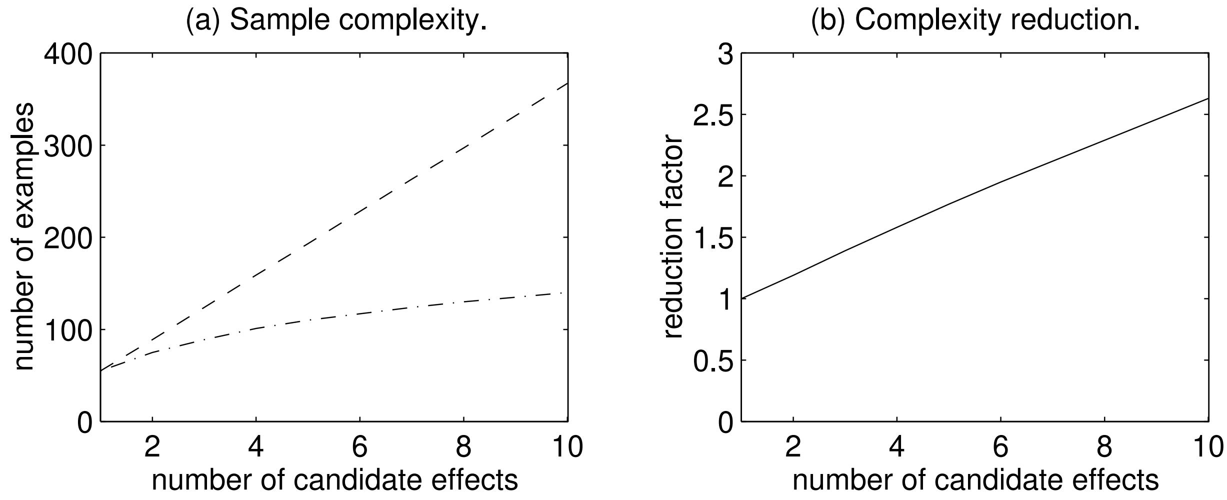


Figure 6.4: Reduction of sample complexity, for $s = 10$, $n_{\max} = 10$, and $\epsilon = \delta = 0.2$. We show (a) the dependency between the number j of candidate effects and the number m of learning examples, for the full hypothesis space (dashed line) and the reduced space (dash-and-dot line), as well as (b) the dependency between j and the reduction factor (solid line).

of examples eliminated the need for a large sample. The construction of an intelligent example generator is an open research direction, related to the automatic construction of training problems in other learning systems.

De-selection of primary effects

The *Chooser* and *Completer* algorithms do *not* demote primary effects. After they have chosen some candidate effect as a new primary effect, it remains primary throughout the learning process and becomes a part of the final selection, even though later choices of primary effects can make it redundant. Moreover, some effects may become redundant if the user either sets a higher limit C for the allowed cost increase, or modifies the domain description, for example, by adding new operators or revising inference rules. To avoid the problem of redundancy, we need an algorithm for identifying and demoting the unnecessary effects.

In Figure 6.5, we give a tentative algorithm for *de-selecting* redundant effects, which loops through all primary effects in the current selection and tests their redundancy. When processing a primary effect, the algorithm demotes it to a candidate effect and then calls the *Test-Completeness* procedure, which checks the completeness of the resulting selection. If this procedure does not find an example demonstrating incompleteness, then the effect remains demoted.

Determining the appropriate number m' of learning examples is an open problem. Note that a large number of examples may *not* guarantee a high probability of completeness, because some de-selected effects may become necessary after demoting effects of other operators. To restore completeness, we have to re-apply the *Completer* algorithm, which may select additional primary effects among the demoted effects.

De-Select-Effects(C, ϵ, δ)
The algorithm inputs the allowed cost increase, C, and the success-probability parameters, ϵ and δ.
It also accesses the operators and inference rules, with the old selection of primary effects.

Determine the required number m' of learning examples,
  which depends on the PAC-learning parameters, ϵ and δ.
For every uninstantiated operator and inference rule $step_u$:
  For every primary effect *eff* of $step_u$:
    Demote *eff* to a side effect of $step_u$.
    If *Test-Completeness*$(step_u, C, n')$ returns false,
      then promote *eff* back to a primary effect.

Test-Completeness$(step_u, C, m')$
Repeat m' times:
  Produce a state I that satisfies the preconditions of $step_u$.
  Generate a full instantiation $step_i$ that matches the state I.
  Produce the goal $G(step_i, I)$ of a replacing sequence.
  Search for a replacing sequence with a cost at most $C \cdot cost(step_u)$.
  If the search fails, then return false (the selection is not complete).
Return true (the selection is probably complete).

Figure 6.5: De-selection of redundant primary effects.

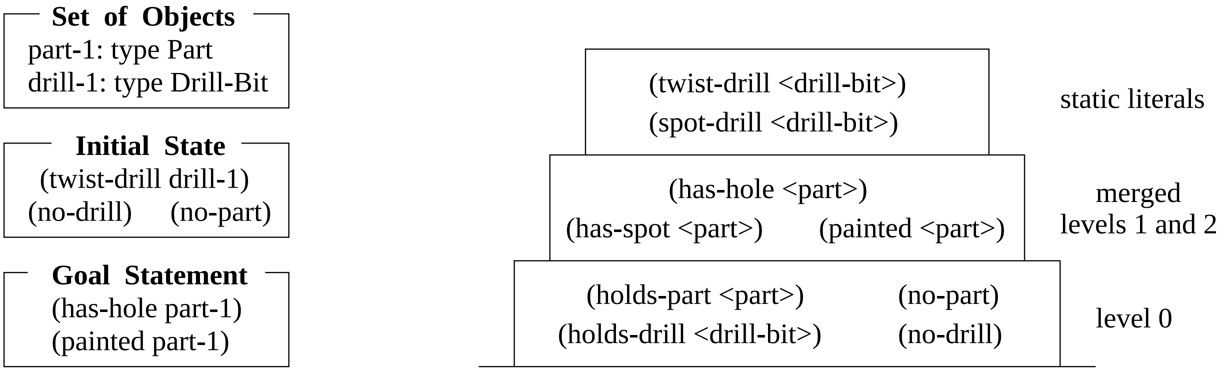
We have *no* empirical results on the performance of the de-selection algorithm. Since completeness does not depend on the value of m', we conjecture that the algorithm will perform well with a small number of learning examples; however, the following use of *Completer* incurs high computational costs, which may outweigh the benefits of the improved selection.

Closely related problems include the design of faster de-selection techniques, as well as the investigation of the trade-off between their running time and the resulting efficiency improvements. Another related problem is the development of procedures that automatically evaluate the trade-off between the branching factor and solution cost, and select an appropriate cost-increase limit C.

6.2.3 Abstraction for the full PRODIGY language

We have extended Knoblock's technique for generating abstraction hierarchies and designed the *Abstractor* algorithm, which constructs abstractions for the full domain language of the PRODIGY4 system. The algorithm imposes constraints on the abstraction levels of preconditions and effects, encodes these constraints as an *abstraction graph,* and then converts the graph into a hierarchy of predicates.

We then showed the relationship between primary effects and abstraction, and used it to develop the *Margie* algorithm, which combines the automatic selection of primary effects with the generation of an abstraction hierarchy. We also designed two auxiliary description



(a) Drilling problem that causes backtracking.

(b) Modified abstraction hierarchy.

Figure 6.6: Modifying the hierarchy in the Drilling Domain, to prevent backtracking across levels.

changers for improving the quality of abstraction, *Refiner* and *Relator*. The *Refiner* algorithm finds the right partial instantiation of predicates, thus preventing the use of over-general predicates, without the risk of a combinatorial explosion. The *Relator* procedure enables the abstraction generator to ignore irrelevant predicates.

The running time of abstraction algorithms is small in comparison with problem-solving time. We tested the resulting abstractions in several domains and demonstrated that they usually reduce search, but may cause the construction of unnecessarily lengthy solutions.

The abstraction generators exploit the syntactic properties of the domain description, which makes them sensitive to a specific encoding of operators and inference rules, and minor syntactic changes may affect the quality of the abstraction hierarchy. The two auxiliary algorithms reduce the dependency on syntactic features of the domain, but do not eliminate it.

The main open problem is the development of abstraction techniques that use semantic analysis of the domain properties. An effective semantic technique should be insensitive to minor isomorphic changes of the domain encoding. Another related problem is extending the classical abstraction techniques for use with other types of search systems and domain languages.

We plan to investigate semantic techniques for revising the abstraction graph in the process of problem solving. As a first step, we have designed a learning algorithm that eliminates backtracking across abstraction levels. The algorithm observes the problem-solving process, detects backtracking episodes, and merges the corresponding components of the abstraction graph.

For example, suppose that we use an abstraction problem solver in the Drilling Domain, described in Section 4.1.2, with the abstraction hierarchy given in Figure 4.2. We consider a problem of drilling and painting a mechanical part, similar to the example in Figure 4.3; however, we now do not allow the use of a spot drill. We give the encoding of this new problem in Figure 6.6(a).

The problem solver first constructs the abstract solution given in Figure 4.4 and then tries to refine it at level 1, which causes a failure and backtracking to level 2. The learning

Removing operators: Identifying unnecessary operators and deleting them from the domain description (Section 6.3.1).

Generating macro operators: Replacing some operators in the domain description with macros constructed from these operators (Section 6.3.2).

Generating new predicates: Replacing some predicates in the domain description with new ones, constructed from conjunctions and disjunctions of the old predicates (Section 6.3.3).

Figure 6.7: Some description changes that has *not* been used in the SHAPER system. We plan to explore them and incorporate into the system, as a part of future work.

algorithm detects this backtracking and merges level 1 with level 2, thus producing the abstraction hierarchy in Figure 6.6(b), and the solver switches to the use of this new hierarchy.

Thus, the learning algorithm ensures that every abstract solution has a low-level refinement, by reducing the number of levels in the hierarchy. It improves the abstraction hierarchy in the process of problem solving, without a preliminary learning stage.

We have no empirical results on the effectiveness of this algorithm in reducing search. Experiments with *Abstractor* have shown that backtracking to higher levels causes a significant increase in search time. We therefore conjecture that the elimination of backtracking would improve performance, despite the reduction in the number of levels.

Bacchus and Yang [1994] developed a different method to prevent backtracking across levels, and used it in the HIGHPOINT abstraction generator. Their algorithm uses rigid syntactic constraints on the relative importance of predicates, which often result in overconstraining and collapse of the hierarchy. It is also more sensitive to the encoding details than the *Abstractor* algorithm. On the positive side, the HIGHPOINT algorithm is very fast, and the resulting hierarchy does not require modification in the process of problem solving.

6.3 Unexplored description changes

The current library of description-changing algorithms is quite small, which limits the capabilities of the SHAPER system. We intend to construct a larger library and use it to evaluate the scalability of SHAPER's top-level control. In particular, we may extend the library by adding some learning systems developed for the PRODIGY architecture, such as ANALOGY [Veloso, 1994] and HAMLET [Veloso and Borrajo, 1994].

We also plan to continue the development of new description changers, which may include not only general-purpose learning algorithms, but also specialized changers for some large-scale domains. The domain-specific algorithms will enable the system to perform more effective description improvements in selected domains. For example, we may design a changer algorithm that simplifies transportation problems by selecting appropriate hubs.

In particular, we intend to explore the description changes listed in Figure 6.7, and construct both general-purpose and specialized algorithms for performing these changes in the SHAPER system. To illustrate the utility of these description improvements, we give

	delivery to the same moon				to different moons		**mean**
	1 pack	2 packs	3 packs	4 packs	2 packs	3 packs	
with extra **fly** operations	0.1	107.9	> 1800.0	> 1800.0	12.2	> 1800.0	> 920.0
without extra operations	0.1	0.8	4.6	52.5	0.2	0.4	9.8

Table 6.2: PRODIGY running times (in seconds) for six problems in the Three-Rocket Domain. The results show that the deletion of unnecessary instances of the **fly** operator reduces the search time by two orders of magnitude.

examples of their use in a simple transportation domain (Section 6.3.1) and in the Tower-of-Hanoi domain (Sections 6.3.2 and 6.3.3).

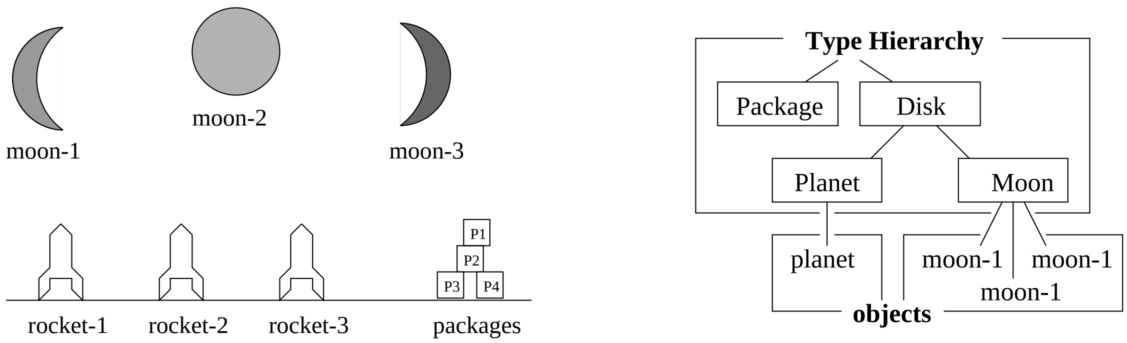
6.3.1 Removing unnecessary operators

In Chapter 1, we have observed that the use of unnecessary extra operators may worsen the performance of a problem solver. For instance, if we add two-disk moves to the Tower-of-Hanoi puzzle, then the PRODIGY system needs more time for solving this puzzle (see Section 1.2.3.

Veloso gave a more dramatic example that illustrates the same point. She constructed the Three-Rocket Domain, where unnecessary operations increase the search time by two orders of magnitude. The domain includes a planet, three moons, three rockets, and several packages (see Figure 6.8a). Initially, all rockets and packages are on the planet. Every rocket can carry any number of packages to any moon; however, it *cannot* return from the moon. The task of a problem solver is to find a plan for delivering certain packages to certain moons. We do not care about the final locations of rockets, as long as every package has reached its proper destination. In Figures 6.8(b) and 6.8(c), we show two problems and their solutions. In the first problem, we have to send three boxes to **moon-1**, whereas the second problem requires delivering boxes to three different moons.

The domain description in Figure 6.8(a) causes an extensive search [Stone and Veloso, 1994]. The PRODIGY system tries to use the same rocket for multiple flights, which causes the exploration of irrelevant branches of the search space. If we increase the number of moons and rockets, the problem-solving time grows exponentially. To improve the efficiency, we construct all instantiations of the **fly** operator and then remove unnecessary instances from the domain description. For example, we may replace the general **fly** operation with three more specific operations, given in Figure 6.8(d). These new operators explicitly encode the knowledge that each rocket can fly only once.

Thus, we have removed some actions from the domain description, leaving a subset of actions sufficient for solving all delivery problems. The new description enables PRODIGY to construct transportation plans with little search. In Figure 6.8, we show the resulting reduction of problem-solving time, for six different problems. When using the general **fly** operator, the PRODIGY system solved three problems and hit the 1800-second time bound on three other problems. After we deleted the unnecessary instances of **fly**, the system solved all problems within 60 seconds.



load(<pack>,<rocket>,<place>)
<pack>: type Package
<rocket>: type Rocket
<place>: type Place
Pre: (at <pack> <place>)
(at <rocket> <place>)
Eff: del (at <pack> <place>)
add (in <pack> <rocket>)

unload(<pack>,<rocket>,<place>)
<pack>: type Package
<rocket>: type Rocket
<place>: type Place
Pre: (in <pack> <rocket>)
(at <rocket> <place>)
Eff: del (in <box> <rocket>)
add (at <box> <place>)

fly(<rocket>,<moon>)
<rocket>: type Rocket
<moon>: type Moon
Pre: (at <rocket> planet)
Eff: del (at <rocket> planet)
add (at <rocket> <moon>)

(a) Three-Rocket Domain.

Initial State
(at rocket-1 planet)
(at rocket-2 planet)
(at rocket-3 planet)
(at pack-1 planet)
(at pack-2 planet)
(at pack-3 planet)

Goal Statement
(at pack-1 moon-1)
(at pack-2 moon-1)
(at pack-3 moon-1)

Solution
load(pack-1, rocket-1, planet)
load(pack-2, rocket-1, planet)
load(pack-3, rocket-1, planet)
fly(rocket-1, moon-1)
unload(pack-1, rocket-1, moon-1)
unload(pack-2, rocket-1, moon-1)
unload(pack-3, rocket-1, moon-1)

(b) Delivery of three packages to the same moon.

Initial State
(at rocket-1 planet)
(at rocket-2 planet)
(at rocket-3 planet)
(at pack-1 planet)
(at pack-2 planet)
(at pack-3 planet)

Goal Statement
(at pack-1 moon-1)
(at pack-2 moon-2)
(at pack-3 moon-3)

Solution
load(pack-1, rocket-1, planet)
fly(rocket-1, moon-1)
unload(pack-1, rocket-1, moon-1)
load(pack-2, rocket-2, planet)
fly(rocket-2, moon-2)
unload(pack-2, rocket-2, moon-2)
load(pack-3, rocket-3, planet)
fly(rocket-3, moon-3)
unload(pack-3, rocket-3, moon-3)

(c) Delivery of three packages to different moons.

fly(rocket-1, moon-1)
Pre: (at rocket-1 planet)
Eff: del (at rocket-1 planet)
add (at rocket-1 moon-1)

fly(rocket-2, moon-2)
Pre: (at rocket-2 planet)
Eff: del (at rocket-2 planet)
add (at rocket-2 moon-2)

fly(rocket-3, moon-3)
Pre: (at rocket-3 planet)
Eff: del (at rocket-3 planet)
add (at rocket-3 moon-3)

(d) Search-saving encoding of the **fly** operation.

Figure 6.8: Removing unnecessary operators in the Three-Rocket Domain: Initial description causes an extensive search, whereas the new encoding of the **fly** operation enables PRODIGY to solve delivery problems with little search.

	number of a problem						mean
	1	2	3	4	5	6	time
standard domain description	2.0	34.1	275.4	346.3	522.4	597.4	296.3
use of a hand for moving disks	0.6	0.6	3.9	11.7	1.5	2.8	3.5
trays in place of pegs	35.1	4.2	> 1800.0	> 1800.0	479.0	> 1800.0	> 986.4
standard desc. with abstraction	0.5	0.4	1.9	0.3	0.5	2.3	1.0

Table 6.3: PRODIGY performance on six problems in the Tower-of-Hanoi Domain, using different domain descriptions. We give running times in seconds for the standard domain encoding (see Figure 6.9a), domain with a hand for moving disks (Figure 6.9b), Holte's domain with trays in place of pegs (Figure 6.9c), and standard encoding with an abstraction hierarchy.

6.3.2 Replacing operators with macros

The learning of macro operators is one of the oldest approaches to improving domain descriptions, which dates back to the GPS problem solver [Newell *et al.*, 1960] and STRIPS system [Fikes *et al.*, 1972]. Researchers have applied this approach in many AI systems and observed that simple techniques for generating macro operators usually give disappointing results [Minton, 1985; Etzioni, 1992].

On the other hand, a synergetic use of macros with other description changes often helps to improve performance. For instance, Korf [1985a,1985b] integrated macro operators with an implicit use of abstraction and primary effects. Yamada and Tsuji [1989] utilized macros in conjunction with heuristics for selecting appropriate operators. The authors of the Soar system developed an advanced strategy for learning macros, called *chunking,* and combined it with mechanisms for partially instantiating learned macros and deleting unnecessary instances [Laird *et al.*, 1986; Tamble *et al.*, 1990]. In the $\mathcal{S}$HAPER system, we may use a generator of macro operators as an auxiliary mechanism for improving the effectiveness of other description changers.

For instance, consider the description of the Tower-of-Hanoi puzzle in Figure 6.9(b). According to this domain encoding, we use a robot hand for moving disks in the puzzle, and the standard move consists of two steps: picking a disk from its initial location and putting it onto another peg. Note that the *Abstractor* algorithm fails to generate a hierarchy for this version of the puzzle, because the predicate (in-hand <disk>) requires additional constraints and causes the hierarchy to collapse into a single level.

We may convert this version of the puzzle into the standard description (see Figure 6.9a) by replacing the operators with two-step macros, as shown in Figure 6.10. The conversion simplifies the domain description and allows us to discard the in-hand predicate, but it does not improve efficiency of search. Surprisingly, this simplification causes a significant *worsening* of performance (see Table 6.3).

On the other hand, if we use macro operators in conjunction with abstraction, then they enable us to reduce the problem-solving time. After we construct the macros and eliminate the in-hand predicate, *Abstractor* generates the usual three-level hierarchy (see Figure 6.10c), which improves the efficiency of PRODIGY search. We give the results of using this hierarchy in the last row of Table 6.3.

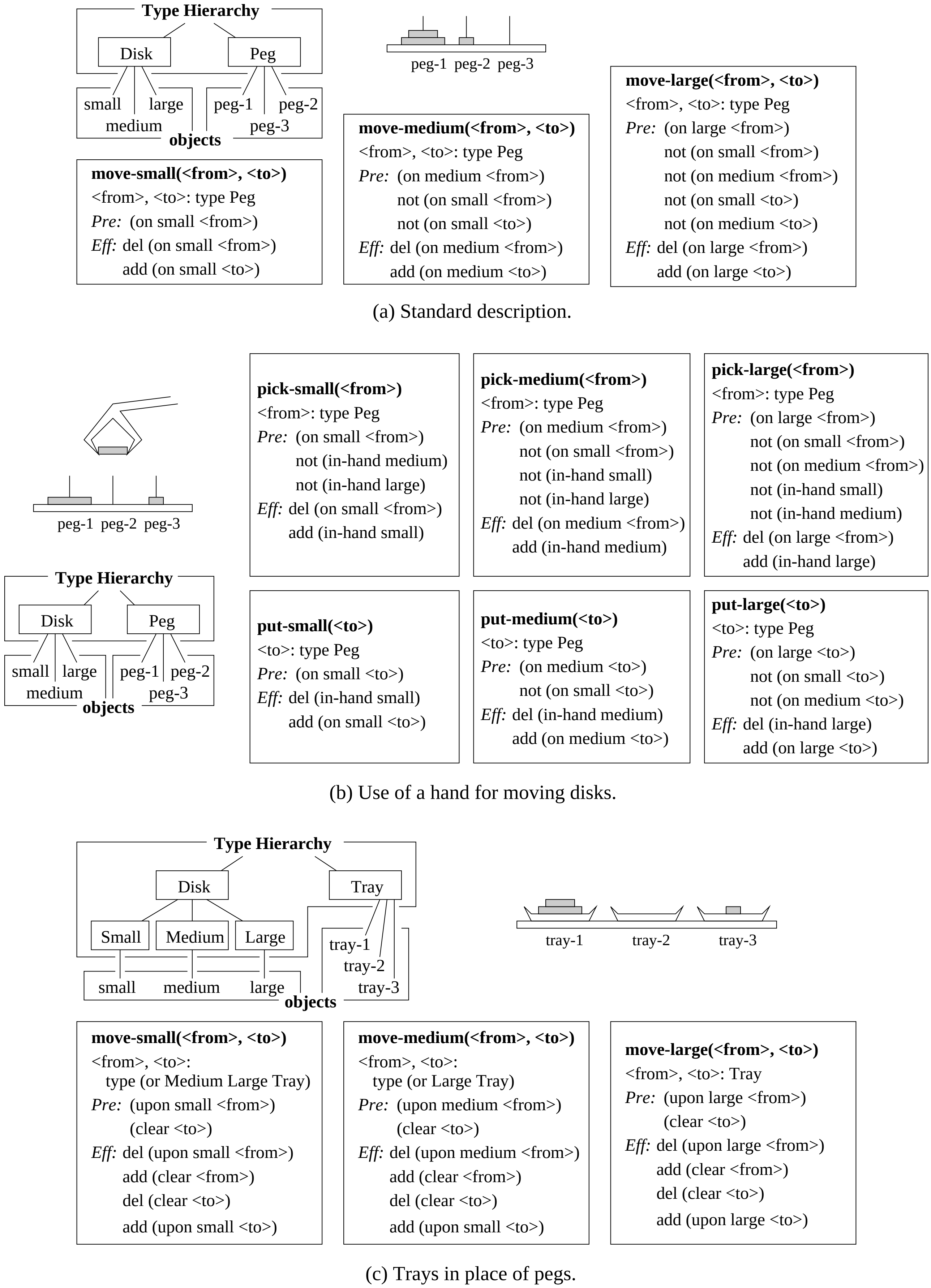
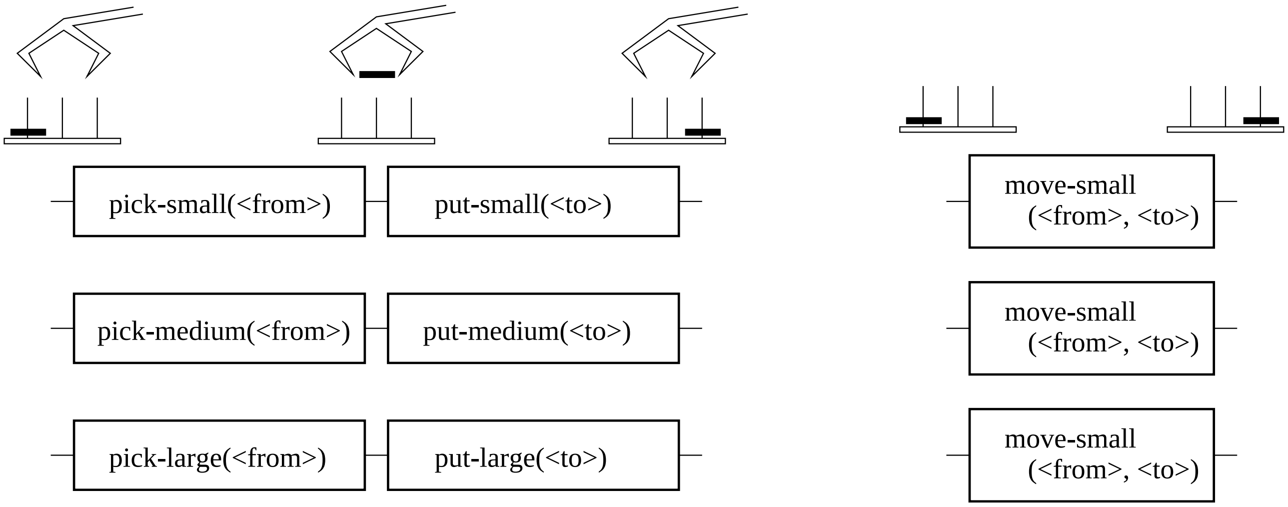
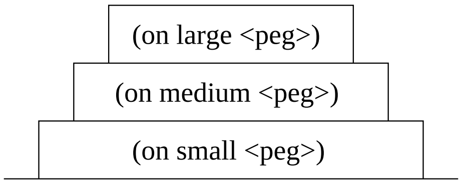


Figure 6.9: Alternative descriptions of the Tower-of-Hanoi Domain.



(a) Sequences of original operators.

(b) Corresponding macros.



(c) Abstraction hierarchy based on macro operators.

Figure 6.10: Replacing operators with macros in the Tower-of-Hanoi domain given in Figure 6.9(b). This description change does *not* reduce search by itself; however, it enables the *Abstractor* algorithm to construct a three-level abstraction hierarchy.

6.3.3 Generating new predicates

When the human operator encodes a new domain, she has to specify predicates for representing states of the simulated world. The choice of predicates determines the explicit features of the state description, and affects the performance of search and learning algorithms.

In particular, Holte has observed that most abstraction algorithms are sensitive to the user's selection of predicates. To illustrate this observation, he has considered the version of the Tower-of-Hanoi domain in Figure 6.9(c), with trays in place of pegs.

Note that the meaning of the upon predicate in this domain differs from that of the on predicate in the standard description. If a disk is the largest in a stack, than it is upon the tray; otherwise, it is upon the next larger disk. The (clear <disk>) predicate means that <disk> is the topmost in a stack, that is, there is no smaller disks above it; similarly, (clear <tray>) means that the specified tray is empty. For example, if all three disks are stacked on the first tray, then we encode the state as follows:

```
(upon large tray-1)     (clear small)
(upon medium large)     (clear tray-2)
(upon small medium)     (clear tray-3)
```

```
(on large <tray>) =
(upon large <tray>)
```

```
(on medium <tray>) =
(or (upon medium <tray>)
    (and (upon large <tray>)
         (upon medium large)))
```

```
(on large <tray>) =
(or (upon small <tray>)
    (and (upon medium <tray>)
         (upon small medium))
    (and (upon large <tray>)
         (upon small large))
    (and (upon large <tray>)
         (upon medium large)
         (upon small medium)))
```

(a) Use of conjunctions and disjunctions.

```
(on <disk> <tray>) =
(or (upon <disk> <tray>)
    (and (above <other-disk> <tray>)
         (upon <disk> <other-disk>)))
```

(b) Use of recursion.

Figure 6.11: Defining new predicates for the Tower-of-Hanoi domain given in Figure 6.9(c). The resulting predicates allow the conversion to the standard encoding of the puzzle (see Figure 6.9a), which improves the efficiency and enables further improvement through the use of abstraction.

Although this description of the Tower-of-Hanoi puzzle is isomorphic to other versions of the puzzle, it makes most problems hard for the PRODIGY system. We show the resulting running times in the next-to-last row of Table 6.3; note that PRODIGY solves only three problems within a 1800-second time bound. Moreover, the changer algorithms in the current version of the $\mathcal{S}$HAPER system fail to improve the domain description.

To remedy this situation, we need to develop a changer algorithm that generates appropriate new predicates, defined through conjunctions and disjunctions of old ones, and uses them to construct a new description of operators. For example, we may define the `on` predicate in the Tower-of-Hanoi domain through partial instantiations of the `upon` predicate, as shown in Figure 6.11(a), which allows the conversion of Holte's domain description into the standard description.

Note that we may use recursion to construct a more general definition of `(on <disk> <tray>)`, given in Figure 6.11(b), which is valid for an arbitrary number of disks. The development of algorithms that use recursion and other advanced techniques in problem reformulation is a challenging research problem [Schmid and Wysotzki, 1996; Mühlpfordt and Schmid, 1998].

6.4 Toward a theory of description changes

Researchers have implemented many systems that automatically improve domain descriptions, by static analysis and learning (see the review of related work in Section 1.3.2); however, they have done little investigation of the common principles underlying these systems.

An important research direction is the development of general methods for the design and evaluation of description-changing mechanisms.

We summarize preliminary results of developing a framework for the development of changer algorithms (Section 6.4.1), and then outline two open problems related to the analysis of representation improvements. First, we point out the need for standard techniques that allow the evaluation of new description changers and problem solvers (Section 6.4.2). Second, we discuss the use of analytical techniques for estimating the utility of specific domain descriptions (Section 6.4.3).

6.4.1 Systematic approach to the design of changer algorithms

The design and implementation of effective description-changing algorithms is usually a complex research task, and there are no standard techniques or guidelines for approaching this task. The related open problems include formalizing the methods used in the work on changer algorithms, developing a systematic approach to the implementation of these algorithms, and applying this approach to designing new description changers.

We have described a technique for specifying important properties of description changers (see Section 1.4.2), which is the first step toward a general framework for the development and analysis of description-changing mechanisms in a variety of AI systems. Although our specifications are semi-informal, they simplify the task of designing description changers and evaluating their performance. In particular, they help to abstract the major decisions in the development of a changer algorithm from details of implementation. When constructing a new changer, we first determine its desirable properties, and then implement a learning or static-analysis algorithm with those properties.

We intend to formalize the structure and language of specifications, study methods for determining which specifications describe useful description changes, and techniques for implementing changer algorithms according to specifications. We also plan to apply these techniques to constructing new description changers for use in the $\mathcal{S}$HAPER system.

A more challenging problem is to develop a system that *automatically* generates new description changers, which may include the construction of algorithms according to the user's specification, as well as the learning of description-changing techniques by analyzing examples of alternative domain descriptions. This research direction is related to Minton's [1996] recent work on the automated construction of constraint-satisfaction programs.

6.4.2 Framework for the analysis of description changers

We may estimate the utility of a representation-changing algorithm by analyzing its effect on the search space of a problem solver. In particular, researchers have applied this approach in evaluating the efficiency of search with macro operators [Korf, 1987; Etzioni, 1992], abstraction hierarchies [Knoblock, 1991; Bacchus and Yang, 1992], and control rules [Cohen, 1992], as well as in comparing alternative commitment strategies [Minton *et al.*, 1991; Knoblock and Yang, 1994]. The search-space analysis helps to predict the performance of a speed-up technique before implementing it.

We used a similar approach to analyzing the efficiency of search with primary effects and predicted an exponential efficiency improvement. The analysis revealed the factors that affect the trade-off between problem solving time and solution quality. Experiments on the use of primary effects in ABTWEAK and PRODIGY confirmed the analytical predictions.

Even though researchers have analyzed several types of description changes, they have not developed a general framework for the evaluation of search-reduction techniques. We plan to generalize the previous analytical results and apply them to the evaluation of new changer algorithms. The long-term purpose of this work is the development of standard formal methods for estimating the efficiency improvements, as well as combining these methods with empirical evaluation techniques.

We also intend to address the main drawbacks of previous analytical techniques. First, the developed techniques sometimes do not provide sufficient accuracy, leading to disparate predictions of the search reduction. For instance, the proposed estimates for ordered abstraction vary from exponential speed-up [Knoblock, 1991] to gross inefficiency [Smith and Peot, 1992].

Second, the analysis of classical problem solvers usually does not address the trade-off between their search time and solution quality. We made a first step to bridging this gap in the evaluation of primary effects, which shows the trade-off between search reduction and solution costs.

Third, researchers usually estimate problem-solving time by the number of nodes in the search space. This technique gives accurate predictions for the efficiency of breadth-first search; however, it is less effective for the analysis of depth-first systems, which often find a solution after exploring a small portion of the available space. Observe that the analysis of search with primary effects has the same drawback and, hence, the analytical predictions often differ from empirical results on the use of primary effects in PRODIGY.

6.4.3 Analysis of specific descriptions

Most analytical techniques are aimed at evaluating the average-case effectiveness of description changers. An important related problem is the evaluation of specific domain descriptions. For example, we may need to estimate the utility of a specific abstraction hierarchy or selection of primary effects. The $\mathcal{S}$HAPER system allows the use of evaluation heuristics in selecting appropriate descriptions, and accurate heuristics may significantly improve the system's performance.

We give preliminary results on the analytical evaluation of the chosen primary effects. The evaluation is based on Inequality 3.5, which gives an approximate condition for search reduction (see Section 3.3). Recall that this condition is based on the comparison of the cost increase C with an expression that includes four other characteristics of the domain (see Figure 3.12): the total number PE of primary effects, the number E of all effects, the number N of nonstatic predicates, and the branching factor BF. The use of primary effects improves the efficiency if

$$C < \frac{\log E + \log BF - \log(2 \cdot N)}{\log PE + \log BF - \log(2 \cdot N)}.$$

After selecting primary effects, the $\mathcal{S}$HAPER system may compute the five values used in this inequality and check whether the selection satisfies the efficiency condition. It can readily determine the values of PE, E, and N, in one pass through the domain encoding. To estimate the average branching factor BF, the system has to analyze the search spaces of several problems.

The computation of the average cost increase C requires a modified version of the *Completer* algorithm, which constructs replacing sequences for a collection of learning examples and returns the mean of the resulting cost increases. Note its running time is much greater than the time required to estimate the other four values.

For example, suppose that we apply this analysis to the Robot Domain in Figure 3.2 (see Section 3.1.2), with the primary effects given in Table 3.5. The total number of primary effects is $PE = 7$, the number of all effects is $E = 10$, the number of nonstatic predicates is $N = 3$, and the average cost increase C is close to 1. The average branching factor BF depends on a specific solver and its search heuristics; in most cases, it is between 2 and 20. These values satisfy the condition for the search reduction, which implies that the selected primary effects improve performance.

The described evaluation has several drawbacks and requires further improvements. In particular, the technique for estimating the cost increase C takes significant time and requires a generator of learning examples. Furthermore, we have derived Expression 3.5 by analyzing the size of the available search space, and it may give inaccurate results for depth-first problem solvers.

Part III

Top-level control

Chapter 7

Generation and use of multiple representations

Every problem-solving effort must begin with creating a representation for the problem—a problem space in which the search for the solution can take place.
— Herbert A. Simon [1996], *The Sciences of the Artificial.*

We have reviewed problem-solving algorithms in the PRODIGY system (Chapter 2), and described learning and static-analysis algorithms for changing the descriptions of PRODIGY domains (Part II). We now present top-level tools for the centralized, synergetic use of the available algorithms and then discuss a utility model for evaluating their effectiveness.

The top-level tools form the system's "control center," which allows the human operator to use problem-solving and description-changing algorithms. We also use the control center as the intermediate layer between the system's algorithms and the top-level learning mechanism that automatically controls these algorithms.

We will describe the automatic control of the system in Chapters 8–10. The control mechanism selects appropriate description changers and problem solvers, and accesses them through the same set of top-level tools as the human user. We provide additional user tools that allow sharing of responsibilities between the human operator and automatic control.

We begin by describing the top-level operations in the $\mathcal{S}$HAPER system, including specification of initial domain descriptions, application of changer algorithms to generate new descriptions, and use of PRODIGY search algorithms with available domain descriptions to solve problems (Section 7.1).

We then abstract from specific features of the PRODIGY architecture and develop a generalized model of generating and using multiple domain descriptions (Section 7.2). We utilize this model in constructing the automatic-control mechanism, which is independent of specific PRODIGY features and can be applied in other problem-solving systems.

We also develop a general model for evaluating the results of problem solving, which allows many different types of utility functions (Section 7.3). We apply this model to statistical analysis of the system's performance, which underlies the automatic selection of solver algorithms and domain descriptions.

Finally, we discuss the simplifying assumptions that underlie both the $\mathcal{S}$HAPER system and the abstract model of problem solving with multiple descriptions (Section 7.4). We also review the role of the human user in providing the initial knowledge and guiding the system's search for effective representations.

7.1 Use of problem solvers and description changers

We describe the use of problem-solving and description-changing algorithms in the $\mathcal{S}$HAPER system. We consider the generation and storage of domain descriptions, their use in constructing new representations, and problem solving with the resulting representations. We also discuss the use of conditions that limit the applicability of the available algorithms and domain descriptions. We summarize the tools available to the user for manual control of the system, and identify the subset of these tools used by the automatic-control mechanism.

We first describe the storage of multiple domain descriptions and the basic operations on descriptions (Sections 7.1.1 and 7.1.2). We then describe the use of problem-solving and description-changing algorithms (Section 7.1.3 and 7.1.4), and the construction of new representations (Section 7.1.5). Finally, we summarize the main top-level operations (Sections 7.1.6).

7.1.1 Elements of the domain description

We have defined a *problem description* as an input to a problem solver. A *domain description* is the part of the problem description common for all problems in the domain. The PRODIGY system allows the user to store the encoding of a domain in a separate "domain" file, and to input specific problems by loading the rest of their description from "problem" files.

We have presented the main elements of a PRODIGY domain description in Chapter 2, and Sections 3.1.2 and 4.1.2; it includes object types, operators, inference and control rules, primary effects, and an abstraction hierarchy.

The $\mathcal{S}$HAPER system uses multiple domain descriptions in problem solving. We have considered the generation and improvement of three elements of the description: primary effects, abstraction, and control rules. These elements can vary from one description to another. We do *not* allow multiple versions of the other description elements in the current implementation of $\mathcal{S}$HAPER; thus, the object types, operators, and inference rules are the same in all alternative descriptions of a domain.

We have introduced this limitation because the implemented description changers do not modify object types, operators, or inference rules. We use the limitation *only* in defining data structures for storing alternative descriptions, which is a part of the lower layer in the system's implementation. The other mechanisms do not rely on this restriction, and we can readily extend the system to allow automatic changes of other description elements.

Storage and use of description elements

The PRODIGY system keeps the domain description in a large "central" data structure, which contains all six elements of the description, as well as the current problem and the state of the system.

If we use multiple versions of a description element, we keep them in separate data structures. We have implemented structures for the separate storage of primary effects, abstraction, and control-rule sets. If we need to use a new description element in problem solving, we first incorporate it into the central domain description.

For example, suppose that we need to use a new abstraction in problem solving, and the domain data structure currently contains a different abstraction. We first extract the current abstraction from the domain structure and store it separately. We then incorporate the new abstraction into the domain description and solve problems with the resulting description.

We have developed procedures for incorporating new elements into the domain data structure, as well as for extracting elements of the central domain description and storing them separately. We have also implemented procedures for comparing description elements and for loading the user's specification of new elements, which we outline next.

Detecting identical elements

We have designed algorithms for detecting identical primary effects, abstraction graphs, and control-rule sets. We use them to compare alternative description elements, generated by different changer algorithms. The detection of identical descriptions improves the accuracy of the statistical performance analysis, used for the automatic selection of representations.

The identity test for primary-effect selections is a simple effect-by-effect comparison. If some effect is primary in one selection but not in the other, then the selections are distinct.

We detect identical abstractions by comparing the corresponding abstraction graphs, which have been described in Chapter 4.2.2. We first check whether the graphs have the same number of components and their components are pairwise identical. If the graphs pass this test, we then check whether the transitive closures of the graphs have the same edges.

The comparison of control-rule sets is less reliable. We have not provided a means for checking the equivalence of two control rules. The system detects the identity of two sets only if they are composed of the same rules. If two sets contain distinct control rules, they are considered different even if their rules are equivalent.

Note that checking the equivalence of control rules is a complex problem. The conditions of PRODIGY control rules often include calls to Lisp functions, and therefore the general case of the comparison problem is undecidable.

Other comparison operations

We also provide operations for checking whether a primary-effect selection is a subset of another selection, whether an abstraction graph is finer-grained than another graph, and whether a control-rule set belongs to another set. We use these operations to construct conditions of heuristic preference rules for selecting among available representations; we will describe these rules in Chapter 10.

We say that a selection of primary effects is a *subset* of another selection if every primary effect of the first selection is also primary in the second one. The test procedure compares two selections effect by effect.

We say that an abstraction graph is *finer-grained* than another graph if we can construct the second graph from the first one by adding new edges and then collapsing connected com-

Test-Finer-Granularity(Abs_1,Abs_2)
1. Verify that Abs_1 has at least as many components as Abs_2.
2. For every strongly connected component of Abs_1,
   verify that it is a subset of some component of Abs_2.
3. For every two components of Abs_1,
   if there is a path joining these components in Abs_1,
   then verify that either
   - they are subsets of the same component of Abs_2 or
   - there is a path joining their respective superset components in Abs_2.

Figure 7.1: Checking whether an abstraction graph Abs_1 is finer-grained than a graph Abs_2; if at least one of the tests gives a negative result, then Abs_1 is *not* finer-grained than Abs_2.

ponents. To put it more formally, for every two literals l_1 and l_2 in the domain description, if there is a path from l_1 to l_2 in the first graph, then there is also a path from l_1 to l_2 in the second graph. Since edges of the abstraction graph encode constraints on the abstraction levels of literals, a finer-grained graph imposes fewer constraints and may allow the construction of a hierarchy with more levels. In Section 4.3, we have discussed the notion of finer granularity in more detail.

In Figure 7.1, we outline an algorithm that checks whether an abstraction graph Abs_1 is finer-grained than another graph Abs_2. If at least one of the tests listed in Figure 7.1 gives a negative result, then Abs_1 is *not* finer-grained, and the algorithm terminates with the negative answer.

The algorithm is based on testing whether strongly connected components of Abs_1 are subsets of Abs_2's components. Recall that we represent an abstraction graph as a collection of strongly connected components, with edges between them (see Section 4.2.2). Every component of a graph encodes a set of literals. This representation allows a fast test of the subset relationship between a component of Abs_1 and a component of Abs_2. Note that Step 1 of the algorithm is for efficiency; it is not needed for correctness of the test.

Finally, we provide a procedure that checks whether a set of control rules is contained in another set; that is, every rule from the first set is also in the second one. Since the system cannot check the equivalence of two rules, it identifies the subset relationship only if the larger set contains all the same rules, not merely equivalent ones.

Loading the user's specifications

We extended the PRODIGY domain language to allow the specification of multiple primary-effect selections, abstractions, and control-rule sets, in a domain-encoding file. When loading the domain, the system creates a library of the specified description elements. The user may later specify additional elements and add them to the library. The stored description elements are used in constructing domain descriptions.

7.1.2 Domain descriptions

We next disucss the storage and use of multiple descriptions in the $\mathcal{S}$HAPER system. The storage mechanism is based on the fact that descriptions differ only in primary effects, abstraction, and control rules. We store these three elements for each of the multiple descriptions. The other, invariant parts of the domain description are stored in the central data structure that represents the domain.

When using a description to solve some problem or generate a new description, we first incorporate the corresponding primary effects, abstraction, and control rules into the central data structure. We then use a solver or changer algorithm with the updated data structure.

The mechanism does *not* require storing a separate copy of the domain structure for each description and, thus, it allows significant memory savings. Its main drawback is the limitation on the changeable parts of the description, which reduces the system's flexibility.

Description triples

A *description triple* is a data structure that includes a selection of primary effects, an abstraction hierarchy, and a set of control rules. We store alternative domain description in the form of such triples.

Some of the three slots in a description triple may be empty. If the primary-effect slot is empty, then all effects are considered primary, which is equivalent to problem solving without the use of primary effects. If the abstraction slot is empty, then the abstraction has one level, which means problem solving without abstraction. Finally, if the control-rule slot is empty, then the solver algorithm does not use control rules.

We have provided a procedure for incorporating a description triple into the domain data structure. The procedure first adds the three elements to the domain structure; it then processes the updated structure to improve the efficiency of using the resulting description. We have also implemented a procedure that extracts the three current description elements from the domain structure and stores them as a new triple.

In addition, we have provided a procedure for detecting identical description triples. The procedure performs element-by-element comparison of triples, using the element-identity tests. The test of description identity is sound, but *not* complete. The incompleteness results from the system's inability to recognize equivalent control rules.

Finally, we implemented a user-interface command for defining new description triples, by combining elements from the library of the user-specified primary-effect selections, abstraction hierarchies, and control-rule sets. This command allows the user to add new domain descriptions to the system's collection of alternative descriptions.

Applicability conditions

We often construct new descriptions for a specific class of problems. The use of such a specialized description for other problems may lead to inefficiency, incompleteness, or even an execution error. Specialized descriptions may be specified by the user or generated automatically. For example, if we apply *Abstractor* to construct a hierarchy for a limited goal set, then we cannot use the new hierarchy in solving problems with other goals.

We use *applicability conditions* to encode the limitations on the use of descriptions. We define an applicability condition for every description triple. If a problem does not satisfy a triple's condition, we should not use this description in solving it. If the human operator tries to use the description anyways, the system gives a warning and asks for confirmation.

An applicability condition consists of three parts: a set of allowed goals, a set of static literals, and a collection of Lisp functions that input a PRODIGY problem and return `true` or `false`. A problem satisfies the condition if the following three restrictions hold:

1. All goals of the problem belong to the set of allowed goals.
2. The problem's initial state contains all static literals specified in the condition.
3. All functions from the third part of the condition return `true` for this problem.

The system generates automatically the first two parts of an applicability condition; however, the user may also specify or modify them. These two parts allow the system to restrict the goals and static properties of the domain, and use the restrictions in constructing an abstraction hierarchy and selecting primary effects.

If the goal-set slot of an applicability condition is empty, then a problem may have any goals. Similarly, if the static-literal slot is empty, then there is no restriction on a problem's initial state.

The third part of an applicability condition is for additional user-specified restrictions. The user encodes her limitations on appropriate problems in the form of Lisp functions.

We have implemented a procedure for constructing a conjunction of several applicability conditions. The goal set in the conjunction condition is the intersection of the goal sets in original conditions, the static-literal set is the union of the original sets, and the function list is the union of function lists.

We have also provided an extension to the PRODIGY language for defining the applicability conditions of user-specified description triples, and implemented a user-interface command for modifying the conditions of automatically generated descriptions.

Construction history

The system generates new descriptions by applying changer algorithms to old descriptions. After constructing a new description, SHAPER stores it as a triple and adds a record on the "construction history." The record includes the changer algorithm that generated the new description, the old triple to which the algorithm was applied, and the construction time. We also add the new triple to the old triple's list of "child" descriptions.

Sometimes, the application of different changer algorithms to different domain descriptions results in the construction of the same new description. The system then stores multiple records, which encode all construction histories.

We may use construction records to trace the full history of a description. Suppose that the system has applied several changer algorithms in a row and thus produced a series of descriptions. If we need to determine the history of the last description in the series, we use construction records to trace all intermediate descriptions, the changers used to generated them, and the overall running time. We use this information in selecting problem solvers, and in deciding which changer algorithms can make further improvements.

7.1.3 Problem solvers

A *problem solver* in the SHAPER system is an algorithm that inputs a problem, domain description, and time bound, and then searches for a solution to a problem. A solver terminates when it finds a solution, exhausts its search space, or hits the time bound.

The top-level evaluation and control mechanism allows the use of any problem-solving algorithms in the system. The current implementation, however, supports only algorithms that use the PRODIGY data structures; in practice, it limits us to the use of PRODIGY search engines.

We used this limitation *only* in the lower-layer procedures that control the application of solver algorithms. The other parts of SHAPER do not rely on this restriction, and the implementation of more general application procedures would allow the use of solver algorithms from other problem-solving systems.

We now outline the use of problem solvers in SHAPER. In particular, we describe fixed sequences of changer and solver algorithms, and conditions that limit the applicability of solver algorithms.

Solver algorithms and sequences

We constructed multiple problem solvers by setting different values of the knob variables that determine the behavior of PRODIGY's general search engine. We have described the main knobs in Section 2.4.2; they control the use of decision points, backtracking, trade-off between backward and forward chaining, operator and goal preferences, and search depth.

A problem solver in SHAPER must *not* have knob variables; that is, the user must fix all knob values when defining a solver. If the user needs to employ several different knob settings, she has to define several solvers.

The input of a solver algorithm consits of a problem, description triple, and time bound. The user has the option to omit a time bound, in which case the algorithm may run indefinitely long. If it does not find a solution or exhaust the search space in reasonable time, the user has to invoke the keyboard interrupt.

We often perform a series of problem-specific description changes before applying a solver. For example, we may select primary effects for a specific problem, generate an abstraction for the problem's goal set, and then apply a solver algorithm.

We thus apply a sequence of changer algorithms followed by a solver algorithm. We may define a fixed sequence of algorithms, which includes several problem-specific changers and a solver, and apply it to different problems. We call it a *solver sequence*.

Note that the application of a solver sequence is similar to the application of a solver algorithm. The sequence inputs a problem, description, and time bound, and attempts to find a solution. If the system hits the time bound during the execution of algorithms in the sequence, it terminates the execution. If one of the changer algorithms fails to produce a new description, the system also terminates the execution of the sequence.

The top-level control operations do *not* distinguish between solver sequences and "simple" solvers. In other words, we view these sequences as special cases of problem solvers. We will say more about the use of solver sequences in Section 7.2.1.

Restricting domain descriptions

We often use solvers that work only for specific classes of domain descriptions. For example, if a solver algorithm is based on the use of abstraction, then it requires descriptions with an abstraction hierarchy. On the other hand, if a solver sequence includes an abstraction generator, we should apply it to descriptions without a hierarchy.

For every solver algorithm and solver sequence, we may specify a condition that limits appropriate descriptions. We check the triple that encodes the description, as well as its construction history. The condition consists of five parts:

1. Specified description elements; that is, the slots of a triple that must not be empty.
2. Elements *not* specified in a triple; that is, the slots that must be empty.
3. Changer algorithms used in constructing a description or its predecessors.
4. Changers *not* used in constructing a description or its predecessors.
5. A Lisp function that inputs a description and returns true or false.

We use a description only if it satisfies all five parts of the condition. That is, all elements specified in Part 1 are present in the description and all elements of Part 2 are absent; all changers listed in Part 3, and none of the changers listed in Part 4, have been used in constructing the description; and the function in Part 5 returns true for the description.

For example, we may specify that the description must have primary effects and abstraction, but not control rules; that we must have used *Abstractor* and *Chooser*, but not *Margie*, in generating the description; and that it has to satisfy a test function that encodes several additional constraints.

When specifying a condition, we may skip any of its five parts. For example, we may omit the restriction on the changers used in generating the description. The system uses only the explicitly specified parts of the condition in selecting appropriate descriptions.

Restricting problems

We may further limit the use of a solver algorithm or solver sequence by restricting the problems to which we apply it. The encoding of this restriction is similar to the applicability condition of a domain description. The restriction consists of three parts: a set of allowed goals; a set of static literals that must hold in a problem's initial state; and a Lisp function that inputs a description triple and PRODIGY problem, and returns true or false.

Note that the third part differs in two ways from the applicability condition of a domain description. First, we use one test function rather than a collection of functions. We never need to compute a conjunction of solvers' restrictions, which makes the use of a single test function more convenient. Second, the test function inputs not only a problem but also a description.

When we use a solver with some specific description, we have to compute the conjunction of their applicability conditions. We have implemented a procedure that performs this computation in two steps. First, it makes the solver's condition more specific, by substituting the specified description into the test function. Second, it generates the conjunction of the resulting specific condition and the description's applicability condition.

We have developed an extension to the PRODIGY language that allows the user to encode a restriction on appropriate domain descriptions and problems. If the user later tries to use a solver with a description or problem that does not satisfy the restriction, the system gives a warning and asks for confirmation.

7.1.4 Description changers

A *description changer* is an algorithm that inputs a domain description and time bound, and tries to construct a new description by improving the original one. A changer terminates when it generates a new description, finds out that it cannot improve the input description, or runs out of time.

We explain the role of description changers in $\mathcal{S}$HAPER, define sequences of changer algorithms, and discuss conditions that limit the applicability of changers.

Changer algorithms and sequences

We have presented several novel algorithms for improving domain descriptions in Part II. We constructed $\mathcal{S}$HAPER's description changers by setting specific values of the knob variables in these algorithms.

A description changer in $\mathcal{S}$HAPER must *not* have knob variables; that is, the user must fix all knob values when defining a changer. A changer's only inputs are a description triple and time bound; the bound specification is optional. Note that all changer algorithms in Part II terminate in polynomial time. In practice, they usually take much less time than problem solving, and their invocation without a time bound never results in excessively long execution.

We may arrange several changer algorithms in a sequence and always use them together. We call this arrangement a *changer sequence*. When we apply a changer sequence, the system consecutively executes its algorithms. It returns the final description and discards the intermediate descriptions. If some algorithm in the sequence fails to construct a new description, then the system terminates the execution of the sequence and returns failure. If the system hits the time bound, it also terminates the execution of the sequence. We may specify a bound for the overall sequence, as well as bounds for individual algorithms in the sequence.

Note that the application of a changer sequence is similar to the application of a changer algorithm. The sequence inputs a description triple and time bound, and attempts to generate a new description. The top-level control does *not* distinguish between changer sequences and "simple" changers. We will say more about the use of changer sequences in Section 7.2.1.

Problem-specific changers

Some changer algorithms use information about a specific given problem to generate a more effective representation. For example, problem-specific versions of *Chooser* and *Margie* utilize the knowledge of a problem's goals, static literals, and available instances of domain variables.

A problem-specific changer inputs a description triple, current problem, and, optionally, a time bound. If it successfully generates a new description, the system uses it *only* for solving the current problem. We use such changers to construct solver sequences.

SHAPER does *not* reuse the resulting descriptions for solving other problems. After applying a solver sequence, the system discards all generated descriptions and keeps only the resulting solution.

Restricting input descriptions

We usually need to limit the applicability of changer algorithms and changer sequences. For example, we should apply *Abstractor* only if the current description has no abstraction hierarchy; similarly, we should use *Chooser* only if we have not yet selected primary effects. We thus restrict descriptions that can be an input to a changer. We encode this restriction in the same way as a solver's condition for limiting the appropriate descriptions (see page 246).

If the system has used a changer in constructing some description, then it will *not* apply the changer again to this description. This default restriction prevents SHAPER from applying the same changer several times in a row. If this restriction is not appropriate in some situations, the user may overwrite it.

Restricting the applicability of generated descriptions

Some changer algorithms can utilize a restriction on possible problems in generating new descriptions. For example, *Abstractor* can generate a description for a limited goals set, and *Margie* uses information about fixed static literals.

We may include such a restriction in the specification of a description changer. If the behavior of the underlying algorithm depends on knob variables, we have to set appropriate knob values, which enable the algorithm to take advantage of the specified restrictions.

We encode this restriction in the same way as a solver's restriction on appropriate problems. That is, we may specify a set of allowed goals; a set of static literals that must hold in a problem's initial state; and a Lisp function that inputs a description triple and PRODIGY problem, and returns true or false.

After a changer generates a new description, we have to construct an applicability condition for this description. We compute it as the conjunction of two other conditions. The first condition in the conjunction is the changer's problem restriction. The second one is the applicability condition of the description that we have used as the changer's input.

7.1.5 Representations

Informally, a representation is a specific approach to solving a problem or collection of problems, which determines the actions during the search for a solution. Researchers used this intuition in formulating several alternative definitions of a representation. For example, Larkin and Simon [1987] defined it as data structures and programs operating on them, and Korf [1980] as a space of world states (see Section 1.1.2 for the discussion of main definitions). We now formalize the notion of representations in our model.

Representations in $\mathcal{S}$HAPER

A *representation* in the $\mathcal{S}$HAPER system is a domain description with a problem solver that uses the description. If the solver is does not make any random choices, the representation uniquely determines the search space for every problem and the order of expanding nodes in this space.

The system automatically constructs representations, by pairing the available solvers with domain descriptions. For every solver, it identifies all descriptions that match the solver's condition and generates the corresponding representations. $\mathcal{S}$HAPER uses not only simple solvers but also solver sequences in constructing representations. The user has an option to prune some representations. Moreover, she may pair solvers with descriptions that do *not* match their conditions, thus creating additional representations.

Applicability conditions

We may limit the use of a representation by an applicability condition, encoded in the same way as conditions for domain descriptions (see Section 7.1.2). If the human operator applies the representation to a problem that does not satisfy this condition, the system signals a warning.

When pairing a solver with a description, $\mathcal{S}$HAPER constructs the conjunction of their applicability conditions and uses it as a condition for the resulting representation. In Section 7.2.3, we will describe the generation of representations and their conditions in more detail. As usual, the user has an option to modify the generated conditions by hand.

7.1.6 Control center

The main top-level operations in the $\mathcal{S}$HAPER system include applying a problem solver, specifying a domain description, and using a changer algorithm to generate an improved description. These three operations form the core of the system's "control center."

The user accesses the system through procedures that perform the three main operations. These procedures enable the user to specify domain descriptions, invoke the automatic improvement of descriptions, and use the resulting descriptions in problem solving.

$\mathcal{S}$HAPER's automatic-control mechanism, which will be described in Chapters 8–10, also accesses solvers and changers through these top-level procedures. The automatic control uses two of the three main operations: application of solver algorithms and use of changers to improve domain descriptions.

Applying a problem solver

The first operation is the application of a solver algorithm, to find a solution to a PRODIGY problem. We have to specify a solver, problem, description triple, and, optionally, a time bound. The system incorporates the triple's elements into the domain data structure, and then calls the solver algorithm, which uses the resulting domain description to search for a solution. After the solver's termination, the system adds the result to the library of problem-solving episodes.

The automatic-control mechanism uses the solver and description from one of the available representations, and limits the search time by a finite bound; the problem must satisfy the applicability condition of the representation. We will describe methods for automatic selection of a representation and time bound in Chapters 8 and 9.

The human user may choose any solver and description for solving a problem. If the selected solver and description do not belong to any of the available representations, or the problem does not satisfy their applicability condition, then the system will warn the user and ask for confirmation.

Specifying a description

The second operation is the specification of a description triple. The user first defines description elements and then combines them into triples. She may specify descriptions in the domain-encoding files and add more descriptions later, in the process of using the domain. The automatic control does *not* use this operation.

The system needs at least one initial description, since it can generate new descriptions only by applying changers to old ones. If the user does not specify any descriptions, the system constructs the default description without primary effects, abstraction, and control rules.

Generating a new description

Finally, the third operation is the use of a description changer, to generate a new description. We have to specify a changer, its input description triple, and, optionally, time bound. The system incorporates the triple's elements into the domain data structure, and then calls the changer algorithm to improve the resulting description.

If the algorithm successfully changes the description, the system stores the resulting primary effects, abstraction, and control rules as a new triple. Then, the system generates the applicability condition for the new description and makes a record of the construction history. Finally, it identifies the solvers whose applicability conditions match the new description and creates a new representation for each of these solvers.

The automatic-control mechanism applies a changer only if the input description satisfies its applicability condition. The mechanism does *not* provide a time bound and thus allows the changer to run until its own termination. Note that all implemented changers in the SHAPER system have polynomial time complexity; in practice, they always terminate in reasonable time.

The human user may apply a changer even if the input description does not satisfy its conditions. The system then gives a warning and asks to confirm the application. After the application, the user may delete some of the new representations.

7.2 Generalized description and representation spaces

We have described the role of problem solvers and description changers in SHAPER. The problem solvers in SHAPER are PRODIGY search algorithms; the description changers are the learning and static-analysis algorithms designed to improve the performance of PRODIGY search. The top-level use of these algorithms, however, is *not* specific to PRODIGY. We can implement multiple descriptions, solvers, and changers in other planning and problem-solving systems, and use them in the same way. Moreover, if several different problem-solving architectures use the same domain language, we may view their search algorithms as alternative problem solvers and combine them into a common system.

Suppose that we define a data structure for storing multiple domain descriptions in some problem-solving architecture, and implement several problem-solving and description-changing algorithms. We may then abstract from the underlying architecture and view descriptions, solvers, and changers as basic low-level objects, with certain axiomatic properties.

This abstraction enables us to define a space of descriptions and develop methods for search in this space. We will design algorithms for search in the space of descriptions and use them for automating the selection of appropriate problem solvers and description changers. The selection algorithms are not PRODIGY-specific; we may use them in other systems.

We begin by defining the properties of descriptions, solvers, and changers in our abstract model (Section 7.2.1). We then use these properties to formalize the notion of a description space (Section 7.2.2). Finally, we show how a description space gives rise to a representation space and discuss the use of these spaces (Section 7.2.3).

7.2.1 Descriptions, solvers, and changers

We consider four main types of objects in the study of representation changes: problems, domain descriptions, problem solvers, and description changers. We also use conditions that restrict the applicability of descriptions, solvers, and changers. We now identify the basic properties of these objects, which are necessary for the analysis of generalized descriptions and representations. We will discuss the limitations of objects with these properties and the simplifying assumptions of the analysis in Section 7.4.

Domain descriptions

A description is a specific domain encoding, used for solving problems in the domain. We may limit the use of a description by a condition on the relevant problems. Formally, a condition is a boolean function, *p-cond*(*prob*), defined on the problems in the domain. If this boolean value for a problem is false, we do not use the description to solve the problem.

For example, suppose that we have used *Abstractor* to construct a hierarchy for a limited set of goals. The *p-cond* function then checks that a problem's goals belong to the set. If the goals are not in the set, we do not use that hierarchy to solve the problem.

Problem solvers

A solver is an algorithm that searches for a solution to a specified problem. The algorithm inputs a domain description and a problem, and runs until it either solves the problem or exhausts the search space without finding a solution. The former outcome is called *success* and the latter is *failure.* If the search space is infinite and the algorithm never finds a solution, it may run forever.

We sometimes view a solver execution as a call to the *Apply-Solver* procedure, which inputs a solver, description, and problem, and then "supervises" the solver application:

Apply-Solver
Inputs: solver, description, problem.
Possible outcomes: finding a solution, failure termination, infinite execution.

Since real-world algorithms cannot run forever, we may have to interrupt a solver that has run too long. Recall that PRODIGY search algorithms allow us to specify an interrupt time bound. We will describe the use of time bounds in Section 7.2.3.

Solver operators

We may restrict the use of a solver by two conditions. The first condition limits domain descriptions used with the solver. Formally, it is a boolean function, *d-cond*(*desc*), defined on the domain descriptions. The second condition limits problems for each description. Thus, it is a function of a description and problem, *dp-cond*(*desc*, *prob*). We can apply a solver to a problem *prob* with a description *desc* only when both functions evaluate to `true`.

A *solver operator* is a solver together with the two conditions that determine its applicability. We denote it by a triple (*solver*, *d-cond*, *dp-cond*). These conditions are hard constraints, aimed to avoid inappropriate use of the solver. We will describe soft constraints, for encoding solver-selection preferences, in Chapter 10.

Problem-independent description changers

A problem-independent changer is an algorithm that inputs a domain description and converts it into a new description. It constructs a description for solving multiple problems, rather than for one specific problem. The algorithm may successfully generate a new description or terminate with failure. For example, *Abstractor* terminates with failure if the hierarchy collapses into a single level.

Some changers allow us to specify a restriction on the problems that can be solved with a new description. A changer then uses this restriction in generating a description. For example, *Abstractor* can use the limitation on the allowed goals, and *Chooser* utilizes the information about static predicates. We may then use the resulting description only for solving problems that satisfy the restriction.

The description changers used in the $\mathcal{S}$HAPER system always terminate. Unlike problem solvers, they never result in infinite execution. For this reason, we do not consider the run-forever possibility in their analysis.

We view a changer execution as a call to the *Apply-Indep-Changer* procedure, which inputs a changer, initial description, and restriction on the problems, and then applies the changer to the description:

Apply-Indep-Changer
Inputs: description changer, initial description, restriction on the problems.
Possible outcomes: generating a new description, failure termination.

Problem-specific description changers

A problem-specific changer generates a new description for solving a given problem instance. The knowledge of the problem often enables the algorithm to construct a better description. The downside is that we cannot re-use the description; we discard it after solving the problem. We use the *Apply-Spec-Changer* procedure to supervise the execution of problem-specific changers:

Apply-Spec-Changer
Inputs: changer, initial description, problem.
Possible outcomes: generating a new description, failure termination.

Changer operators

We may restrict the use of a problem-independent changer by two conditions. The first condition, *d-cond*(*desc*), limits input descriptions. We apply the changer only to domain descriptions that satisfy this condition. The second condition, *dp-cond*(*desc*,*prob*), limits the problems that we solve with a newly generated description.

A *changer operator* is a changer together with the two conditions, denoted by a triple (*changer*, *d-cond*, *dp-cond*). We use only problem-independent changers in operators.

Suppose that we apply a changer operator to some description *desc*, and the applicability of *desc* is limited by some condition *p-cond*. If *desc* satisfies *d-cond*, we apply the changer to *desc*, with the restricting condition formed by the conjunction of *dp-cond* and *p-cond*. We use the new description only for problems that satisfy the conjunction of the two conditions, $dp\text{-}cond(desc, prob) \wedge p\text{-}cond(prob)$.

The role of changer operators in the automatic exploration of alternative descriptions is similar to the role of classical operators in PRODIGY search. The conditions of changer operators are hard constraints, which rule out inappropriate description changes. We will describe the use of preferences in Chapter 10.

Solver sequences

We may use a solver operator that contains a sequence of one or more problem-specific changers and a problem solver. When applying this operator to solve a problem, we first execute the changers, which make several consecutive description changes, and then use the solver with the final description.

A sequence of several problem-specific changers with a solver at the end is called a *solver sequence*. Note that a solver sequence satisfies our definition of a solver: if we apply it to solve a problem, it may output a solution, fail (if some changer or the solver fails), or run forever (if the solver runs forever). We do not include the intermediate descriptions in the output, since we cannot use them for other problems.

We therefore view a solver sequence as a type of solver, and do not distinguish it from a "simple" solver in the study of methods for constructing and selecting representations.

Changer sequences

A *changer sequence* is a sequence of problem-independent changers. The application of the sequence involves executing the changers in order, thus making several consecutive description changes. The output of the application is the final description.

The behavior of a sequence is similar to the behavior of a problem-independent changer: it may generate a new description or terminate with failure (if one of the changers fails). We therefore do not distinguish between changer sequences and simple changers.

7.2.2 Description space

When we load a PRODIGY domain into $\mathcal{S}$HAPER, the system constructs initial descriptions of the domain, and forms a library of changer operators for use in this domain, according to the user's specification. If the user does not specify initial descriptions, the system uses the basic default description as the only initial description.

$\mathcal{S}$HAPER generates new descriptions by applying changer operators first to initial descriptions and then to newly generated ones. The *description space* of a domain is the set of all descriptions that $\mathcal{S}$HAPER can potentially generate, using the available changer operators. The search in this space involves the construction of new descriptions and their use in problem solving.

The descriptions that have already been generated form the expanded part of the space. We now review the data stored for the expanded space and describe the addition of new descriptions. We will then present the construction of representations from the generated descriptions and their use in problem solving.

Description nodes

We store the expanded part of the description space as a collection of nodes. A node consists of a description, the applicability condition for this description, and some information about the history of creating it. We create a new node by applying a changer operator to one of the old nodes. We may apply a changer only if the description in the old node satisfies the operator's condition. We say that the newly generated node is a child of the old node.

A node's history information includes pointers to the parent node, to the operator that generated the node, and to all children (if any). We use this information in the conditions of solver and changer operators, since the applicability of solvers and changers often depends on the history of constructing a description.

Identity testing

The application of different changers may sometimes result in generating the same description. When adding a description node, we check whether its description is different from descriptions in the other nodes.

If the new description is identical to some old description, we merge the new node with the old one. The result of the merge is a node with the same description, whose applicability condition is the disjunction of the original nodes' conditions. The merged node inherits children from the old node and parents from both old and new node. Note that the resulting node may have multiple parents.

The identity test in SHAPER is sound but not complete. The system recognizes identical abstraction hierarchies and selections of primary effects, but it may not detect identical control rules (see Section 7.1.1). SHAPER may thus create two distinct nodes with identical descriptions, though it rarely occurs in practice. The only negative effect of this situation is a decrease in the accuracy of the description-quality statistics (see Chapter 8).

Rejection rules

We allow the use of rejection rules, which prune some nodes from the description space. The user may specify a set of rules for all domains and additional rules for each specific domain.

Formally, a *rejection rule* is a boolean function, whose argument is a description node. If it returns `true`, we do not add the node to the space. The rejection may depend on the node's description, applicability condition, and history of constructing the node.

After generating a description node, SHAPER fires the rejection rules. If the node matches some rule, the system does not add it to the expanded space.

Comparison rules

We also use rules that compare description nodes with each other and prune nodes with inferior descriptions. Formally, a *comparison rule* is a boolean function, *better-node*$(node_1, node_2)$, whose arguments are two distinct description nodes. If it returns `true`, then the description of $node_1$ is *definitely* better than that of $node_2$, and we prune $node_2$ from the expanded space.

We consider the "better-than" relationship transitive. Suppose that, according to the comparison rules, $node_1$ is better than $node_2$ and $node_2$ is better than $node_3$. We then assume that $node_1$ is better than $node_3$, even if no rule compares them directly. We thus prune both $node_2$ and $node_3$.

If $node_1$ is better than $node_2$ and $node_2$ is better than $node_1$, we assume that the utility of the two descriptions is very similar, and we may prune either of them. If the "better-than" relationship forms a longer loop, we prune all but one nodes of the loop.

After generating a new description node, SHAPER compares it with the old nodes. If some old node is better than the new one, SHAPER does not add the new node to the expanded space. If some old node is worse then the new one, the system makes the old node *inactive.*

SHAPER does not use inactive descriptions in problem solving or in generating new descriptions; however, it does not remove them from the description space. This approach enables us to preserve the structure of the expanded space, as well as the statistics on problem solving with inactive descriptions, accumulated before their inactivation.

Summary of generating a node

The *Make-Description* algorithm in Figure 7.2 is the summary of applying a changer operator. The algorithm inputs an operator and a description node. The operator includes a description changer, denoted *changer*, a condition for its applicability, *d-cond*, and a restriction *dp-cond* on the problems to be solved with the new description. The initial node

Make-Description(*changer*, *d-cond*, *dp-cond*; *init-desc*, *p-cond*)
1. If *desc* does not satisfy *d-cond*, then terminate.
2. Define applicability condition *new-p-cond* such that, for every *prob*,
   new-p-cond(*prob*) = *p-cond*(*prob*) $\wedge$ *dp-cond*(*desc*, *prob*).
3. Call *Apply-Indep-Changer*(*changer*, *init-desc*, *new-p-cond*).
4. If it fails to produce a new description, then terminate;
   Else it returns some description *new-desc*.
5. If some old node has a description identical to *new-desc*, and some condition *old-p-cond*,
   then replace its condition with *old-p-cond* $\vee$ *new-p-cond*,
   add a parent pointer to the initial node, and terminate.
6. Make a node *new-node*, with description *new-desc*, condition *new-p-cond*,
   and parent pointer to the initial node.
7. If *new-node* matches some rejection rule, then mark *new-node* "rejected."
8. For every old node, *old-node*:
   For every comparison rule, with condition *better-node*:
   If *better-node*(*old-node*, *new-node*), then mark *new-node* "rejected;"
   Else, if *better-node*(*new-node*, *old-node*), then inactivate *old-node*.
9. If *new-node* is marked "rejected," then terminate.
10. Add *new-node* to the expanded description space.
11. Call *Make-Reps*(*new-desc*, *new-p-cond*).

Figure 7.2: Applying changer operator (*changer*, *d-cond*, *dp-cond*) to initial node (*init-desc*, *p-cond*).

includes a domain description, *init-desc*, and a restriction *p-cond* on the problems that can be solved with the node's description.

The algorithm first checks whether the initial description satisfies the condition of the changer operator (step 1 of the algorithm). If it does, the changer generates a new description node (steps 2–4). If some old node already has this description, the algorithm modifies the node's condition and history, and then terminates (step 5). Otherwise, it makes a new node (step 6).

The application of rejection and comparison rules may result in discarding the new node or inactivating some old nodes (steps 7–9). Even if we reject the new node, we inactivate the old nodes that are worse than the new one, according to the comparison rules. We therefore have to fire all comparison rules even when rejecting the new node.

Rejection and comparison rules are *hard* constraints, designed to prune description nodes. We do *not* use soft preference constraints for selecting among nodes; however, we allow use of preferences in selecting among representations, which are constructed from the descriptions.

After adding a new node to the description space (step 10), SHAPER generates representations based on the node's description (step 11). The construction and use of representations is the subject of Section 7.2.3.

The SHAPER system is able to keep multiple domains in memory and interleave problem solving in different domains. The system expands a *completely separate* description space for every domain. The simultaneous use of several domains enables the system to accumulate some "historical" experience on the generation and use of descriptions, and transfer this experience across domains (Sections 10.2 and 11.1).

7.2.3 Representation space

We next formalize the use and construction of representations in our abstract model. The *representation space* of a domain is the set of representations that the system can potentially generate, based on the domain's description space and available problem solvers.

Use of representations

We view a representation as a triple $(desc, solver, p\text{-}cond)$, where the first element is a description, the second is a solver, and the third is a boolean function, defined on the problems in the domain, that determines the applicability of the representation. If the *p-cond* function returns false for a problem, we do not use the representation to solve the problem.

Since the use of a representation may result in an infinite execution of the solver, we always set a finite time bound and interrupt the solver upon reaching this bound.

We view the use of a representation as a call to the *Apply-Rep* procedure, which inputs a representation, problem, and time-bound, and then uses the solver and description from this representation to solve the problem. We refer to the use of this procedure as *applying a representation* to a problem or *running a representation* on a given problem.

This procedure may give four different outcomes. First, *Apply-Rep* may reject the problem, if it does not match the representation's applicability condition. Second, the solver may find a solution; we call it a *successful termination* or simply *success.* Third, it may terminate with failure after exhausting the search space. Finally, the procedure may interrupt the solver upon reaching the time bound. We now give a summary of the procedure's inputs and outcomes:

> **Apply-Rep**
> *Inputs:* representation, problem, time bound.
> *Possible outcomes:* finding a solution, failure termination,
> rejecting the problem, time-bound interrupt.

Note that successes, failures, and rejections occur before reaching a time bound, and thus they do not require an external interrupt of the solver algorithm. We sometimes refer to these three outcomes as *termination outcomes*, to distinguish them from interrupts. This distinction becomes important in the statistical analysis of the performance of available representations (see Chapter 8).

Failures, rejections, and interrupts

A failure outcome shows the absence of a solution in the solver's space, which may be useful information. In particular, if the solver is complete, failure means that the problem has no solution. Thus, a failure outcome may be more valuable than a rejection or interrupt. This situation is different from the application of changer operators. A failure of a changer does not give any information that we utilize later. We thus do *not* distinguish between failures and rejections resulting from the changer applications.

A representation may contain a solver sequence rather than a simple solver. Recall that a sequence consists of several problem-specific changers and a problem solver. If we use such a representation, one of the changers may fail to generate a description, resulting in an unsuccessful outcome. We view such an outcome as a rejection, rather than a failure, because it does not indicate the absence of a solution in the solver's search space.

In SHAPER, the time for rejecting a problem is usually much less than the time for other outcomes; however, sometimes it is not the case. The use of complex functions in applicability conditions, as well as problem-specific changers that frequently fail, may result in significant rejection times. We therefore must consider a possibility of time loss due to rejections in the analysis of the system's behavior.

If the use of a representation has resulted in an interrupt, *Apply-Rep* returns the expanded part of the problem's search space. If we later try to solve the problem with the same representation and larger time bound, we may continue the expansion of the space, rather than starting from scratch. This reuse of spaces, however, is limited by the memory available for storing them.

Constructing representations from a description

The generation of new representations parallels the expansion of the description space. After adding a new description node, say (*desc*, *p-cond*), the SHAPER system produces the corresponding representations, by combining the description with available solver operators.

For every solver operator (*solver*, *d-cond*, *dp-cond*), the system checks whether the new description *desc* matches the operator's condition *d-cond*. If it does, the system makes a representation (*desc*, *solver*, *new-p-cond*), which contains the new description and the operator's solver. The applicability condition *new-p-cond* is defined as the conjunction of the node's condition and the solver's condition for this description:

$$\text{For every } \textit{prob}, \text{ we have } \textit{new-p-cond}(\textit{prob}) = \textit{p-cond}(\textit{prob}) \wedge \textit{dp-cond}(\textit{desc}, \textit{prob}).$$

If a description does not match any solver operators, it does not give rise to any representations. Such a description is not used in problem solving, but it may be an intermediate step in constructing other descriptions.

Rejection and comparison rules

We allow the use of rejection and comparison rules to prune representations. The semantics of these rules is the same as that of the rules for pruning description nodes.

If a new representation matches a rejection rule, or if it is worse than some old representation according to a comparison rule, we discard it. If an old representation is worse than a new one, we inactivate the old representation and do not use it in problem solving. If we inactivate some description node (see the *Make-Description* algorithm in Figure 7.2), then we also inactivate all representations that contain it.

We use not only pruning rules, but also several types of preference rules in selecting among representations. We will describe the use of preferences in Chapter 10.

Make-Reps(*desc*, *p-cond*)
For every solver operator (*solver*, *d-cond*, *dp-cond*):
  If *desc* satisfies *d-cond*, then:
    Define applicability condition *new-p-cond* such that, for every *prob*,
      new-p-cond(*prob*) = *p-cond*(*prob*) ∧ *dp-cond*(*desc*, *prob*).
    Define representation *new-rep* as (*desc*, *solver*, *new-p-cond*).
    Call *Add-Rep*(*new-rep*).

Add-Rep(*new-rep*)
If *new-rep* matches some rejection rule, then mark *new-rep* "rejected."
For every old representation, *old-rep*:
  For every comparison rule, with condition *better-rep*:
    If *better-rep*(*old-rep*, *new-rep*), then mark *new-rep* "rejected;"
    Else, if *better-rep*(*new-rep*, *old-rep*), then inactivate *old-rep*.
If *new-rep* is not marked "rejected," then add it to the expanded representation space.

Figure 7.3: Generating representations for a new description node (*desc*, *p-cond*).

Summary of generating representations

We summarize the algorithm for generating new representations in Figure 7.3. Shaper invokes this algorithm after creating a new description node, with description *desc* and applicability condition *p-cond*. The algorithms add representations that correspond to the new node.

The pseudocode in Figures 7.2 and 7.3 does not include the inactivation of old representations that follows the inactivation of a description node. Shaper performs this inactivation when executing the last line of step 8 in the *Make-Description* algorithm.

If the Shaper system interleaves problem solving in several domains, then it keeps a separate description space and solver-operator library for every domain and, hence, it expands a completely separate representation space for each domain.

7.3 Utility functions

We describe general utility functions for evaluating the results of problem solving, which enable us to formalize the notions of problem-solving effectiveness and representation quality. They provide a means for an objective comparison of different representations. We use these utility measures in developing statistical methods for selection among representations.

We evaluated the use of primary effects and abstractions along three dimensions: the number of solved problems, the running time, and the cost of resulting solution plans. The use of utility functions combines these dimensions and allows us to decide whether a gain along some dimension compensates a loss along another.

7.3.1 Gain function

We assume that an application of a problem-solving algorithm may result in one of four outcomes. First, the algorithm may find a solution to a problem. Second, it may terminate with failure, after exhausting the available search space without finding a solution. Third, we may interrupt the algorithm, if it has reached some pre-set time bound without termination. Finally, the algorithm may reject the problem without trying to solve it. For example, the SHAPER system rejects a problem when the goals do not belong to the limited goal set used in constructing the current representation.

Intuitively, we have to pay for the running time of the algorithm, and we get a reward if it finds a solution. The payment for the time may depend on a specific problem; the reward depends on the problem and solution. The overall *gain* is a function of a problem, time, and the result of problem solving. Note that we call it a *gain function*, rather than using a more generic term *utility function*, to avoid confusion with other utility measures.

We denote this function by $gain(prob, time, result)$, where *prob* is the problem, *time* is the running time, and *result* is the result. The latter may be some solution to the problem, or one of the three unsuccessful outcomes: termination with failure (denoted fail), interrupt upon hitting a time bound (intr), or the algorithm's rejection of the problem (reject). Note that fail, intr, and reject are *not* variables and, hence, we do not italicize them.

The specification of a particular gain function is the user's responsibility. The function encodes the value of different problem-solving outcomes for the user. It gives the user a means to communicate her value judgments to the system.

We assume that the gain function is defined for all problems in a domain, all time values, and all possible outcomes. We impose four constraints on the possible gain function, which determine the necessary basic properties of problem-solving gains.

1. The gain decreases with time:

   For every *prob*, *result*, and $time_1 < time_1$, we have
   $gain(prob, time_1, result) \geq gain(prob, time_2, result)$.

2. If we interrupt the system or reject a problem without spending any running time, then the gain is zero:

   For every *prob*, we have $gain(prob, 0, \mathsf{intr}) = gain(prob, 0, \mathsf{reject}) = 0$.

3. The interrupt gives the same gain as the rejection of a problem, if the running time is the same in both cases:

   For every *prob* and *time*, we have
   $gain(prob, time, \mathsf{intr}) = gain(prob, time, \mathsf{reject})$.

4. The gain of solving a problem or failing upon the exhaustion of the search space is at least as large as the gain of the interrupt after the same running time:

   For ever *prob*, *time*, and *soln*,
   (a) $gain(prob, time, soln) \geq gain(prob, time, \mathsf{intr})$,
   (b) $gain(prob, time, \mathsf{fail}) \geq gain(prob, time, \mathsf{intr})$.

Note that we do *not* assume that gain is a continuous function of time. A gap in the function may represent a decrease in a solution's value upon reaching a deadline. Also note that a failure may give a larger gain than an interrupt. For example, a failure of a complete solver tells us that the problem has no solution, and this information may be worth extra gain.

Constraint 2 means that the gain of doing nothing is exactly zero. Since we always have a do-nothing option, negative gain is never desirable. Constraints 1 and 2 imply that interrupts and rejections never give a positive gain. Constraints 2 and 4 ensure that the gain of instantly finding a solution is nonnegative.

We define a relative *quality* of solutions in terms of the gain function. Suppose that $soln_1$ and $soln_2$ are two different solutions for some problem, *prob*.

> $soln_1$ has higher quality than $soln_2$ if, for every *time*,
> $gain(prob, time, soln_1) \geq gain(prob, time, soln_2)$.

This relationship between solutions is a partial order. If $soln_1$ gives larger gains than $soln_2$ for some running times and lower gains for others, then neither of them has higher quality than the other. For example, a conference paper may be more valuable than a technical report if it is finished before the deadline, but less valuable past the deadline.

7.3.2 Additional constraints

We now discuss extra constraints on gain functions, used in some derivations. We do *not* assume that they hold for all gain functions. We will specify explicitly which of these constraints are needed in specific derivations.

5. As the running time approaches infinity, the gain approaches negative infinity. In other words, the gain has no finite lower bound:

> For every *prob* and *result*, and for every negative real value g,
> there is *time* such that $gain(prob, time, result) \leq g$.

Informally, it means that we cannot buy infinite running time for a finite payment. For every problem and solution, there is a "threshold" time such that the gain is positive if the running time is smaller than the threshold, and negative otherwise.

6. Failure termination upon exhausting the search space gives the same gain as an interrupt after the same running time:

> For every *prob* and *time*, we have
> $gain(prob, time, \mathsf{fail}) = gain(prob, time, \mathsf{intr})$.

We use this constraint when a problem solver is incomplete, and the exhaustion of its search space does not mean that the problem in unsolvable.

7. If $soln_1$ gives a larger gain than $soln_2$ for zero running time, it gives a larger gain for all other times:

For every *prob*, *time*, $soln_1$, and $soln_2$,
if $gain(prob, 0, soln_1) \geq gain(prob, 0, soln_2)$,
then $gain(prob, time, soln_1) \geq gain(prob, time, soln_2)$.

Suppose that this constraint holds for a domain. Since *gain* is a real-valued function, we can compare the quality of any two solutions to a given problem. In other words, In other words, for every problem in the domain, the problem's solutions are totally ordered by their quality. We may then define a function *quality*(*prob*, *result*) that provides a numerical measure of the solution quality. The function must satisfy the following conditions, for every problem *prob* and every two results of solving it, $result_1$ and $result_2$:

- $quality(prob, \textsf{intr}) = 0$.
- If, for every *time* value, $gain(prob, time, result_1) = gain(prob, time, result_2)$, then $quality(prob, result_1) = quality(prob, result_2)$.
- If, for some *time* value, $gain(prob, time, result_1) > gain(prob, time, result_2)$, then $quality(prob, result_1) > quality(prob, result_2)$.

The quality value of every result is no smaller than that of an interrupt; therefore, quality values are nonnegative.

Most $\mathcal{S}$HAPER's domains have natural quality measures that satisfy these conditions. For example, suppose that we use the measure described in Section 2.3.1; that is, we evaluate a solution by the total cost of its operators. Suppose further that, for each problem *prob*, there is some maximal acceptable cost, $cost_{\max}(prob)$, and solutions with larger costs are not satisfactory. We may define a quality function as

$$quality(prob, soln) = cost_{\max}(prob) - cost(soln).$$

If we use some quality measure, we may define problem-solving gain as a function of problem, running time, and solution quality, $gain_q(prob, time, quality)$. This function must satisfy the following condition:

For every *prob*, *time*, and *result*,
$gain_q(prob, time, quality(prob, result)) = gain(prob, time, result)$.

We first observe that this function is well-defined:

If $quality(prob, result_1) = quality(prob, result_2)$, then we have
$gain_q(prob, time, quality(prob, result_1)) = gain_q(prob, time, quality(prob, result_2))$;

therefore, the imposed condition specifies exactly one value of $gain_q$ for each triple of *prob*, *time*, and *quality*. We also note that gain is an increasing function of quality:

If $quality_1 \leq quality_2$, then $gain_q(prob, time, quality_1) \leq gain_q(prob, time, quality_1)$.

We next consider another constraint on *gain*, which is stronger than Constraint 7; that is, this new constraint implies Constraint 7.

8. We can decompose the gain into the payment for running time and the reward for solving a problem:

> For every *prob*, *time*, and *result*, we have
> $gain(prob, time, result) = gain(prob, time, \mathsf{intr}) + gain(prob, 0, result)$.

The first term in the decomposition is the payment for time; it is nonpositive. The second term is the reward, which is nonnegative. If this constraint holds, we may define the quality function as $quality(prob, result) = gain(prob, 0, result)$; then, $gain_q$ is linear on the solution quality:

$$gain_{\mathrm{q}}(prob, time, quality) = gain(prob, time, \mathsf{intr}) + quality.$$

9. The sum payment for two interrupted runs that take $time_1$ and $time_2$ is the same as the payment for a run that takes $time_1 + time_2$:

> For every *prob*, $time_1$, and $time_2$, we have
> $gain(prob, time_1, \mathsf{intr}) + gain(prob, time_2, \mathsf{intr}) = gain(prob, time_1 + time_2, \mathsf{intr})$.

This constraint implies that the interrupt gain is proportional to time:

$$gain(prob, time, \mathsf{intr}) = time \cdot gain(prob, 1, \mathsf{intr}).$$

The value of $gain(prob, 1, \mathsf{intr})$ represents the price of a unit time. If we use Constraints 8 and 9 together, we get the following decomposition of the gain function:

$$gain(prob, time, \mathsf{intr}) = time \cdot gain(prob, 1, \mathsf{intr}) + gain(prob, 0, result).$$

7.3.3 Representation quality

We now derive a utility function for evaluating a representation. The quality of a representation depends not only on the representation itself, but also on a gain function, choice of time bounds, and distribution of problems.

We consider a fixed representation and suppose that we do *not* use time bounds. We assume that the problem solver never makes random choices. Thus, for every problem *prob*, the representation uniquely determines the running time, $time(prob)$, and the result of problem solving, $result(prob)$. Since we do not interrupt the solver, $time(prob)$ may be infinite. The result may be a solution, failure, rejection, or infinite run.

If we use a time bound B in solving *prob*, then the running time and result are as follows:

$$\begin{aligned} time' &= \min(B, time(prob)) \\ result' &= \begin{cases} result(prob), & \text{if } B \geq time(prob) \\ \mathsf{intr}, & \text{if } B < time(prob) \end{cases} \end{aligned}$$

Thus, the choice of a bound B determines the time and result; therefore, it determines the gain. We denote the function that maps problems and bounds into gains by $gain'$:

$$gain'(prob, B) = gain(prob, time', result'). \tag{7.1}$$

Since programs in the real world cannot run forever, we must always set a finite time bound. We will describe $\mathcal{S}$HAPER's heuristics for choosing a bound in Chapters 8 and 9. The heuristics determine a function that maps every *prob* into a time bound $B(prob)$. The gain of solving *prob* is then $gain'(prob, B(prob))$.

If the gain function satisfies Constraint 7 of Section 7.3.2 and we use a quality measure *quality*(*prob*, *result*), we can define *gain'* in terms of the solution quality. First, suppose that we use a fixed representation *without* time bounds. Then, for every *prob*, the representation uniquely determines the solution quality:

$$quality_{\mathrm{p}}(prob) = \begin{cases} quality(prob, result(prob)), & \text{if } time(prob) \text{ is finite} \\ quality(prob, \mathsf{intr}), & \text{if } time(prob) \text{ is infinite} \end{cases}$$

If we use a time bound B, it determines a different quality value; we denote this value by *quality'*:

$$quality'(prob, B) = \begin{cases} quality_{\mathrm{p}}(prob), & \text{if } B \geq time(prob) \\ quality(prob, \mathsf{intr}), & \text{if } B < time(prob) \end{cases}$$

We can now express *gain'* through the function $gain_{\mathrm{q}}$, defined in Section 7.3.2:

$$gain'(prob, B) = gain_{\mathrm{q}}(prob, time', quality'(prob, B)) \tag{7.2}$$

We define a representation utility by averaging the gain over all problems [Koenig, 1997]. We denote the set of problems in a domain by $\mathcal{P}$ and assume that it is finite or countably infinite. We also assume a fixed probability distribution on $\mathcal{P}$, which determines the chance of encountering each problem. For every *prob*, the distribution defines some probability $p(prob)$. The sum of these probabilities, $\sum_{prob \in \mathcal{P}} p(prob)$, is 1.

If we select a problem at random, according to the probability distribution, then the expected problem-solving gain is as follows:

$$G = \sum_{prob \in \mathcal{P}} p(prob) \cdot gain'(prob, B(prob)). \tag{7.3}$$

We use G as a utility function, which allows us to evaluate the representation and bound-selection heuristics. It unifies the three quality dimensions, which include near-completeness, running time, and solution quality.

7.3.4 Use of multiple representations

The $\mathcal{S}$HAPER system generates multiple representations of a problem domain, and uses learning techniques (Chapters 8 and 9) and heuristics (Chapter 10) to select a representation for each given problem. We generalize Equation 7.3 to define a utility function for the use of a fixed collection of representations.

We denote the number of available representations by k. The gain function may depend on the representation, so we consider k distinct gain functions, $gain_1, ..., gain_k$. Since different representations require different choices of time bounds, we also consider the use of k distinct bound-selection functions, $B_1, ..., B_k$. When we solve a problem *prob* with representation i (where $1 \leq i \leq k$), we set the time bound $B_i(prob)$ and use the gain function $gain_i$ to estimate the resulting gain.

For every representation i, we may define the function $gain'_i$, in the same way as we have defined $gain'$ in Section 7.3.3. The gain of solving $prob$ with representation i is then $gain'_i(prob, B_i(prob))$.

The representation-selection techniques, used in the system, determine a function that maps every problem $prob$ into some representation $i(prob)$, where $1 \le i(prob) \le k$.

When solving $prob$, we first identify the corresponding representation $i(prob)$ and then choose the time bound $B_{i(prob)}(prob)$. If we select a problem at random, according to a probability distribution $\mathcal{P}$, then the expected gain is as follows:

$$G = \sum_{prob \in \mathcal{P}} p(prob) \cdot gain'_{i(prob)}(prob, B_{i(prob)}(prob)). \tag{7.4}$$

Note that this equality holds not only for finite, but also for countably infinite collections of representations; however, we will consider only finite collections in developing $\mathcal{S}$HAPER's techniques for selecting representations.

We have assumed that the selection of a representation and bound takes very little computation and does not reduce the overall gain. We will show experimentally that this assumption holds in $\mathcal{S}$HAPER (see Section 8.3). If the selection takes considerable time, we need to adjust Equation 7.4 to account for the cost of computing $i(prob)$ and $B_{i(prob)}(prob)$.

The utility value G depends on the gain function, probability distribution, choice of representation, and time bound. The first two parameters are properties of the world, and the system has no control over them.

The $\mathcal{S}$HAPER system gets the user-specified gain function as a part of the input and gradually learns the probability distribution in the process of problem solving. $\mathcal{S}$HAPER's top-level control algorithms choose a representation and adjust bound-selection heuristics to maximize the expected gain.

7.3.5 Summing gains

When we solve several problems or find several solutions to the same problem, we are interested in the overall gain. We now discuss the rules for determining the total gain.

Suppose that we apply problem-solving algorithms to problems $prob_1, prob_2, ..., prob_n$. We may use different representations and gain functions in solving these problems. We denote the time for solving $prob_i$ by $time_i$, the result by $result_i$, and the corresponding gain function by $gain_i$.

A. If $prob_1, prob_2, ..., prob_n$ are all distinct,
the total gain is $\sum_{i=1}^{n} gain_i(prob_i, time_i, result_i)$.

We use this rule in the $\mathcal{S}$HAPER system, even though it does not always hold in the real world. For example, if two problems are parts of a larger problem, solving one of them may be worthless without solving the other. On the other hand, if either part readily gives a key to the larger problem, solving one of them may be no less valuable then solving both.

If we try to solve a problem with different representations, and all attempts except one result in an interrupt or rejection, the total gain is also the sum of the individual gains:

B. If $result_1, result_2, ..., result_{n-1}$ for some *prob* are all interrupts and rejections, the total gain of solving *prob* is $\sum_{i=1}^{n} gain_i(prob, time_i, result_i)$.

We cannot use this rule if two different runs result in finding a solution or exhausting the search space. For example, if two runs give the same solution, the rule would count the reward twice. We use additional constraints on the gain function to define the total gain in such cases.

Suppose that the gain function satisfies Constraint 6; that is, the failure gain is equal to the rejection gain. We then use Rule B for summing the gain when one run gives a solution and all others result in failures, interrupts, and rejections. If we do not use SHAPER for finding a second solution to an already solved problem, then Rules A and B always enable us to compute the total gain.

If we allow search for multiple solutions to a problem, we need Constraints 6 and 8 for defining the total gain. We compute the total payment for time as the *sum* of payments for individual runs, and the reward for solving a problem as the *maximum* of rewards for individual solutions:

C. If $result_1, result_2, ..., result_n$ are all for the same problem *prob*, the total gain is $\sum_{i=1}^{n} gain_i(prob, time_i, \mathsf{intr}) + \max_{i=1}^{n} gain_i(prob, 0, result_i)$.

This rule is also a simplification, which may not hold in the real world, since obtaining multiple solutions to a problem is sometimes more valuable than finding only the best solution.

If we use description-changing algorithms, we include the payment for their use in the total-gain computation. Since a description changer does not solve any problems, its "gain" is negative. The potential reward is the gain of problem solving with a new description.

We assume that the gain of applying a changer is a function of the initial description and running time, $gain_c(desc, time)$, and that this function has the same properties as the interrupt gain of problem solvers:

- For every *desc*, we have $gain_c(desc, 0) = 0$
- If $time_1 < time_2$, then $gain_c(desc, time_1) \geq gain_c(desc, time_2)$

The gain does *not* depend on the result of a description change; however, the result affects the consequent problem-solving gains.

We compute the total gain by *summing* all description-changing gains and adding them to the total problem-solving gain. This rule is analogous to Rules A and B for totaling the gains of problem solving.

7.4 Simplifying assumptions and the user's role

We have formalized the use of description and representation spaces, and described general gain functions for evaluating the system's performance. This generalized model allows us to abstract from specific features of PRODIGY. We use it to design a search engine for the exploration of the representation space. We can use this engine in many different problem-solving systems, and for the automatic selection among available systems.

The mechanism for guiding the representation search is based on the statistical analysis of past performance (Chapters 8 and 9) and several types of preference rules (Chapter 10). The automatic statistical analysis enables the system to learn the performance of the available representations and select the most effective among them. When statistical data are not sufficient, the system relies on the preference rules.

We now discuss the simplifying assumptions, which limit the applicability of the developed model (Section 7.4.1). We then describe the role of the human user in guiding $\mathcal{S}$HAPER, and summarize her responsibilities and options (Section 7.4.2).

7.4.1 Simplifying assumptions

We divide the assumptions into three groups. First, we review the simplifications used to formalize the representation space. Second, we summarize the assumptions that underlie the evaluation model and statistical analysis of the performance. Third, we introduce additional assumptions, used in selecting representations and time bounds.

Behavior of solvers and changers

We begin by reviewing the assumptions that underlie the generalized model of search in description and representation spaces (Section 7.2).

First, we have assumed that solvers and changers are be *sound*; that is, they always produce correct solutions and valid domain descriptions.

Second, solvers do *not* use *any-time* behavior. That is, a solver finds a solution and terminates, rather than outputting successively better solutions. We may readily extend the representation-space model and evaluation methods for any-time algorithms; however, $\mathcal{S}$HAPER's statistical analysis does *not* account for such algorithms.

If a solver does not find a solution, it may terminate with failure, hit a time bound, or reject a problem. We allow a rejection by the solver itself or by its applicability condition. Even though the rejection time is usually very small compared to problem solving, we do *not* neglect it in the analysis.

Description changers also do not use any-time behavior. A changer generates a new description or fails. We do not distinguish between a changer's failures and rejections.

Performance analysis

We now review the assumptions underlying the evaluation model (Section 7.3), which is based on Constraints 1–4 (Section 7.3.1) and Rules A–C for summing gains (Section 7.3.5).

We assume the availability of functions that provide an exact (or sufficiently accurate) gain computation for every problem-solving episode. The statistical analysis of performance does *not* account for the time of the gain computation, nor for the time of the analysis itself. We thus consider the time of these operations negligible. Their actual running time in $\mathcal{S}$HAPER is smaller than that of problem solving by two orders of magnitude (see Sections 9.2.2 and 9.3.2).

Since the SHAPER system does not run solvers and changers in parallel, we have defined utility only for the sequential application of algorithms. The use of parallelism would require a significant extension to the evaluation model.

The statistical analysis is valid only if the performance of representations does not change over time, which requires two assumptions. First, we use the *stationarity assumption* [Valiant, 1984]; that is, we suppose that the distribution of problems does not change over time. Second, we assume that the performance of a representation does not improve due to learning. SHAPER may use learning to construct a representation, but the system does not improve the representation further in the process of problem solving.

Some PRODIGY algorithms do not satisfy this assumption. For example, the ANALOGY solver [Veloso, 1994] accumulates and reuses search episodes, and its efficiency improves with solving more problems. We may use such algorithms in SHAPER, but the statistical analysis does not take the efficiency improvement into account. The system underestimates the effectiveness of these solvers, judging by their initial performance, and may favor the use of other algorithms.

Even if solvers do not use learning, the system's overall performance improves over time. The improvement comes from generating new descriptions and applying statistical analysis to select effective descriptions and solvers.

We have assumed that the solvers do not make any random choices, which simplifies the evaluation model (Section 7.3.3). This assumption is *not* required for the statistical analysis. We readily extend the model to randomized algorithms, by viewing every possible behavior on a problem as a separate problem-solving episode.

Additional assumptions

We introduce some assumptions that simplify the selection and use of representations. We plan to investigate their relaxation as a part of future work.

First, SHAPER views solvers as black boxes; it calls a selected solver and then "passively" waits for a termination or elapse of a pre-set time. The system does not provide any guidance or performance analysis *during* the solver execution. In particular, we do not use the dynamic adjustment of the time bound based on the findings during the solver's run.

Second, we do not interrupt description changers. The implemented changers always terminate in finite time. Moreover, their running time is polynomial, which allows us to predict it much more accurately than solvers' search time. We therefore run changers without time bounds.

Third, we assume that the system faces one problem at a time. Thus, it does not have to select among available problems or decide on the order of solving them. The choice among problems would require the comparison of gain estimates for different problems. The ordering should also depend on the possibility of transferring experience from one problem to another.

Finally, SHAPER does not interleave the use of different representations in solving a problem, even though interleaving search techniques is sometimes more effective than trying them one after another.

7.4.2 Role of the user

The SHAPER system automates the construction and use of domain descriptions, and the selection among problem solvers; however, it relies on the user for certain initial knowledge. We discuss the user's role and options in guiding the system.

First and foremost, the user must encode domains in the PRODIGY language [Carbonell *et al.*, 1992], and provide problem-solving and description-changing algorithms. The PRODIGY architecture includes a number of solvers [Stone and Veloso, 1994; Veloso *et al.*, 1995; Fink and Blythe, 1998], as well as the changers described in Part II. The user may construct additional solvers and changers by adjusting the knobs of the existing algorithms, or provide her own new algorithms.

We have discussed some methods and general guidelines for designing new algorithms that improve domain descriptions (see Section 1.4.2); however, the development of novel algorithms is usually a complex research problem. Future work may include automatic construction of new solvers and changers, which is a challenging and largely unexplored research area. Minton [1996] has made some steps in this direction, in his work on automatically configuring constraint-satisfaction algorithms.

Second, the user must specify gain functions for all solvers and changers. The gain functions provide information on the value of solving problems and cost of executing SHAPER's algorithms. In other words, they encode the user's value judgments. We have extended the PRODIGY language for specifying these functions and implemented a procedure for verifying their compliance with the constraints of Section 7.3.

The rest of the user's input is optional; it includes solver and changer operators, control rules, initial descriptions, estimates of problem complexity, information on problem similarity, and sample problems used in learning.

The specification of solver and changer operators includes arranging the available algorithms into sequences and providing applicability conditions for these sequences. The purpose of applicability conditions, as well as rejection and comparison rules, is to prune parts of description and representation spaces. The operator conditions and pruning rules form *hard* constraints for guiding SHAPER's search.

The user may also provide preference rules, which encode *soft* constraints. We describe the main types of preference rules and mechanisms for their use in Chapter 10. Some of these rules incorporate simple learning techniques.

We extended the PRODIGY language for defining global operators and control rules, as well as additional operators and rules for specific domains. If the user does not specify operators, SHAPER does not use algorithm sequences, and assumes that all solvers and changers are always applicable.

We encoded several operators based on the implemented solvers and changers, as well as control rules for use of these operators. An extension to the library of algorithms would require additional operators and rules. Future work may include automatic generation of solver and changer sequences, applicability conditions, and control rules. In particular, we may try to adapt techniques for learning macros [Korf, 1985b; Shell and Carbonell, 1989; Driskill and Carbonell, 1996], operator preconditions [Gil, 1992; Wang, 1996], and control rules [Minton, 1988; Borrajo and Veloso, 1996] to use in the description space.

If the user provides her own domain descriptions, SHAPER uses them to form initial nodes in the description space. A specification of a domain description includes an abstraction hierarchy, selection of primary effects, and set of PRODIGY control rules. If the user does not specify descriptions, SHAPER creates an initial node with the "basic" description, which has no abstraction, primary effects, or control rules.

The user may provide procedures for estimating problem complexity and similarity between problems. The system utilizes this information in its performance analysis. We have not investigated methods for learning complexity estimates or similarity.

Finally, the user may give a collection of "easy" sample problems, or a function that generates such problems, for testing the performance of representations before solving hard problems (Section 10.3).

Chapter 8

Statistical selection among representations

Good ideas are based on past experience.
— George Polya [1957], *How to Solve It.*

Problem-solving gain depends on four factors: the gain function, the probability of encountering each problem, the representation, and the strategy for selecting time bounds (see Equation 7.4). The system has control over the last two factors, representation and time bound. The system's control module makes three types of top-level decisions, which determine the effectiveness of problem solving:

1. when to generate new representations;
2. which of the available representations to use for each given problem;
3. what time bound to set for the selected representation.

We now describe statistical techniques for selecting among available representations and setting time bounds. Then, in Section 11.1, we will describe a mechanism for deciding when to generate new representations. SHAPER uses these techniques to navigate the space of possible representations. The implemented procedures for choosing representations are very fast, and their computational cost is usually negligible compared to that of solver and changer algorithms.

First, we formalize the statistical problem of estimating the expected performance of a representation (Section 8.1). Then, we derive a solution to this problem (Sections 8.2 and 8.3) and show how to use it in selecting a representation and time bound (Sections 8.4 and 8.5). Finally, we give experimental results that confirm the effectiveness of the selection algorithm (Section 8.6 and 8.7).

8.1 Selection task

We begin with an outline of our results (Section 8.1.1), and then formalize the selection task in the SHAPER system (Section 8.1.2).

8.1.1 Previous and new results

Researchers have long realized the importance of automatic evaluation and selection of search methods, and developed techniques for various special cases of this problem. In particular, Horvitz described a framework for evaluating algorithms based on trade-offs between computation cost and solution quality, and used this framework in automatic selection of sorting algorithms [Horvitz, 1988]. Breese and Horvitz designed a decision-theoretic algorithm that evaluates different methods of belief-network inference and selects the optimal method [Breese and Horvitz, 1990]. Hansson and Mayer [1989], and Russell [1990] applied related evaluation and selection techniques to the problem of choosing promising branches of a search space.

Russell, Subramanian, and Parr formalized a general problem of selecting among alternative problem-solving methods, and used dynamic programming to solve some special cases of this problem [Russell *et al.*, 1993]. Minton developed an inductive learning system that configures constraint-satisfaction programs, by selecting among alternative search strategies [Minton, 1996; Allen and Minton, 1996].

Hansen and Zilberstein studied trade-offs between running time and solution quality in simple any-time algorithms, and designed a dynamic-programming technique for deciding when to terminate search [Hansen and Zilberstein, 1996]. Mouaddib and Zilberstein developed a similar technique for hierarchical knowledge-based algorithms [Mouaddib and Zilberstein, 1995].

We found that the previous results are not applicable to selecting representations in the SHAPER system, because the developed techniques rely on the analysis of a sufficiently large sample of past performance data. When we apply PRODIGY search engines to a new domain or generate new representations, we usually have little or no prior data. Acquisition of sufficient data is often impractical, because experimentation may prove significantly more expensive than solving given problems.

We have therefore developed a novel selection technique, which makes the best use of the available data, even when they do not provide an accurate estimate. We describe a learning algorithm that accumulates data on the performance of available representations, in the process of problem solving, and uses these data to select the representation that maximizes the expected gain.

We also consider the task of setting a time bound for the chosen representation. The previous results for deciding on the time of terminating any-time algorithms are not applicable to this task, because PRODIGY problem solvers do not use any-time behavior and do not satisfy the assumptions used in past studies. We provide a statistical technique for selecting time bounds, and demonstrate that determining an appropriate bound is as crucial for efficient problem solving as choosing the right representation.

The described techniques are aimed at selecting a representation and time bound *before* solving a given problem. We do not provide a mechanism for switching a representation or revising the selected bound *during* the search for a solution. Developing a revision mechanism is an important open problem.

We begin by formalizing the statistical problem of estimating the expected problem-solving gain (Section 8.1.2). We derive a solution to this problem (Sections 8.2 and 8.3), show how to use it in selecting a representation and time bound (Sections 8.4), and discuss heuristics for making a selection in the absence of statistical data (Section 8.5). Then, we give results from using the developed selection technique (Sections 8.6 and 8.7). Note that we do not need a perfect estimate of the expected gain; *we only need accuracy sufficient for selecting the right representation and near-optimal time bound.*

The statistical technique of Sections 8.2–8.6 chooses a representation and bound that maximize the average gain, but it does *not* adjust its choice to a specific problem. In terms of Section 7.3.4, this limitation means that it chooses a fixed representation i and bound B (see Equation 7.4). It does not construct a function $i(prob)$ for selecting a representation and functions $B_1(prob), ..., B_k(prob)$ for selecting time bounds.

In Chapter 9, we will lift this limitation and describe extensions to the statistical technique, which account for specific problem features. We will describe the use of an estimated problem size and similarity between problems.

In Chapter 10, we will present an alternative general mechanism for choosing representations, based on the use of preference rules, and describe its synergy with statistical learning. We use this mechanism to implement several selection techniques, including learning of preferences, evaluation of representations by solving special test problems, and the use of human advice.

8.1.2 Example and general problem

We now give an example of a situation where we need to select a representation by analyzing past performance data. We then generalize this example and state the statistical problem of choosing from available representations. We use the evaluation model of Section 7.3 in formalizing the problem.

Suppose that we use the PRODIGY system to construct plans for transporting packages by vans between different locations in a city [Veloso, 1994]. We consider the use of three different representations. The first of them is based on the use of the SAVTA search algorithm [Veloso and Stone, 1995], described in Section 2.2.5, with control rules designed by Veloso [1994] and Pérez [1995]. The SAVTA algorithm applies the selected actions to the current state of the simulated transportation world as early as possible; we call the representation based on this algorithm Apply.

The second representation uses the SABA algorithm (see Section 2.2.5) with the same set of control rules. Since SABA delays the application of the selected actions and forces more emphasis on the backward search [Veloso and Stone, 1995], we call this representation Delay. Finally, the third representation, called Abstract, uses a combination of SAVTA with the problem-specific version of the *Abstractor* algorithm, outlined in Section 5.3. *Abstractor* constructs an abstraction hierarchy for each given problem, and SAVTA uses the resulting hierarchy in problem solving.

If we use one of these representations to solve a problem, we may get one of three outcomes: the system may solve the problem; it may terminate with failure, after exhausting the available search space; or it may interrupt the search for a solution, upon reaching the time bound. Apply, Delay, and Abstract are applicable to all problems in the transportation domain, and they never give the fourth possible outcome, a rejection of the problem.

#	time (sec) and outcome			# of	#	time (sec) and outcome			# of
	Apply	Delay	Abstract	packs		Apply	Delay	Abstract	packs
1	1.6 s	1.6 s	1.6 s	1	16	4.4 s	68.4 s	4.6 s	4
2	2.1 s	2.1 s	2.0 s	1	17	6.0 s	200.0 b	6.2 s	6
3	2.4 s	5.8 s	4.4 s	2	18	7.6 s	200.0 b	7.8 s	8
4	5.6 s	6.2 s	7.6 s	2	19	11.6 s	200.0 b	11.0 s	12
5	3.2 s	13.4 s	5.0 s	3	20	200.0 b	200.0 b	200.0 b	16
6	54.3 s	13.8 f	81.4 s	3	21	3.2 s	2.9 s	4.2 s	2
7	4.0 s	31.2 f	6.3 s	4	22	6.4 s	3.2 s	7.8 s	4
8	200.0 b	31.6 f	200.0 b	4	23	27.0 s	4.4 s	42.2 s	16
9	7.2 s	200.0 b	8.8 s	8	24	200.0 b	6.0 s	200.0 b	8
10	200.0 b	200.0 b	200.0 b	8	25	4.8 s	11.8 f	3.2 s	3
11	2.8 s	2.8 s	2.8 s	2	26	200.0 b	63.4 f	6.6 f	6
12	3.8 s	3.8 s	3.0 s	2	27	6.4 s	29.1 f	5.4 f	4
13	4.4 s	76.8 s	3.2 s	4	28	9.6 s	69.4 f	7.8 f	6
14	200.0 b	200.0 b	6.4 s	4	29	200.0 b	200.0 b	10.2 f	8
15	2.8 s	2.8 s	2.8 s	2	30	6.0 s	19.1 s	5.4 f	4

Table 8.1: Performance of Apply, Delay, and Abstract on thirty transportation problems.

In Table 8.1, we give the results of solving thirty transportation problems, with each of the three representations. We denote successes by s, failures by f, and interrupts upon hitting the time bound by b. Note that our data in this example are only for illustrating the selection problem, and *not* for the purpose of a general comparison of these three search techniques. Their relative performance may be very different in other domains. In particular, the control rules in the transportation domain were developed for SAVTA, which gives it an advantage over SABA.

Even though each representation outperforms the others on at least one transportation problem (see Table 8.1), a glance at the data reveals that Apply's performance in this domain is probably the best among the three. We use statistical analysis to confirm this intuitive conclusion and to estimate its statistical significance. We also show how to select a time bound for the chosen representation.

If we identify several distinct problem types in a domain, we may discover that different types require different representations and time bounds. In other words, the appropriate choice of a representation may depend on the properties of a problem. We will analyze such situations in Section 9.3.

We suppose that the user has specified some gain function, and that we have sample data on the past performance of every representation. We need to select a representation and time bound that maximize the expected gain G, determined by Equation 7.3.

We can estimate the expected gain for all candidate representations and time bounds, using past performance data, and select the representation and bound that give the largest estimate. We distinguish between past terminations and interrupts in deriving the estimate, but we do not make any distinction among the three types of terminations (successes, failures, and rejections). We thus consider the following statistical problem.

Problem: *Suppose that a representation terminated on n problems, called $pt_1, ..., pt_n$, and* *gave an interrupt (upon hitting a time bound) on m problems, called* $pb_1, ..., pb_m$. *The termination times were* $t_1, ..., t_n$ *and the corresponding results were* $result_1, ..., result_n$*; the interrupt times were* $b_1, ..., b_m$. *Given a gain function and a time bound B, estimate the expected gain and determine the standard deviation of the estimate.*

The termination results, denoted $result_1, ..., result_n$, may include solutions, failures, and rejections. On the other hand, all interrupts by definition give the same result, denoted intr.

We need a gain estimate that makes the *best use of the available data*, even if these data are not sufficient for statistical significance. We cannot ask for more data, since experimentation is usually much more expensive than solving a given problem. In addition, we should ensure that *statistical computations take little time,* especially since the statistical model does not account for this addition to the overall problem-solving time.

We now give an example of a gain function; we will use this function to evaluate gains in the transportation domain. Suppose that we get a certain reward R for solving a transportation problem, and that we have to pay for each second of running time. In this example, if the system solves a given problem, the overall gain is $(R - time)$. If the system fails or hits a time bound, it is $(-time)$. Thus, the gain is a linear function of time:

For every *prob*, *time*, and *soln*,
(a) $gain(prob, time, soln) = R - time$,
(b) $gain(prob, time, \mathsf{fail}) = -time$,
(c) $gain(prob, time, \mathsf{intr}) = -time$.

We assume that this gain function is the same for all three representations in the transportation example.

Suppose that we choose one of the three representations and use some fixed time bound B. For each *prob*, the representation uniquely determines the (finite or infinite) running time until termination, $time(prob)$. If we use the example gain function, we may rewrite Equation 7.1 as follows:

$$gain'(prob, B) = \begin{cases} R - time, & \text{if } B \geq time(prob) \text{ and the outcome is success} \\ -time, & \text{if } B \geq time(prob) \text{ and the outcome is failure} \\ -B, & \text{if } B < time(prob) \end{cases}$$

We need to select the representation and time bound B that maximize the expected value of $gain'(prob, B)$.

8.2 Statistical foundations

We now derive a solution to the statistical problem. That is, we evaluate the results of problem solving with a fixed representation and time bound, and determine the estimate of the expected gain and the standard deviation of the estimate.

We assume, for convenience, that the termination and interrupt times are sorted in increasing order; that is, $t_1 \leq t_2 \leq ... \leq t_n$ and $b_1 \leq b_2 \leq ... \leq b_m$. We first consider the case where the time bound B is no larger than the lowest of the past time bounds; that is,

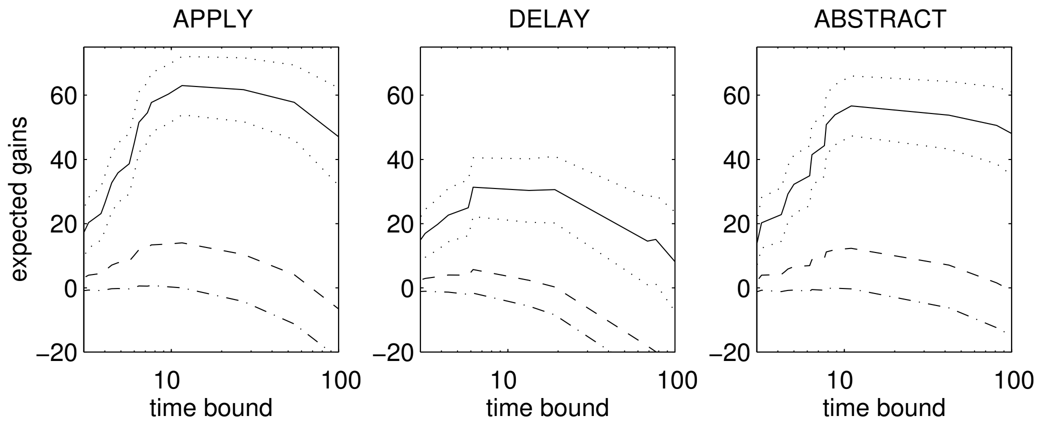


Figure 8.1: Dependency of the expected gain on the time bound, for a reward of 10.0 (dash-and-dot lines), 30.0 (dashed lines), and 100.0 (solid lines). The dotted lines show the standard deviation of the expected gain for the 100.0 reward.

$B \leq b_1$. We denote the number of termination times that are no larger than B by c; that is, $t_c \leq B < t_{c+1}$.

We estimate the expected gain by averaging the gains that would be obtained in the past, if we used the time bound B for all problems. The problem solver would terminate without hitting the time bound on the problems $pt_1, ..., pt_c$, earning the gains $gain(pt_1, t_1, result_1)$,..., $gain(pt_c, t_c, result_c)$. It would hit the time bound on $pt_{c+1}, ..., pt_n$, resulting in the negative gains $gain(pt_{c+1}, B, \mathsf{intr})$,..., $gain(pt_n, B, \mathsf{intr})$. It would also hit the bound on $pb_1, ..., pb_m$, resulting in the negative gains $gain(pb_1, B, \mathsf{intr})$,..., $gain(pt_m, B, \mathsf{intr})$. The estimate of the expected gain is equal to the mean of these $n + m$ gain values:

$$\frac{\sum_{i=1}^{c} gain(pt_i, t_i, result_i) + \sum_{i=c+1}^{n} gain(pt_i, B, \mathsf{intr}) + \sum_{j=1}^{m} gain(pb_j, B, \mathsf{intr})}{n + m}. \tag{8.1}$$

For example, suppose that we solve transportation problems with Abstract, and that we use the example gain function with reward $R = 30.0$. The gain of solving a problem is $(30.0 - time)$ and the negative gain of a failure or interrupt is $(-time)$. If we apply Equation 8.1 to estimate the gain for the bound $B = 6.0$, using the data in Table 8.1, we find that the expected gain is 6.0. If we raise the bound to 8.0, the expected gain becomes 11.1.

In Figure 8.1, we show the dependency of the expected gain on time bound for our three representations. We give the dependency for three different values of the reward R, 10.0 (dash-and-dot lines), 30.0 (dashed lines), and 100.0 (solid lines).

Since we have computed the mean gain for a randomly selected sample of problems, it may be different from the mean of the overall population. We estimate the standard deviation of the expected gain using the formula for the deviation of a sample mean:

$$\sqrt{\frac{SqrSum - \frac{Sum^2}{n+m}}{(n+m) \cdot (n+m-1)}}, \tag{8.2}$$

where

$$Sum = \sum_{i=1}^{c} gain(pt_i, t_i, result_i) + \sum_{i=c+1}^{n} gain(pt_i, B, \mathsf{intr}) + \sum_{j=1}^{m} gain(pb_j, B, \mathsf{intr}),$$
$$SqrSum = \sum_{i=1}^{c} gain(pt_i, t_i, result_i)^2 + \sum_{i=c+1}^{n} gain(pt_i, B, \mathsf{intr})^2 + \sum_{j=1}^{m} gain(pb_j, B, \mathsf{intr})^2.$$

This formula is an approximation based on the *Central Limit Theorem*, which states that the distribution of sample means is close to normal (see, for example, Mendenhall's textbook [1987]). The accuracy of the approximation improves with sample size; for thirty or more sample problems, it is near-perfect.

For example, if we use Abstract with reward 30.0 and time bound 6.0, the standard deviation of the expected gain is 2.9. In Figure 8.1, we use dotted lines to show the standard deviation of the gain estimate for the 100.0 reward, for all three representations. The lower dotted line is "one standard deviation below" the estimate, and the upper line is "one standard deviation above."

We have so far assumed that $B \leq b_1$. We now consider the case when B is larger than d of the past interrupt times; that is, $b_d < B \leq b_{d+1}$. We cannot use $b_1, b_2, ..., b_d$ directly in the gain estimate, because the use of the time bound B would cause the problem solver to run beyond these old bounds. The collected data do not tell us the results of solving the corresponding past problems with the time bound B. We estimate these results by "re-distributing" the probabilities of reaching these low time bounds among the other outcomes.

If we had not interrupted the problem solver at b_1 in the past, it would have terminated at some larger time or hit a larger time bound. We may estimate the expected gain using the data on the past problem-solving episodes in which the solver ran beyond b_1. We get this estimate by averaging the gains for all the larger-time outcomes of problem solving. To incorporate this averaging into the computation, we remove b_1 from the sample and distribute its chance to occur among the larger-time outcomes.

To implement this re-distribution, we assign weights to all past outcomes. Initially, the weight of every outcome is 1. After removing b_1, we distribute its weight among all the larger-than-b_1 outcomes. If the number of such outcomes is a_1, each of them gets the weight of $1 + \frac{1}{a_1}$. Note that $b_2, ..., b_d$ are all larger than b_1, and thus they all get the new weight.

We next remove b_2 from the sample and distribute its weight, which is $1 + \frac{1}{a_1}$, among the larger-than-b_2 outcomes. If the number of such outcomes is a_2, then we increase their weights by $(1 + \frac{1}{a_1}) \cdot \frac{1}{a_2}$; that is, their weights become $(1 + \frac{1}{a_1}) \cdot (1 + \frac{1}{a_2})$. We repeat the distribution process for all the interrupt times smaller than B; that is, for $b_3, ..., b_d$.

We illustrate the re-distribution technique using the data on Abstract's performance (see Table 8.1). Suppose that we interrupted Abstract on problem 4 after 4.5 seconds of the execution and on problem 7 after 5.5 seconds, thus obtaining the data shown in Table 8.2(a), and that we need to estimate the gain for $B = 6.0$. This bound B is larger than two interrupt times.

We first have to distribute the weight of b_1. In this example, b_1 is 4.5 and there are 21 problems with larger times. We remove 4.5 from the sample data and increase the weights of the larger-time outcomes from 1 to $1 + \frac{1}{21} = 1.048$ (see Table 8.2b). We next perform the distribution for b_2, which is 5.5. The table contains 15 problems with larger-than-b_2 times. We distribute b_2's weight, 1.048, among these 15 problems, thus increasing their weight to $1.048 + \frac{1.048}{15} = 1.118$ (Table 8.2c).

#	Abstract's time
1	1.6 s
2	2.0 s
3	4.4 s
4	4.5 b
5	5.0 s
6	81.4 s
7	5.5 b
8	200.0 b
9	8.8 s
...	
29	10.2 f
30	5.4 f

(a)

$\longrightarrow$

weight	time
1.000	1.6 s
1.000	2.0 s
1.000	4.4 s
–	–
1.048	5.0 s
1.048	81.4 s
1.048	5.5 b
1.048	200.0 b
1.048	8.8 s
...	
1.048	10.2 f
1.048	5.4 f

(b)

$\longrightarrow$

weight	time
1.000	1.6 s
1.000	2.0 s
1.000	4.4 s
–	–
1.048	5.0 s
1.118	81.4 s
–	–
1.118	200.0 b
1.118	8.8 s
...	
1.118	10.2 f
1.048	5.4 f

(c)

Table 8.2: Distributing the weights of interrupt times among the larger-time outcomes.

We have thus removed the interrupt times $b_1, b_2, ..., b_d$ from the sample, and assigned weights to the termination times and remaining interrupt times. We denote the resulting weights of the termination times $t_1, ..., t_c$ by $u_1, ..., u_c$; recall that these termination times are no larger than B. All termination and interrupt times larger than B have the same weight, which we denote by u_*. Note that the sum of the weights is equal to the number of problems in the original sample; that is, $\sum_{i=1}^{c} u_i + (n+m-c-d) \cdot u_* = n+m$.

We have obtained $n+m-d$ weighted times. We use them to compute the estimate of the expected gain, which gives us the following expression:

$$\frac{\sum_{i=1}^{c} u_i \cdot gain(pt_i, t_i, result_i) + u_* \cdot \sum_{i=c+1}^{n} gain(pt_i, B, \mathsf{intr}) + u_* \cdot \sum_{j=d+1}^{m} gain(pb_j, B, \mathsf{intr})}{n+m}. \quad (8.3)$$

Similarly, we use the weights in computing the standard deviation of the gain estimate:

$$\sqrt{\frac{SqrSum - \frac{Sum^2}{n+m}}{(n+m)\cdot(n+m-d-1)}}, \quad (8.4)$$

where

$$Sum = \sum_{i=1}^{c} u_i \cdot gain(pt_i, t_i, result_i) + u_* \cdot \sum_{i=c+1}^{n} gain(pt_i, B, \mathsf{intr}) + u_* \cdot \sum_{j=d+1}^{m} gain(pb_j, B, \mathsf{intr}),$$
$$SqrSum = \sum_{i=1}^{c} u_i \cdot gain(pt_i, t_i, result_i)^2 + u_* \cdot \sum_{i=c+1}^{n} gain(pt_i, B, \mathsf{intr})^2 + u_* \cdot \sum_{j=d+1}^{m} gain(pb_j, B, \mathsf{intr})^2.$$

The application of these formulas to the data in Table 8.2(c), for Abstract with reward 30.0 and time bound 6.0, gives the expected gain 6.1 and the standard deviation 3.0.

If B is larger than the largest of the past bounds (that is, $B > b_m$) and the largest time bound is larger than all past termination times (that is, $b_m > t_n$), then the re-distribution procedure does not work. We need to distribute the weight of b_k among the larger-time outcomes, but the sample has no such outcomes. In this case, the data are not sufficient for the statistical analysis because we do not have any past experience with large enough time bounds.

8.3 Computation of the gain estimates

We now describe an algorithm that computes the gain estimates and their deviations according to Equations 8.3 and 8.4, for multiple values of the time bound B.

We have assumed in deriving the gain estimate that we try to solve each problem once, and do not return to the same problem later. In practice, if we have interrupted the search, we may later return to the same problem and continue the search from the point where we have stopped. Note that we need to store the expanded search space in order to continue the search later. Thus, the reuse of the previously expanded spaces requires extra memory, but it may save significant time.

We develop a gain-estimate algorithm that takes into account the reuse of expanded spaces. We modify the equations for the gain estimate and its deviation, to account for the space reuse, and then give the pseudocode for statistical computations.

Suppose that we have tried to solve a problem *prob* in the past and hit a time bound B_o (where the subscript "o" stands for "old"). We may store the complete space of the interrupted search, or part of the space, expanded until reaching some bound b_o, where $b_o \leq B_o$. If we reuse the space expanded by the time b_o, we refer to b_o as the *reused time*.

Assume that, if we did not reuse an old search space, then the running time until termination on *prob* would be *time*(*prob*). If we reuse the search space expanded by b_o, the running time until termination is $time(prob) - b_o$.

Now suppose that we are solving *prob* again, with the same representation and a new time bound B. We measure time from the beginning of the *first* problem-solving attempt. That is, if we reuse the space expanded in time b_o, then the new search begins at time b_o and reaches the bound at time B; the running time of the new problem-solving attempt is bounded by $(B - b_o)$. Since we know that the representation hits the bound B_o without solving the problem, we must set a large bound for the new attempt; that is, $B > B_o$.

We modify Equation 7.1 for solving a problem with time bound B and reused time b_o:

$$gain'(prob, B) = \begin{cases} gain(prob, time(prob) - b_o, result(prob)), & \text{if } B \geq time(prob) \\ gain(prob, B - b_o, \mathsf{intr}), & \text{if } B < time(prob) \end{cases}$$

We next determine the expected value of the problem-solving gain, based on the past performance data. First, we remove the previous interrupted attempt to solve the given problem from the performance data. We denote the termination times in the resulting sample by $t_1, ..., t_n$ and interrupt times by $b_1, ..., b_m$.

We denote the number of past termination times that are no larger than B_o by e; that is, $t_e \leq B_o < t_{e+1}$. Note that, since $B_o < B$, we have $e \leq c$. We know that $time(prob) > B_o$ and, hence, we must not use the past termination times $t_1, ..., t_e$ in computing the expected gain. We remove these times from the sample, and use the remaining termination times $t_{e+1}, ..., t_n$ and interrupt times $b_1, ..., b_m$ to estimate the gain. Thus, the sample now comprises $(n+m-e)$ past outcomes.

We apply the estimation technique of Section 8.2 to the reduced sample. If some interrupt times $b_1, ..., b_d$ are smaller than B, we distribute their weights among the larger-time outcomes. We denote the resulting weights of the termination times $t_{e+1}, ..., t_c$ by $u'_{e+1}, ..., u'_c$, and the weight of all other times by u'_*. The sum of all weights is equal to the number of

c	number of the already processed termination times; the next termination time to process will be t_{c+1}
d	number of the already processed interrupt times
h	number of the already processed time bounds
u	weight assigned to the times that are larger than the current time bound B_{h+1}
T_Sum	weighted sum of the gains for the processed terminations; that is, $\sum_{i=e+1}^{c} u'_i \cdot gain(pt_i, t_i - b_o, result_i)$
I_Sum	unweighted sum of the interrupt gains for the current bound B_{h+1}; that is, $\sum_{i=c+1}^{n} gain(pt_i, B - b_o, \mathsf{intr}) + \sum_{j=d+1}^{m} gain(pb_j, B - b_o, \mathsf{intr})$
Sum	weighted sum of the gains for all sample problems, for the current bound B_{h+1}
T_SqrSum	weighted sum of the squared gains for the processed terminations; that is, $\sum_{i=e+1}^{c} u'_i \cdot gain(pt_i, t_i - b_o, result_i)^2$
I_SqrSum	unweighted sum of the squared interrupt gains for the bound B_{h+1}
SqrSum	weighted sum of the squared gains for all sample problems, for the bound B_{h+1}

Figure 8.2: Variables used in the gain-estimate algorithm in Figure 8.3.

problems in the reduced sample; that is, $\sum_{i=e+1}^{c} u'_i + (n+m-c-d) \cdot u'_* = n+m-e$. We compute the estimate of the expected gain as follows:

$$\frac{\sum_{i=e+1}^{c} u'_i \cdot gain(pt_i, t_i - b_o, result_i) + u'_* \cdot \sum_{i=c+1}^{n} gain(pt_i, B - b_o, \mathsf{intr}) + u'_* \cdot \sum_{j=d+1}^{m} gain(pb_j, B - b_o, \mathsf{intr})}{n+m-e}. \tag{8.5}$$

The standard deviation of this estimate is

$$\sqrt{\frac{SqrSum - \frac{Sum^2}{n+m-e}}{(n+m-e) \cdot (n+m-e-d-1)}}, \tag{8.6}$$

where

$$Sum = \sum_{i=e+1}^{c} u'_i \cdot gain(pt_i, t_i - b_o, result_i) + u'_* \cdot \sum_{i=c+1}^{n} gain(pt_i, B - b_o, \mathsf{intr}) + u'_* \cdot \sum_{j=d+1}^{m} gain(pb_j, B - b_o, \mathsf{intr}),$$

$$SqrSum = \sum_{i=e+1}^{c} u'_i \cdot gain(pt_i, t_i - b_o, result_i)^2 + u'_* \cdot \sum_{i=c+1}^{n} gain(pt_i, B - b_o, \mathsf{intr})^2 + u'_* \cdot \sum_{j=d+1}^{m} gain(pb_j, B - b_o, \mathsf{intr})^2.$$

We now describe the computation of the gain estimate and its deviation, for multiple candidate bounds $B_1, ..., B_l$. We list the variables used in the computation in Figure 8.2 and give the pseudocode in Figure 8.3.

The algorithm determines weights and computes gain estimates in one pass through the sorted list of termination times, interrupt times, and time bounds. When processing a termination datum, the algorithm increments the sum of the weighted gains and the sum of their squares. When processing an interrupt datum, the algorithm modifies the weight value. When processing a time bound, the algorithm uses the accumulated sums of gains and squared gains to compute the gain estimate and deviation for this bound.

Note that we execute Step 2 of the algorithm at most $n - e$ times, Step 3 at most m times, and Step 4 at most l times. We compute one value of the gain function at Step 2, and

Estimate-Gains

The input of the algorithm includes: the gain function *gain*; the sorted list of termination times $t_{e+1}, ..., t_n$, with the corresponding problems $pt_{e+1}, ..., pt_n$ and results $result_{e+1}, ..., result_n$; the sorted list of interrupt times $b_1, ..., b_m$, with the corresponding problems $pb_1, ..., pb_m$; a sorted list of candidate time bounds $B_1, ..., B_l$; and the reused time b_o. The variables used in the computation are described in Figure 8.2.

Set the initial values:

$c := e;\ d := 0;\ h := 0$

$u := 1;\ T\_Sum := 0;\ T\_SqrSum := 0$

Repeat the computations until finding the gains for all time bounds; that is, until $h = l$*:*

1. Select the smallest among the following three times: t_{c+1}, b_{d+1}, and B_{h+1}.
2. If the termination time t_{c+1} is selected, increment the related sums:
   $T\_Sum := T\_Sum + u \cdot gain(pt_{c+1}, t_{c+1} - b_o, result_i)$
   $T\_SqrSum := T\_SqrSum + u \cdot gain(pt_{c+1}, t_{c+1} - b_o, result_i)^2$
   $c := c + 1$
3. If the interrupt time b_{e+1} is selected:
   If no termination times are left (that is, $c = n$),
   then terminate (the data are not sufficient to estimate the gains for the remaining bounds).
   Else, distribute the interrupt's weight among the remaining times, by incrementing u and d:
   $u := u + u \cdot \frac{n+m-e-c-d}{n+m-e-c-d-1}$
   $d := d + 1$
4. If the time bound B_{h+1} is selected:
   First, compute the unweighted sum of interrupt gains and the sum of their squares:
   $I\_Sum := \sum_{i=c+1}^{n} gain(pt_i, B_{h+1} - b_o, \mathsf{intr}) + \sum_{j=d+1}^{m} gain(pb_j, B_{h+1} - b_o, \mathsf{intr})$
   $I\_SqrSum := \sum_{i=c+1}^{n} gain(pt_i, B_{h+1} - b_o, \mathsf{intr})^2 + \sum_{j=d+1}^{m} gain(pb_j, B_{h+1} - b_o, \mathsf{intr})^2$
   Next, compute the overall sums of the sample-problem gains and their squares:
   $Sum := T\_Sum + u \cdot I\_Sum$
   $SqrSum := T\_SqrSum + u \cdot I\_SqrSum$
   Now, compute the gain estimate and deviation, for the bound B_{h+1}:
   Gain estimate: $\frac{Sum}{n+m-e}$
   Estimate deviation: $\sqrt{\frac{SqrSum - Sum^2/(n+m-e)}{(n+m-e)\cdot(n+m-e-d-1)}}$
   Finally, increment the number of processed bounds:
   $h := h + 1$

Figure 8.3: Computing the gain estimates and estimate deviations.

$n+m-e-d$ different values at Step 4. If the time complexity of computing the gain function is *Comp*(*gain*), than the overall complexity of the algorithm is $O(l \cdot (n+m-e) \cdot Comp(gain))$.

The computation of *I_Sum* and *I_SqrSum* in Step 4 is the most time-consuming part of the algorithm; however, we may significantly speed up this computation for some common special cases of the gain function. We consider two special cases, which allow us to compute the gain estimates in linear time.

First, suppose that the interrupt gains do not depend on a specific problem; that is, we have an interrupt gain function $gain_{\mathrm{i}}(time)$ such that:

For every *prob* and *time*,
$gain(prob, time, \mathsf{intr}) = gain_{\mathrm{i}}(time)$.

We then compute *I_Sum* and *I_SqrSum* in constant time:

$I\_Sum = (n + m - c) \cdot gain_{\mathrm{i}}(B_{h+1} - b_o)$,
$I\_SqrSum = (n + m - c) \cdot gain_{\mathrm{i}}(B_{h+1} - b_o)^2$.

Since we execute Step 4 at most l times, we need to compute at most l values of $gain_{\mathrm{i}}$ and, hence, the overall time complexity of the algorithm is $O((l + n - e) \cdot Comp(gain) + m)$. In particular, if the computation of the gain function takes constant time, then the overall complexity is linear, $O(l + n + m - e)$.

Next, we consider a gain function that satisfies Constraints 8 and 9 of Section 7.3.2. Then, the interrupt gain is proportional to the running time:

$$gain(prob, time, \mathsf{intr}) = time \cdot gain(prob, 1, \mathsf{intr}).$$

We use this property in computing the values of *I_Sum* and *I_SqrSum*:

$I\_Sum = (B_{h+1} - b_o) \cdot \left(\sum_{i=c+1}^{n} gain(pt_i, 1, \mathsf{intr}) + \sum_{j=d+1}^{m} gain(pb_j, 1, \mathsf{intr})\right)$,
$I\_SqrSum = (B_{h+1} - b_o) \cdot \left(\sum_{i=c+1}^{n} gain(pt_i, 1, \mathsf{intr})^2 + \sum_{j=d+1}^{m} gain(pb_j, 1, \mathsf{intr})^2\right)$.

We denote the four summation terms in this computation as follows:

$Sum_c = \sum_{i=c+1}^{n} gain(pt_i, 1, \mathsf{intr})$,
$SqrSum_c = \sum_{i=c+1}^{n} gain(pt_i, 1, \mathsf{intr})^2$,
$Sum_d = \sum_{j=d+1}^{m} gain(pb_j, 1, \mathsf{intr})$,
$SqrSum_d = \sum_{j=d+1}^{m} gain(pb_j, 1, \mathsf{intr})^2$.

We may pre-compute these sums, for all values of c and d, before executing the statistical algorithm. We present pseudocode for finding them in Figure 8.4; its time complexity is $O((m + n - e) \cdot Comp(gain))$.

We use the pre-computed sums at Step 4 of the statistical algorithm, which allows us to find the values of *I_Sum* and *I_SqrSum* in constant time. The overall complexity of the algorithm, including the pre-computation, is $O((m + n - e) \cdot Comp(gain) + l)$. If we can compute each gain value in constant time, then the complexity is $O(l + n + m - e)$.

Before calling the statistical algorithm, we need to sort the termination times, interrupt times, and time bound. We also need to remove the times $t_1, ..., t_e$ from the sample of past

Pre-Compute-Sums$(gain; pt_{e+1}, ..., pt_n; pb_1, ..., pb_m)$

Set the initial values:
$Sum_n := gain(pt_n, 1, \mathsf{intr})$; $SqrSum_n := gain(pt_n, 1, \mathsf{intr})^2$
$Sum_m := gain(pb_m, 1, \mathsf{intr})$; $SqrSum_m := gain(pb_m, 1, \mathsf{intr})^2$

Repeat for c from $n-1$ downto e:
$Sum_c := Sum_{c+1} + gain(pt_c, 1, \mathsf{intr})$
$SqrSum_c := SqrSum_{c+1} + gain(pt_c, 1, \mathsf{intr})^2$

Repeat for d from $m-1$ downto 1:
$Sum_d := Sum_{d+1} + gain(pb_d, 1, \mathsf{intr})$
$SqrSum_d := SqrSum_{d+1} + gain(pb_d, 1, \mathsf{intr})^2$

Figure 8.4: Pre-computing the sums of interrupt gains.

data. The complexity of these operations is $O((l+n+m) \cdot \log(l+n+m))$, but in practice they take much less than the rest of the computation.

We implemented the algorithm in Common Lisp and tested it on a Sun 5, the same computer as we used for running problem solvers and description changers. For the example gain function described in the end of Section 8.1.2, the running time of the implemented algorithm is about $(l+n+m) \cdot 3 \cdot 10^{-4}$ seconds.

8.4 Selection of a representation and time bound

We describe the use of the statistical estimate to choose among representations and to determine appropriate time bounds. We provide heuristics for combining the exploitation of past experience with exploration of new alternatives, and for making a choice in the absence of past data.

The basic technique is to estimate the gain for a number of time bounds, for each available representation, and select the representation and time bound that maximize the gain. For instance, if we solve problems in the transportation domain and use the example gain function with reward 30.0, than the best choice is Apply with the time bound 11.6, which gives the expected gain of 14.0. This choice corresponds to the maximum of the dashed lines in Figure 8.1. If the expected gain for all time bounds is negative, than we are better off not solving the problem at all. For example, if the only available representation is Delay and the reward is 10.0 (see the dash-and-dot line in Figure 8.1), we should skip the problem.

We describe a technique for incremental learning of the performance of available representations. We may begin problem solving with no data on the performance of available representations, or with some sample of past results. We accumulate additional data as we solve more problems. For each new problem, we use statistical estimates to select a representation and time bound. After using the selected representation, we add the result to the performance data.

Note that we have to choose a representation and time bound even if we have no past experience. Also, we sometimes need to deviate from the maximal-expectation choice in order to explore new opportunities. If we began with no past data and always used the selection that maximizes the expected gain, we would be stuck with the representation that yielded the first positive gain, and we would never choose a time bound higher than the first termination time.

The use of the statistical estimates in incremental learning causes a deviation from rigorous statistics: the resulting termination and interrupt times are not independent, because the representation and time bound used for each problem depend on the times for solving the previous problems. In spite of this violation of rigor, the technique gives good results in practice.

We have not constructed a statistical model for combining exploitation and exploration. Instead, we provide a heuristic solution, which has worked well for selecting representations in the $\mathcal{S}$HAPER system. We first describe the choice of candidate time bounds, for which we estimate problem-solving gains (Section 8.4.1). We consider the task of learning an appropriate time bound for a fixed representation (Section 8.4.2), and then show how to select a representation (Section 8.4.3).

8.4.1 Choice of candidate bounds

We have described a statistical algorithm that estimates gains for a finite list of candidate time bounds (see Figure 8.3). We thus need to select candidate bounds based on the available performance data.

We use past termination times as candidate time bounds, and compute expected gains only for these bounds. If we computed the gain for some other time bound B, we would get a smaller estimate than for the closest lower termination time t_i, where $t_i < B < t_{i+1}$. Extending the time bound from t_i to B would not increase the number of terminations on the past problems and, hence, it would not increase the gain estimate.

If some of the previously solved problems gave negative gains, we prune the corresponding termination times from the selection of candidate bounds. To put it formally, suppose that the data sample contains a termination time t_i, with the corresponding problem pt_i and $result_i$. If $gain(pt_i, t_i, result_i) \leq 0$, then we do not use t_i as a candidate bound, because this bound would give a smaller gain estimate than t_{i-1}.

Now suppose that we have tried to solve some problem in the past and hit a time bound B_o, and that we need to estimate gains for a new attempt to solve this problem. Then, all candidate bounds must be larger than B_o. Thus, if $t_e \leq B_o < t_{e+1}$, we use only $t_{e+1}, ..., t_n$ as candidate bounds. If we reuse the expanded search space, with the reused time b_o, then we modify the condition for pruning the termination times that give negative gains: $gain(pt_i, t_i - b_o, result_i) \leq 0$.

We multiply the selected bounds by 1.001, in order to avoid the chance of interrupting the solver too early because of rounding errors. If several candidate bounds are "too close" to each other, we drop some of them, to reduce the computation. In our implementation, we consider two bounds too close if they are within the factor of 1.05 from each other.

We summarize the algorithm for generating candidate bounds in Figure 8.5. We used this algorithm to select time bounds for constructing the graphs in Figure 8.1.

Generate-Bounds($gain; t_{e+1}, ..., t_n; pt_{e+1}, ..., pt_n; result_{e+1}, ..., result_n; b_o$)

Set the initial values:
$Bounds := \emptyset$ (list of candidate bounds)
$B := 0$ (largest generated bound; 0 if none)

Repeat for i from e + 1 to n:
If $t_i \geq 1.05 \cdot B$ and $gain(pt_i, t_i - b_o, result_i) > 0$, then:
$B := 1.001 \cdot t_i$
$Bounds := Bounds \cup \{B\}$

Figure 8.5: Generating candidate time bounds, based on the past termination times.

8.4.2 Setting a time bound

We consider the use of a fixed representation to solve a sequence of problems, and describe a technique for learning an appropriate time bound in the process of problem solving. To illustrate the use of this technique, we learn time bounds for Apply, Delay, and Abstract in the transportation domain.

If we have no previous data on using a representation, we set some default bound. We will discuss heuristics for selecting this initial bound in Section 8.5.1. If we use the example gain function, described in the end of Section 8.1.2, we set an initial bound equal to the reward R. This heuristic is based on the observation that, for PRODIGY search engines, the probability of solving a problem, say, within the next second, usually declines with the passage of search time. If a solver has not terminated within half a minute, chances are it will not find a solution in the next half minute either. Thus, if the reward is 30.0 and the solver has already run for 30.0 seconds, it is time to interrupt the search.

Now suppose that we have accumulated some performance data, which enable us to determine the time bound that maximizes the gain estimate. To encourage exploration, we select the largest bound whose gain estimate is "not much different" from the maximum. Let us denote the maximal estimate by $g_{\max}$ and its standard deviation by $\sigma_{\max}$. Suppose that the estimate for some bound is g and its deviation is σ. Then, the expected difference between the gain g and the maximal gain is $g_{\max} - g$. If we assume that the estimates are normally distributed, then the standard deviation of the expected difference is $\sqrt{\sigma_{\max}^2 + \sigma^2}$. Note that this estimate of the deviation is an approximation, because the distribution for small samples may be Student's rather than normal, and because $g_{\max}$ and g are not independent variables, as they are computed from the same data.

We say that g is "not much different" from the maximal gain if the ratio of the expected difference to its deviation is bounded by some constant. In most experiments, we set this constant to 0.1, which tends to give good results:

$$\frac{g_{\max} - g}{\sqrt{\sigma_{\max}^2 + \sigma^2}} \leq 0.1.$$

We thus select the largest time bound whose gain estimate g satisfies this condition. We

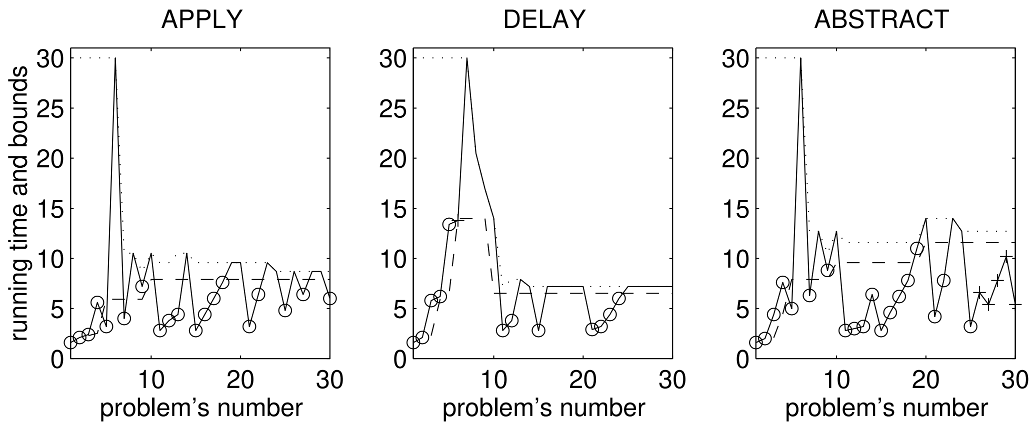


Figure 8.6: Results of incremental learning of a time bound: running times (solid lines), time bounds (dotted lines), and maximal-gain time bounds (dashed lines). The successes are marked by circles and the failures by pluses.

have also experimented with other constants, from 0.02 to 0.5, for bounding $\frac{g_{\max}-g}{\sqrt{\sigma_{\max}^2+\sigma^2}}$. We describe the results of these experiments in Sections 12.3, 13.3, 14.3, and 15.3.

In Figure 8.6, we present the results of this selection strategy, with the bounding constant 0.1. In this experiment, we determine time bounds for the transportation domain; we use the example gain function with a reward of 30.0. We apply each of the three representations to solve the thirty transportation problems from Table 8.1. The horizontal axes show the number of a problem, from 1 to 30, and the vertical axes are the running time. The dotted lines show the selected time bounds and the dashed lines mark the time bounds that give the maximal gain estimates. The solid lines show the running times; they touch the dotted lines where the system hits the time bound. The successfully solved problems are marked by circles and the failures are shown by pluses.

Apply's total gain is 360.3, which gives an average of 12.0 per problem. If we used the maximal-gain time bound, 11.6, for all problems, the average gain would be 14.0 per problem. Thus, the use of incremental learning has yielded a near-optimal gain, in spite of the initial ignorance. The time bounds used with this representation (dotted line) converge to the estimated maximal-gain bounds (dashed line), since the deviations of the gain estimates decrease as we solve more problems. Apply's estimate of the maximal-gain bound, after solving all problems, is 9.6. This estimate differs from the 11.6 bound, based on the data in Table 8.1, because the use of bounds that ensure a near-maximal gain has prevented sufficient exploration.

Delay's total gain is 115.7, or 3.9 per problem. If we used the data in Table 8.1 to find the optimal bound, which is 6.2, and solved all problems with this bound, we would earn 5.7 per problem. Thus, the incremental-learning gain is about two-thirds of the gain that could be obtained based on advance knowledge. Finally, Abstract's total gain is 339.7, or 11.3 per problem. The estimate based on Table 8.1 gives the bound 11.0, which would result in earning 12.3 per problem. Unlike Apply, both Delay and Abstract have found the optimal bound.

Note that the main "losses" in incremental learning occur on the first ten problems, when the past data are not sufficient for selecting an appropriate time bound. After this initial period, the choice of a time bound becomes close to the optimal.

Also note that the choice of a time bound converged to the optimal in two out of three experiments. Further tests have shown that insufficient exploration prevents finding the optimal bound in about half of all cases. If we encourage more exploration by increasing the upper limit for $\frac{g_{\max}-g}{\sqrt{\sigma_{\max}^2+\sigma^2}}$, then the selected bound converges to the optimal more often; however, the overall performance worsens due to larger time losses on unsolved problems.

The total time of the statistical computations while solving the thirty problems is 0.26 seconds, which makes less than 0.01 per problem. This time is negligible in comparison with the problem-solving time, which averages at 6.5 seconds per problem for Apply, 7.7 per problem for Delay, and 7.1 per problem for Abstract.

8.4.3 Selecting a representation

We next describe the use of incremental learning to select an appropriate representation, from the pool of available representations.

If we have no data on the performance of some representation, we always select this unknown representation. The optimistic use of the unknown encourages exploration during early stages of learning. In Section 8.5.4, we will discuss the motivation underlying this exploratory strategy. If we have no data on several representations, we use preference heuristics to select among them (Chapter 10); if there is no applicable heuristics, we make a random selection.

If we have past performance data for all representations, we first select a time bound for each representation, using the technique of Section 8.4.2, and determine the gain estimates and their standard deviations for the selected time bounds. Then, for every representation, we estimate the probability that it is the best among the available representations. Finally, we make a weighted random selection; the chance of choosing a representation is equal to the probability that it is the best. This probabilistic selection results in a frequent use of representations that perform well, but it also encourages some exploratory use of poor performers.

We now describe a technique for estimating the probability that a representation is the best. Suppose that we have k different representations. We select one of them, whose gain estimate is g and deviation of the estimate is σ, and compare it with the other available representations. Recall that we compute g from the sample of past performance data, for the selected time bound. We denote the expected problem-solving gain, for the selected bound, by G; we have defined this value in Equation 7.3. Then, g is an unbiased statistical estimate of G. Finally, we denote the gain estimates of the other representations by $g_1, ..., g_{k-1}$ and the corresponding deviations by $\sigma_1, ..., \sigma_{k-1}$.

First, suppose that we know the exact value of G; that is, we know the mean gain computed over the population of *all* possible problems (see Equation 7.3). The selected representation is the best if G is larger than the expected gains of the other available representations. We use the statistical z-test to determine the probability that G is the largest among the expected gains.

We begin by finding the probability that G is greater than the expected gain of another representation i, whose gain estimate is g_i and deviation is σ_i. The expected difference between the two gains is $G - g_i$, and the standard deviation of the difference is σ_i. The *z value* is the ratio of the expected difference to its standard deviation; that is, $z = \frac{G-g_i}{\sigma_i}$. The z-test converts this value into the probability that the expected gain for the selected representation is larger than that for representation i. Note, however, that the z-test uses the value of G, which cannot be found from the sample data. We denote the resulting probability by $p_i(G)$.

We next determine the probability $p(G)$ that G is larger than the expected gains of all other representations. If the gain estimates $g_1, ..., g_{k-1}$ are independent, the probabilities $p_1(G), ..., p_{k-1}(G)$ are also independent, and we compute $p(G)$ as their product:

$$p(G) = \prod_{i=1}^{k-1} p_i(G).$$

Since we cannot compute G from the available data, we need to use its estimate g in the statistical computation. The distribution of the possible values of G is approximately normal, with mean g and standard deviation σ; that is, its probability density function is as follows:

$$f(G) = \frac{e^{-(G-g)^2/(2\sigma^2)}}{\sigma \cdot \sqrt{2\pi}}.$$

To determine the probability p that the selected representation is the best, we integrate over possible values of G:

$$p = \int_{-\infty}^{\infty} p(G) \cdot f(G)\, d\,G = \int_{-\infty}^{\infty} \prod_{i=1}^{k-1} p_i(G) \cdot \frac{e^{-(G-g)^2/(2\sigma^2)}}{\sigma \cdot \sqrt{2\pi}}\, d\,G. \tag{8.7}$$

Note that we have used two simplifying assumptions to derive this expression. First, we have assumed that the sample means $g, g_1, ..., g_{k-1}$ are normally distributed. If we compute these values from small samples of past data, their distributions may not be normal. Second, we have considered the distributions of $g, g_1, ..., g_{k-1}$ to be independent. If we use incremental learning, then the choice of a representation for each problem depends on the representations used for the previous problems, and the data samples collected for different representations are not independent.

Even though Equation 8.7 is an approximation, it is satisfactory for the learning algorithm. We use the probability p only for our "occasional exploration" heuristic, which does not require high accuracy in determining the exploration frequency. We implemented the computation of p using numerical integration on the interval from $g - 4\sigma$ to $g + 4\sigma$, with step 0.1σ. That is, we approximate the integral by the following sum:

$$p_{\text{best}} = \sum_{j=-40}^{40} \prod_{i=1}^{k-1} p_i(g + 0.1j\sigma) \cdot \frac{e^{-(0.1j\sigma)^2/(2\sigma^2)}}{10 \cdot \sqrt{2\pi}}. \tag{8.8}$$

For example, suppose that we need to select among Apply, Delay, and Abstract based on the data in Table 8.1. We select bound 13.1 for Apply, which gives a gain estimate of 13.5 with a deviation of 3.3; bound 5.3 for Delay, with a gain estimate of 5.3 and its deviation

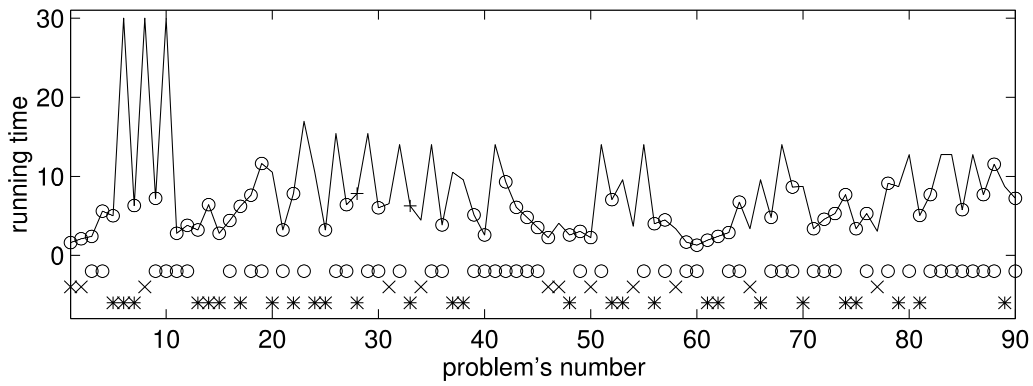


Figure 8.7: Results of incremental selection of a representation and time bound, on ninety transportation problems. The graph shows the running times (solid line), successes (o) and failures (+), and the selection made among Apply (o), Delay (x), and Abstract ($*$).

3.0; and bound 13.2 for Abstract, with a gain of 11.2 and deviation 3.2. We use Equation 8.8 to determine the probability that Apply is better than the other two representations, which gives us 0.68. Similarly, the probability that Delay is best is 0.01, and Abstract's chance of being best is 0.31. We now choose one of the representations randomly; the chance of choosing each representation equals its probability of being the best.

We show the results of using this strategy in the transportation domain, for the example gain function with the reward of 30.0, in Figure 8.7. In this experiment, we first use the thirty problems from Table 8.1 and then sixty additional transportation problems. The horizontal axis shows the number of a problem and the vertical axis is the running time. We mark successes by circles and failures by pluses. The rows of symbols below the curve show the choice of a representation: a circle for Apply, a cross for Delay, and an asterisk for Abstract.

The total gain is 998.3, which gives an average of 11.1 per problem. The overall time of the statistical computations is 0.78, or about 0.01 per problem. The selection converges to the use of Apply with the time bound 12.7, which is optimal for this set of ninety problems. If we used the final selection on all the problems, we would earn 13.3 per problem. Note that the convergence is slower than in the bound-selection experiments (see Figure 8.6), because we test each representation only on about a third of all problems.

8.5 Selection without past data

When the system faces a new problem domain, it has no past performance data. Thus, it needs to select the initial representation and time bound without performance analysis. It also has to decide when to switch from this initial selection to incremental learning.

We provide general heuristics for these decisions, and allow the user to overwrite them with domain-specific heuristics. We have *not* used statistics in developing the general heuristics; improving them is an open problem, which requires an analytical or empirical investigation of initial choices.

We discuss methods for determining initial time bounds (Section 8.5.1), describe a heuristic computation of these bounds (Section 8.5.2), and identify some of their drawbacks (Section 8.5.3). Then, we give heuristics for switching to incremental learning and for the initial choice of representation (Section 8.5.4).

8.5.1 Initial time bounds

We describe the user's options for specifying the computation of initial bounds, and present the system's default heuristic for this computation.

User-specified bounds

The user may provide Lisp procedures that determine initial time bounds, which take precedence over the default heuristic. She may specify a separate procedure for each domain and each gain function. The procedure inputs a representation and, optionally, a specific problem; it must output a positive real value. If the value does not depend on the representation and problem, the user may specify it by a number rather than a procedure.

Alternatively, the user may provide initial-bound procedures that are applicable to multiple domains and gain functions. These procedures input not only a representation and problem, but also a domain and gain function. For every such procedure, the user may specify a condition that limits its applicability. If several procedures are applicable for the same domain and gain, the system chooses one of them at random.

Default bounds

We next describe a general heuristic for computing initial time bounds, used in the absence of domain-specific heuristics.

Note that the incremental-learning procedure never considers time bounds that are larger than the initial bound (see Section 8.4.2); therefore, the initial choice should *overestimate* the optimal bound. On the other hand, an unnecessarily large initial bound may cause high losses at the beginning of learning (for example, see Figures 8.6 and 8.7).

The system chooses an initial bound B that satisfies the following condition, for all problems in the domain and all possible results; this condition almost always gives a larger-than-optimal bound:

$$gain(prob, B, result) \leq 0. \tag{8.9}$$

We now give an informal justification for this heuristic. Note that, if we get a negative gain for solving a problem, then we should have skipped the problem rather than solving it. On the other hand, a larger-than-B time bound would encourage the system to solve negative-gain problems. Thus, the optimal bound should be no larger than B.

If the payment for time is *linear*, we may provide additional intuitive support. For PRODIGY search engines, the probability of solving a problem, say, within the next second, usually declines with the passage of search time. If the system has not solved the problem within B seconds, chances are it will not find a solution after running for B more seconds. On the other hand, possible rewards for solving the problem are smaller than the cost of a B-second run. Thus, if the system has already reached the bound B, it is time to interrupt the search.

We enforce a global upper limit on all time bounds, which is currently set to 1800.0 seconds, that is, thirty minutes. The system never goes over this limit, even if Inequality 8.9 requires a larger bound.

The reason for this particular limit is the memory requirements of PRODIGY. We have run the system on a Sun 5 computer with 256-megabyte memory, which can support thirty to forty minutes of search. If the search engine runs longer, it exhausts the available memory and causes and error termination.

Note, however, that the use of some global limit would be essential even if the system did not cause memory problems. The purpose of this limit is to prevent impractically large initial bounds, which may lead to huge losses.

8.5.2 Computation of initial bounds

We describe two techniques for computing initial time bounds, based on Inequality 8.9. The first technique is for linear gain functions, which satisfy Constraints 8 and 9 of Section 7.3.2. The second technique is for gains defined through solution quality.

In practice, most gain functions fall into one of these two categories. If a function does *not* belong to either category, then the system uses the maximal allowed bound, 1800.0 seconds, which may result in significant initial losses.

Linear gain function

If problem-solving gain satisfies Constraints 8 and 9, then the user may specify it by two functions: the payment for a unit time, $pay(prob)$, and the reward for solving the problem, $R(prob, result)$. The system computes the gain as follows:

$$gain(prob, time, result) = -time \cdot pay(prob) + R(prob, result).$$

For this specification of a gain function, $\mathcal{S}$HAPER computes the initial bound B through the minimum of pay and maximum of R, taken over all problems:

$$B = \frac{\max R(prob, result)}{\min pay(prob)}.$$

The resulting bound satisfies Inequality 8.9 for all problems.

If pay and R are not constants, the user should provide some means for determining the minimum of pay and the maximum of R. If she does not, the system computes pay for several problems and uses the minimum of the resulting values as an approximation of the overall minimum. Similarly, it finds the R values for several problems and uses their maximum.

Gain defined through quality

If gain satisfies Constraint 7 of Section 7.3.2, then the user may specify a solution-quality function, $quality(prob, result)$, and express gain through quality, $gain_q(prob, time, quality)$. Given this specification, the system can compute a time bound that satisfies Inequality 8.9 for a specific given problem, though *not* for all problems.

The computation requires the use of an upper bound on solution quality for a given problem; we denote this bound by *qual-bd*(*prob*). The user should provide a means for finding such a bound. In practice, most quality functions have a readily computable bound. For example, if we define the quality through the total cost of operators in the plan (see Section 7.3.2), then we compute its upper bound as the quality of a zero-cost solution.

Observe that the function $gain_{\mathrm{q}}(prob, time, qual\text{-}bd(prob))$ is an upper bound on *prob*'s gain, for each specific running time. Therefore, if a time bound B satisfies the following equation, then it also satisfies Inequality 8.9:

$$gain_{\mathrm{q}}(prob, B, qual\text{-}bd(prob)) = 0.$$

The function $gain_{\mathrm{q}}$ is a decreasing function of *time* (see Constraint 1 in Section 7.3.1), which allows us to solve the equation numerically, by a binary search. The system searches for B between 0 and the 1800.0-second limit. If the solution to the equation is greater than 1800.0, the system chooses the 1800.0 bound.

$\mathcal{S}$HAPER computes an initial bound by finding bounds for several available problems and taking their maximum. That is, it uses the maximum over a sample of problems instead of the overall maximum. We specify the sample size by a knob variable, which is currently set to 10.

8.5.3 Drawbacks of the default bounds

The described heuristic usually gives reasonable initial bounds; however, it has several drawbacks, which sometimes result in generating inappropriate bounds. We discuss some situations where the heuristic gives undesirable results.

Too large bounds

If the gain function does not satisfy Constraint 5, then the gain of solving some problems may be positive for arbitrarily large running time, which means that Inequality 8.9 has no solution and the heuristic suggests unbounded search. The system then enforces the 1800.0-second bound, but it often proves unnecessarily large.

If Constraint 5 holds, the inequality has a solution for every problem; however, the bound that satisfies it for *all* problems may be impractically large because of outliers. If the domain has infinitely many problems, their common bound may be infinite. The system avoids such situations by finding a bound that satisfies a sample of problems rather than all problems.

Too small bounds

Occasionally, the heuristic underestimates the optimal bound. We illustrate this possibility with an artificial example, using the linear gain function defined in Section 8.1.2, with the reward of 2.5. If the system solves a problem, the gain is $(2.5 - time)$; otherwise, the negative gain is $(-time)$. We show this function in Figure 8.8(a).

Suppose that the probability of solving a randomly selected problem in exactly 1.0 second is 1/2, the chance of solving it in 2.0 seconds is 1/4, the chance that it takes 3.0 seconds

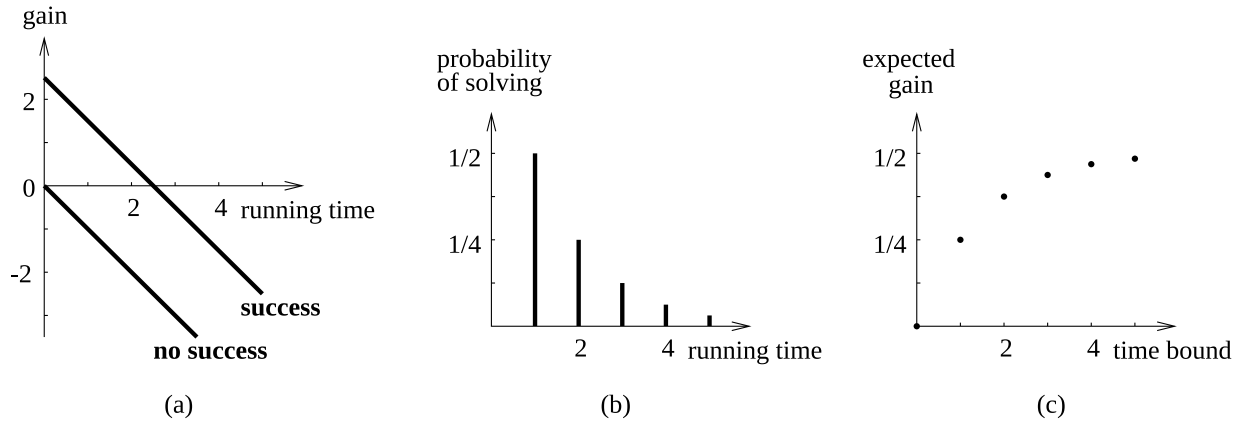


Figure 8.8: Problem distribution that causes underestimation of the optimal bound.

is 1/8, and so on (see Figure 8.8b). If we set the time bound to n seconds, the expected gain is $\frac{1}{2} \cdot (1 - \frac{1}{2^n})$, as shown in Figure 8.8(c). Thus, the expected gain monotonically grows with time bound, which means that the system should *not* bound the search; however, the heuristic would impose a bound of 2.5 seconds.

In Section 8.6.1, we will give a more practical example of underestimating the optimal bound, in the PRODIGY Logistics Domain, where the initial bound proves smaller than optimal by a factor of 1.08. We could avoid most underestimations if we increased the heuristic bounds by some pre-set constant factor; however, it would also increase the losses due to unnecessarily large initial bounds.

Nonlinear gain

The initial-bound heuristic does not account for the shape of nonlinear gain functions and may give misleading results for such functions. For example, consider the nonlinear function in Figure 8.9(a). According to this function, the payment for a unit time becomes smaller after 4.0 seconds of execution:

$$gain = \begin{cases} 4.0 - time, & \text{if success and } time \leq 4.0 \\ -time, & \text{if interrupt and } time \leq 4.0 \\ 4.0 - (\frac{time}{4} + 3.0), & \text{if success and } time > 4.0 \\ -(\frac{time}{4} + 3.0), & \text{if interrupt and } time > 4.0 \end{cases}$$

Suppose that the system solves all problems, the search always takes a natural number of seconds, and it is never longer than eight seconds. If all eight running times are equally likely (see Figure 8.9b), then the dependency of the expected gain on bound is as shown in Figure 8.9(c), and the optimal bound is 8.0 seconds (or more).

The heuristic, however, gives an initial bound of 4.0 seconds, because it does not account for the reduction in the cost of a unit time. Intuitively, if the system has run for 4.0 seconds, it should continue execution, to take advantage of the cost reduction. Developing a heuristic that identifies such situations is an open problem.

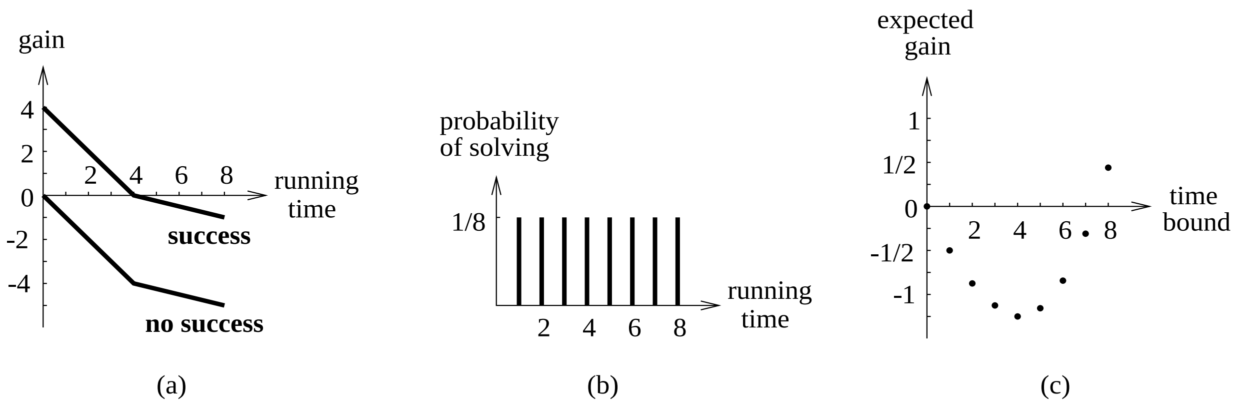


Figure 8.9: Nonlinear gain function that causes underestimation of the optimal bound.

8.5.4 Other initial decisions

We now discuss two other decisions that must be made at the beginning of the statistical exploration. First, the system has to determine when to switch from the initial time bound to incremental learning of the bound. Second, it has to select the order of exploring new representations.

Switching to incremental learning

When a representation has no past data, the system uses it with the initial time bound until successfully solving two problems, and then switches to incremental learning. If the representation gives only interrupts and failures, the system gives up on it after a pre-set number of trials. By default, the system gives up after seven trials without any success or after fifteen trials with only one success.

The user has an option to overwrite these knob values. She may indicate that the system must switch to learning after a different number of successes, and that it must give up after a different number of unsuccessful trials. She may either specify these numbers as constants or provide Lisp procedures that compute them for each given domain, representation, and gain function.

Alternatively, the user may provide a boolean Lisp function that determines when to start learning. The system invokes this function before solving each new problem, until it signals the beginning of learning. Similarly, the user may specify a Lisp function that signals when to give up on a poorly performing representation.

Note that the role of this give-up function is different from pruning rules. The system applies pruning rules to discard some of the newly generated representations, *before* using them in problem solving (see Sections 7.2.2 and 7.2.3). On the other hand, the give-up function identifies representations that have proved inefficient *during* problem solving.

Exploring new representations

The system accumulates initial data for all available representations before switching to incremental learning. If the data for some representation are not sufficient for statistical selection of a time bound, the system chooses this unexplored representation for solving the next problem. If several representations have insufficient data, the system chooses among them at random.

This optimistic use of the unknown encourages exploration during early stages of learning. If the number of problems in a domain is significantly larger than the number of representations, then early identification of effective representations justifies the cost of the exploration.

On the other hand, if we need to solve only a few problems, the exploration may prove too costly. A better strategy is to try only the most promising representations and disregard the others. The default exploration procedure does not consider this strategy; however, the user may limit the exploration by providing appropriate rejection and comparison rules, which prune undesirable representations (see Sections 7.2.2 and 7.2.3). She may also provide preference rules, described in Chapter 10, which control the exploration. For example, they may enforce a specific exploration order or delay experimenting with unpromising representations.

8.6 Empirical examples

We have shown the effectiveness of statistical selection in a simple transportation domain. We now give results of using this technique in two other domains. First, we describe experiments with an extension of the transportation domain (Section 8.6.1). Second, we determine how long one should wait on the phone, before hanging up, when there is no answer (Section 8.6.2).

8.6.1 Extended transportation domain

We consider transportation problems that require the use of airplanes and vans [Veloso, 1994]. We use planes for transporting packages between cities, and vans for local delivery within cities. The relative performance of Apply, Delay, and Abstract on such problems differs from their performance in the simple domain used in Sections 8.1–8.4. In Table 8.3, we give the results of applying each of the three representations to thirty problems.

We use the example gain function with a reward $R = 400.0$; that is, the gain of solving a problem is $(400 - time)$, and the negative gain of a failure or interrupt is $(-time)$. We present the results of incremental learning of a time bound in Figure 8.10. The Apply learning gives a gain of 110.1 per problem and eventually selects a bound of 127.5. The optimal bound for this set of problems is 97.0. If we used the optimal bound for all problems, we would earn 135.4 per problem.

Delay earns 131.1 per problem and chooses a bound of 105.3 at the end of the learning process. The actual optimal bound for Delay is 98.4, the use of which on all problems would give a per-problem gain of 153.5. Finally, Abstract earns 243.5 per problem and chooses a bound of 127.6. The optimal bound for Abstract is 430.8, which would give a per-problem gain of 255.8.

Even though the bound learned for Abstract is much smaller than optimal (127.6 versus 430.8), the resulting gain is close to optimal. The reason is that, in this experiment, the dependency of expected gain on time bound has a long plateau, and the choice of a bound within the plateau does not make much difference.

#	time (sec) and outcome			# of	#	time (sec) and outcome			# of
	Apply	Delay	Abstract	packs		Apply	Delay	Abstract	packs
1	4.7 s	4.7 s	4.7 s	1	16	35.1 s	21.1 s	6.6 f	2
2	96.0 s	9.6 f	7.6 f	2	17	60.5 s	75.0 f	13.7 s	2
3	5.2 s	5.1 s	5.2 s	1	18	3.5 s	3.4 s	3.5 s	1
4	20.8 s	10.6 f	14.1 s	2	19	4.0 s	3.8 s	4.0 s	1
5	154.3 s	31.4 s	7.5 f	2	20	232.1 s	97.0 s	9.5 f	2
6	2.5 s	2.5 s	2.5 s	1	21	60.1 s	73.9 s	14.6 s	2
7	4.0 s	2.9 s	3.0 s	1	22	500.0 b	500.0 b	12.7 f	2
8	18.0 s	19.8 s	4.2 s	2	23	53.1 s	74.8 s	15.6 s	2
9	19.5 s	26.8 s	4.8 s	2	24	500.0 b	500.0 b	38.0 s	4
10	123.8 s	500.0 b	85.9 s	3	25	500.0 b	213.5 s	99.2 s	4
11	238.9 s	96.8 s	76.6 s	3	26	327.6 s	179.0 s	121.4 s	6
12	500.0 b	500.0 b	7.6 f	4	27	97.0 s	54.9 s	12.8 s	6
13	345.9 s	500.0 b	58.4 s	4	28	500.0 b	500.0 b	16.4 f	8
14	458.9 s	98.4 s	114.4 s	8	29	500.0 b	500.0 b	430.8 s	16
15	500.0 b	500.0 b	115.6 s	8	30	500.0 b	398.7 s	214.8 s	8

Table 8.3: Performance in the extended transportation domain.

Note that Abstract's optimal bound is larger than the initial bound (430.8 versus 400.0). Since the learning algorithm never sets a bound higher than the initial one, the use of this initial bound prunes the optimal bound from consideration. This experiment illustrates the imperfection of the heuristic for selecting an initial time bound.

We show the results of incremental selection of a representation in Figure 8.11. In this experiment, we first use the thirty problems from Table 8.3 and then sixty additional problems. The representation converges to the choice of Abstract with a time bound of 300.6, and gives a gain of 207.0 per problem. The best choice for this set of problems is the use of Abstract with a time bound of 517.1, which would give a per-problem gain of 255.8. We identified this optimal choice in a separate experiment, by running every representation on all ninety problems.

8.6.2 Phone-call domain

We next illustrate the use of the statistical technique in a very different domain. We apply it to select a time bound when calling a friend on the phone. The algorithm determines how many seconds (or rings) one should wait for an answer before handing up.

In Table 8.4, we give the time measurements on sixty phone calls, rounded to 0.05 seconds. We made these calls to sixty different people at their home numbers. We measured the time from the beginning of the first ring, skipping the static silence of the connection delays. A success occurred when our party answered the phone. A reply by an answering machine was considered failure.

We first consider the gain function that gives $(R - time)$ for a success and $(-time)$ for a failure or interrupt. We thus assume that the caller is not interested in leaving a message, which means that a reply by a machine gets a reward of zero. The reward R for reaching

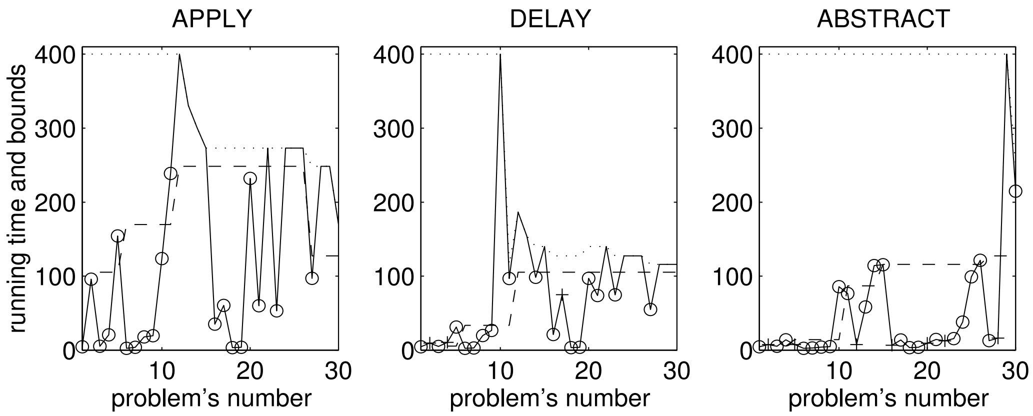


Figure 8.10: Incremental learning of time bounds in the extended transportation domain: running times (solid lines), time bounds (dotted lines), and maximal-gain time bounds (dashed lines). The successes are marked by circles and the failures by pluses.

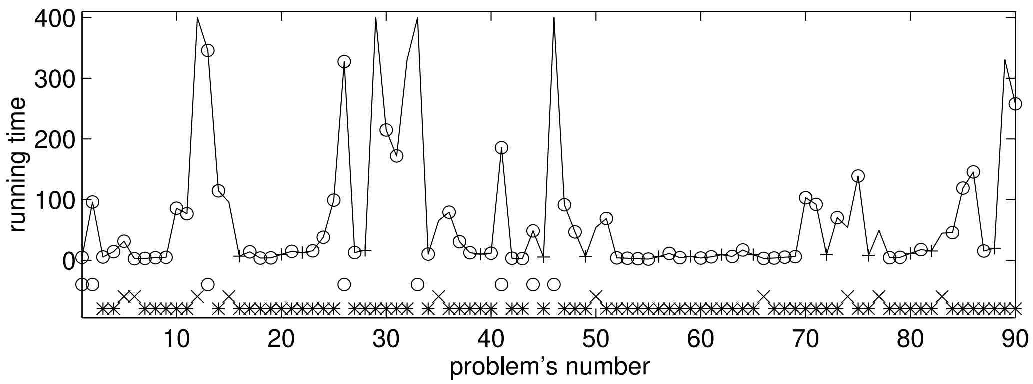


Figure 8.11: Selection of a representation in the extended transportation domain: the running times (solid line), successes (o) and failures (+), and the selection made among Apply (o), Delay (x), and Abstract (∗).

the other party may be determined by the time that the caller is willing to wait in order to talk now, as opposed to hanging up and calling again later.

In Figure 8.12(a), we show the dependency of the expected gain on time bound, for the rewards of 30.0 (dash-and-dot line), 90.0 (dashed line), and 300.0 (solid line). The optimal bound for the 30.0 and 90.0 rewards is 14.7 (three rings); the optimal bound for the 300.0 reward is 25.5 (five rings).

If the caller plans to leave a message, then the failure reward is not zero, though it may be smaller than the success reward. We denote the failure reward by R_f and define the gain as follows:

$$gain = \begin{cases} R - time, & \text{if success} \\ R_f - time, & \text{if failure} \\ -time, & \text{if interrupt} \end{cases}$$

In Figure 8.12(b), we show the expected gain for the success reward of 90.0 with three different failure rewards, 10.0 (dash-and-dot line), 30.0 (dashed line), and 90.0 (solid line).

#	time	#	time	#	time	#	time	#	time
1	5.80 f	13	11.45 f	25	11.30 f	37	26.70 f	49	10.05 s
2	8.25 s	14	3.70 s	26	10.20 f	38	6.20 s	50	6.50 s
3	200.00 b	15	7.25 s	27	4.15 s	39	24.45 f	51	15.10 f
4	5.15 s	16	4.10 s	28	14.70 s	40	29.30 f	52	25.45 s
5	8.30 s	17	8.25 s	29	2.50 s	41	12.60 s	53	20.00 f
6	200.00 b	18	5.40 s	30	8.70 s	42	26.15 f	54	24.20 f
7	9.15 s	19	4.50 s	31	6.45 s	43	7.20 s	55	20.15 f
8	6.10 f	20	32.85 f	32	6.80 s	44	16.20 f	56	10.90 s
9	14.15 f	21	200.00 b	33	8.10 s	45	8.90 s	57	23.25 f
10	200.00 b	22	200.00 b	34	13.40 s	46	4.25 s	58	4.40 s
11	9.75 s	23	10.50 s	35	5.40 s	47	7.30 s	59	3.20 f
12	3.90 s	24	14.45 f	36	2.20 s	48	10.95 s	60	200.00 b

Table 8.4: Waiting times (seconds) in sixty phone-call experiments.

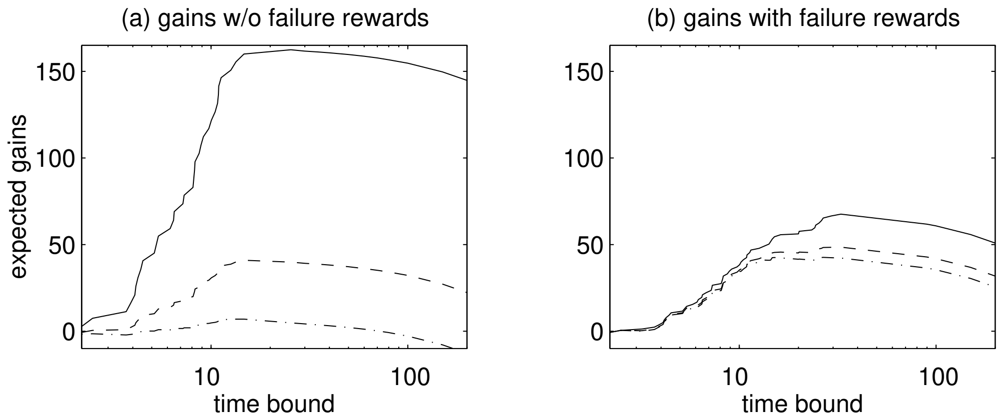


Figure 8.12: The dependency of the expected gain on the time bound in the phone-call domain: (a) for the rewards of 30.0 (dash-and-dot line), 90.0 (dashed line), and 300.0 (solid line); (b) for the success reward of 90.0 and failure rewards of 10.0 (dash-and-dot line), 30.0 (dashed line), and 90.0 (solid line).

The optimal bound for the 10.0 failure reward is 26.7 (five rings); for the other two rewards, it is 32.9 (six rings).

In Figure 8.13, we show the results of selecting a time bound incrementally, for the 90.0 success reward and zero failure reward. The learned time bound converges to the optimal bound, 14.7. The average gain obtained during the learning process is 38.9 per call. If we used the optimal bound for all calls, we would earn 41.0 per call.

The experiments in the transportation domain and phone-call domain show that the learning algorithm usually finds a near-optimal time bound after solving twenty or thirty problems, and that the gain obtained during learning is close to optimal.

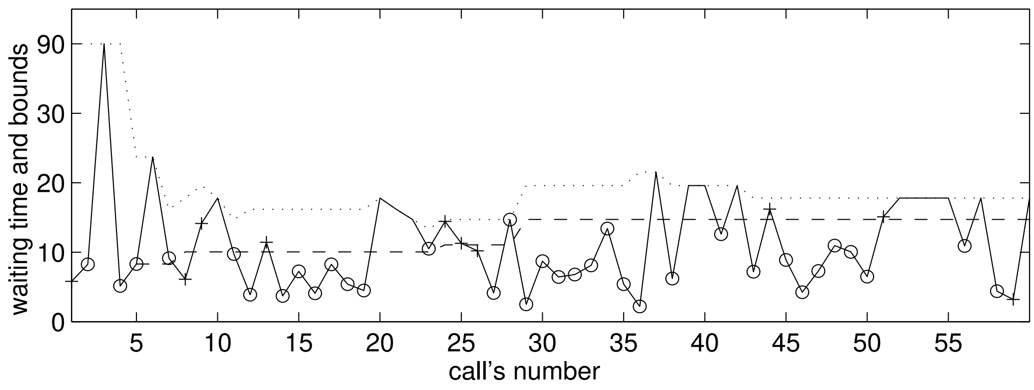


Figure 8.13: Incremental learning of a time bound in the phone-call domain.

8.7 Artificial tests

We now present a series of experiments with artificially generated values of success and failure times. The "running times" in these experiments are the values produced by a random-number generator. The artificial data enable us to perform controlled experiments with known distributions.

The learning mechanism has proved effective for all tested distributions. The experiments have demonstrated that the gain obtained during incremental learning is usually close to optimal. We have not found a significant difference in performance for different distributions.

We use the linear gain function with a reward $R = 100.0$ in all experiments; that is, the success gain is $(100.0 - \mathit{time})$ and the negative gain of a failure or interrupt is $(-\mathit{time})$. We consider the following four distributions of success and failure times:

Normal: The normal distribution of success and failure times corresponds to the situation where the running time for most problems is close to some "typical" value, and problems with much smaller or much larger times are rare.

Log-Normal: The distribution of times is called log-normal if time logarithms are distributed normally. Intuitively, this distribution occurs when the "complexity" of most problems is close to some typical complexity and the problem-solving time grows exponentially with complexity.

Uniform: The times are distributed uniformly if they belong to some fixed interval and all values in this interval are equally likely; thus, there is no "typical" running-time value.

Log-Uniform: The logarithms of running times are distributed uniformly. Intuitively, the complexity of problems is within some fixed interval, and running time is exponential in complexity.

For each of the four distribution types, we ran multiple tests, varying the values of the following parameters:

Success and failure probabilities: We varied the probabilities of success, failure, and infinite looping.

Mean and deviation: We experimented with different values of the mean and standard deviation of success-time and failure-time distributions.

Length of the problem sequence: We tested the incremental-learning mechanism on sequences of 50, 150, and 500 problems.

We ran fifty independent experiments for each setting of the parameters and averaged their results. Thus, every graph with artificial-test results (Figures 8.14–8.19) shows the average of fifty experiments.

Since the learning technique has proved effective in all these tests, as well as in selecting among representations in the SHAPER system, we conjecture that it also works well for most other distributions. We plan to experiment with a wider variety of distributions and identify situations in which the technique does not give good results, as a part of future work.

Experiments with short and long problem sequences

We present the results of learning a time bound on sequences of fifty and five hundred problems. The success probability in these experiments is 1/2, the failure probability is 1/4, and the probability of infinite looping is also 1/4. The mean of success times is 20.0 and their standard deviation is 8.0; the failure-time mean is 10.0 and the standard deviation is 4.0. We experimented with all four distribution types. For each distribution, we ran fifty experiments and averaged their results.

In Figure 8.14, we summarize the results for fifty-problem sequences. The horizontal axes in all graphs show the number of a problem in a sequence. In the top row of graphs, we give the average per-problem gain obtained up to the current problem. The circles mark the gain that the system would obtain if it knew the distribution in advance and used the optimal time bound for all problems. The vertical bars show the width of the distribution of gain values obtained in different experiments. Each bar covers two standard deviations up and down, which means that 95% of the experiments fall within it.

In the middle row of graphs, we show the selected time bounds. In the bottom row, we give the system's estimates of the optimal time bound (recall that the selected bounds are larger than optimal, to encourage exploration). The crosses mark the values of the optimal time bounds. Note that the system's estimates of the optimal bounds converge to their real values.

In Figure 8.15, we present similar results for 500-problem sequences. In these experiments, per-problem gains come closer to the optimal values, but still do not reach them. The difference between the obtained and optimal gains comes from losses during early stages of learning and from the use of larger-than-optimal bounds.

Varying success and failure probabilities

We give the results of learning a time bound for different probabilities of success and failure. The means and standard deviations of the success and failure times are the same as in the previous experiments.

We summarize the results in Figure 8.16. The top row of graphs is for a solver that succeeds, fails, and goes into an infinite loop equally often; that is, the probability of each

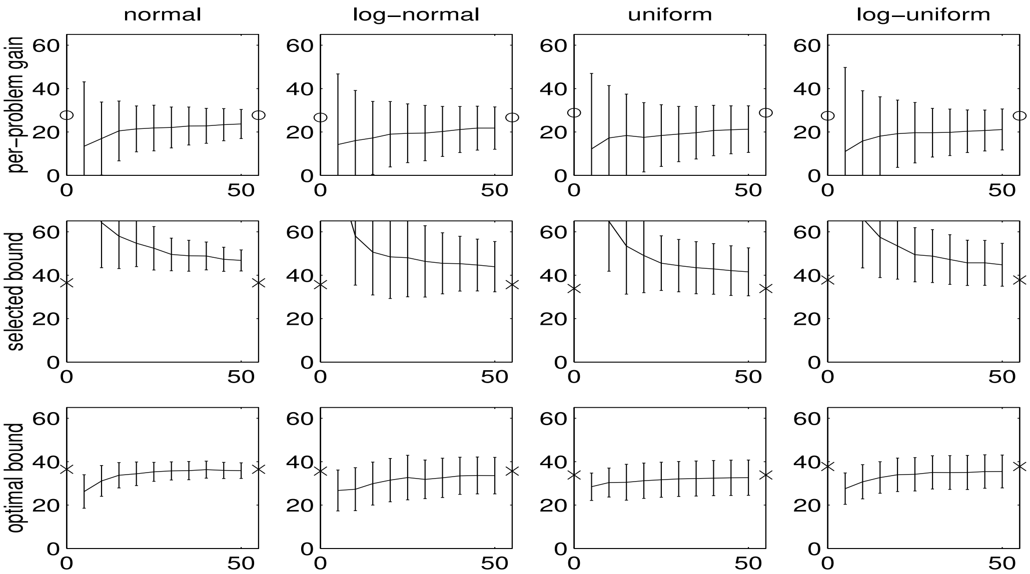


Figure 8.14: Per-problem gains (top row), time bounds (middle row), and estimates of the optimal time bounds (bottom row) for incremental learning on fifty-problem sequences. The crosses mark the optimal time bounds and the circles show the expected gains for the optimal bounds.

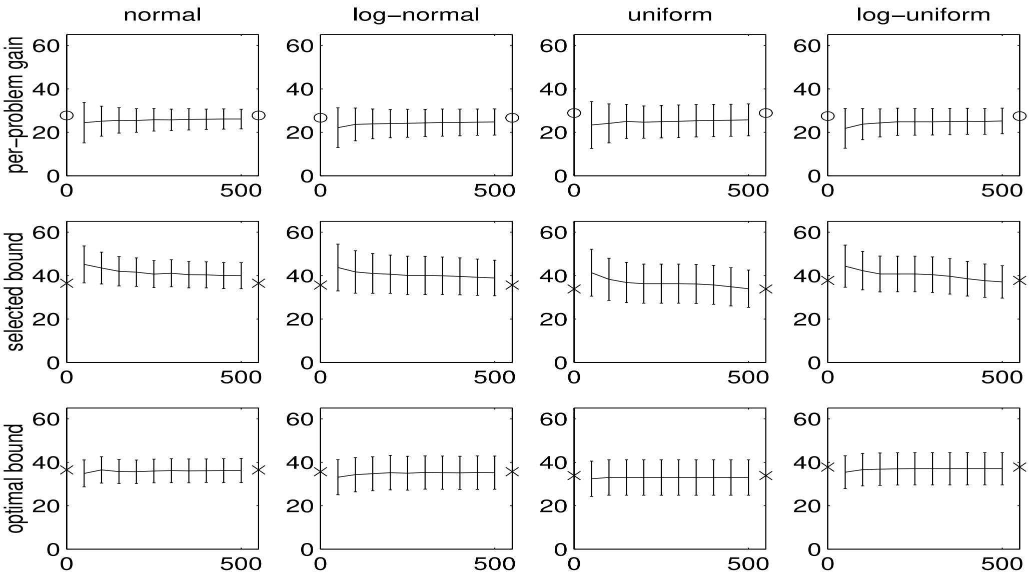


Figure 8.15: Per-problem gains (top row), time bounds (middle row), and estimates of the optimal time bounds (bottom row) for incremental learning on 500-problem sequences.

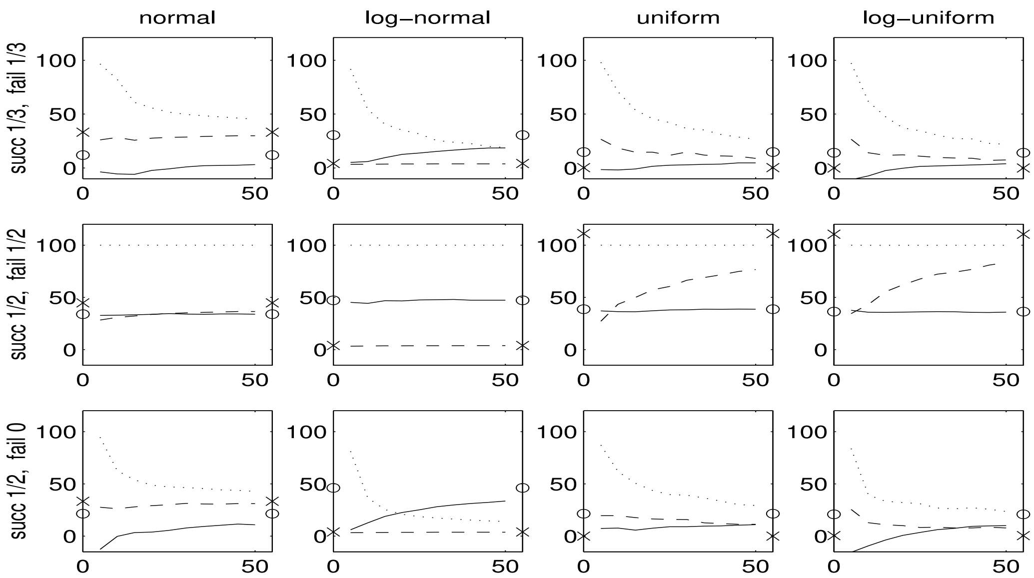


Figure 8.16: Per-problem gains (solid lines), time bounds (dotted lines), and estimates of the optimal time bounds (dashed lines) for different success and failure probabilities. The crosses mark the optimal time bounds and the circles show the expected gains for the optimal bounds. We give the values of success probability ("succ") and failure probability ("fail") to the left of each row.

outcome is 1/3. In the middle row of graphs, we give the results for a solver that succeeds half of the time and fails half of the time, and never goes into an infinite loop. Finally, the bottom row is for a solver that succeeds half of the time and loops forever otherwise.

The solid lines show the average per-problem gain up to the current problem; the dotted lines are the selected time bounds; and the dashed lines are the estimates of the optimal bound. The crosses mark the optimal time bounds, and the circles are the expected gains for the optimal bounds.

Note that, when the probability of infinite looping is zero (the middle row), any large time bound gives near-optimal results, because we never need to interrupt a solver. Thus, the system never changes the initial time bound and gets near-optimal gains from the very beginning.

Varying the means of time distributions

We now vary the mean value of failure times. We keep the mean success time equal to 20.0 (with standard deviation 8.0), and experiment with failure means of 10.0 (with deviation 4.0), 20.0 (with deviation 8.0), and 40.0 (with deviation 16.0). We give the results in Figure 8.17.

The gains for normal and log-normal distributions come closer to the optimal values than the gains for uniform and log-uniform distributions. This observation suggests that the learning technique works better for the first two distributions. The difference, however, is not statistically significant.

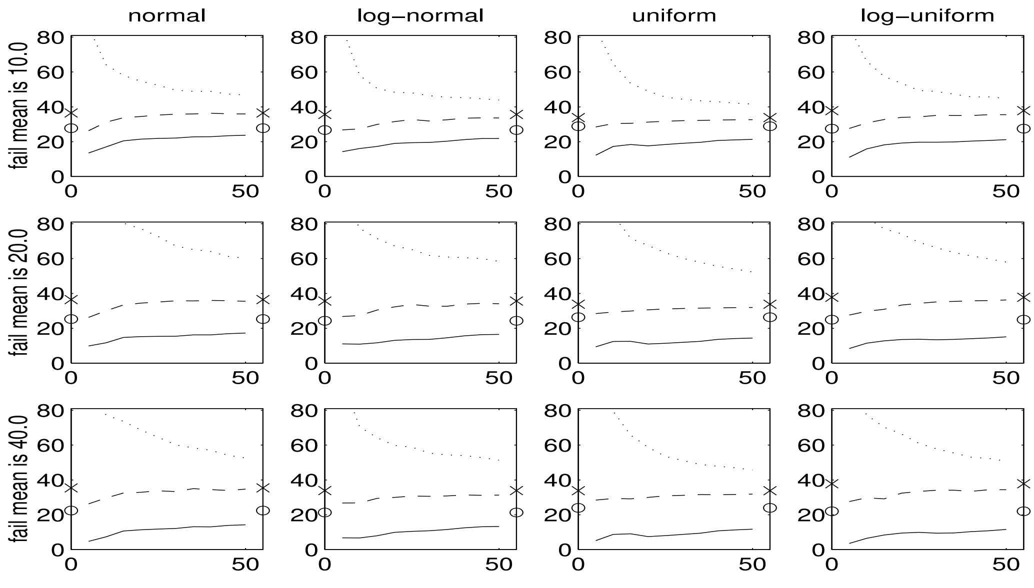


Figure 8.17: Per-problem gains (solid lines), time bounds (dotted lines), and estimates of the optimal time bounds (dashed lines) for different mean values of failure times. The mean of success times is 20.0 in all experiments.

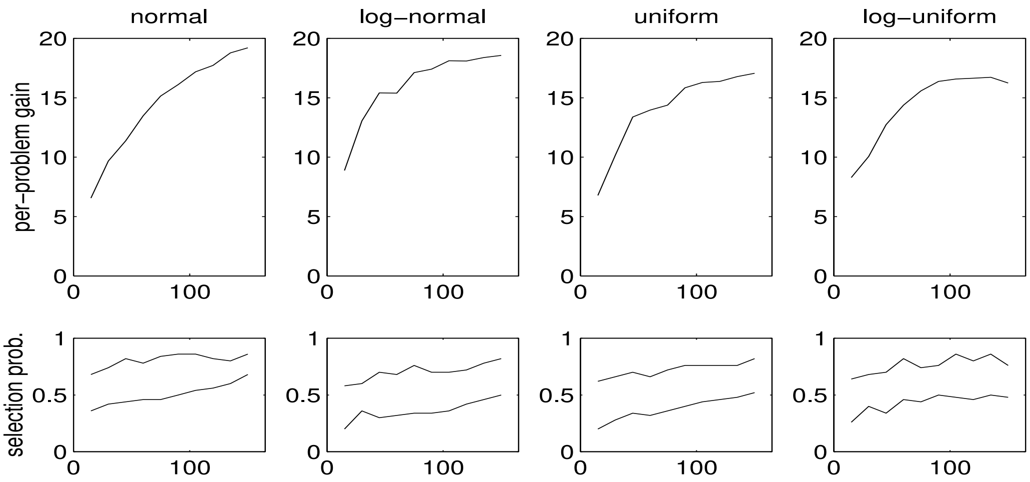


Figure 8.18: Incremental selection among three representations, where the average gain for the first representation is 10% larger than that for the second one and 20% larger than that for the third one. We show the average per-problem gains (the top row of graphs) and the probability of selecting each representation (the bottom row).

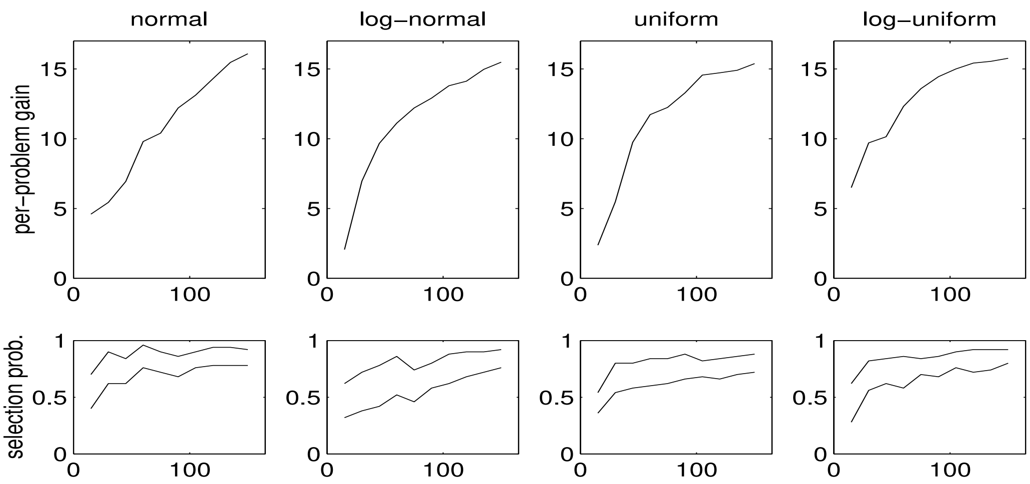


Figure 8.19: Incremental selection among three representations, where the average gain for the first representation is 30% larger than that for the second one and 60% larger than that for the third one.

Selection of a representation

Finally, we show the results of incremental selection among three representations, on 150-problem sequences. In the first series of experiments, we adjusted mean success and failure times in such a way that the optimal per-problem gain for the first representation was 10% larger than that for the second representation and 20% larger than that for the third representation.

We give the results in Figure 8.18. In the top row of graphs, we show the average per-problem gain. In the bottom row, we give the probability of choosing each representation, for the experiments without and with problem sizes. The distance from the bottom of the graph to the lower curve is the probability of selecting the first representation, the distance between the two curves is the chance of selecting the second representation, and the distance from the upper curve to the top is the third representation's chance. The graphs show that the probability of selecting the first representation (which gives the highest gain) increases during the learning process. The probability of selecting the third (worst-performing) representation decreases faster than that of the second representation.

In the second series of experiments, the optimal gain of the first representation was 30% larger than that of the second representation and 60% larger than that of the third representation. We give the results in Figure 8.19. Note that the probability of selecting the first representation grows much faster, due to the larger difference in the expected gains.

Chapter 9

Extensions to the statistical technique

We have considered the task of selecting a representation and time bound that will work well for most problems in a domain, and developed a statistical technique for this task. If we have additional information about a given problem, we can adjust the selection to this problem. We utilize problem-specific information in estimating expected gains, and then use the resulting estimates to choose a representation and time bound.

First, we describe the construction and use of problem-specific gain functions (Section 9.1). Second, we take into account estimated problem complexity (Section 9.2) and similarity among problems (Section 9.3).

9.1 Problem-specific gain functions

We have assumed in the statistical analysis that the user specifies a real-valued gain function, $gain(prob, time, result)$, defined for *all* problems, running-time values, and problem-solving results. We have used it in deriving statistical estimates in Sections 8.2 and 8.3. If this function satisfies certain conditions, we may construct a problem-specific gain function and use it to compute gain estimates for the given problem.

We first illustrate the use of this technique for the example gain function. We then generalize the developed technique and state conditions for its applicability (Section 9.1.2).

9.1.1 Example of problem-specific estimates

We explain the use of problem-specific estimates for the linear gain function defined in Section 8.1.2. If the system successfully solves a problem, the gain is $(R - time)$; if it fails or hits the time bound, the negative gain is $(-time)$. We assume, for simplicity, that the system never rejects a problem.

Suppose that some problems are more important than others, and the user specifies different reward values for different problems. In other words, the user provides a function $R(prob)$, which maps problems into their rewards.

Assume that a representation solved ns problems, failed on nf problems, and gave an interrupt on m problems. The success times were $ts_1, ..., ts_{ns}$, with the corresponding rewards

$R_1, ..., R_{ns}$, the failure times were $tf_1, ..., tf_{nf}$, and the interrupt times were $b_1, ..., b_m$. Note that we disregard the rewards of the unsolved problems.

We can use Equation 8.3 to estimate the expected gain for a specific time bound B. We denote the number of success times that are no larger than B by cs, the number of failure times no larger than B by cf, and the number of interrupts no larger than B by d. We distribute the weights of $b_1, ..., b_d$ among the larger-time outcomes (see Section 8.3), and denote the weights of $ts_1, ..., ts_{ns}$ by $us_1, ..., us_{ns}$, the weights of $tf_1, ..., tf_{nf}$ by $uf_1, ..., uf_{nf}$, and the weight of all other times by u_*. We substitute these values into Equation 8.3 and get the following estimate of the expected gain:

$$\frac{\sum_{i=1}^{cs} us_i \cdot (R_i - ts_i) + \sum_{i=1}^{cf} uf_i \cdot (-tf_i) + u_* \cdot (ns + nf + m - cs - cf - d) \cdot (-B)}{ns + nf + m}. \tag{9.1}$$

Next suppose that we fix some reward R_* and use it for *all* problems. That is, the gain of solving any problem is now $(R_* - time)$. If we use Equation 8.3 to find a gain estimate for this fixed reward, we get the following expression:

$$\frac{\sum_{i=1}^{cs} us_i \cdot (R_* - ts_i) + \sum_{i=1}^{cf} uf_i \cdot (-tf_i) + u_* \cdot (ns + nf + m - cs - cf - d) \cdot (-B)}{ns + nf + m}. \tag{9.2}$$

Now assume that the rewards change from problem to problem and we need to compute the gain estimate for a specific problem, whose reward is R_*. If we use Equation 9.1, we get an estimate for a randomly selected problem. On the other hand, Equation 9.2 incorporates the problem-specific knowledge of the reward value, and thus it gives a more accurate estimate for the given problem.

We use Equation 9.2 to determine expected gains for specific problems; however, we can do it only if the reward function $R(prob)$ does *not* correlate with the problem complexity. We formalize this condition in terms of the correlation between rewards and running times.

We define $time(prob)$ as a (finite or infinite) running time on $prob$ without a time bound. The outcome of problem solving without a bound may be a success, failure, or infinite run. We divide the problems into three groups, according to the outcomes, and use this division to state applicability conditions:

Condition 9.1 *We use Equation 9.2 if the following two conditions hold:*

1. *The reward $R(prob)$ does not correlate with $time(prob)$.*
2. *The mean reward of the success-outcome problems is equal to the mean reward of the failure-outcome problems and to that of the infinite-run problems.*

We always assume that these two conditions are satisfied, unless we have opposite information. We avoid the problem-specific computation only when we can show, with statistical significance, that one of the conditions does not hold.

If Condition 9.1 is not satisfied, Equation 9.2 may give misleading results, which worsen the system's performance. For example, suppose that the system successfully solves all problems and the reward perfectly correlates with the running time, specifically, $R(prob) =$

$0.9 \cdot \mathit{time}(\mathit{prob})$. Then, for every problem *prob*, the gain of solving it is negative, $-0.1 \cdot \mathit{time}(\mathit{prob})$.

If we apply the problem-independent gain computation (Equation 9.1) to past performance data, it gives negative gain estimates for all nonzero time bounds. The system determines that the best course is to avoid problem solving, which prevents further losses.

On the other hand, if we apply the problem-specific computation, we average past running times and disregard past rewards (see Equation 9.2). We then get high gain estimates for problems with large rewards, and negative estimates for problems with small rewards. These misleading estimates cause the system to solve problems with larger-than-average rewards, which leads to additional losses.

9.1.2 General case

We now extend the problem-specific evaluation to more general gain functions. The extension is based on the use of the solution-quality function, described in Section 7.3.2.

We have derived the problem-specific estimate of Section 9.1.1 by substituting a fixed problem into the gain function. To compute an estimate for a given problem prob_*, we replace the general gain expression $(R(\mathit{prob}) - \mathit{time})$ by the problem-specific function $(R(\mathit{prob}_*) - \mathit{time})$.

We cannot use the substitution for a general gain function, $\mathit{gain}(\mathit{prob}, \mathit{time}, \mathit{result})$, because the gain depends not only on the problem but also on the specific solution to the problem. The function $\mathit{gain}(\mathit{prob}_*, \mathit{time}, \mathit{result})$ is defined *only* for the solutions of prob_*, and does not allow the computation of the gains for the problems in a data sample.

To construct a problem-specific function, we assume that $\mathit{gain}(\mathit{prob}, \mathit{time}, \mathit{result})$ satisfies Constraint 7 of Section 7.3.2. Then, we can define a measure of solution quality, $\mathit{quality}(\mathit{prob}, \mathit{result})$, and express the gain in terms of quality, $\mathit{gain}_{\mathrm{q}}(\mathit{prob}, \mathit{time}, \mathit{quality})$. We obtain a problem-specific function by substituting prob_* into $\mathit{gain}_{\mathrm{q}}$, and use it to derive a problem-specific version of Equation 8.3, which gives us the following gain estimate:

$$\frac{\mathit{Sum}}{n+m}, \tag{9.3}$$

where

$$\mathit{Sum} = \textstyle\sum_{i=1}^{c} u_i \cdot \mathit{gain}_{\mathrm{q}}(\mathit{prob}_*, t_i, \mathit{quality}(\mathit{pt}_i, \mathit{result}_i)) + u_* \cdot \sum_{i=c+1}^{n} \mathit{gain}(\mathit{prob}_*, B, \mathsf{intr})$$

$$+ u_* \cdot \textstyle\sum_{j=d+1}^{m} \mathit{gain}(\mathit{prob}_*, B, \mathsf{intr}).$$

We use the same gain values in the problem-specific version of Equation 8.3, which gives us the standard deviation of the resulting estimate:

$$\sqrt{\frac{\mathit{SqrSum} - \frac{\mathit{Sum}^2}{n+m}}{(n+m) \cdot (n+m-d-1)}}, \tag{9.4}$$

where Sum is the same as in Equation 9.3 and

$$\begin{aligned} SqrSum = {} & \textstyle\sum_{i=1}^{c} u_i \cdot gain_{\mathrm{q}}(prob_*, t_i, quality(pt_i, result_i))^2 + u_* \cdot \sum_{i=c+1}^{n} gain(prob_*, B, \mathsf{intr})^2 \\ & + u_* \cdot \textstyle\sum_{j=d+1}^{m} gain(prob_*, B, \mathsf{intr})^2. \end{aligned}$$

Next, we give problem-specific versions of Equations 8.5 and 8.6. We use these equations if we have tried to solve $prob_*$ in the past and interrupted the solver at some bound B_o. We assume that e of the past terminations are no larger than B_o, that is, $t_e \leq B_o < t_{e+1}$. We allow the reuse of the previously expanded search space, and denote the reused time by b_o. We compute the expected gain for $prob_*$ as follows (see Section 8.3 for a detailed description of the notation):

$$\frac{Sum}{n+m-e}, \tag{9.5}$$

where

$$\begin{aligned} Sum = {} & \textstyle\sum_{i=e+1}^{c} u'_i \cdot gain_{\mathrm{q}}(prob_*, t_i - b_o, quality(pt_i, result_i)) \\ & + u'_* \cdot \textstyle\sum_{i=c+1}^{n} gain(prob_*, B - b_o, \mathsf{intr}) \\ & + u'_* \cdot \textstyle\sum_{j=d+1}^{m} gain(prob_*, B - b_o, \mathsf{intr}). \end{aligned}$$

The standard deviation of this estimate is

$$\sqrt{\frac{SqrSum - \frac{Sum^2}{n+m-e}}{(n+m-e) \cdot (n+m-e-d-1)}}, \tag{9.6}$$

where Sum is the same as in Equation 9.5 and

$$\begin{aligned} SqrSum = {} & \textstyle\sum_{i=e+1}^{c} u'_i \cdot gain_{\mathrm{q}}(prob_*, t_i - b_o, quality(pt_i, result_i))^2 \\ & + u'_* \cdot \textstyle\sum_{i=c+1}^{n} gain(prob_*, B - b_o, \mathsf{intr})^2 \\ & + u'_* \cdot \textstyle\sum_{j=d+1}^{m} gain(prob_*, B - b_o, \mathsf{intr})^2. \end{aligned}$$

We now state the applicability condition for these equations, in terms of the gain's correlation with running time and solution quality. We define $time(prob)$ as the running time on $prob$ without a time bound, and $quality(prob)$ as the quality for the corresponding result, which may be a solution, failure, rejection, or infinite run.

Condition 9.2 *We use problem-specific estimates if, for every fixed pair of time and quality values, $time_*$ and $quality_*$, the following three conditions hold:*

1. *The gain function $gain_{\mathrm{q}}(prob, time_*, quality_*)$ does not correlate with $time(prob)$.*
2. *For the success-outcome problems, the gain function $gain_{\mathrm{q}}(prob, time_*, quality_*)$ does not correlate with $quality_{\mathrm{p}}(prob)$.*
3. *The mean value of $gain_{\mathrm{q}}(prob, time_*, quality_*)$ for the success-outcome problems is equal to its mean value for the failure-outcome problems, to that for the rejection-outcome problems, and to that for the infinite-run problems.*

Even though the use of Condition 9.2 gives good practical results, it is neither necessary nor sufficient for ensuring unbiased problem-specific estimates. Finding a more accurate applicability condition is an open problem, which we plan to address in the future.

9.2 Use of problem size

If the system can estimate the relative sizes of problems, then it can improve performance by adjusting the choice of a representation and time bound to the problem size.

We define a *problem size* as an easily computable positive value that correlates with the problem complexity: the larger the value, the longer it usually takes to solve the problem. Finding an accurate measure of complexity is often a difficult task; however, many domains have features that provide at least a rough complexity estimate. For example, in the transportation domain, we may estimate the problem complexity by the number of packages to be delivered. In the rightmost column of Tables 8.1 and 8.3, we show the number of packages in each of the sample problems.

Note that measures of a problem size are usually domain-specific, and the choice of a good measure is the user's responsibility. We allow the user to specify different size measures for different representations.

We apply least-squares regression to find an approximate dependency between problem size and running time (Section 9.2.1) and use this dependency in estimating the expected gains (Section 9.2.2). We give the results of using size in the transportation domain (Section 9.2.3) and then test the regression technique on artificial data (Section 9.2.4).

9.2.1 Dependency of time on size

We describe the use of linear regression to find the dependency between the sizes of sample problems and the times required to solve them. We use three separate regressions: the first regression for the success times, the second one is for the failure times, and the third is for the rejection times.

In $\mathcal{S}$HAPER, success usually occurs after exploring a small part of the search space, whereas failure requires the exploration of the entire space, and the dependency of the success time on the problem size is quite different from that of the failure time. The rejection times are different from both success and failure times, because we usually perform the rejection test by a separate procedure, which does not use the system's search engines.

We allow the user to enable some of the three regressions and turn off the others. In particular, we usually do not use regression for rejection times, because, in most cases, the time for rejection tests is either negligibly small or constant.

We assume that the dependency of time on size is either polynomial or exponential. If it is polynomial, than the logarithm of time depends linearly on the logarithm of size; for an exponential dependency, the time logarithm depends linearly on size. We thus use linear regression to find both polynomial and exponential dependencies.

We use the least-squares technique to perform regression. In Figure 9.1(a) and 9.1(b), we give the regression formulas for a polynomial dependency between size and time; the regression for an exponential dependency is similar. We denote the number of sample problems by n, the problem sizes by $size_1, ..., size_n$, and the corresponding running times by $time_1, ..., time_n$.

We evaluate the regression results using the t-test. The t value in this test is the ratio of the estimated slope of the regression line to the standard deviation of the slope estimate.

(a) Approximate dependency of the running time on the problem size:
$\log time = \alpha + \beta \cdot \log size$;
that is, $time = e^{\alpha} \cdot size^{\beta}$.

(b) Regression coefficients:
$\beta = \frac{\sum_{i=1}^{n} \log size_i \cdot \log time_i - SizeSum \cdot TimeSum/n}{SizeSqrSum - SizeSum^2/n}$,
$\alpha = \frac{TimeSum}{n} - \beta \cdot \frac{SizeSum}{n}$,
where
$TimeSum = \sum_{i=1}^{n} \log time_i$,
$SizeSum = \sum_{i=1}^{n} \log size_i$,
$SizeSqrSum = \sum_{i=1}^{n} (\log size_i)^2$.

(c) The t value, for evaluating the regression accuracy:
$t\text{-}value = \frac{\beta}{TimeDev} \cdot \sqrt{SizeSqrSum - \frac{SizeSum^2}{n}}$,
where
$TimeDev = \sqrt{\frac{1}{n-2} \cdot \left(\sum_{i=1}^{n} (\log time_i)^2 - \frac{TimeSum^2}{n} - \beta \cdot \left(\sum_{i=1}^{n} \log size_i \cdot \log time_i - \frac{SizeSum \cdot TimeSum}{n}\right)\right)}$.

Figure 9.1: Regression coefficients and t value for the polynomial dependency of time on size.

We give the formula for computing this value in Figure 9.1(c). The *TimeDev* value in this formula is the standard deviation of time logarithms. It shows how much, on average, time logarithms deviate from the regression line.

The t-test converts the t value into the probability that the use of regression gives *no better* prediction of running time than ignoring the sizes and simply taking the mean; in other words, it is the probability that regression does not help. This probability is called the *P value*; it is a function of the t value and the number n of sample problems. When the regressed line gives a good fit to the sample data, the t value is large and the P value is small.

In Figure 9.2, we give the results of regressing the success times for the sample transportation problems from Table 8.1; we do *not* show failure regression. The top three graphs give the polynomial dependency of the success time on the problem size; the bottom graphs are for the exponential dependency. The horizontal axes show the problem sizes (that is, the number of packages), and the vertical axes are the times. The circles show the sizes and times of the problem instances; the solid lines are the regression results. For each regression, we give the t value and the corresponding interval of the P value.

We use regression only if the P value is smaller than a certain bound. In our experiments, we set this bound to 0.2; that is, we used problem size only for $P < 0.2$. This test ensures that we use size only if it provides a good correlation with problem complexity. If the size measure proves inaccurate, then the gain-estimate algorithm ignores sizes. We use the 0.2 bound rather than more "customary" 0.05 or 0.02 because an early detection of a dependency between size and time is more important for the overall efficiency than establishing a high certainty of the dependency.

For example, all three polynomial regressions in the top row of Figure 9.2 pass the $P < 0.2$

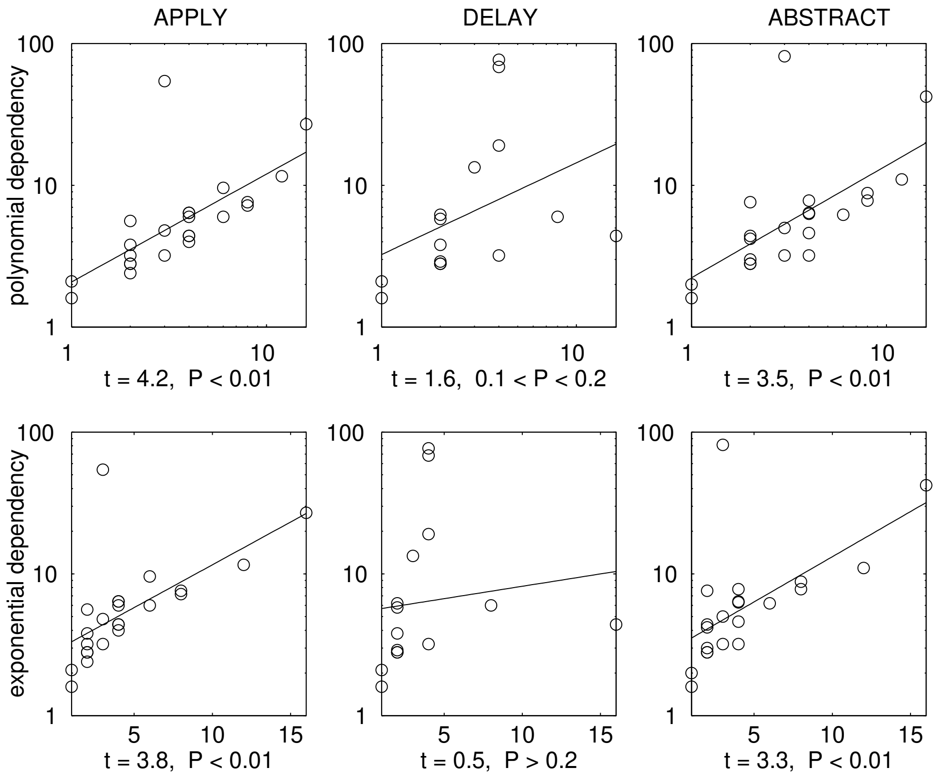


Figure 9.2: The dependency of the success time on the problem size. The top graphs show the regression for a polynomial dependency, and the bottom graphs are for an exponential dependency.

test. The exponential regressions for Apply and Abstract also satisfy this condition. On the other hand, the exponential regression for Delay fails the test (see the middle bottom graph in Figure 9.2).

The choice between polynomial and exponential regression is based on the t-test results: we prefer the regression with the larger t value. In the example of Figure 9.2, the polynomial regression wins for all three representations.

The user has an option to select between the two regressions herself. For example, she may insist on the use of the exponential regression. We also allow the user to set a regression slope. This option is useful when the human operator has a good notion of the slope value and the past data are not sufficient for an accurate estimate. If the user specifies a slope, the algorithm uses her value in regression; however, it compares the user's value with the regression estimate of Table 9.1, determines the statistical significance of the difference, and gives a warning if the user's estimate is off with high probability.

Note that the least-squares regression and the related t-test make quite strong assumptions about the nature of the distribution. First, for problems of fixed size, the distribution of the time logarithms must be normal; that is, time must be distributed log-normally. Second, for all problem sizes, the standard deviation of the distribution must be the same. Regression, however, usually provides a good approximation of the dependency between size and time, even when these assumptions are not satisfied.

The computational complexity of regression is linear in the number of data points. For

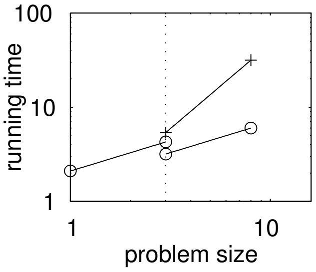


Figure 9.3: Scaling two success times (o) and a failure time (+) of Delay to a 3-package problem.

n terminations and m interrupts, the implemented Lisp procedure on a Sun 5 performs both polynomial and exponential regression, along with the related t-tests, in approximately $(n+m) \cdot 7 \cdot 10^{-4}$ seconds.

During the incremental learning, the system does *not* perform regression from scratch for each new problem. Instead, it stores the sums used in regression computation (see Figure 9.1), and updates them incrementally, after adding each new problem to the sample. The system updates the sums, recomputes the regression coefficients, and finds the new t-values in constant time, about $8 \cdot 10^{-4}$ seconds in total.

9.2.2 Scaling of past running times

The use of the problem size in estimating the gain is based on "scaling" the times of sample problems to a given size. We illustrate it in Figure 9.3, where we scale Delay's times of a 1-package success, an 8-package success, and an 8-package failure for estimating the gain on a 3-package problem (the 3-package size is marked by the vertical dotted line). To scale a problem's time to a given size, we draw the line with the regression slope through the point representing the problem (see the solid lines in Figure 9.3), to the intersection with the vertical line through the given size (the dotted line). The ordinate of the intersection is the scaled time.

If the size of the problem is $size_o$, the running time is $time_o$, and we need to scale it to $size$, using a regression slope β, then we compute the scaled time $time$ as follows:

Polynomial regression:
$\log time = \log time_o + \beta \cdot (\log size - \log size_o)$;
that is, $time = time_o \cdot \left(\frac{size}{size_o}\right)^{\beta}$.

Exponential regression:
$\log time = \log time_o + \beta \cdot (size - size_o)$;
that is, $time = time_o \cdot \exp(\beta \cdot (size - size_o))$.

We use the slope of the success regression in scaling success times (see the lines through circles in Figure 9.3), the slope of the failure regression in scaling failures (the line through pluses), and the slope of the rejection regression in scaling rejection times. The slope for scaling an interrupt time should depend on whether the system would succeed, fail, or reject

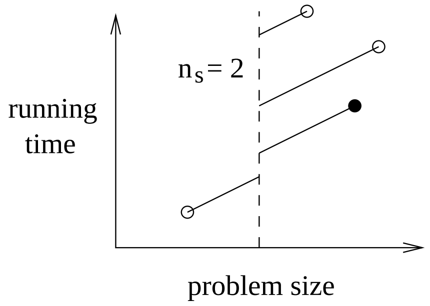


Figure 9.4: Computation of n_s for an interrupt point: the system scales all successes times (o) and the interrupt time (•) to a fixed size, using the success slope.

the problem if we did not interrupt it; however, we do not know which of these three outcomes would occur.

We "distribute" each interrupt point among success, failure, and rejection slopes, according to the probabilities that the continuation of the execution would result in the respective outcomes. We thus break an interrupt point into three weighted points, whose total weight is 1.

To determine the three weights, we may scale the success, failure, and rejection times in the data sample to the size of the problem that caused the interrupt. The weights are proportional to the number of successes, failures, and rejections that are *larger* than the interrupt time. If the number of larger success times is n_s, the number of larger failures is n_f, and that of larger rejections is n_r, then the corresponding weights are $\frac{n_s}{n_s+n_f+n_r}$, $\frac{n_f}{n_s+n_f+n_r}$, and $\frac{n_r}{n_s+n_f+n_r}$.

The implemented computation of n_s, n_f, and n_r is somewhat different, for efficiency reasons. The algorithm does *not* scale success, failure, and rejection times to every interrupt point; instead, it scales them to some fixed size, and then scales interrupt points to the same size. In Figure 9.4, we illustrate the computation of n_s. Note that the algorithm scales every interrupt three times: it uses the success slope for determining n_s, the failure slope for n_f, and the rejection slope for n_r.

After scaling the times of the sample problems to a given size, we use the technique of Sections 8.2 and 8.3 to compute the gain estimate and its standard deviation (see Equations 8.5 and 8.6). The only difference is that we reduce the second term in the denominator for the standard deviation by 2, because the success and failure regressions reduce the number of degrees of freedom of the sample data. Thus, we modify Equation 8.6 and compute the deviation as follows:

$$\sqrt{\frac{SqrSum - \frac{Sum^2}{n+m-e}}{(n+m-e)\cdot(n+m-e-d-3)}}. \tag{9.7}$$

The running time of the scaling procedure is proportional to the number of data points. The Lisp implementation on a Sun 5 scales a termination point in $2 \cdot 10^{-4}$ seconds and distributes an interrupt in $6 \cdot 10^{-4}$ seconds. Thus, for a sample of n terminations and m interrupts, the scaling time is $(2 \cdot n + 6 \cdot m) \cdot 10^{-4}$ seconds.

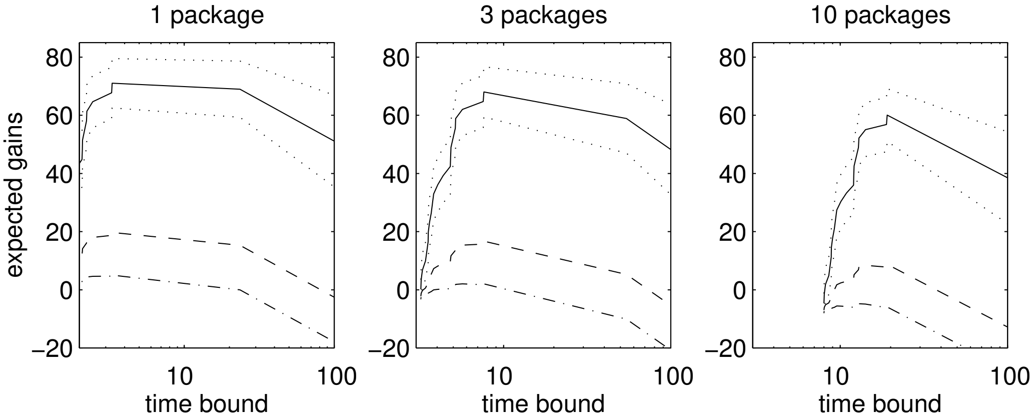


Figure 9.5: Dependency of Apply's expected gain on time bound in the simple transportation domain, for rewards of 10.0 (dash-and-dot lines), 30.0 (dashed lines), and 100.0 (solid lines). The dotted lines show the standard deviation for the 100.0 reward.

We now give the overall running time of the statistical computation, for the example gain function from Section 8.1.2. The computation includes polynomial and exponential regression, related t-tests, scaling the sample times to a given problem size, and determining the expected gains for l time bounds. If the system performs regression from scratch, the total time is $(3 \cdot l + 9 \cdot n + 13 \cdot m) \cdot 10^{-4}$ seconds. If it incrementally updates the regression coefficients and t-values, then the time is $(3 \cdot l + 2 \cdot n + 8 \cdot m + 8) \cdot 10^{-4}$ seconds.

9.2.3 Results in the transportation domain

We illustrate the use of size by experiments in the transportation domain. We use the number of packages to be delivered as a measure of the problem size, and estimate problem-solving gains for specific sizes. In Figure 9.5, we show the dependency of the expected gain on time bound when using Apply on 1-package, 3-package, and 10-package problems. We computed this dependency for the simple transportation domain with the example gain function (Section 8.1.2), using the performance data in Table 8.1.

If we use problem size in the incremental-selection experiments of Sections 8.4 and 8.6, we get larger gains in all eight experiments. In Table 9.1, we give the per-problem gains in these experiments, with and without size.

The results demonstrate that the use of problem size increases the gain, though not by much. In Section 9.2.4, we show that the use of regression gives a more significant gain increase when running times better correlate with problem size. The average running time of regression and scaling is 0.03 seconds per problem. Thus, the time of the statistical computation for using problem size is much smaller than the resulting gain increase.

In Figure 9.6, we give a more detailed comparison of gains with and without regression, for the bound-selection experiments of Section 8.4. The horizontal axes show the number of a problem, from 7 to 30. We skip the first six problems, because the algorithm does not use size in selecting the time bounds for these problems: it has not yet accumulated enough data for regression with sufficiently small P value.

	w/o sizes	with sizes
transportation by vans (Section 8.4)		
Apply's bound selection	12.0	12.2
Delay's bound selection	3.9	4.7
Abstract's bound selection	11.3	11.9
selection of a representation	11.1	11.8
transportation by vans and airplanes (Section 8.6)		
Apply's bound selection	110.1	121.6
Delay's bound selection	131.1	137.4
Abstract's bound selection	243.5	248.3
selection of a representation	207.0	215.6

Table 9.1: Per-problem gains in the learning experiments, with and without the use of size.

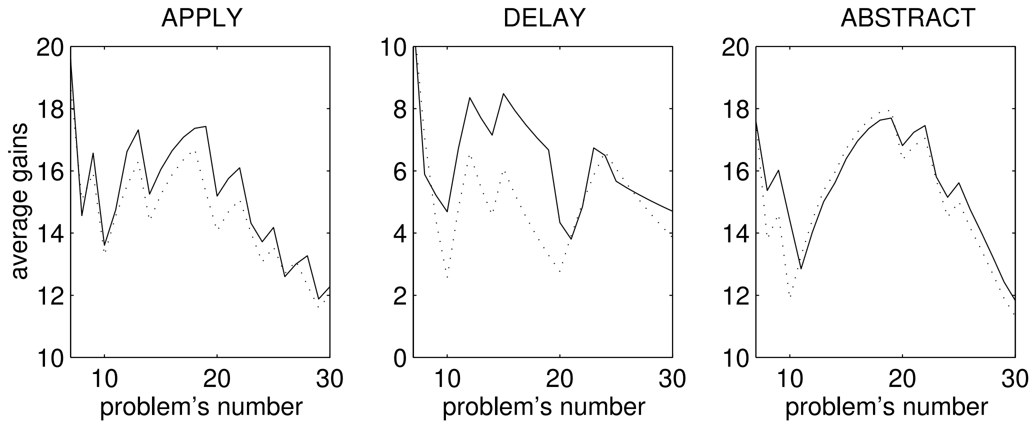


Figure 9.6: Average per-problem gains without regression (dotted lines) and with regression (solid lines), during incremental learning of a time bound.

The vertical axes show the average per-problem gain up to the current problem. For example, the left end of the curve shows the average gain for the first seven problems, and the right end gives the average for all thirty problems. The gain declines for problems 20 to 30 because these problems happen to be harder, on average, than the first twenty problems (see Table 8.1). The dotted lines give the average gains without the use of problem size, and the solid lines are for the gains obtained with regression.

The graphs show that the use of problem size usually, though not always, provides a small improvement in performance. The apparent advantage of regression in Delay's learning is mostly due to the choice of low time bounds for problems 9 and 10, which cannot be solved in feasible time. This luck in setting low bounds for two hard problems is not statistically significant. If the algorithm does not use problem size, it hits the time bounds of 16.9 and 14.0 on these problems (see Figure 8.6) and falls behind in its per-problem gain.

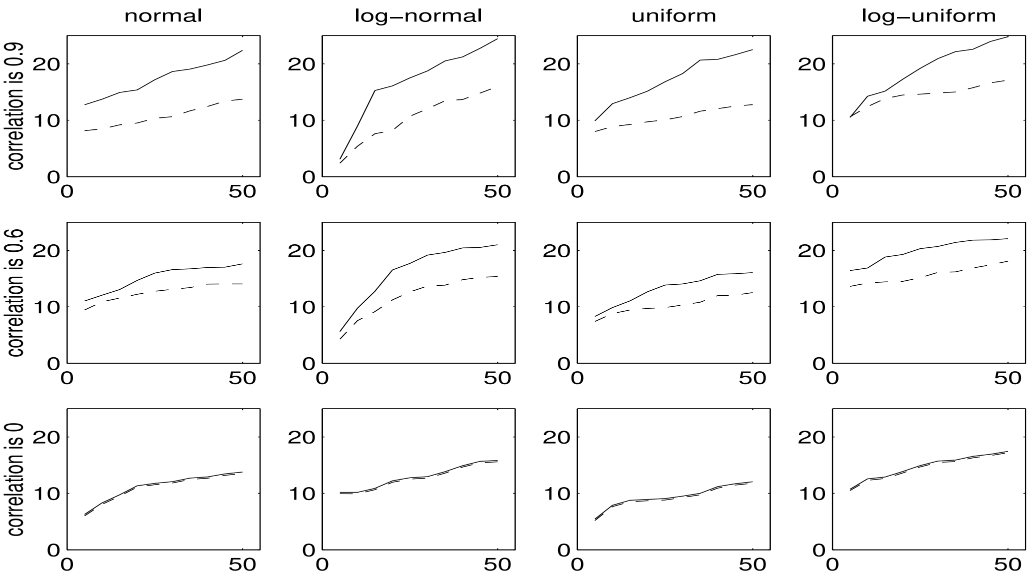


Figure 9.7: Per-problem gains with the use of size (solid lines) and without size (dashed lines), for different correlations between size logarithms and time logarithms.

9.2.4 Experiments with artificial data

We test the regression technique on artificially generated values of running times. The experimental setup is the same as in the artificial tests of Section 8.7. That is, we use the linear gain function with a reward $R = 100.0$, and consider four types of running-time distribution: normal, log-normal, uniform, and log-uniform. The experiments demonstrate that regression improves the performance when there is a correlation between time and size, and does not worsen the results when there is no correlation.

Selecting a time bound

We give the results of learning a time bound on 50-problem sequences, for different time distributions and different correlations between time and size, and compare the gains obtained with and without regression.

Problem sizes in this experiment are natural numbers from 1 to 10, selected randomly. The logarithms of mean success and failure times are proportional to the problem-size logarithms. We adjusted the standard-deviation values to obtain desired correlations between time logarithms and size logarithms. We used the correlation of 0.9 in the first series of experiments and 0.6 in the second series. Finally, we ran a series of experiments with zero correlation; the mean times in this series were the same for all problem sizes.

We give the results in Figure 9.7, where solid lines show the average per-problem gains with regression, and the dashed lines give the gains obtained without regression. The use of regression improves the performance and the improvement is greater for larger correlations. If there is no correlation, the system disregards the results of regression and performs identically with and without size.

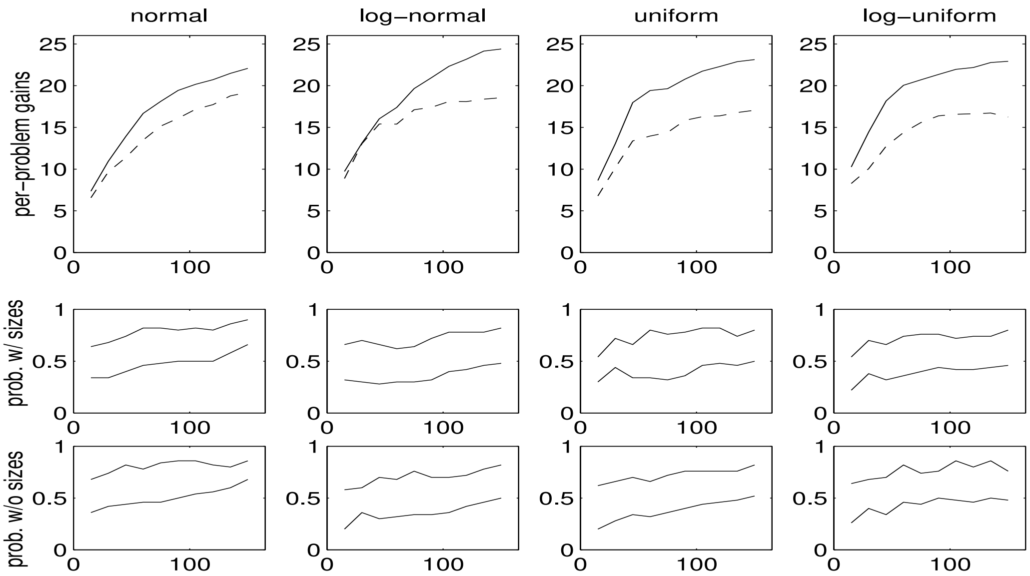


Figure 9.8: Incremental selection among three representations, where the average gain for the first representation is 10% larger than that for the second one and 20% larger than that for the third one. We show the average per-problem gains in the experiments with and without regression (the top row of graphs), and the probability of selecting each representation (the other two rows).

Choosing a representation

We next show the results of the incremental selection among three representations, with the same distributions as in the representation-selection experiments of Section 8.7 (see Figures 8.18 and 8.19). We present two series of experiments, with 150-problem sequences.

In the first series, the optimal gain for the first representation is 10% larger than that for the second representation and 20% larger than that for the third one. We summarize the results in Figure 9.8. The top row of graphs shows the average per-problem gain with the use of problem size (solid lines) and without size (dotted lines). The other two rows give the probability of choosing each representation, in the experiments with and without size.

In the second series, the optimal gain of the first representation is 30% larger than that of the second one and 60% larger than that of the third one. We give the results of this series in Figure 9.9.

9.3 Similarity among problems

We have estimated the expected gain by averaging the gains of *all* sample problems. If we know which of them are similar to a new problem, we may improve the estimate accuracy by averaging only the gains of these similar problems.

We encode similarity among problems by a hierarchy of problem groups (Section 9.3.1), and use this hierarchy in estimating expected gains (Section 9.3.2). We illustrate it by experiments in the transportation and phone-call domain (Section 9.3.3).

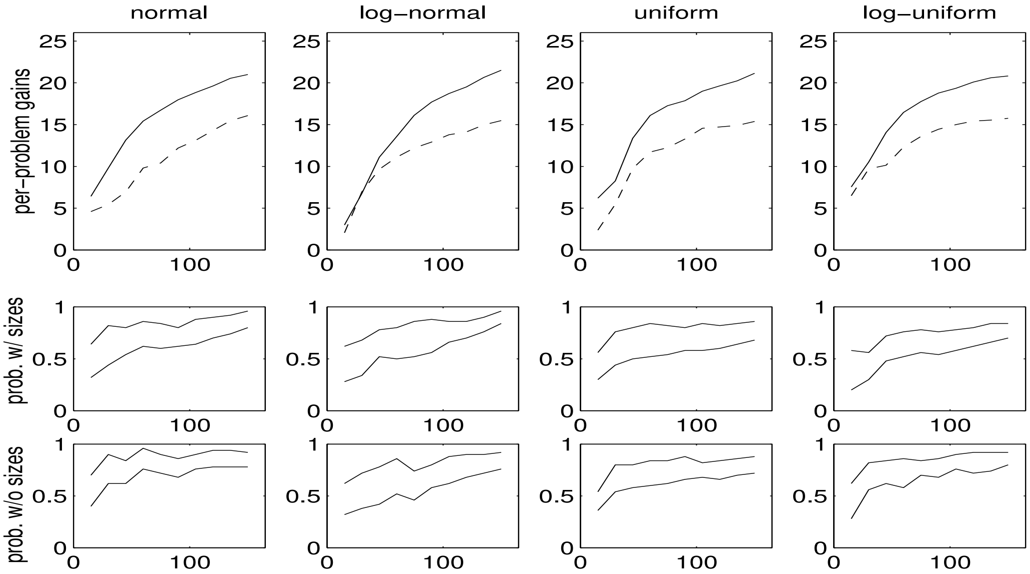


Figure 9.9: Incremental selection among three representations, with and without regression. The average gain for the first representation is 30% larger than that for the second one and 60% larger than that for the third one.

#	time (sec) and outcome			# of
	Apply	Delay	Abstract	conts
1	2.3 s	2.3 s	2.1 s	1
2	3.1 s	5.1 s	4.1 s	2
3	5.0 s	20.2 s	4.8 s	3
4	3.3 s	8.9 s	3.2 s	2
5	6.7 s	36.8 s	6.4 s	4
6	200.0 b	200.0 b	10.1 f	8
7	3.2 s	3.2 s	3.2 s	2
8	24.0 s	200.0 b	26.3 s	8
9	4.8 s	86.2 s	3.4 s	4
10	8.0 s	200.0 b	9.4 s	6

Table 9.2: Performance of Apply, Delay, and Abstract on ten container-transportation problems.

9.3.1 Similarity hierarchy

We describe similarity by a tree-structured *similarity hierarchy.* The leaf nodes of the hierarchy are groups of similar problems. The other nodes represent weaker similarity among groups. We assume that each problem belongs to exactly one group, and that determining a problem's group takes little computational time.

For example, we may divide the transportation problems into within-city and between-city deliveries. We extend this example by a new type of problem, which involves the transportation of containers within a city. A van can carry only one container at a time, which sometimes makes container delivery harder than package delivery. In Table 9.2, we give the performance of Apply, Delay, and Abstract on ten container-transportation problems. If we subdivide within-city problems into package deliveries and container deliveries, we get the similarity hierarchy shown in Figure 9.10(a).

The construction of a hierarchy is presently the user's responsibility. We plan to address the problem of learning a hierarchy automatically in future work. We allow the user to construct a separate hierarchy for each representation or a common hierarchy for all repre-

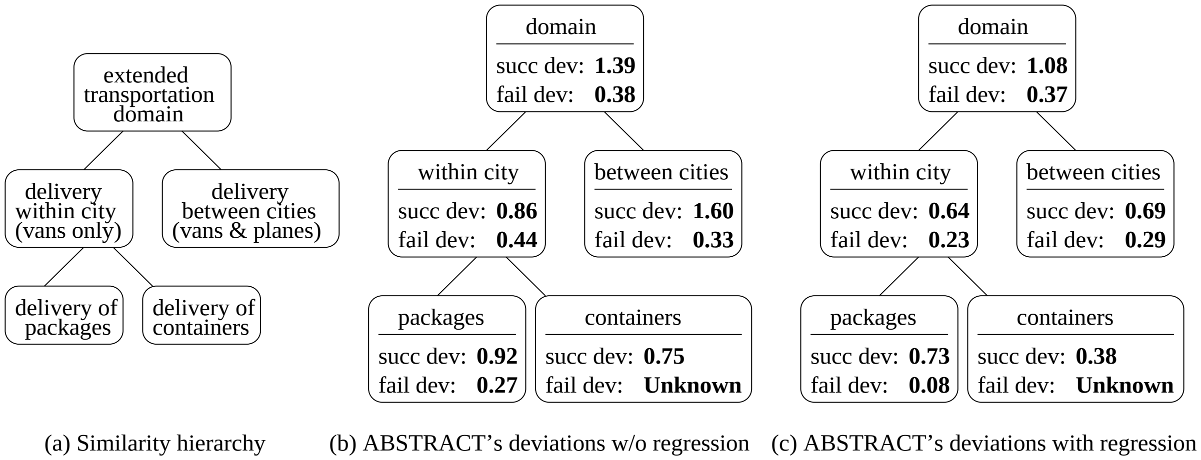


Figure 9.10: Similarity hierarchy and the deviations of Abstract's success and failure logarithms.

sentations. We also allow the use of different problem-size measures for different groups of problems.

We may estimate the similarity of problems in a group by the standard deviation of the logarithms of running times, computed for the sample problems that belong to the group:

$$TimeDev = \sqrt{\frac{1}{n-1} \cdot \left(\sum_{i=1}^{n} (\log time_i)^2 - \frac{(\sum_{i=1}^{n} \log time_i)^2}{n} \right)}. \tag{9.8}$$

We compute the deviations separately for successes and failures, and use these values as a heuristic measure of the hierarchy's quality. In the current implementation, we do *not* include the rejection-time deviations into the quality measure, because rejection times do not depend on problem groups in any of $\mathcal{S}$HAPER's domains.

The smaller the success and failure deviations for the leaf groups, the better the user's hierarchy. If some deviation value is larger than a pre-set threshold, the system gives a warning. In the implementation, we set this threshold to 2.0.

If we use regression, we apply it separately to each group of the similarity hierarchy. If regression confirms the dependency between problem size and time, we compute the deviation of time logarithms by a different formula, given in the last line of Figure 9.1.

For example, the deviation values for Abstract in the transportation domain are as shown in Figure 9.10. We give the deviations computed without regression in Figure 9.10(b), and the deviations for gain estimates with regression in Figure 9.10(c). The values show that within-city problems are more similar to each other than between-city problems.

Note that the deviations of the logarithms do not change if we multiply all times by the same factor, which means that they do not depend on the speed of a specific computer that executes the code. Also, the deviation values do not change, on average, with the addition of more problems to the sample.

9.3.2 Choice of a group in the hierarchy

We may estimate the expected gain for a new problem by averaging the gains of the sample problems that belong to the same leaf group. Alternatively, we may use a larger sample from one of its ancestors. The leaf group has less data than its ancestors, but the deviation of these data is smaller. We need to analyze this trade-off when selecting between the leaf group and its ancestors. Intuitively, we should use ancestral groups during early stages of incremental learning, and move to leaf groups after collecting more data.

We present a heuristic (rather than a statistical technique) for selecting between a group and its parent, based on two tests. The first test is aimed at identifying the difference between the distribution of the group's problems and the distribution of the other problems in the parent's sample. If the two distributions prove different, we use the group rather than its parent for estimating the problem-solving gain. If not, we perform the second test, to determine whether the group's sample provides a more accurate performance estimate than the parent's sample. We now describe the two tests in detail.

If we do not use regression, then the first test is the statistical t-test that determines whether the mean of the group's time logarithms differs from the mean of the other time logarithms in the parent's sample. We perform the test separately for successes and failures. In our experiments, we consider the means different when we can reject the null-hypothesis that they are equal with 0.75 confidence. If we use regression and it confirms the dependency between size and time, then we use a different t-test. Instead of comparing the means of time logarithms, we determine whether the regression lines are different with confidence 0.75.

A statistically significant difference for either successes or failures is a signal that the distribution of the group's running times differs from the distribution for the other problems in the group's parent. Therefore, if we need to estimate the gain for a new problem that belongs to the group, the use of the parent's sample may bias the prediction. We thus should use the group rather than its parent.

For example, suppose that we use the data in Tables 8.1, 8.3, and 9.2 with the hierarchy in Figure 9.10(a), and we need to estimate Abstract's gain on a new problem that involves the delivery of packages within a city. We consider the choice between the corresponding leaf group and its parent. In this example, we do *not* use regression.

The estimated mean of the success-time logarithms for the package-delivery problems is 4.07, and the standard deviation of this estimate is 0.20. The estimated mean for the other problems in the parent group, which are the container-delivery problems, is 4.03, and its deviation is 0.16. The difference between the two means is *not* statistically significant. Since the container-transportation sample has only one failure, we cannot estimate the deviation of its failure logarithms; therefore, the difference between the failure-logarithm means is also considered insignificant.

If we apply regression to this example and use the t-test to compare the regression slopes, it also shows that package-transportation and container-transportation times are not significantly different.

The second test is the comparison of the standard deviations of the mean estimates for the group and its parent. The deviation of the mean estimate is equal to the deviation of the time logarithms divided by the square root of the sample size, $\frac{TimeDev}{\sqrt{n}}$. We compute it separately for success times and failure times. We use this value as an indicator of the sample's accuracy in estimating problem-solving gain: the smaller the value, the greater the accuracy. This indicator accounts for the trade-off between the deviation of the running-time distribution and the sample size. It increases with an increase in the deviation and decreases with an increase in the sample size.

If the group's deviation of the mean estimate is smaller than that of the group's parent, for either successes or failures, then the group's sample is likely to provide a more accurate gain estimate; thus, we prefer the group to its parent. On the other hand, if the parent's mean-estimate deviation is smaller for both successes and failures, and the comparison of the group's mean with that of the other problems of the parent sample has not revealed a significant difference, then we use the parent to estimate the gain for a new problem.

Suppose that we apply the second test to the group selection for estimating Abstract's gain on within-city package delivery. The standard deviation of the mean estimate of the success-time logarithms, for the corresponding leaf group, is 0.20; the deviation for its parent is 0.16. The deviation of the mean estimate of the failure-time logarithms is also smaller for the parent. Since the first test has not revealed a significant difference between the group's times and the other times in the parent's sample, we prefer the use of the parent.

After selecting between the leaf group and its parent, we use the same two tests to choose between the resulting "winner" and the group's grandparent. We then compare the new winner with the great-grandparent, and so on. In our example, we need to compare the selected parent group with the top-level node (see Figure 9.10a). After applying the first test, we find out that the mean of the group's success logarithms is 4.03, and the corresponding mean for the other problems in the top node's sample is 5.39. The difference between these means is statistically significant. We thus prefer the group of within-city problems to the top-level group.

In Figure 9.11, we summarize the algorithm for selecting a group of the similarity hierarchy. It inputs the leaf group of a given problem and returns an ancestor group for use in estimating the problem-solving gains. Note that, for every ancestor group, the algorithm has to determine several sums over its size and time values. We use the following sums in statistical computations of the procedures *Test-1* and *Test-2*:

$$\sum_{i=1}^{n} size_i \qquad \sum_{i=1}^{n} size_i^2 \qquad \sum_{i=1}^{n} \log size_i \qquad \sum_{i=1}^{n} (\log size_i)^2$$

$$\sum_{i=1}^{n} \log time_i \qquad \sum_{i=1}^{n} (\log time_i)^2 \qquad \sum_{i=1}^{n} size_i \cdot \log time_i \qquad \sum_{i=1}^{n} \log size_i \cdot \log time_i$$

We compute these sums separately for successfully solved problems and for failures. When performing *Test-1*, we use not only the sums for the group's problems, but also the sums of the *other* problems in the group's parent.

We store all these sums for every group of the hierarchy, and update them incrementally. When adding a new problem to the past performance data, we increment the sums of the corresponding leaf group and all its ancestors. Thus, the running time for adding a problem is proportional to the depth of the hierarchy; that is, its complexity is $O(depth)$. We use the pre-computed sums in statistical tests, which allows the computation of *Test-1* and *Test-2* in constant time. Thus, the time complexity of *Select-Group* is also $O(depth)$.

We measured the execution time for a Lisp implementation on a Sun 5 computer, using the example gain function in the incremental learning. The statistical computation for each

Select-Group(*Leaf-Group*)
The input is a leaf group of a similarity hierarchy; the output is the selected ancestor group.

Set the initial values:
  Current-Group := *Leaf-Group*
  Best-Group := *Leaf-Group*

Repeat while Current-Group is not the root of the hierarchy:
  If *Test-1*(*Current-Group*) and *Test-2*(*Best-Group*, *Parent*(*Current-Group*)),
    then *Best-Group* := *Parent*(*Current-Group*)
  Current-Group := *Parent*(*Current-Group*)

Return *Best-Group*

Test-1(*Group*)
The test returns true if the distribution of the group's problems is *not* significantly different from the distribution of the *other* problems in the parent's sample.

If a t-test confirms that either
  ∘ the success times of *Group* differ from the other success times in *Parent*(*Group*), or
  ∘ the failure times of *Group* differ from the other failure times in *Parent*(*Group*),
 then return false;
Else, return true.

(We apply one of the following two t-tests:
  If we use size and the regression is statistically significant for both sets of time values,
    then we determine whether the regression lines are different with confidence 075;
  Else, we determine whether the means of time logarithms are different with confidence 075.)

Test-2($Group_1$, $Group_2$)
Here $Group_2$ is an ancestor of $Group_1$. The test returns true if $Group_2$ provides a more accurate estimate than $Group_1$ for both the expected success time and the expected failure time.

Set the following values:
  ns_1, nf_1: the number of successes (ns_1) and failures (nf_1) in $Group_1$
  ns_2, nf_2: the number of successes and failures in $Group_2$
  $S\_TimeDev_1, F\_TimeDev_1$: the *TimeDev* values for successes and failures in $Group_1$
  $S\_TimeDev_2, F\_TimeDev_2$: the *TimeDev* values for $Group_2$

  (We use one of the following two expressions for computing a *TimeDev* value:
    If we use size and the regression is statistically significant,
      then we apply the formula in the last line of Figure 9.1;
    Else, we compute it using Equation 9.8.)

Perform the test:
  If $\frac{S\_TimeDev_2}{ns_2} < \frac{S\_TimeDev_1}{ns_1}$ and $\frac{F\_TimeDev_2}{nf_2} < \frac{F\_TimeDev_1}{nf_1}$,
    then return true;
  Else, return false.

Figure 9.11: Selecting a group of a similarity hierarchy for the gain-estimate computation.

	using leaf groups	using the top group	heuristic group selection
	without the use of problem size		
Apply's bound selection	11.8	10.5	12.1
Delay's bound selection	7.0	4.7	7.5
Abstract's bound selection	19.5	18.1	19.5
selection of a representation	13.1	11.1	13.4
	with the use of problem size		
Apply's bound selection	16.3	11.1	16.8
Delay's bound selection	12.1	5.2	12.0
Abstract's bound selection	22.6	18.4	22.6
selection of a representation	19.4	13.7	21.0

Table 9.3: Per-problem gains in learning experiments, for different group-selection techniques.

new problem includes performing the necessary regressions, selecting a group, scaling the times of this group to the size of the new problem, and determining the expected gains for l time bounds. It also includes adding the results of solving the problem to the past data. The total time of these operations is about $(3 \cdot l + 2 \cdot n + 8 \cdot m + 22 \cdot depth) \cdot 10^{-4}$ seconds. This time is still very small compared to PRODIGY's problem-solving time.

We have considered several alterations of the described group-selection heuristic in our experiments. In particular, we tried replacing the deviation of time logarithms with the deviation of times divided over their mean. In most cases, the use of this measure led to the same selection. We also tried to use either success or failure times, rather than both successes and failures. This alternative proved to be a less effective strategy. When successes are much more numerous than failures, which happens in most domains, the results of using successes and ignoring failures are near-identical to the results of using both success and failures; however, when the number of successes and failures is approximately equal, the use of both successes and failures gives better performance.

9.3.3 Examples of using similarity

We give results of using a similarity hierarchy in selecting representations and time bounds. We describe experiments in the transportation and phone-call domain, and show that the use of similarity increases problem-solving gains in both domains.

Transportation domain

We present the results of using the similarity hierarchy of Figure 9.10, and compare them with the results obtained without a hierarchy. We ran the bound-selection experiments on a sequence of seventy transportation problems, which was constructed by interleaving the problem sets of Tables 8.1, 8.3, and 9.2. We then ran experiments on choosing among Apply, Delay, and Abstract, using a sequence of 210 problems.

In Table 9.3, we give the mean per-problem gains obtained in these experiments. In the first column, we show the results of using only leaf groups in estimating the gains. In the

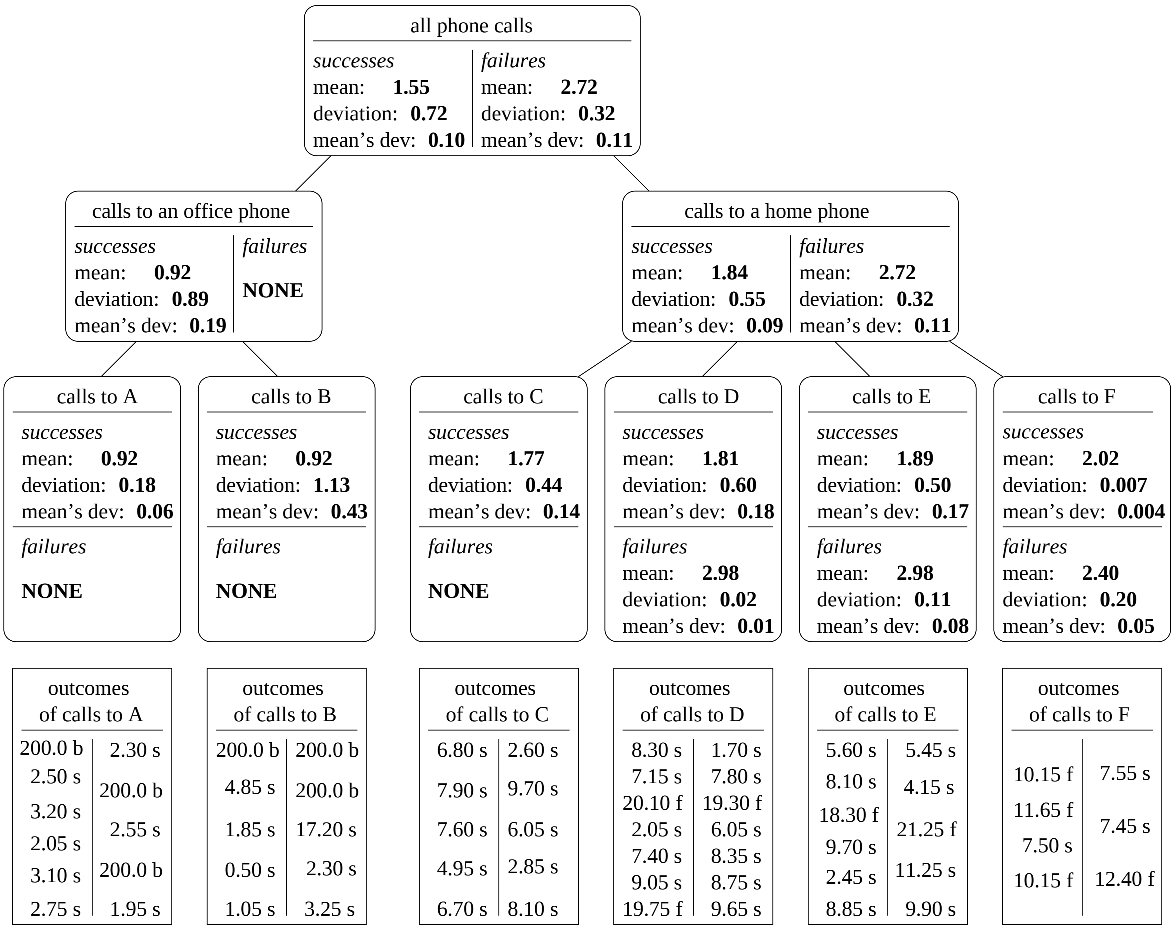


Figure 9.12: Similarity hierarchy and call outcomes in the phone-call domain.

second column, we give the results of using the top-level group for all estimates, which means that we do not distinguish among the three problem types. The third column contains the results of using the similarity hierarchy, with our heuristic for the group selection.

We first ran the experiments using both success and failure times in the group selection, and then re-ran them using only success times. In all eight cases, the results of using both successes and failures were identical to the results of using successes.

The experiments demonstrate that the use of the complete hierarchy gives larger gains than either the leaf groups or the top-level group; however, the improvement is not large.

Phone-call domain

We next use a similarity hierarchy in selecting a time bound for phone calls. We consider the outcomes of sixty-three calls to six different people. We phoned two of them, say A and B, at their office phones; we phoned the other four, C, D, E, and F, at their homes. We show the similarity hierarchy and the call outcomes in Figure 9.12.

For each group in the hierarchy, we give the estimated mean of success and failure time logarithms ("mean"), the deviation of the time logarithms ("deviation"), and the deviation of the mean estimate ("mean's dev"). The mean of success-time logarithms for calls to offices is significantly different from that for calls to homes, which implies that the distribution of office-call times differs from the distribution of home-call times.

The mean success logarithms for persons A and B are *not* significantly different from each other. Similarly, the success means of C, D, and E do *not* differ significantly from the mean of the home-call group. On the other hand, the success mean of F *is* significantly different from the mean for the other people in the home-call group, implying that the time distribution for F differs from the rest of its parent group. Finally, the failure-logarithm means of D, E, and F are all significantly different from each other.

We ran incremental-learning experiments on these data with the success reward of 90.0 and zero failure reward. An experiment with the use of the leaf groups for all gain estimates yielded a gain of 57.8 per call. We then ran an experiment using the home-call and office-call groups for all estimates, without distinguishing among different people within these groups, and obtained an average gain of 56.3. We next used the top-level group for all estimates, which yielded 55.9 per call. Finally, we experimented with the use of our heuristic for choosing between the leaf groups and their ancestors based on the means and deviations of time logarithms; the gain in this experiment was 59.8 per call. If we knew the time distributions in advance, determined the optimal time bound for each leaf group, and used these optimal bounds for all calls, then the average gain would be 61.9.

The phone-call experiments have confirmed that a similarity hierarchy improves performance, though not by much. Note that the gain obtained with the hierarchy is significantly closer to the optimal than the gain from the use of leaf groups or the top-level group.

Chapter 10

Preference rules

The $\mathcal{S}$HAPER system allows the use of heuristic rules in selecting representations. We have already discussed rejection and comparison rules, which prune ineffective descriptions and representations (see Sections 7.2.2 and 7.2.3). We now introduce preference rules, which generate judgments about performance of the remaining representations, *without* pruning any of them.

The preference-rule mechanism provides a general means for developing top-level control strategies, to supplement statistical learning. We use preference rules to encode the initial knowledge about relative performance of representations and to implement simple learning techniques. This mechanism also allows the user to control the trade-off between exploitation and exploration in statistical learning.

The system uses preference rules in conjunction with the statistical analysis of past performance. When $\mathcal{S}$HAPER has little performance data, it mostly relies on the preferences; after accumulating more data, it switches to the statistical analysis.

We present the encoding and application of preference rules (Section 10.1), and two learning mechanisms implemented through preference rules (Sections 10.2 and 10.3). Then, we describe storage of generated preference judgments and resolution of conflicts between them (Section 10.4). Finally, we discuss techniques for combining preference rules with statistical selection (Section 10.5).

10.1 Preferences and preference rules

We first describe the syntax and semantics of preference rules (Section 10.1.1), and then overview the main types of these rules (Section 10.1.2).

10.1.1 Encoding and application of rules

We present the encoding of preference rules and their use to compare representations. We begin by defining the notion of a *preference*, which is an instance of representation comparison, and then present a mechanism for generating preferences. Finally, we discuss some constraints that we follow in constructing preference rules.

Add-Preference(*prefer-rep*, *priority*, *certainty*; rep_1, rep_2)
If *prefer-rep*(rep_1, rep_2), then:
Compute the certainty, $cert := certainty(rep_1, rep_2)$.
If $cert > 0.5$,
then add the preference $(rep_1, rep_2, cert, priority(rep_1, rep_2))$;
Else, add the preference $(rep_2, rep_1, 1 - cert, priority(rep_1, rep_2))$.

Figure 10.1: Application of a preference rule; representations rep_1 and rep_2 must be distinct.

Preferences

A *preference* is an expectation that a certain domain representation, rep_1, should be better than another representation, rep_2. The system may rely on this expectation when it does not have past data for the statistical comparison of rep_1 and rep_2.

We represent the reliability of a preference by two values, called priority and certainty. The *priority* is a natural number that serves for resolving conflicts among preferences. If the system gets preferences that contradict each other, it chooses among them according to their priorities. We will give the conflict-resolution algorithm in Section 10.4.3.

The *certainty* is an approximate probability that the expectation is correct, that is, rep_1 is indeed better than rep_2. This probability value must be no smaller than 0.5, because otherwise rep_2 would be preferable to rep_1. The system uses certainties to control the trade-off between exploitation and exploration in early stages of statistical learning (see Section 10.5). Note that certainties are *not* used in conflict resolution.

We denote a preference by a quadruple $(rep_1, rep_2, prior, cert)$, where rep_1 and rep_2 are two specific representations, *prior* is a priority (natural number), and *cert* is a certainty (between 0.5 and 1).

Rule encoding

A *preference rule* is a heuristic for generating preferences, which inputs two distinct representations and determines whether one of them is better than the other.

We encode a rule by an applicability condition, *prefer-rep*(rep_1, rep_2), and two functions, *priority*(rep_1, rep_2) and *certainty*(rep_1, rep_2). The condition determines whether we can use the rule to compare the given representations. When the rule is applicable, the first function returns the priority of the preference, and the second function gives its certainty. If the resulting certainty value is larger than 0.5, then rep_1 is preferable to rep_2; if it is smaller, the system generates the opposite preference. If the certainty is exactly 0.5, the rule is considered inapplicable. We summarize the algorithm for applying a preference rule in Figure 10.1.

Preference rules provide less accurate comparison of representations than statistical selection; in particular, they do *not* account for gain functions, properties of specific problems, or a probability distribution of problems in a domain. These limitations simplify the design and encoding of preference rules, and enable us to develop fast procedures for use of preferences.

We allow the user to add new rules, as well as delete or modify old rules, in the process of solving problems. These operations require the appropriate revisions of previously generated preferences; we discuss them in Section 10.4.1.

Antisymmetry and consistency

A well-designed preference rule should not generate conflicting preferences. In particular, if the application of a rule to some pair of representations produces a preference, then its application to the reversed pair should not produce a different preference.

We may avoid such conflicts by enforcing the following property of the applicability conditions:

> **Antisymmetry**
> For every condition *prefer-rep*, and representations rep_1 and rep_2,
> $\textit{prefer-rep}(rep_1, rep_2)$ is false or $\textit{prefer-rep}(rep_2, rep_1)$ is false.

That is, if a rule is applicable to a pair of representations, then it is *not* applicable to the reversed pair.

Antisymmetry is convenient for avoiding simple conflicts of a rule with itself, but it sometimes proves too restrictive. We may enforce a weaker property, which also prevents conflicts:

> **Consistency**
> For every rule (*prefer-rep*,*priority*,*certainty*), and representations rep_1 and rep_2,
> if $\textit{prefer-rep}(rep_1, rep_2)$ and $\textit{prefer-rep}(rep_2, rep_1)$ are both true, then
> (a) $\textit{priority}(rep_1, rep_2) = \textit{priority}(rep_2, rep_1)$ and
> (b) $\textit{certainty}(rep_1, rep_2) = 1 - \textit{certainty}(rep_2, rep_1)$.

We found these constraints useful for designing preference rules, and used either Antisymmetry or Consistency in each of the experimental domains. Note, however, that these properties are recommendations rather than requirements, and their violation does *not* result in errors or unresolvable conflicts. The user may enforce or disregard either property, at her discretion.

We provide an optional procedure for verifying these properties during the application of preference rules. The human operator may use it to monitor the violations of Antisymmetry or Consistency.

10.1.2 Types of rules

We currently use three types of preference rules in the $\mathcal{S}$HAPER system: user rules, counting rules, and testing rules. User rules are completely specified by the human operator, whereas the other two types allow limited learning.

User rules

We extended the PRODIGY domain language to allow specification of preference rules. This extension enables the human operator to encode her knowledge and combine it with automatic statistical selection.

When encoding a rule, the user provides its applicability condition, priority, and certainty. The condition is a Lisp function that inputs two representations and returns true or false. It may take into account not only the structure of the representations, but also their construction history (see Section 7.1.2).

We built a small library of functions that are often used to construct applicability conditions for PRODIGY representations. In particular, we implemented procedures for detecting identical primary effects, abstraction hierarchies, and control-rule sets. We also provided functions for checking whether a selection of primary effects is a subset of another selection, an abstraction is finer grained than another abstraction, and a control-rule set belongs to another set. We have described these procedures in Section 7.1.1.

The priority specification is a Lisp function that inputs two representations and returns a priority value. If the priority of a preference rule does not depend on the specific representations, the user may specify it by a natural number rather than a function. If the user does not specify priority, the system uses the default value. For user rules, this default is 0, the lowest priority.

The certainty is also specified by a Lisp function or a number; certainty values must be reals between 0 and 1. If the user does not provide this specification, the system uses the default value, which is presently set to 2/3.

The user may provide preference rules for specific domains, as well as multi-domain rules. She may restrict the applicability of every multi-domain rule to a certain group of domains, by coding a Lisp function that inputs a pointer to a domain and determines whether the rule is applicable in this domain.

Counting and testing rules

If the user specifies the applicability condition of a rule, the system can automatically learn its certainty. If the resulting certainty is larger than 0.5, then the first representation in every matching pair is preferable to the second one; otherwise, the system prefers the second representation. We use two different mechanisms for learning certainties, which give rise to counting rules and testing rules.

A *counting rule* is a simple mechanism for computing certainty from past performance data. When the system applies such a rule, it identifies the pairs of old representations that match the rule's condition, and uses statistical analysis of past performance to compare representations in these pairs. It determines the percentage of pairs in which the first representation gives larger gains than the second one, and uses this percentage as the rule's certainty. For example, suppose that the procedure has identified four pairs of old representations that match the applicability condition, and determined that the first representation is better than the second one in three of these pairs. Then, the system sets the rule's certainty to 0.75.

A *testing rule* compares representations by their performance on a collection of test problems. When two representations match the condition of a testing rule, the system applies them to solve a sequence of small problems, and determines the percentage of problems on which the first representation outperforms the second one. This percentage becomes the certainty of the resulting preference. For example, if the first representation has won on three out of four test problems, the system prefers it to the second one with certainty 0.75. We must ensure that test problems take much less time than real problems. Otherwise, the time saved by making the right selection may prove smaller than the time spent for testing.

The user should specify not only applicability conditions but also priorities of counting and testing rules. By default, priorities of counting rules are higher than user-rule priorities and lower than testing-rule priorities. A rule may be used in a specific domain or in multiple domains. If a counting rule is applicable to multiple domains, then the system determines its certainty based on the data accumulated in all domains. We now give a more detailed description of counting and testing rules, and present algorithms for computing their certainty.

10.2 Counting rules

Suppose that the human operator has identified some condition relevant to the efficiency of representations, but she does not know whether it implies good or poor performance. For example, an experienced operator should know that abstraction affects efficiency; however, she may not know whether it helps or hurts in a specific domain.

Then, she may encode a *counting preference rule* based on this condition. When the human operator provides such a rule, she does not specify the direction and certainty of preferences. The system automatically determines the certainty, after accumulating some performance data. To compute the rule's certainty, SHAPER identifies the matching pairs of old representations, and counts the pairs in which the first representation gives larger gains than the second one.

Counting rules are usually more reliable than user rules; however, the system cannot apply them until it accumulates relevant data. We describe the computation of the certainties and priorities of counting rules.

Certainty computation

To find the certainty of a counting rule, the system identifies all pairs of representations that match the rule's condition. For each pair, it tries to determine which of the two representations performed better in the past. When the available data are not sufficient for identifying the better representation, SHAPER skips the pair.

Suppose that the first representation wins in n_w pairs and loses in n_l pairs. If neither of these values is zero, we compute the rule's certainty as $\frac{n_w}{n_w+n_l}$. If n_l is zero, we compute it as $\frac{n_w}{n_w+1}$ rather than using the "absolute" certainty of 1. Similarly, if n_w is zero, the certainty is $\frac{1}{1+n_l}$. If the data are not enough for determining the better representation in *any* pair, then both n_w and n_l are zero, and the system does *not* use the rule until it accumulates more data.

Comparison of representations

The comparison of the representations in a pair is based on their expected gains. The system uses past data to estimate the optimal time bound and the mean gain for this bound (see Sections 8.3 and 8.4), and chooses the representation with the larger gain. If the two gains prove equal, $\mathcal{S}$HAPER skips the pair. We do *not* use problem sizes in comparing representations; thus, gains are averaged over all possible sizes. Since the system finds the optimal expected gains in the course of statistical learning, it does not have to run statistical computations when applying counting rules.

By default, the system does not test statistical significance of the difference between expected gains. This default allows the computation of certainty with little performance data, in early stages of learning. The user has an option to enforce a significance test. She may specify the required confidence value for a domain, as well as for individual counting rules. When the difference does not prove significant, $\mathcal{S}$HAPER concludes that the available data are insufficient and skips the pair.

Counting procedures

We summarize the procedures for use of counting rules in Figure 10.2. The *Add-Pairs* procedure identifies the pairs of representations that match the conditions of counting rules. For every condition *prefer-rep*, it stores the matching pairs in *pairs*[*prefer-rep*]. When the system adds a new representation, it invokes this procedure to determine new matching pairs.

The *Count-Certainty* procedure computes the certainty of a specific counting rule. For every matching pair, it calls *Use-Pair* to compare the two representations. Then, it counts the number of wins and losses by the first representation, and determines the percentage of wins. When the system gets new performance data, it invokes *Count-Certainty* to recompute rule certainties, and then updates the certainties of all generated preferences.

The *Use-Pair* procedure inputs two representations and compares their expected gains, computed for the optimal time bounds. If the comparison does not pass the statistical z-test, the procedure returns tie. The minimal passing z-value, denoted *min-z*, depends on the user's confidence requirement. By default, the system sets *min-z* to zero and does not check statistical significance.

Global certainty

When $\mathcal{S}$HAPER works with multiple domains, it creates a *separate* representation space for every domain. We apply control heuristics and learning techniques to individual domains, and usually do *not* transfer learned knowledge across domain boundaries. Only two of $\mathcal{S}$HAPER's learning mechanisms transfer experience among different domains. We now describe one of them, for computing the certainty of counting rules. The other mechanism, presented in Section 11.1, is for analyzing the performance of description changers.

Suppose that a counting rule is applicable in multiple domains, and the current domain does not have enough data for the certainty computation. If the system has computed the rule's certainties in some other domains, it can determine a *global* certainty of the rule. The system uses this certainty until accumulating domain-specific data, and then computes a *local* certainty.

Suppose that the rule's certainty is strictly larger than 0.5 in m_w domains and strictly smaller than 0.5 in m_l domains. That is, m_w is the number of domains where the system

Add-Pairs(*new-rep*)
For every counting preference rule, with condition *prefer-rep*:
For every active old representation, *old-rep*:
If *prefer-rep*(*new-rep*, *old-rep*),
then *pairs*[*prefer-rep*] := *pairs*[*prefer-rep*] $\cup$ {(*new-rep*, *old-rep*)}.
If *prefer-rep*(*old-rep*, *new-rep*),
then *pairs*[*prefer-rep*] := *pairs*[*prefer-rep*] $\cup$ {(*old-rep*, *new-rep*)}.

Count-Certainty(*prefer-rep*)
The input is the condition of a counting rule.

Set the initial values:
$n_w := 0$ (number of wins by rep_1)
$n_l := 0$ (number of losses by rep_1)

For every pair (rep_1, rep_2) *in pairs*[*prefer-rep*]*:*
Call *Use-Pair*(rep_1, rep_2).
If it returns win, then $n_w := n_w + 1$.
If it returns loss, then $n_l := n_l + 1$.

Compute the certainty:

$$\frac{\min(n_w, 1)}{\min(n_w, 1) + \min(n_l, 1)}$$

Use-Pair(rep_1, rep_2)

Determine the gain for the first representation:
If rep_1 has no performance data, then return tie.
Else, find the expected gain, g_1, and its deviation, σ_1.

Determine the gain for the second representation:
If rep_2 has no performance data, then return tie.
Else, find the expected gain, g_2, and its deviation, σ_2.

Compare the gains:
If $\frac{|g_1 - g_2|}{\sqrt{\sigma_1^2 + \sigma_2^2}} \leq$ *min-z*, then return tie.
If $g_1 > g_2$, then return win; else, return loss.

Figure 10.2: Use of counting rules: Identification of matching pairs and certainty computation.

prefers the first representation of a matching pair, and m_l is the number of domains with the opposite preference. Then, the global certainty is $\frac{m_w}{m_w+m_l}$. If m_w or m_l is zero, we treat it in the same way as in the local-certainty computation. If m_l is zero, then the certainty is $\frac{m_w}{m_w+1}$; if m_w is zero, then it is $\frac{1}{1+m_l}$; finally, if both m_w and m_l are zero, the system does not compute a global certainty.

When the user encodes a rule, she may specify its initial certainty. Then, the system uses this certainty until it accumulates enough data for computing a global certainty. When the rule's certainty switches from initial to global, the system updates the certainties of all generated preferences. When the certainty in the current domain switches from global to local, it also updates the preference certainties.

User-specified priorities

The priority of a counting rule may depend on whether the system uses global or local certainty. Accordingly, the user may provide two priority functions.

When the system uses global certainty, it applies the first function to compute the priorities of preferences. After switching to local certainty, it applies the second function to re-compute their priorities. The re-computation must not reduce the priorities of preferences. In other words, the priority value for the local certainty must be no smaller than that for the global certainty. When the system detects a violation of this condition, it signals a warning.

Default priorities

We first describe the computation of the default priority for global certainties. This default value is greater than the priorities of all user rules in the domain. To compute it, the system determines the maximum of user-rule priorities, and adds 1.

Since the user may specify priorities by functions, their maximal values may not be known. In this case, the system determines the maximal priority of the generated preferences, and uses it instead of the overall maximum. If some user rule generates a preference with a greater priority, the system increases the maximum and then updates the priorities of counting preferences.

The computation of the default priority for local certainties is similar. Its value is greater than all user-rule priorities and all global-certainty priorities. Thus, the system determines the maximum of all these priorities, and increases it by 1.

10.3 Testing rules

We next describe *testing rules*, which provide a mechanism for evaluating the performance of representations on small problems, before solving real problems. The user has to provide the applicability conditions of testing rules (see Section 10.1.1). The system then compares the performance of matching representations on test problems. These rules are usually more reliable than counting and user rules, though less reliable than statistical selection.

The main drawback of testing rules is their computational cost. We must ensure that the time for solving test problems is smaller than the resulting savings on real problems.

Note that we *cannot* amortize the computational cost over a long sequence of real problems, because the system discards all preferences after the initial stage of statistical learning.

We present a procedure that applies testing rules and computes the certainty of the resulting preferences. We then discuss the role of the human operator and her options in tuning this procedure.

Certainty computation

When two representations match a testing rule, the system compares their performance on a series of test problems. For each problem, it determines which representation gives the greater gain. If the first representation wins n_w times and loses n_l times, the certainty of preferring it is $\frac{n_w}{n_w+n_l}$; however, if $n_l = 0$, the system sets the certainty to $\frac{n_w}{n_w+1}$ rather than 1; similarly, if $n_w = 0$, the certainty is $\frac{1}{1+n_l}$.

We summarize the certainty computation in Figure 10.3. Note that the procedure does *not* count ties, for example, when both representations hit the time bound. It also disregards the problems that result in rejection by either representation.

If SHAPER applies a testing rule several times, it uses the same collection of problems for all applications. It may use different collections for different rules, or the same common collection for all rules, at the user's discretion. The system stores the results of solving test problems, for every representation. If later comparisons involve the same representation, it retrieves old results rather than solving the same problems anew.

Test problems

The human operator has to supply test problems. For every rule, she specifies either a collection of problems or a procedure that generates them on demand. She should ensure that the problems take little computation and provide a fair performance comparison.

The system has *no* means for producing test problems automatically. Development of a domain-independent generator of sample problems is an unexplored direction of AI research, relevant to many learning systems.

Gain function

By default, the gain function for test problems is the same as the function for real problems. The user has an option to provide a different gain measure. Sometimes, this option allows us to improve the testing accuracy.

If the real gain function is not known during testing, the system uses the following linear function, where the reward R is strictly larger than the time bound:

$$gain(prob, time, result) = \begin{cases} R - time, & \text{if success} \\ -time, & \text{if failure} \\ -R, & \text{if interrupt} \end{cases}$$

According to this function, success always gives a positive gain. We have not defined gain for rejections, because the system does not use them in comparing representations.

Test-Certainty$(rep_1, rep_2, probs)$
The input includes two representations, rep_1 and rep_2,
and a collection of test problems, $probs$.

Set the initial values:
$n_w := 0$ (number of wins by rep_1)
$n_l := 0$ (number of losses by rep_1)

For every prob in probs:
Determine a time bound B for solving $prob$.
Call $\textit{Use-Problem}(rep_1, rep_2, prob, B)$.
If it returns win, then $n_w := n_w + 1$.
If it returns loss, then $n_l := n_l + 1$.

Compute the certainty:

$$\frac{\min(n_w, 1)}{\min(n_w, 1) + \min(n_l, 1)}$$

Use-Problem$(rep_1, rep_2, prob, B)$

Test the first representation:
Apply rep_1 to $prob$, with the time bound B.
If the result is a rejection, then return tie.
Else, determine the problem-solving gain, g_1.

Test the second representation:
Apply rep_2 to $prob$, with the time bound B.
If the result is a rejection, then return tie.
Else, determine the problem-solving gain, g_2.

Compare the gains:
If $g_1 > g_2$, then return win.
If $g_1 < g_2$, then return loss.
Else, return tie.

Figure 10.3: Computing the preference certainty for a testing rule.

Number of problems

We use a knob to specify the default number of problems for testing representations. Its current value is 10; that is, when two representations match a testing rule, the system compares them on ten problems. If the number of available problems is smaller, the system uses all of them. When the user encodes a testing rule, she may specify a different number of tests.

Default time bounds

We next describe the computation of time bounds for test problems. These bounds must ensure that the testing time is much smaller than the time for solving real problems.

$\mathcal{S}$HAPER determines the initial bound B for real problems (see Section 8.5.1), and limits the testing time by a certain fraction of B. The coefficient for computing this limit through B is a knob, currently set to 0.5; that is, the testing-time limit is $T = 0.5 \cdot B$.

When the system compares two representations on n problems, it sets the time bound to $\frac{T}{2 \cdot n}$, which ensures that the overall testing time is at most T. If $\mathcal{S}$HAPER solves some problems faster, it may spend more time on the remaining tests, so it re-sets the bound accordingly. If the system has solved some problems during previous comparisons, it retrieves the past results rather than solving them anew, and allocates more time for the other problems.

User-specified bounds

The user has an option to provide different default bounds for specific domains, as well as explicit bounds for individual testing rules. For each rule, she may specify either a fixed bound or a function that computes a bound for each problem.

If the user provides a time bound and does *not* specify the number of test problems, the system solves problems until it reaches the time limit T. It stops at T even if it has used less than ten problems. If the user sets both a time bound and a problem number, then the system disregards the limit T.

Default priority

The default priority of testing rules is larger than all priorities of counting and user rules. To compute this default, the system determines the maximum of all counting and user priorities, and increases it by 1. If the system later revises the estimated maximum, then it updates the priorities of all testing preferences. The update procedure is the same as for the priorities of counting rules (see Section 10.2).

10.4 Preference graphs

A *preference graph* is a data structure for storage, efficient access, and analysis of generated preferences. It is a directed graph, whose nodes are representations or groups of representations, and edges are preferences.

For every problem domain, the system maintains a *full preference graph* and *reduced preference graph.* The first graph comprises *all* preferences, whereas the second graph is the result of pruning duplicate preferences and resolving conflicts. The system uses the reduced graph in selecting preferable representations and in controlling an early stage of statistical learning (see Section 10.5).

We first describe the full graph and its use for storing preferences (Section 10.4.1). We then present the reduced graph (Section 10.4.2) and give algorithms for its construction (Sections 10.4.3 and 10.4.4).

10.4.1 Full preference graph

The system stores the results of applying preference rules in the *full preference graph.* The available representations are nodes of this graph, and all generated preferences are its edges. Note that the full graph may contain multiple edges connecting the same pair of nodes.

Some representations in the full graph may be *inactive* (see Sections 7.2.2 and 7.2.3), which means that they are no longer used in problem solving. We do *not* compare an inactive representation with other representations; however, we use its incoming and outgoing preference edges to identify transitive preferences (see Section 10.4.3). If an inactive representation has no incoming or no outgoing edges, we remove it from the graph.

The system modifies the full preference graph after adding a new representation, inactivating an old representation, adding a new preference rule, or deleting or modifying an old rule. We now consider each of these modifications.

Adding and inactivating representations

When $\mathcal{S}$HAPER generates a new representation (see the algorithm in Figure 7.3), it adds the corresponding node to the full preference graph. The system applies the available preference rules to compare the new representation with active old representations and adds appropriate preferences. We give pseudocode for this procedure, called *Add-Rep-Node*, in Figure 10.4; the procedure calls the *Add-Preference* function, presented in Figure 10.1.

If an old representation becomes inactive, the system checks whether the corresponding node of the preference graph has both incoming and outgoing edges. If not, the system removes the node from the graph.

Adding and removing preference rules

When the user adds a new preference rule, the system applies it to compare active representations and adds the resulting preferences. If the user removes an old rule, $\mathcal{S}$HAPER deletes the corresponding preferences and then removes inactive nodes that no longer have incoming or outgoing edges. In Figure 10.4, we present these procedures, called *Add-Pref-Rule* and *Remove-Pref-Rule.*

When applying a preference rule, the system stores pointers from the rule to the resulting preferences. These pointers allow fast identification of all preferences that correspond to the rule, which improves the efficiency of the rule removal.

Add-Rep-Node(*new-rep*)
For every active old representation, *old-rep*:
For every preference rule, (*prefer-rep*, *priority*, *certainty*):
Call *Add-Preference*(*prefer-rep*, *priority*, *certainty*; *new-rep*, *old-rep*).
Call *Add-Preference*(*prefer-rep*, *priority*, *certainty*; *old-rep*, *new-rep*).

Add-Pref-Rule(*prefer-rep*, *priority*, *certainty*)
For every active representation, rep_1:
For every other active representation, rep_2:
Call *Add-Preference*(*prefer-rep*, *priority*, *certainty*; rep_1, rep_2).

Remove-Pref-Rule
For every preference (rep_1, rep_2, *prior*, *cert*) that corresponds to this rule:
Remove this preference from the preference graph.
If rep_1 is inactive, and it has no outgoing edges,
then remove the representation rep_1 from the graph.
If rep_2 is inactive, and it has no incoming edges,
then remove the representation rep_2 from the graph.

Figure 10.4: Modification of the full preference graph.

Modifying preference rules

If the user modifies the condition of a rule, the system treats it as a removal of the old rule and addition of a new one. That is, it deletes the corresponding old preferences and then applies the modified rule to all pairs of active representations.

On the other hand, if the user makes changes to the priority and certainty function of a rule, and does *not* alter the condition, then the system modifies the corresponding old preferences rather than re-applying the rule. In particular, it may have to change the directions of some preferences.

10.4.2 Reduced preference graph

The full graph comprises *all* generated preferences, which may form conflicts. Some of the preferences may be unreliable because of low certainty. The system pre-processes the graph before using it to select representations: it prunes the low-certainty preferences, resolves conflicts, and identifies groups of equally preferable representations. These operations result in constructing a new graph, called the *reduced preference graph*.

Main features

We begin by listing the main properties of the reduced graph, which differentiate it from the full graph and allow efficient selection of preferable representations.

First, the nodes of the reduced graph are *groups* of representations. When the system judges several representations equally preferable, it combines them in a group and views it as a single node. The edges of the graph are preferences between groups.

Second, the certainties of all preferences are above some pre-set threshold, which equals 2/3 in the current implementation. Moreover, the graph has *no* multiple edges connecting the same pair of nodes. That is, for every two nodes, there is at most one preference edge between them.

Third, the reduced graph is *acyclic*, that is, preference edges do *not* form loops. The system eliminates loops in two steps: it removes the edges that conflict with higher-priority preferences and then combines equal-preference representations into groups. Edges of the reduced graph have certainties, but no priorities. We discard priorities because their only purpose is resolving conflicts, and the graph has no conflicting preferences. We denote a preference without a priority by a triple $(rep_1, rep_2, cert)$.

Outline of the construction

We next outline the main operations for converting the full preference graph into the reduced graph, which include pruning low-certainty and duplicate edges, resolving conflicts, and identifying equally preferable representations.

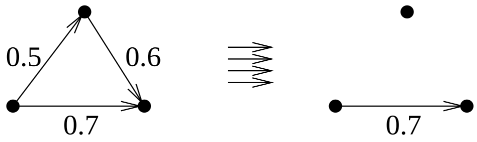
Low-certainty edges If the certainty of some edges is smaller than the 2/3 threshold, the system prunes them, as illustrated in Figure 10.5(a).

Multiple edges If several preference edges connect the same two nodes in the same direction, then the system keeps the highest-priority edge and prunes the other edges. If several edges have this highest priority, the system keeps the one with the highest certainty. Note that the priority takes precedence: the system prunes lower-priority edges even if they have larger certainty. We show the deletion of multiple edges in Figure 10.5(b), where thicker edges have higher priorities and numbers show certainties.

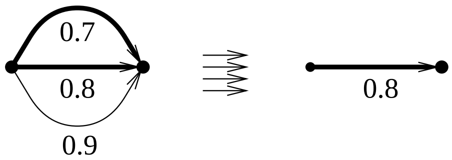
Simple conflicts If two opposite edges with *different* priorities connect the same two nodes, we say that they form a *simple conflict*. The system resolves it by removing the lower-priority edge, as shown in Figure 10.5(c). If opposite edges have the *same* priority, the system does *not* remove either of them.

Path conflicts Suppose that, for some preference edge $(rep_1, rep_2, prior, cert)$, there is an opposite multi-edge path from rep_2 to rep_1, and the priorities of all edges in this path are *strictly greater* than *prior*. We call this situation a *path conflict* and resolve it by pruning the edge from rep_1 to rep_2, as illustrated in Figure 10.5(d). We distinguish between simple and path conflicts for efficiency reasons: it enables us to develop faster algorithms, described in Section 10.4.3.

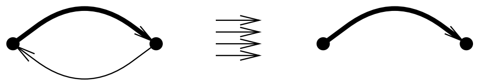
Preference groups The preference graph may have loops even after resolving all simple and path conflicts. We consider the representations in a loop to be equally preferable. The system identifies groups of equally preferable representations and then determines preferences between these groups. We give an example of such groups in Figure 10.5(e).



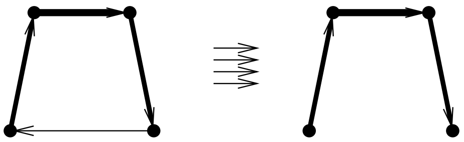
(a) Low-certainty edges.



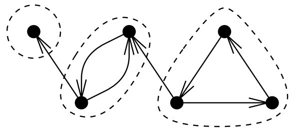
(b) Multiple edges.



(c) Simple conflict.



(d) Path conflict.



(e) Preference groups.

Figure 10.5: Main operations for constructing the reduced preference graph: pruning low-priority and multiple edges, resolving conflicts, and identifying preference groups. We show higher-priority preferences by thicker lines and specify certainties by numbers.

10.4.3 Constructing the reduced graph

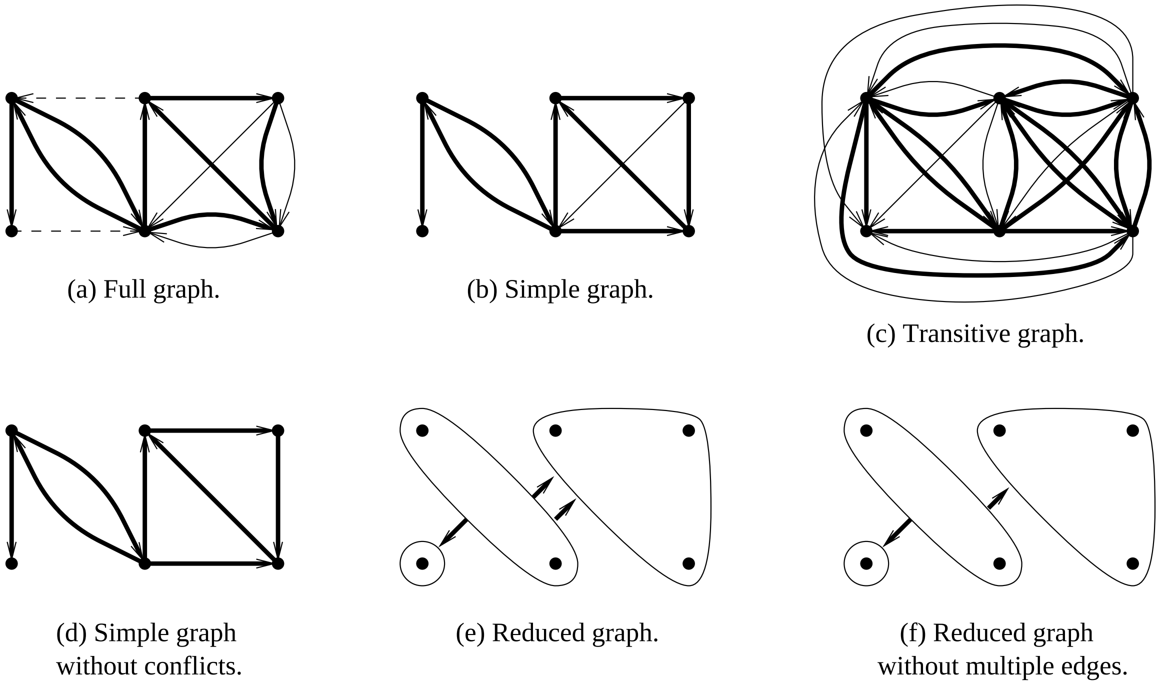
We describe a technique for constructing the reduced preference graph, which involves the generation of two intermediate graphs, called *simple* and *transitive* graphs. The first intermediate graph is the result of pruning "undesirable" edges. The second graph is the transitive closure of preferences, which serves to identify path conflicts. We illustrate the main steps of the construction in Figure 10.6.

The algorithms for building the reduced graph rely on the priorities of edges and make little use of certainties. We have restricted the use of certainties because, in most cases, they are only a rough approximation of actual probabilities. A more advanced treatment of certainties is an open problem.

Simple graph

The construction begins with the removal of low-certainty edges, multiple edges, and simple conflicts (see Figures 10.5a–c). In Figure 10.7, we give an algorithm that performs the removal and outputs the resulting new graph, called the *simple preference graph* (see Figure 10.6b). The nodes of the new graph are the same as in the full graph, and its edges are a subset of the full graph's edges.

The algorithm uses two matrices, *max-prior* and *max-cert*, indexed on the representations.



(a) Full graph. (b) Simple graph. (c) Transitive graph.

(d) Simple graph without conflicts. (e) Reduced graph. (f) Reduced graph without multiple edges.

Figure 10.6: Steps of generating the reduced graph. We show preferences with certainty at least 2/3 by solid lines, and low-certainty preferences by dashed lines; thicker lines denote higher priorities.

The first matrix is for computing the maximal priority of multiple preference edges, and the second is for the maximal certainty of the highest-priority edges.

To analyze the time complexity, we denote the number of representations by k and the number of preferences in the full graph by m_{f}. The complexity of Steps 1 and 4 is $O(k^2)$ and that of Steps 2 and 3 is $O(m_{\mathrm{f}})$. In the implementation, we replaced the matrices with more complex structures and reduced the complexity of Steps 1 and 4 to $O(\min(k^2, m_{\mathrm{f}}))$, but the practical reduction of running time proved insignificant.

Note that the removal of multiple edges and simple conflicts is an efficiency measure. Skipping it would not violate the correctness of the subsequent construction, but may significantly slow down the computation of the transitive closure.

Transitive graph

The identification of path conflicts (see Figure 10.5d) is based on the construction of the *transitive preference graph*, which is the transitive closure of the simple graph (see Figure 10.6c). The nodes of this new graph are the same as in the simple graph, whereas its edges are a superset of the simple graph's edges, defined as follows:

> For every two representation nodes rep_1 and rep_2,
> if the simple graph has some path from rep_1 to rep_2,
> then the transitive graph has an edge from rep_1 to rep_2.

Edges of the transitive graph have priorities, but no certainties. Their priorities are determined by the path priorities in the simple graph. The priority of a path is the *minimal*

Build-Simple-Graph

1. *Set the initial values:*
   For every representation rep_1:
   For every other representation rep_2:
   $max\text{-}prior[rep_1, rep_2] := 0$
   $max\text{-}cert[rep_1, rep_2] := 0$

2. *Compute the maximal priorities:*
   For every preference $(rep_1, rep_2, prior, cert)$ in the full graph:
   If $prior > max\text{-}prior[rep_1, rep_2]$ and $cert \geq 2/3$,
   then $max\text{-}prior[rep_1, rep_2] := prior$.

   (For every rep_1 and rep_2, the $max\text{-}prior[rep_1, rep_2]$ value is now equal to the maximal priority of preference edges from rep_1 to rep_2.)

3. *Compute the maximal certainties:*
   For every preference $(rep_1, rep_2, prior, cert)$ in the full graph:
   If $prior = max\text{-}prior[rep_1, rep_2]$ and $cert > max\text{-}cert[rep_1, rep_2]$,
   then $max\text{-}cert[rep_1, rep_2] := cert$.

   (For every rep_1 and rep_2, the $max\text{-}cert[rep_1, rep_2]$ value is now equal to the maximal certainty of the highest-priority preference edges from rep_1 to rep_2.)

4. *Build the simple graph:*
   Create a graph with the same representations as in the full graph, and no preferences.
   For every representation rep_1:
   For every other representation rep_2:
   If $max\text{-}cert[rep_1, rep_2] \geq 2/3$ (certainty above the threshold)
   and $max\text{-}prior[rep_1, rep_2] \geq max\text{-}prior[rep_2, rep_1]$ (no simple conflict),
   then add the preference $(rep_1, rep_2, max\text{-}prior[rep_1, rep_2], max\text{-}cert[rep_1, rep_2])$.

Remove-Conflicts

For every preference $(rep_1, rep_2, prior, cert)$ in the simple graph:
If the transitive graph has an edge from rep_2 to rep_1 and its priority is larger than $prior$,
then remove the preference $(rep_1, rep_2, prior, cert)$ from the simple graph.

Figure 10.7: Construction of the simple preference graph.

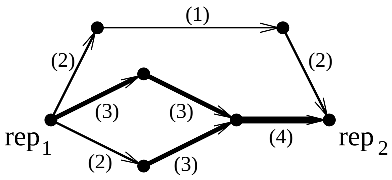


Figure 10.8: Example of a transitive preference, where the numbers denote edge priorities (not certainties). The priorities of paths from rep_1 to rep_2 are 1, 2, and 3. The priority of the resulting transitive preference is the maximum of the path priorities, which is 3.

priority of its edges, and the priority of the transitive edge from rep_1 to rep_2 is the *maximum* of the priorities of paths from rep_1 to rep_2. For example, the graph in Figure 10.8 has three paths from rep_1 to rep_2, two of which share an edge. Their priorities are 1, 2, and 3, and the priority of the resulting transitive edge is 3.

The construction of a transitive graph and computation of its edge priorities is a special case of the *all-pairs algebraic-path problem* [Carre, 1971; Lehmann, 1977], solved by generalized shortest-path algorithms (for example, see the textbook by Cormen *et al.* [1990]). If the simple graph has k representations and m_s preferences, the complexity of constructing the transitive graph is $O(k^2 \cdot \log k + k \cdot m_s)$, which is the most time-consuming step in generating the reduced preference graph.

The system uses the transitive graph to detect path conflicts of the simple graph. We give an algorithm for resolving path conflicts in Figure 10.7, and illustrate the results of its execution in Figure 10.5(d). The complexity of the algorithm is $O(m_s)$.

Reduced graph

After removing conflicts, the system identifies groups of equally preferable representations (see Figure 10.6e). By definition, these groups are strongly connected components of the simple graph, and the complexity of their identification is $O(k + m_s)$ (see the textbook by Cormen *et al.* [1990]).

The system creates the *reduced graph*, whose nodes are the resulting groups and whose edges are *all* between-group edges of the simple graph, as shown in Figure 10.6(e). Finally, the system removes multiple edges of the resulting graph, using the same algorithm as for constructing the simple graph (see Figure 10.7), which takes $O(m_s)$ time. We show the resulting graph in Figure 10.6(f). After constructing this graph, the system discards edge priorities, since they are no longer needed for conflict resolution.

The overall time complexity of all steps, including the construction of the simple, transitive, and reduced graphs, is $O(m_f + k^2 \cdot \log k + k \cdot m_s)$, where k is the number of representations, m_f is the number of edges in the full graph, and m_s is the number of edges in the reduced graph.

10.4.4 Modifying the reduced graph

We described the algorithms for updating the full preference graph in Section 10.4.1. They include addition and inactivation of representations, as well as addition, removal, and mod-

Update-Transitive-Graph(*new-rep*)
For every old representation *old-rep*$_1$:
  For every other old representation *old-rep*$_2$:
    If there are transitive preferences from *old-rep*$_1$ to *new-rep* and from *new-rep* to *old-rep*$_2$:
      Let *prior* be the smaller of the priorities of these two preferences.
      If there is no preference from *old-rep*$_1$ to *old-rep*$_2$,
        then add this preference and set its priority to *prior*.
      If there is a preference from *old-rep*$_1$ to *old-rep*$_2$ and its priority is below *prior*,
        then set the priority of this preference to *prior*.

Figure 10.9: Updating preference edges between old nodes of the transitive graph.

ification of preference rules. When the system updates the full graph, it has to propagate the changes to the simple, transitive, and reduced graphs.

We have not developed algorithms for propagating the results of addition, removal, or modification of preference rules. If the system makes any of these changes to the full preference graph, it constructs the other graphs from scratch. Usually, these changes occur much less frequently than addition and inactivation of representations, and their efficiency is not essential to the overall performance.

If the system adds a new representation to the full graph (see the *Add-Rep-Node* algorithm in Figure 10.4), it updates the other graphs rather than constructing them anew. First, $\mathcal{S}$HAPER adds the new node to the simple graph and determines its incoming and outgoing edges, using a procedure similar to *Build-Simple-Graph* in Figure 10.7. This procedure processes only the newly added preference edges of the full graph, which are adjacent to the new node. We denote the number of these new edges by m_{new}; the time complexity of the procedure is $O(m_{\text{new}})$.

After modifying the simple graph, $\mathcal{S}$HAPER updates the transitive graph, in two steps. First, it computes all incoming and outgoing transitive edges of the new node, and their priorities. This computation is a special case of the *single-source algebraic path problem,* which takes $O(k + m_s)$ time. Second, the system updates transitive edges between old representations, using the procedure in Figure 10.9. The complexity of this procedure is $O(k^2)$.

Then, $\mathcal{S}$HAPER resolves path conflicts in the simple graph, using the *Remove-Conflicts* algorithm in Figure 10.7; it takes $O(m_s)$ time. Finally, the system constructs the reduced graph; it performs this construction from scratch, in $O(k+m_s)$ time. The overall complexity of adding the new representation to the simple, transitive, and reduced graphs is $O(m_{\text{new}} + k^2)$.

If an old representation becomes inactive, the system may remove it from the full preference graph (see Section 10.4.1). When it happens, $\mathcal{S}$HAPER also removes the inactive representation and its adjacent edges from the other preference graphs. The complexity of this operation is $O(k)$.

10.5 Use of preferences

A well-constructed set of preference rules can serve as an autonomous mechanism for selecting representations; however, its effectiveness would depend on the human operator, who provides the rules. Moreover, it would disregard some relevant information, such as specific gain functions and similarity among problems. We now describe a combination of preferences with statistical learning, which eliminates these limitations and gives better results than either mechanism alone.

Spreading the exploration

If we use statistical learning without preference rules, the system begins with eager exploration: it accumulates performance data for all available representations (see Section 8.5.4). This strategy allows early identification of effective representations, but it usually results in large initial losses. The system amortizes these losses over subsequent problems. If the domain has many more problems than representations, the initial exploration pays off. On the other hand, if we end up solving only a few problems, the exploration may significantly reduce the overall gain.

The use of preference rules allows us to *spread* the exploration process, rather than trying all representations early on. If the system solves a long sequence of problems, the overall exploratory losses are no smaller than in the eager exploration; however, they are not concentrated at the beginning of the sequence.

For each new problem, the system needs to choose between exploring a new representation and using a representation with known performance. In case of exploration, it has to choose among the unexplored representations. The system makes these choices by analyzing the reduced preference graph. We describe an algorithm for making these choices and then discuss the resulting exploration process.

Choosing a representation

We consider a representation *unexplored* as long as the system uses it with the initial time bound, before accumulating enough data for statistical learning (see Section 8.5.4). A node in the reduced graph is unexplored if it contains at least one unexplored representation.

The system identifies the unexplored nodes that do *not* have incoming edges from other unexplored nodes. We call them *fringe nodes;* intuitively, they are the most preferable among unexplored nodes. We illustrate this notion in Figure 10.10, where white circles are unexplored nodes and grey circles are the other nodes. We mark the fringe nodes by thick lines.

Suppose that some fringe nodes do not have *any* incoming edges; that is, the preferences give *no* evidence that these nodes are worse than any other nodes. For example, the graph in Figure 10.10(a) has two such nodes. Then, the system picks one of the unexplored representations in these nodes and uses it in problem solving. As long as the graph has such fringe nodes, the system eagerly explores their representations and does not use statistical selection.

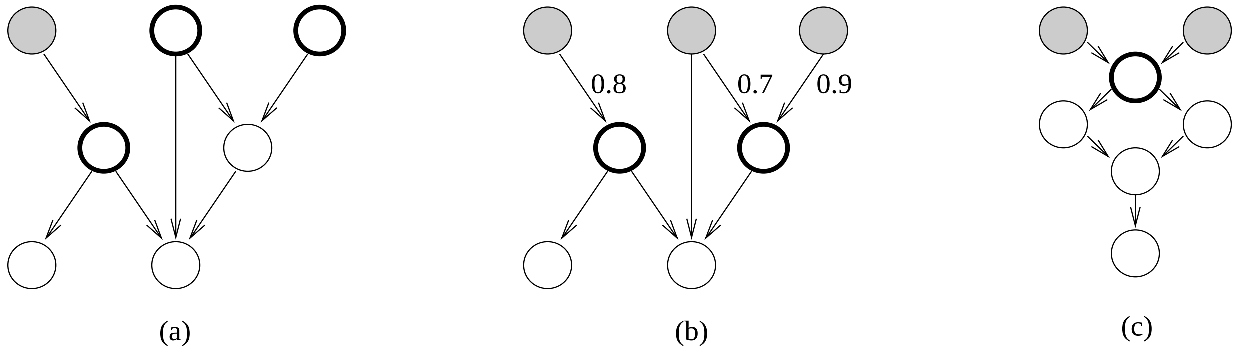


Figure 10.10: Fringe nodes in the reduced preference graph. We show unexplored nodes by white circles, mark fringe nodes among them by thicker lines, and specify the certainties of some preferences by numbers.

Now suppose that all fringe nodes have incoming edges, as shown in Figure 10.10(b). For every fringe node, the system computes the maximal certainty of incoming preference edges; this value is an estimated probability that the node's representations are less effective than the best representation with known performance. In Figure 10.10(b), these estimates are 0.8 (left) and 0.9 (right).

The system chooses the fringe node with the lowest estimate. Intuitively, it is the most promising of the unexplored nodes. If there are several lowest-estimate nodes, the system picks one of them. In our example, it selects the node whose estimate is 0.8.

Finally, the system decides between using a representation with known performance and trying an unexplored representation. It makes a weighted random decision; the probability of using a known representation is equal to the estimate of the chosen fringe node. In our example, this probability is 0.8. If the random decision favors a representation with known performance, the system invokes the statistical-learning mechanism to select one such representation. Otherwise, it uses one of the unexplored representations from the chosen fringe node.

We summarize the selection procedures in Figure 10.11. *Estimate-Node* inputs an unexplored node, checks whether it is on the fringe, and determines the node's estimate, that is, the maximal certainty of incoming preferences. If the input node is *not* on the fringe, it returns 1. If the node has no incoming preferences, it returns 0. *Choose-Node* identifies the fringe node with the lowest estimate. If the graph has no unexplored nodes, it returns an estimate of 0. Finally, *Choose-Rep* makes the weighted random decision and then chooses a representation.

For every new problem, the system invokes the *Choose-Rep* procedure to select a representation. Thus, $\mathcal{S}$HAPER interleaves statistical learning with exploration. We next outline the resulting learning process.

Exploration process

When $\mathcal{S}$HAPER faces a new domain, it eagerly experiments with preferable representations, until it accumulates initial data for all nodes that do not have incoming preference edges.

Choose-Rep
Call *Choose-Node* to select *chosen-node* and determine *min-estimate*.
With probability *min-estimate*,
  invoke statistical learning to choose among explored representations.
Else, pick an unexplored representation from *chosen-node*.

Choose-Node
chosen-node := none; *min-estimate* := 1
For every unexplored node, *unexp-node*:
  estimate := *Estimate-Node*(*unexp-node*)
  If *estimate* < *min-estimate*,
    then *chosen-node* := *unexp-node*; *min-estimate* := *estimate*.
  If *estimate* = 0,
    then return (*chosen-node*, 0) (it has no incoming preferences).
Return (*chosen-node*, *min-estimate*) (it has some incoming preferences).

Estimate-Node(*unexp-node*)
estimate := 0
For every incoming preference, (*other-node*, *unexp-node*, *cert*):
  If *other-node* is unexplored,
    then return 1 (*unexp-node* is not a fringe node).
  If *cert* > *estimate*,
    then *estimate* := *cert*.
Return *estimate* (*unexp-node* is a fringe node).

Figure 10.11: Selection among available representations.

Then, the system begins to interleave the exploration with statistical learning. It experiments with less and less promising representations, and eventually collects initial data for all representations. The exploration process slows as the system moves to less preferable nodes.

The speed of the exploration depends on the conditions and certainties of preference rules. If the preferences provide a near-total order of nodes, they significantly delay experiments with unpromising representations. For example, the graph in Figure 10.11(c) would result in slower exploration than that in Figure 10.11(b). Large certainties also delay exploration.

The system may add new representations or modify preference rules, which results in changes to the preference graph and causes deviations from the described exploration scheme. In particular, if SHAPER adds representations that do not have incoming edges, it eagerly explores their performance.

Chapter 11

Summary of work on the top-level control

We have presented a detailed report on the development of an AI system for constructing and evaluating new representations. To our knowledge, this work is the first attempt to build a *general-purpose* engine for the automatic coordination of multiple learning and search algorithms.

The report covers the three central parts of the $\mathcal{S}$HAPER system: tools for maintaining the description and representation spaces (Chapter 7), statistical procedures for evaluating representations (Chapters 8 and 9), and a mechanism for utilizing preference heuristics (Chapter 10).

We now outline two other essential parts: a procedure for choosing among the available representation changes (Sections 11.1) and a collection of tools for the optional user participation in the top-level control (Section 11.2). The current implementation of these parts is an ad-hoc solution, which has a number of limitations and requires support of an experienced human operator.

Then, we review the key architectural decisions underlying the $\mathcal{S}$HAPER system and the main results of developing the control module (Section 11.3). We also point out several important limitations of the system and outline some directions for future research.

11.1 Delaying the change of representations

The exploration of a representation space involves two types of top-level decisions: (1) when to generate new representations and (2) which of the available representations to use. We have described a collection of statistical and symbolic techniques for making the second decision (see Chapters 8–10); however, we have *not* addressed the first type of top-level decisions. The development of a general-purpose mechanism for deciding when to apply changer algorithms is an important open problem, and we plan to investigate it in the future.

By default, the system generates *all* possible representations *before* solving any problems. This eager strategy encourages exploration in early stages of statistical learning. If the number of problems is much larger than the number of generated representations, then early selection of effective representations justifies the cost of expanding the full representation space.

The human user has an option to overwrite the default strategy by encoding *suspension and cancellation rules,* which guide the expansion of the representation space. Suspension rules delay the application of changer algorithms, and cancellation rules may later prune some of the delayed description changes. The user may specify a collection of general rules for all domains, as well as specific rules for each domain.

We explain the semantics of suspension and cancellation rules, and give an algorithm for using these rule and resolving conflicts among them (Section 11.1.1). Then, we outline a statistical technique for predicting the performance of changer operators, and discuss its role in selecting the appropriate description changes (Section 11.1.2).

11.1.1 Suspension and cancellation rules

We begin by describing rules that postpone the application of changer operators. Then, we outline a similar mechanism for delaying the use of newly generated representations. Finally, we discuss the frequency of re-checking the applicability conditions of the suspended representation changes.

Suspension rules

A *suspension rule* consists of two boolean functions, denoted *cn-cond*(*changer-op*,*node*) and *suspend-change*(*changer-op*,*node*), which have identical arguments: the first argument is a changer operator, and the second one is a description node. Recall that a changer operator consists of a description-changing algorithm and its applicability conditions (Section 7.2.1), whereas a description node comprises a domain description along with some additional information about its generation and use (Section 7.2.2).

The user encodes the two boolean functions by Lisp procedures, which may access any information about the global state of the $\mathcal{S}$HAPER system. The first function is the condition for using the suspension rule. If *cn-cond*(*changer-op*,*node*) returns false, then the rule gives no information about applying *changer-op* to *node*. On the other hand, if *cn-cond* returns true, then the second function provides a suspension test: if *suspend-change*(*changer-op*,*node*) is true, then the system should delay the application of *changer-op* to *node*; otherwise, it should immediately apply the changer operator.

When $\mathcal{S}$HAPER generates a new description node, it first identifies the applicable changer operators and then determines whether some of them should be delayed. For every operator, the system invokes all available suspension rules. If at least one rule indicates the need for immediate application, then $\mathcal{S}$HAPER applies the changer operator. If some rules indicate the need for delay and no rule suggests immediate application, then $\mathcal{S}$HAPER delays the application. Finally, if the system finds no rules with matching conditions, then by default it immediately applies the operator.

If $\mathcal{S}$HAPER has delayed the application of some changer operators, then it periodically re-invokes the matching suspension rules, to determine whether the delay is over. When at least one rule prompts the immediate use of a suspended operator, the system applies the operator.

Note that, since the Lisp functions in a suspension rule utilize the global state of the system, their output may change over time. In particular, the *suspend-change* function may cause the suspension of a changer operator at the beginning of statistical learning, and trigger its application in a later stage. For example, a suspension rule may prompt the generation of a new domain description after solving a certain number of problems, or after finding out that the old descriptions give poor results.

Cancellation rules

The purpose of cancellation rules is to prune the suspended description changes that have become obsolete. For example, if the available domain descriptions have proved effective, then cancellation rules may prevent the delayed generation of new descriptions.

Formally, a *cancellation rule* is a boolean function, *cancel-change*(*changer-op*,*node*), whose arguments are a changer operator and a description node. If this function returns true, then the $\mathcal{S}$HAPER system never applies *changer-op* to *node*. If the system has suspended some description changes, then it periodically invokes the available cancellation rules and prunes the matching changes.

In Figure 11.1, we give two algorithms for utilizing the available suspension and cancellation rules, which are called *Check-Changers* and *Recheck-Changers*. When $\mathcal{S}$HAPER generates a new description node, it calls the *Check-Changers* procedure to identify the applicable changer operators. First, the procedure identifies all applicable operators and stores them in the node's applicability list. Then, it invokes the suspension rules, which delay the application of some operators, and applies the unsuspended operators. The procedure uses the *Make-Description* function, described in Section 7.2.2 (see Figure 7.2), for applying the appropriate changer operators.

The *Recheck-Changers* procedure is for re-visiting the delayed description changes. The system periodically calls this procedure for every description node that has suspended operators in the applicability list. First, the procedure invokes the available cancellation rules, which prune obsolete changer operators. Then, it re-iterates through the suspension rules, which may trigger the application of some operators.

Suspension of new representations

After applying a changer operator, the $\mathcal{S}$HAPER system combines the resulting domain description with matching problem solvers, thus producing new representations (see Section 7.2.3). The human operator may provide rules for delaying the use of newly generated representations and, thus, postponing the exploration process for these representations. In addition, she may specify rules for cancellation of delayed representations.

A suspension rule for representations consists of two Lisp functions, *r-cond*(*rep*) and *suspend-rep*(*rep*), which input a representation and return true or false. The role of these functions is analogous to that of *cn-cond* and *suspend-change*: the first function is the condition for using the rule, whereas the second is a suspension test. If both functions return true, then the system delays the use of the representation. A cancellation rule for representations is also a boolean Lisp function, whose argument is a representation. The system utilizes cancellation rules for pruning suspended representations.

Check-Changers(*node*)
For every changer operator, *changer-op*:
  If *node* matches the applicability condition of *changer-op*,
    then add *changer-op* to *node*'s applicability list.
For every changer operators, *changer-op*:
  If *node* matches the applicability condition of *changer-op*,
    then call *Check-Change*(*changer-op*,*node*).

Check-Change(*changer-op*,*node*)
For every cancellation rule, *cancel-change*:
  If *cancel-change*(*changer-op*,node),
    then delete *changer-op* from *node*'s applicability list and terminate.
For every suspension rule, (*cn-cond*, *suspend-change*):
  If *cn-cond*(*rep*) and not *suspend-change*(*rep*) (apply the changer),
    then call *Make-Description*(*changer-op*,*node*),
      remove *changer-op* from the applicability list, and terminate.
  If *cn-cond*(*rep*) and *suspend-change*(*rep*) (suspend the application),
    then mark *changer-op* in *node*'s applicability list as "suspended."
If *changer-op*is not marked "suspended,"
  then call *Make-Description*(*changer-op*,*node*),
    remove *changer-op* from the applicability list, and terminate.

Recheck-Changers(*node*)
For every changer operator, *changer-op*, in *node*'s applicability list:
  Call *Recheck-Change*(*changer-op*,*node*).

Recheck-Change(*changer-op*,*node*)
For every cancellation rule,
  If *cancel-change*(*changer-op*,*node*),
    then delete *changer-op* from *node*'s applicability list and terminate.
For every suspension rule, (*cn-cond*, *suspend-change*):
  If *cn-cond*(*rep*) and not *suspend-change*(*rep*) (apply the changer),
    then call *Make-Description*(*node*,*node*),
      remove *changer-op* from the applicability list, and terminate.

Figure 11.1: Delaying the application of changer operators. The *Check-Changers* procedure identifies the changer operators applicable to a new description node, and then uses the available rules to suspend some of these operators. The *Recheck-Changers* procedure re-invokes the suspension and cancellation rules for the suspended operators. Both procedures call the *Make-Description* function, given in Figure 7.2 (page 256), for executing the selected changer operators.

Check-Rep(*rep*)
If the representation *rep* matches some cancellation rule, then terminate.
For every suspension rule, (*r-cond*, *suspend-rep*):
  If *r-cond*(*rep*) and not *suspend-rep*(*rep*) (add the representation to the space),
    then call *Add-Rep*(*rep*) and terminate.
  If *r-cond*(*rep*) and *suspend-rep*(*rep*) (suspend the representation),
    then mark the representation *rep* as "suspended."
If *rep* is marked "suspended,"
  then add it to the list of suspended representations.
Else, call *Add-Rep*(*rep*).

Recheck-Rep(*rep*)
If the representation *rep* matches some cancellation rule,
  then delete *rep* from the list of suspended representation and terminate.
For every suspension rule, (*r-cond*, *suspend-rep*):
  If *r-cond*(*rep*) and not *suspend-rep*(*rep*) (add the representation to the space),
    then call *Add-Rep*(*rep*), delete *rep* from suspended representations, and terminate.

Figure 11.2: Suspending and unsuspending new representations. The *Check-Rep* procedure invokes the suspension and cancellation rules for a newly generated representation, whereas *Recheck-Rep* re-invokes the rules for a suspended representation. These procedures use the *Add-Rep* function, given in Figure 7.3 (page 259), for adding the new representation to SHAPER's space of available representations.

In Figure 11.2, we summarize the two main procedures for suspending, unsuspending, and cancelling representations. After the system has generated a new representation, it calls the *Check-Rep* procedure, which loops through the available rules. If *Check-Rep* does not suspend or cancel the newly generated representation, then it invokes the *Add-Rep* procedure (see Figure 7.3 in Section 7.2.3), which adds the new representation to the expanded space. For every suspended representation, the system periodically invokes the *Recheck-Rep* procedure, which re-iterates through the cancellation and suspension rules.

Note that the role of cancellation rules differs from that of rejection rules, described in Sections 7.2.2 and 7.2.3. The system *repeatedly* invokes cancellation rules for suspended representations, and the output of these rules may change over time, depending on the global state of the system. On the other hand, it invokes rejection rules only *once* for each representation (see the *Add-Rep* algorithm in Figure 7.3), and their output should not change over time.

Frequency of re-invoking the rules

The system invokes the *Recheck-Changers* and *Recheck-Rep* procedures after each problem-solving episode, that is, after every application of solver algorithms. In addition, SHAPER calls these two procedures (1) after applying a changer operator or cancelling a suspended operator, (2) after unsuspending or cancelling a representation, and (3) after discarding an old representation because of unfavorable statistical data.

The execution of suspension and cancellation rules usually takes much less time than the application of solvers; hence, the repetitive calls to *Recheck-Changers* and *Recheck-Rep* do not slow down the system. If the human operator constructs rules that do take significant computational time, she may enforce a less frequent re-invocation of these rules.

In Figure 11.3, we summarize the options for specifying the frequency of using suspension and cancellation rules. The user may utilize any combination of these options to set the default frequency for a problem domain, as well as frequencies of re-invoking specific rules.

For example, if a rule's output does not change after failed and interrupted attempts to solve a problem, then we should restrict its invocation to successful problem-solving episodes. As another example, if the rule's output does not depend on the accumulated statistical data, then we should disable its re-invocation after any problem-solving episodes.

11.1.2 Expected performance of changer operators

When the user specifies a condition for delaying the application of some changer operator, she may need to account for the expected performance of this operator. The main performance factors include the probability of successful application, time to execute the operator, and expected utility of the resulting domain description.

Recall that a changer operator may comprise several changer algorithms, and its application involves the successive execution of these algorithms (see Section 7.2.1). If one of the changer algorithm fails to generate a new description, then the application procedure skips the remaining algorithms and returns a failure. Since all description changers in the SHAPER system terminate in reasonable time, the top-level module never interrupts their execution.

The system includes statistical procedures for estimating the success probability of changer operators and their expected running time, which proved useful for constructing the conditions of suspension and cancellation rules. On the other hand, we have yet to address the problem of predicting the expected utility of a new description. We describe the implemented statistical functions and discuss related directions for future work.

Success probability

The system estimates the success probability for a changer operator by the percentage of its past successes in the current domain. For example, if SHAPER has applied the operator five times and succeeded in four cases, then the chance of success is 0.8. The estimate function returns not only this probability value, but also the number of past operator applications, which allows the user to judge the accuracy of the estimated value. The system also keeps similar statistics for description-changing algorithms, and the user may utilize success probabilities of individual algorithms in constructing suspension rules.

Note that SHAPER does *not* use the success chances of individual description changers in computing the probabilities for changer operators. We experimented with estimating the chances of an operator's success by multiplying the probabilities of the changer algorithms that form the operator; however, this technique proved inaccurate, because the success probabilities are not independent.

Re-invocation after a problem-solving episode

By default, the system re-invokes all suspension and cancellation rules after every application of solver algorithms. When the human operator encodes a new rule, she may completely disable its use after problem-solving episodes. Alternatively, she may enforce a less frequent re-invokation, using any combination of the following three options:

- re-invoke after successfully solving a problem
- re-invoke after failure to solve a problem
- re-invoke after interrupting a problem solver

The system also supports the option of re-visiting a rule *only* after the use of a matching description. If the human user enables this restriction for a rule that delays description changes, then $\mathcal{S}$HAPER will re-invoke the rule for a pair (*changer-op*, *node*) only after running some solver with *node*'s description. Similarly, if the user chooses this option for a rule that suspends or cancels representations, then the rule will fire for a representation *rep* only after problem solving with *rep*'s domain description.

Re-invocation after applying or cancelling a changer operator

By default, $\mathcal{S}$HAPER re-invokes the available rules after every application of a changer operator and every cancellation of a suspended operator. The human user may disable the invocation of a rule in either of these cases, or in both cases.

Furthermore, when the user constructs a rule for delaying description changes, she may indicate that it fires *only* after the use of a matching operator. Then, the system will re-invoke the rule for a pair (*changer-op*, *node*) only when applying *changer-op* to another node.

Re-invocation after adding or pruning a representation

The $\mathcal{S}$HAPER system also re-iterates through the suspension and cancellation rules after each of the following operations:

- unsuspending a representation and adding it to the expanded space
- cancelling one of the previously suspended representations
- dropping a representation that has proved ineffective (see Section 8.5.4)

The user may disable the re-invocation of a rule in any of these three cases. Furthermore, she may restrict the system to invoking the rule *only* after unsuspending or pruning a representation with a matching domain description.

Figure 11.3: Options for controlling the re-invocation frequency of suspension and cancellation rules. The human operator may use these options to set the default frequency for a domain, and to adjust the frequencies of individual rules.

Expected running time

The strategy for predicting execution times is different from the computation of success probabilities. The system estimates the running times of individual description changers, and then uses them to compute the expected times for changer operators. This strategy proved more accurate than simple averaging of the past running times of changer operators. Moreover, it enables SHAPER to utilize past data from other domains in computing time estimates for the current domain.

The system keeps the mean running times of all description changers, as well as the standard deviations of the mean estimates. After each application of a changer algorithm, SHAPER updates the corresponding mean and deviation values. The current version of the statistical procedure does *not* distinguish between success and failure times. Since the running time of most changer algorithms in the SHAPER system does not depend on the outcome of their application, we have not implemented a separate estimation of success and failure times.

The statistical procedure determines the expected computational time of a changer operator by summing the times of the corresponding description changers. Suppose that the operator includes k changer algorithms, their estimated mean times are $t_1, t_2, ..., t_k$, and the standard deviations of these mean estimates are $\sigma_1, \sigma_2, ..., \sigma_k$. Then, we compute an operator's running time and deviation as follows:

$$\begin{aligned} t &= t_1 + t_2 + ... + t_k, \\ \sigma &= \sqrt{\sigma_1^2 + \sigma_2^2 + ... + \sigma_k^2}. \end{aligned}$$

The resulting value is an unbiased time estimate for the successful application of the operator. On the other hand, if the operator comprises more than one changer algorithm, then this value may overestimate the mean failure time: If one of the k algorithms fails to generate a new description, then the system does not apply the remaining algorithms and, thus, the operator's running time should be less than t. Developing a more accurate estimate of the failure time is an open problem.

Use of complexity estimates

The human user may provide Lisp functions for evaluating the expected computational complexity of description changes, which improve the accuracy of time estimates for changer algorithms. The role of these functions is similar to problem-size estimates in computing the expected problem-solving time (see Section 9.2).

A *complexity-estimate function* inputs a domain description and returns a positive value, which should correlate with the time of applying a changer algorithm to this description. The user may specify a default complexity function for a domain, as well as more accurate functions for specific changer algorithms.

The statistical procedure applies least-square regression to find a *linear dependency* between complexity estimates and running times, and then performs t-tests to check the statistical significance of the regression (see the description of the t-test in Section 9.2.1). If the dependency passes the significance test, the system utilizes it in estimating running

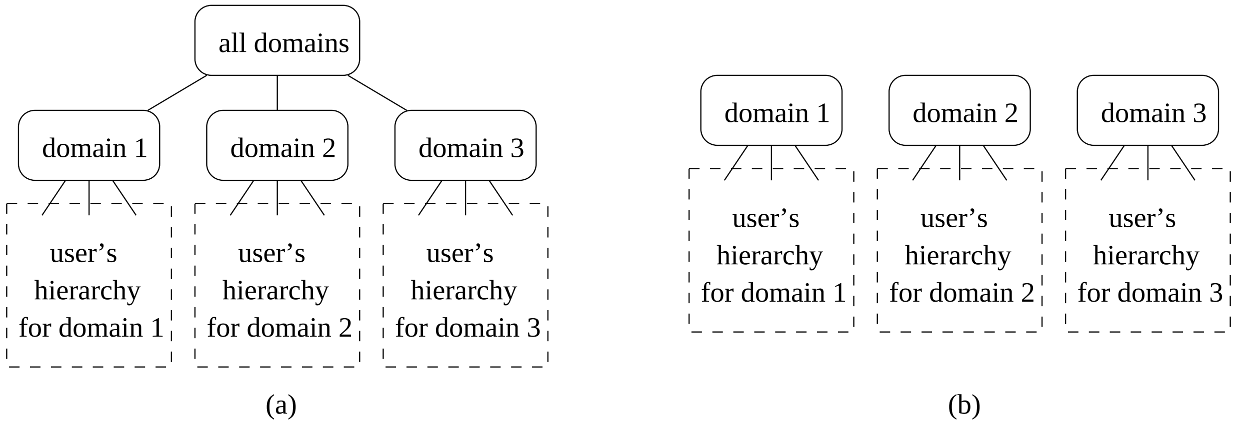


Figure 11.4: Similarity hierarchies for estimating the time of description changes. By default, the system builds a multi-domain hierarchy (a). If the human operator disables the transfer of experience among domains, then the system uses a separate similarity hierarchy for each domain (b).

times of changer operators. If it does not pass the test, then $\mathcal{S}$HAPER disregards complexity estimates.

We have analyzed the time complexity of all changer algorithms in the current version of the $\mathcal{S}$HAPER system (see Part II), and the derived expressions allow the construction of very accurate estimate functions. For example, the running time of the *Chooser* algorithm depends on total number E of all effects, in all operators and inference rules, and on the number N of nonstatic predicates; specifically, the time is proportional to $E \cdot (N + 6)$ (see Section 3.4.1). Thus, we have implemented a complexity-estimate function that determines E and N, and then returns the value $E \cdot (N + 6)$.

Similarity hierarchy

The system also supports the utilization of similarity hierarchies in computing the expected running time. This technique is based on the encoding of similarity among domain descriptions by a tree-structured hierarchy, whose leaf nodes represent the groups of similar descriptions. The encoding is analogous to the similarity hierarchies for estimating problem-solving gains (see Section 9.3).

When $\mathcal{S}$HAPER needs to estimate the time for changing a description, it identifies the corresponding leaf group in the hierarchy and then selects the appropriate ancestor group, using the technique of Section 9.3.2. The statistical procedure utilizes the data in the selected group to compute the expected running time of the changer algorithm.

By default, the system constructs the similarity hierarchy shown in Figure 11.4. The root group of this hierarchy comprises the performance data from *all* domains, whereas the second-level groups are for individual domains. If the user provides hierarchies for specific domains, then $\mathcal{S}$HAPER links them to the corresponding second-level groups (see the dashed rectangles in Figure 11.4).

The system allows the user to construct a common hierarchy for all changer algorithms or, alternatively, a separate hierarchy for each algorithm. It also supports the use of different complexity-estimate functions for different similarity groups.

The resulting similarity hierarchy allows the transfer of experience across domains: if SHAPER has no performance data for the current domain, then it utilizes data from other domains. If the human operator believes that cross-domain transfer is not appropriate for some changer algorithm, then she may disable this transfer, which causes removal of the root group from this algorithm's hierarchy (see Figure 11.4b).

Related problems

The main open problem is to develop a general statistical mechanism for guiding the expansion of the description space, and to integrate it with the use of suspension and cancellation rules. The purpose of this mechanism is to automate the three top-level decisions:

1. when to generate new domain descriptions;
2. which description changes to perform;
3. whether to suspend some representations.

To address this problem, we plan to develop a function for estimating the utility of potential new descriptions. The function must determine whether the application of a given changer operator would produce an effective description, and whether it could be better than all previous descriptions.

We also need to design a control procedure, which will analyze the statistical predictions and make the top-level decisions. It has to take into account the past performance of solver and changer algorithms, as well as the expected suspensions and cancellations. The control mechanism should apply a changer operator only if the predicted performance improvement outweighs the application costs. Furthermore, it should identify the description changes that promise the most significant improvements, and delay the other changes.

11.2 Collaboration with the human user

The developed mechanism for generating and using new descriptions is fully automated, that is, it can process all given problems and explore the representation space and without human assistance. The user has to encode domains and problems, provide a library of solvers and changers, and specify a gain function. After providing these data, she may pass control to the SHAPER system, which will select and invoke appropriate algorithms.

If the human operator wants to share responsibilities with the system, she may *optionally* participate in the top-level control. The system's interface allows her to undertake any control decisions, and leave the other decisions to the system. We outline the structure and role of the interface (Section 11.2.1), and then describe its main parts (Section 11.2.2).

11.2.1 User interface

The SHAPER interface serves four main purposes: specification of problem-solving tasks and related initial knowledge; inspection of the intermediate results and top-level decisions;

manual selection of representations, changer operators, and time bounds; collection of performance data and evaluation of the system's effectiveness.

The current version of the interface has a number of limitations and requires substantial improvements. It is based on a library of ad-hoc tools for an expert user, developed in the process of debugging and testing the system. The user interacts with SHAPER by entering text-based commands in the Lisp window, and some advanced interface options require the implementation of small Lisp procedures. The help messages are limited, and serve only as reminders for an experienced user. Despite these limitations, the implemented interface has proved to be a convenient instrument for controlling the system, studying its behavior, and collecting empirical results.

The manual control of SHAPER comprises three groups of interface commands. The first group is for supplying initial knowledge, providing additional information in later stages of the problem-solving process, and manually correcting and enhancing intermediate results. The second collection of tools allows the human operator to inspect the initial information, new representations, accumulated performance data, and results of statistical learning.

Finally, the last group is for invoking the top-level control module and imposing restriction on its decisions. The control mechanism chooses solvers and changers, calls the selected algorithms, and stores and analyzes the results of their execution. After each application of a solver or changer operator, the system outputs a brief summary of the results, called the *trace,* which helps the user to keep track of SHAPER's progress and its key decisions.

By default, the top-level module runs until processing all input problems; however, the system has several options for specifying conditions for early termination, and later re-invoking the control mechanism. The user may utilize these options to "step" through the problem-solving process. Furthermore, she may modify intermediate results, revise heuristics for guiding the system, or supply additional knowledge.

11.2.2 Main tools

We now give a more detailed description of the interface tools. We begin with the mechanism for specifying search tasks and related initial knowledge (see Figures 11.5 and 11.6), then outline tools for inspecting the search results (Figures 11.7 and 11.8), and finally explain the user's options for guiding selection of solvers and changers (Figures 11.9-11.11).

Initializing and modifying the system's global state

The global state of the SHAPER system includes domains and problems, solver and changer operators, gain functions, representation spaces, user-specified heuristics, accumulated results of problem solving, and other relevant data. The interface includes tools for initializing SHAPER's state and later modifying intermediate states.

The system supports an extended version of the PRODIGY domain language, which serves for specifying search tasks, utility functions, representations, and related heuristics (see Figure 11.5). The user has to create files with encoding of initial data, and load them into the system. She may later load additional files, with extra tasks or new heuristics, as well as modify and re-load the original files. For example, the user may input a new domain after

$\mathcal{S}$HAPER has processed all problems in the initial domains. As another example, she may input additional solver operators, and then modify and re-load preference rules.

The interface also includes a mechanism for specifying some objects directly from the command line, without writing them into a file, as well as tools for deleting old domains and inactivating obsolete representations. Finally, it supports back-up and retrieval of intermediate states. In Figure 11.6, we summarize the options for modifying the global state.

Inspecting top-level decisions and intermediate results

When the control module is active, it outputs a real-time trace of its choices and main results (see Figure 11.7). The user has no control over the contents of the trace, which is a drawback of the current implementation; however, the system includes an option for turning off the trace output.

After $\mathcal{S}$HAPER has reached a termination condition, the user may get more details about the search results. We have implemented a library of functions for inspecting the global state, which can supply a wide range of data, from high-level summary of the system's performance to the contents of specific data structures. In Figure 11.8, we give a brief description of the inspection tools.

Invoking solvers, changers, and the control module

The user can either manually invoke individual problem solvers and description changers, or pass the control to the automatic top-level module. Furthermore, when the control module is inactive, she may modify generated representations and their applicability conditions.

To invoke a solver, the user specifies a representation, problem, and time bound. She may skip the bound specification, and then the system will automatically determine an appropriate time limit. Similarly, she can manually invoke a description improvement, by specifying a changer operator, initial description, and time bound.

When the user starts the automatic control mechanism, she may select the order of processing problems, restrict the allowed choices of representations, specify time bounds for some representations, and provide guidance for selecting and applying changer operators (see Figures 11.9 and 11.10). In addition, she may define a condition for terminating the problem-solving process. When the system's global state satisfies this condition, the top-level module will halt and pass the control back to the user. We list the options for specifying termination conditions in Figure 11.11.

The user may also halt the problem-solving process by the Lisp keyboard interrupt, which causes immediate termination. In this case, the $\mathcal{S}$HAPER discards the results of the unfinished application of a solver or changer operator. If the interrupt occurs during the update of the global state, between solver applications, then it may cause an inconsistency of the global state; however, the probability of this situation is very low, because the Lisp procedure for completing the update consists of several assignment operations and takes less than ten microseconds. We have not addressed the engineering problem of implementing a completely safe interrupt.

Domain language

The $\mathcal{S}$HAPER system supports a collection of tools for describing domains, problems, and related initial knowledge. These tools extend the PRODIGY domain language and enable the human user to specify the following information:

Domains and problems: A domain encoding includes a type hierarchy, operators, and inference rules (Sections 2.2.1 and 2.3.2); the user may provide several alternative encodings of the same domain. A problem instance is specified by an object list, initial state, and goal statement (Sections 2.2.1).

Control rules: The user may construct heuristic rules for guiding the PRODIGY search (Section 2.4.3). She may provide specialized rules for every domain, as well as a set of global rules.

Primary effects and abstraction: A domain encoding may include multiple selections of primary effects (Section 3.1.2) and multiple abstraction hierarchies (Section 4.1.2), which give rise to alternative descriptions. All hierarchies must satisfy the ordered-monotonicity property (Section 4.1.4).

Domain descriptions: A description consists of a pointer to a domain encoding and four optional fields, which include a selection of primary effects, abstraction hierarchy, set of control rules, and restrictions on the allowed problems (Section 7.1.2).

Solver and changer operators: A solver operator consists of a problem-solving algorithm and two conditions for its applicability (Section 7.2.1). It may also include a sequence of problem-specific changers, which improve a problem description before the invocation of the solver algorithm. A changer operator comprises a sequence of problem-independent description changers and two applicability conditions, similar to a solver's conditions. The user may need to specify knob parameters of solver and changer algorithms utilized in the operators.

Gain functions: The gain computation may be specified by a three-argument Lisp function, which inputs a PRODIGY problem, running time, and search outcome, and returns the gain value; it must satisfy Constraints 1–4, given in Section 7.3.1. If gain linear decreases with time (see Constraints 7–9 in Section 7.3.2), then the user may provide a different encoding, which consists of the reward function and unit-time cost.

Size functions and similarity hierarchies: A size function is encoded by a Lisp procedure, which inputs a PRODIGY problem and returns a real-valued estimate of the problem complexity (Section 9.2.1). A similarity hierarchy consists of similarity groups, arranged into a tree, and a Lisp function that identifies the matching leaf group for every given problem (Section 9.3.1). When the user encodes a similarity group, she may optionally include a size function. In addition, she may specify fixed regression slopes for success and failure times.

Initial time bounds: When the system has no past data for determining the appropriate time bounds, it utilizes general heuristics for limiting the search time (Section 8.5.1). The human operator may provide Lisp procedures that determine initial bounds, which take precedence over the default heuristics.

Rejection and comparison rules: The user may provide heuristic rules for pruning ineffective descriptions and representations (Sections 7.2.2 and 7.2.3). The applicability conditions of these rules are encoded by Lisp functions.

Preference rules: A preference rule consists of three functions, which determine its applicability, priority, and certainty (Section 10.1.1). The system supports three types of preference heuristics: user rules, counting rules, and testing rules (Section 10.1.2). If the human operator constructs testing rules, she has to supply a collection of test problems or a Lisp procedure for generating them.

Suspension and cancellation rules: The human operator specifies these rules by boolean Lisp functions, with appropriate arguments (Section 11.1.1).

Figure 11.5: Main elements of the extended PRODIGY language.

Specification of search tasks and related data

The system supports a collection of tools for defining the problem-solving tasks and related heuristics, updating them in later stages, controlling the main parameters of the global state, and manually modifying intermediate results.

Domains, problems, and other initial data
The human user has the following options for the specification of problem-solving tasks and related initial knowledge:

- Load files that encode domains, problems, descriptions, solver and changer operators, and heuristic rules (see Figure 11.5); the user may always add new files, as well as modify and re-load old ones
- Discard some of the previously loaded domains or problems; when discarding a domain, the system deletes the corresponding description and representation space, along with all performance data
- Define a new gain function and add it to the library of available functions

Global state of the SHAPER system
The interface includes commands for activating and inactivating SHAPER's control mechanism, saving and retrieving the learned information, and switching between gain functions:

- Turn on or off the top-level control module; when it is off, the human operator can invoke solvers and changers, but the system does not store performance data or new descriptions
- Save the global state of the system, which includes all domains and problems, expanded description and representation spaces, and accumulated performance data; the user may later restore the saved state
- Reset the system to its initial state, that is, discard the generated descriptions and accumulated performance data, for all domains; the user may also reset the representation space of a specific domain, or discard the performance data of a specific representation
- Change the current domain or Switch to a new gain function; note that the system does *not* restart statistical learning after the change of a gain function, since it may utilize the accumulated data for computing the expected value of the new function (see Section 8.3)

Elements of the representation space
Finally, the interface provides several options for adding and modifying elements of representation spaces:

- Construct a new domain description, by specifying a selection of primary effects, abstraction hierarchy, and collection of control rules; the system adds it to the description space and then generates the corresponding representations
- Specify a new representation, by pairing a domain description with a solver operator; the user can perform this operation even if the description does not match the operator's applicability condition
- Modify the applicability conditions of a solver operator, changer operator, or representation
- Inactivate some descriptions or representations; the user may also reactivate and unsuspend representations

Figure 11.6: Interface tools for specifying initial data and modifying the system's state.

Trace output

The system outputs a trace of its operations, which allows the human user to keep track of the main top-level decisions. The trace includes data about the application of solvers, changers, and heuristic rules.

Solving a problem
When SHAPER needs to solve a problem, it selects a representation and time bound, applies the chosen representation, and then outputs the following trace information:

- Name of the PRODIGY problem
- Selected representation, which includes a domain description and solver sequence; the system outputs the name of the description and the names of all algorithms in the solver sequence
- Selected time bound, expected problem-solving gain for this bound, and estimated optimal bound
- Information about reusing an old search space, specifically, the size and expansion-time of the reused space; SHAPER outputs this data if it has tried to solve the current problem in the past, with the same representation, and stored the expanded space
- Running time and outcome of the problem-solving attempt; the outcome may be a success, failure, rejection of the problem, or interrupt upon hitting the time bound
- Length and cost of the solution; the system gives this information only for successful problem-solving attempts
- Resulting gain, which may depend on the representation, problem, outcome, running time, and solution quality

Constructing a new description
The system applies a changer sequence to some old domain description, then finds solver operators that match the newly generated description, and constructs the corresponding representations. Finally, it applies user-specified rules for rejecting or suspending inappropriate representations. The trace includes the following information:

- Name of the old domain description
- Names of all algorithms in the changer sequence applied to the old description; if the user has specified a time limit for executing the sequence, then the system also outputs this limit
- Running time and outcome of the description-changing attempt; the outcome may be a success, failure to generate a new description, or interrupt upon reaching the time limit
- Name assigned to the newly generated description; the system outputs this information only in case of success
- List of the corresponding new representations, divided into three groups: (1) the ones added to the representation space, (2) the rejected ones, and (3) the suspended ones; for every representation, the system shows its solver sequence
- Gain of executing the changer sequence, which may depend on the algorithms in the sequence, initial description, and running time; recall that this "gain" is always negative, since changers incur computational costs without solving any problems

Pruning or suspending a representation
The system notifies the human operator about the results of utilizing comparison, suspension, and cancellation rules; specifically, it shows

- inactivated representations, after applying a comparison rule
- unsuspended representations, after re-invoking a suspension rule
- cancelled representations, after re-invoking a cancellation rule

For every such representation, SHAPER outputs the name of the corresponding domain description and the names of all algorithms in the solver sequence.

Figure 11.7: Contents of the trace output, which shows SHAPER's top-level decisions.

Inspection tools

The mechanism for inspecting the system's global state includes three groups of commands:

- The first group comprises several tools for accessing high-level information about the available domains, generated representations, and problem-solving results
- The second collection of commands allows the expert user to get more details; in particular, she may check the performance data, statistical evaluation of representations, and results of applying heuristic rules
- Finally, the third group consists of procedures for inspecting and analyzing the contents of all data structures in the implemented system; its purpose is to support low-level manual control and debugging

These tools enable the user to examine the following objects, intermediate results, and top-level decisions:

Main objects
- Domains, problems, solver and changer operators, and gain functions
- Size functions, similarity hierarchies, and heuristic rules
- Primary effects, abstraction hierarchies, and control rules
- Domain descriptions and representations, along with their construction history
- Structure of the expanded description and representation spaces

Search results
- Accumulated performance data, for representations and changer algorithms
- Generated solutions, along with their lengths, costs, and other related data
- Summary of results, including the number of solved problems, total gain and running time, and so on

Heuristic decisions
- Evaluation of problem sizes and estimated complexity of description changes
- Initial choices of time bounds and the number of trials before switching to the statistical computation of bounds
- Structure of the preference graphs and the resulting heuristic selection of representations

Statistical estimates
- For each representation: Probability of choosing it, exploratory and optimal time bound, and expected gain
- For each changer algorithm: Success probability, expected running time, and deviation of the time estimate
- For each similarity group: Regression slopes, t and P values, and deviations of time logarithms
- For each similarity hierarchy: Evaluation of groups in the hierarchy and the choice of appropriate groups

Figure 11.8: Interface tools for examining initial knowledge, search results, and statistical data.

Manual control of problem solving

When the user passes the control to the system's top-level module, she can optionally specify the order of solving the input problem, define constraints on the choice of representations, and supply her own mechanism for setting time bounds.

Problems
The choice of problems and their ordering may affect the learning curve and overall gain; however, we have not automated these decisions. By default, the system processes problems according to their order in the input queue. The human operator has three options for selecting and re-ordering problems. These options are mutually exclusive, that is, they cannot be used together.

Problem sequence: If the user provides a list of problem names, then SHAPER will process the specified problems in order, and halt upon reaching the end of the list.

Choice function: Another option is to implement a Lisp procedure for selecting problems, which takes no arguments and returns either a problem name or the termination signal. The system will call it after every problem-solving attempt, to get the next problem.

Test function: Finally, the third alternative is to provide a function for rejecting inappropriate problems, which inputs the pointer to a problem and returns true or false. The system will use the default processing order, but skip the rejected problems.

In addition, the human operator can completely disable problem solving, that is, disallow the application of solver algorithms, thus forcing SHAPER to concentrate on description changes.

Representations
The user may impose five constraints on the choice of representations. The system selects among the representations that satisfy *all* specified conditions, and disregards the other choices.

Allowed (or disallowed) representations: If the user inputs a list of allowed representations, then SHAPER will not consider any other choices. Alternatively, the user may specify a list of *disallowed* representations.

Allowed (or disallowed) solver operators: The system will choose among the representations that include the specified solver operators, and ignore the ones with disallowed solvers.

Allowed (or disallowed) descriptions: This restriction will rule out the representations that are based on disallowed domain descriptions.

Problem-independent test: The user may provide a test function, which inputs a representation and returns true or false, and then SHAPER will disregard the representations that fail the test.

Problem-specific test: The control mechanism can also utilize a two-argument test function, which inputs a representation along with a given problem and determines whether the representation is fit for this problem.

Time bounds
If the human operator does not trust SHAPER with the choice of time limits, then she can utilize three options for setting her own limits:

Specific bounds: First, she may set fixed time limits for some representations, and then SHAPER will use the specified limits for all problems solved with these representations.

Bound computation: The second option is to provide a Lisp procedure for setting time limits, which inputs a representation and problem, and either returns a time bound or signals that it is unable to choose a bound.

Common bound: Finally, the system can use a common fixed bound with all representations and problems.

If the human operator utilizes all three options, then specific fixed bounds take precedence over the function for computing time limits, whereas the common bound has the lowest priority.

Figure 11.9: Options for manual selection of problems, representations, and time bounds.

Restrictions on description changes

The system has a switch for turning on and off the mechanism for expanding the description space. If it is off, then SHAPER utilizes the available representations and does not construct new ones. When the user turns it on, she can impose three constraints on the use of description changers.

Allowed (or disallowed) changer operators: The user may provide a list of allowed operators, and then SHAPER will not apply any other changers; an alternative option is to specify disallowed operators.

Allowed (or disallowed) initial descriptions: This restriction limits the use of old domain descriptions in constructing new ones. The system will apply changer operators only to the allowed old descriptions.

Test function: The control module can utilize a user-specified test procedure, which inputs a description and changer operator, and determines whether the operator is suitable for improving this description.

In addition, the user may set time limits for some description changers. The mechanism for specifying these limits is similar to that for problem solvers: the system can utilize fixed bounds or a function for computing them.

Figure 11.10: Options for restricting the application of changer operators.

Termination conditions

The human user may apply SHAPER to a specific sequence of problems, restrict the allowed CPU time, limit the desired gain, and encode additional termination conditions by a boolean Lisp function. The system checks all conditions after every application of a solver or changer, and stops the execution in any of the following situations:

- Processing all problems of the given sequence; if the user does *not* specify a sequence, then SHAPER handles problems according to their order in the queue
- Processing a pre-set number of problems from the queue; the user may specify this number only if she does *not* provide a problem sequence
- Spending the alloted CPU time; note that the system checks this condition only between solver applications and, hence, the last solver may run over the alloted limit
- Reaching the threshold on the total gain; the user may specify two threshold values, positive and negative, and then the system stops when the cumulative gain either rises to the positive limit or falls to the negative one
- Satisfying the termination condition specified by a Lisp function; when the human operator designs this function, she may utilize any information about the global state of the system

In addition, the human operator can halt the system by a keyboard interrupt. If the system gets an interrupt, it immediately terminates the search and loses the results of the last problem-solving or description-changing attempt.

Figure 11.11: Conditions for halting the automatic control module.

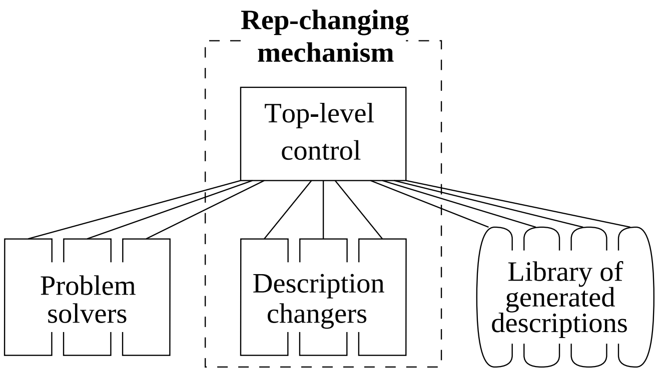


Figure 11.12: Architecture of the *S*HAPER system.

11.3 Contributions and open problems

We have constructed a collection of description-changing algorithms and a top-level control module. In Chapter 6, we summarized the results of developing description changers for the PRODIGY architecture. We now discuss the contributions of our work on the top-level control and identify some open problems.

We review the overall control architecture (Sections 11.3.1 and 11.3.2), the utility model for evaluating the system's performance (Section 11.3.3), and the main control mechanisms (Sections 11.3.4–11.3.6). We summarize the main limitations of these mechanisms (Section 11.3.7) and propose some directions for future research (Section 11.3.8).

11.3.1 Architecture for changing representations

We have developed a mechanism for automatic representation changes, which performs two major tasks: given a problem, it selects a problem solver and, if necessary, improves the domain description before calling the solver. The system accumulates a library of descriptions, and reuses them for solving new problems.

The *S*HAPER system comprises problem-solving engines, description changers, the top-level control module, and a library of domain descriptions. The changers and top-level module compose the mechanism for changing representations. We summarize *S*HAPER's architecture in Figure 11.12. This architecture *integrates multiple solver and changer algorithms,* and allows the use of competing search algorithms in a unified system.

Observe that there is no definite boundary between problem solvers and domain descriptions. For example, we may view control rules that guide PRODIGY search (see Section 2.4.3) as part of a solver algorithm or as a description element. The choice of a specific boundary is an architectural decision: we fix some search-control parameters to construct solver algorithms, and allow modification of other parameters by description changers.

The architecture in Figure 11.12 underlies all our results; however, we conjecture that there are other architectures for automated representation improvement. Major open problems include identifying limits of the proposed architecture and developing other types of representation-changing systems.

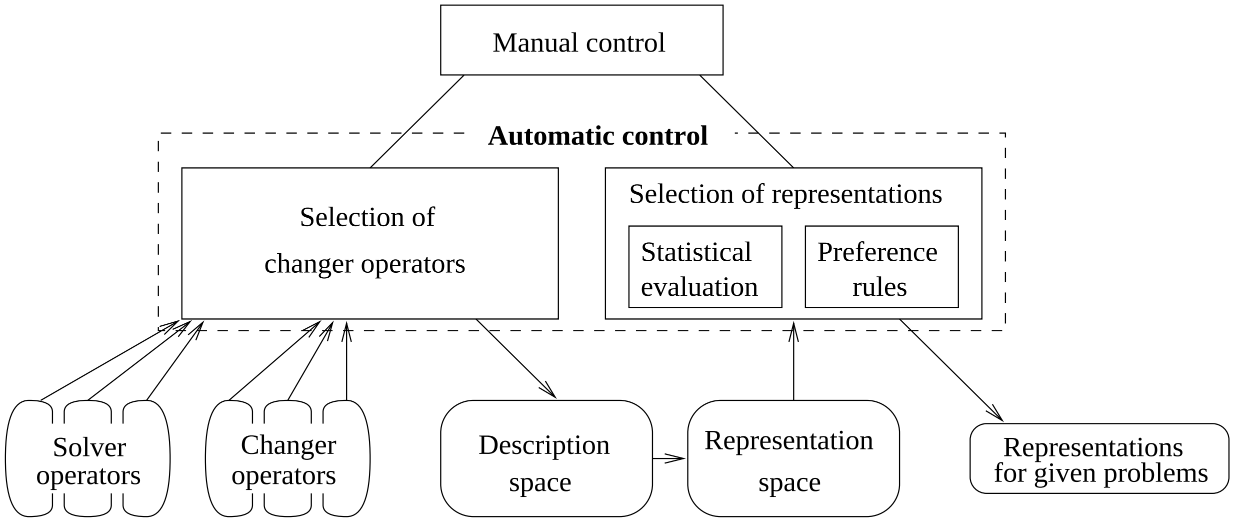


Figure 11.13: Architecture of the top-level control module.

11.3.2 Search among descriptions and representations

The top-level control is based on the concept of *description and representation spaces*. The main objects in these spaces are solver operators, changer operators, description nodes, and representations (see Section 7.2).

We give the architecture of the control module in Figure 11.13, where we show parts of the system by rectangles, and input and output data by ovals. The algorithm for expanding the description space is based on a collection of user-specified rules for selecting changer operators (see Section 11.1). The system constructs representations by pairing new domain descriptions with appropriate solver operators. Then, it identifies an appropriate representation for each given problem, using statistical performance evaluation (Chapters 8 and 9) and preference rules (Chapter 10). The human user has the option of participating in selection of description changers and representations.

In Figure 11.14, we show the main decision points of the control module. When we input a new problem, $\mathcal{S}$HAPER enters this decision cycle and runs until solving the problem or deciding to skip it.

The current system does not have two important decision points, shown in Figure 11.15 by thick lines. First, after constructing a new description, $\mathcal{S}$HAPER should intelligently select solver operators suitable for this description (see Figure 11.15a). Presently, it uses *all* solver operators whose conditions match the description. Second, when the system fails to solve a given problem, it should decide between solving the problem with another old representation and constructing a new representation (Figure 11.15b). Presently, $\mathcal{S}$HAPER does not consider the generation of new representations at this point of the decision cycle, and delays it until processing the next problem.

The system uses intelligent changer algorithms to generate new domain descriptions; hence, it constructs potentially good descriptions and skips a vast majority of "junk" descriptions. For this reason, the number of representations that $\mathcal{S}$HAPER can generate for a particular domain is relatively small, usually within a few hundred. A small representation

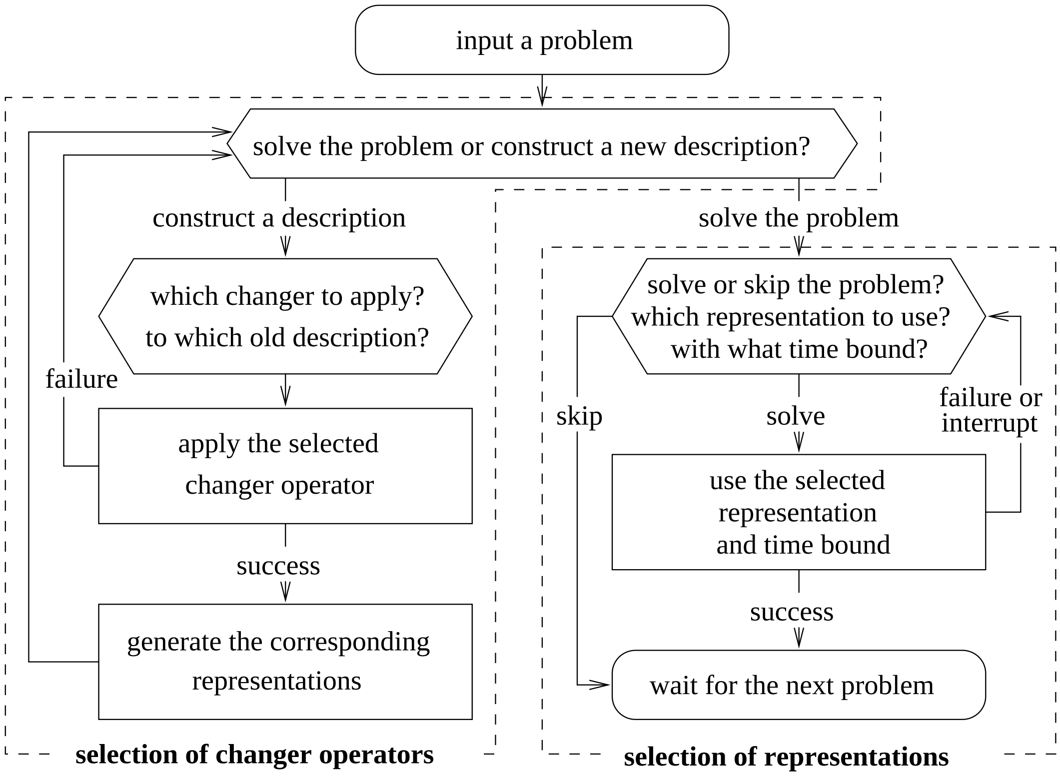


Figure 11.14: Decision cycle of the control module.

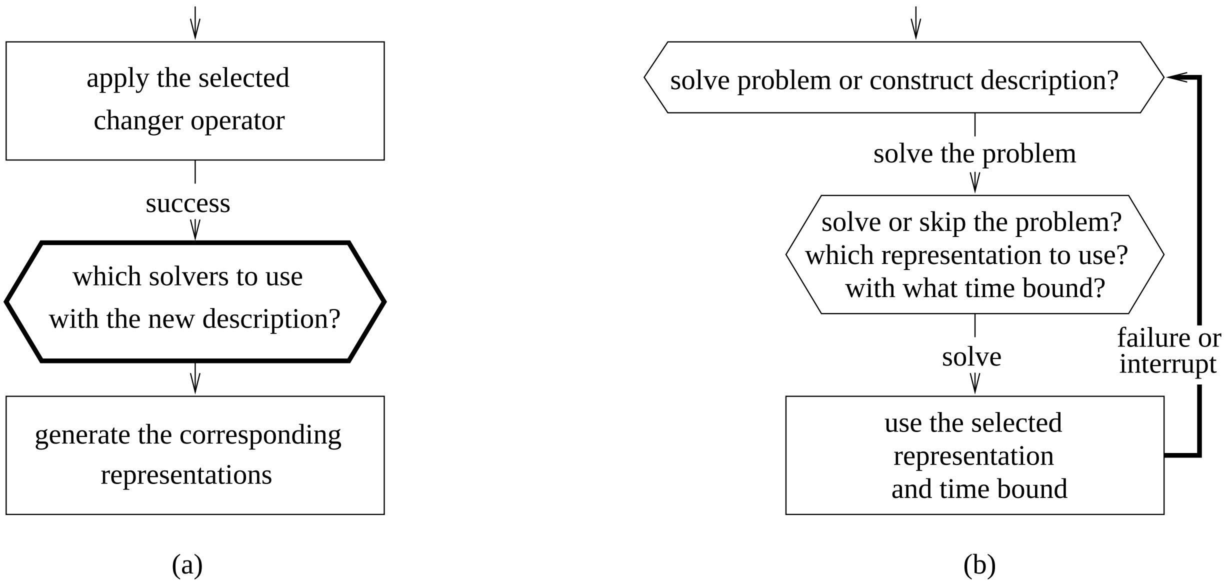


Figure 11.15: Missing decision points.

space is an important advantage over Korf's model, which requires the exploration of a huge space of all possible representations [Korf, 1980].

The exploration of a representation space is expensive, because the execution of changer algorithms takes significant time and the evaluation of new representations requires problem solving. Note that the system does *not* use backtracking in exploring description and representation spaces. Therefore, if the user does not provide rejection or comparison rules, the system stores and evaluates all generated representations. Designing more effective exploration techniques is an important problem.

The developed model does *not* rely on any specific properties of PRODIGY solvers; however, we have used two assumptions that limit its generality. First, all solver and changer algorithms use the same input language. If we allowed multiple input languages, we would have to add a library of language-conversion algorithms and provide a mechanism for their use in expanding the description space. The choice of representations would depend not only on the efficiency considerations but also on the language of given problems. Note that this extension would be necessary for the integrated use of search engines from different problem-solving architectures.

Second, we do *not* improve old descriptions incrementally in the process of problem solving. For example, we do not revise old abstractions or primary effects after failed solving attempts. The incremental modification of descriptions would pose several open problems, including propagation of modifications to the corresponding representations, backtracking over inappropriate modifications, and statistical evaluation of the modification quality.

11.3.3 Evaluation model

We have proposed a model for evaluating problem solvers, which unifies three main dimensions of performance: the number of solved problems, running time, and solution quality. We have considered a generalized gain function and used the expected problem-solving gain as the unified performance measure.

The evaluation model is very general; it allows the comparison of any two competing search algorithms. The performance measure, however, depends not only on an algorithm itself, but also on the domain, problem distribution, gain function, and bound-selection strategy. This dependency formalizes the observation that *no* search technique is universally better than others [Minton *et al.*, 1991; Knoblock and Yang, 1994; Veloso and Blythe, 1994]. The relative performance of problem solvers depends on a specific domain, as well as on the user's value judgments.

The model differs in two important ways from the use of asymptotic complexity bounds in the algorithm theory. First, the performance measure shows the *average-case* behavior, not the worst case. Second, this model is for *empirical* evaluation of performance. We determine the expected gains by testing solvers on sample problems rather than by algorithm analysis.

Note that the performance estimate does *not* directly depend on the completeness of a problem solver. If the search hits a time bound, the negative gain is the same whether the problem has no solution, the solver did not have enough time, or it could not find a solution because of incompleteness. In other words, we do not distinguish between the system's incompleteness and its inability to solve problems in feasible time.

The developed model has several limitations, which we plan to address in the future. First, we have assumed that all solvers are sound, and that the user provides a numerical gain function, defined for all possible outcomes. Lifting these assumptions would require a significant extension to the evaluation and statistical-selection technique. In particular, if the user cannot construct a gain function, the system needs to elicit the utility in the process of problem solving [Ha and Haddawy, 1997].

Second, the arithmetic mean of possible gains is not always an adequate performance measure. For example, when financial experts evaluate risky transactions, they use more sophisticated averaging functions, such as the geometric mean of possible gains or the ratio of the expected gain to its standard deviation (see, for instance, the textbook by Hull [1997]). As another example, the user may be interested in reducing the risk of losses rather than maximizing the expected gains.

Third, our rules for summing gains do not apply to some practical situations. We have already discussed the main limitations of each rule (see Section 7.3.5). Note also that we have not considered parallel execution of problem solvers, which would require a different gain summation.

11.3.4 Statistical selection

We have stated the task of selecting among representations as a statistical problem, derived an approximate solution, and demonstrated its effectiveness in selecting appropriate representations and time bounds. Selection takes little computation and its running time is usually negligible compared to problem-solving time.

We developed a learning algorithm for selecting representations, which combines the exploitation of past experience with the exploration of new alternatives. The algorithm accumulates performance data in the process of problem solving, and its selection accuracy grows with experience. Note that we do *not* separate the learning stage from the use of the learned knowledge.

The selection technique has proved effective for all tested distributions of problem-solving times. It gives good results even when distributions do not satisfy the assumptions of the statistical analysis. The algorithm can use an approximate measure of problem sizes and information about similarity between problems to improve selection accuracy; however, the improvement depends on the user's proficiency in providing an accurate size measure and defining groups of similar problems.

The developed technique is independent of specific search engines and problem domains; in particular, it does *not* rely on any special properties of PRODIGY solvers. We can use the learning algorithm in any AI system that offers a choice among multiple search engines, uses alternative domain descriptions, or allows selection of appropriate values of "knob" parameters. The technique is equally effective for small and large-scale domains. Even though AI problem solving provided the motivation for our work, the statistical model extends to situations outside AI, as illustrated by the phone-call example.

The model raises many open problems, which include relaxing the simplifying assumptions, improving the rigor of the statistical derivations, accounting for more features of real-world situations, and improving the heuristics used with the statistical estimates. We also plan to work on further improvement of the efficiency of statistical computations.

To make the model more flexible, we need to provide a mechanism for switching the representation and revising the time bound in the process of search for a solution. In addition, we plan to study the use of any-time solver algorithms, as well as solvers whose performance improves with experience, which will require an extension to the statistical model. We also need to allow interleaving of problem solving with several promising representations, as well as running competing representations on parallel processors.

If problem solving is *not* much slower than statistical computations, then we need to account for the running time of statistical procedures in estimating the expected gain. In particular, we may have to decide whether the use of size and similarity provides sufficient improvement to justify the extra running time.

We also intend to replace some heuristics with more rigorous techniques. In particular, we need to provide statistical models for selecting time bounds in the absence of past data (see Section 8.5), optimizing trade-offs between exploitation and exploration (Sections 8.4.2 and 8.4.3), and selecting appropriate groups of a similarity hierarchy (Section 9.3.2).

To enhance the use of similarity, we should allow multiple inheritance among groups and make appropriate extensions to the group-selection heuristics. Finally, we need to provide a means for constructing a similarity hierarchy automatically, to minimize the deviation of time logarithms (see Figure 9.1c) within groups.

11.3.5 Other selection mechanisms

We have constructed an alternative selection mechanism, based on the use of preference rules, which is also independent of specific problem-solving engines. This mechanism is a means for developing control strategies. We have used it for encoding the user's knowledge (Section 2.3.3), learning the relative performance of representation classes (Section 10.2), and comparing representations by testing them on small problems (Section 10.3).

We intend to use preference rules in developing other learning techniques, for more efficient selection of representations. In particular, we need to develop methods for learning applicability conditions of preference rules. We also plan to construct a central library of advanced preference heuristics, which improve the system's performance in most domains, and develop specialized heuristics for some large-scale domains. Note, however, that the current conflict-resolution method limits the generality of the preference mechanism. Developing a more flexible method is an important open problem.

We can use preference rules without statistical selection; however, the combination of the two mechanisms gives better results than either mechanism alone. Their synergy enables us to merge semantic knowledge with incremental learning. We use statistical evaluation as the main selection mechanism, and invoke preference rules when statistical data are insufficient. In particular, we use preferences to control the trade-off between exploitation and exploration. Preference rules usually improve the performance in the early stages of learning, whereas statistics becomes more important in a longer run.

We have provided tools for optional user participation in the construction and selection of representations. The user can make any part of the required decisions, and leave the other decisions up to the system. If the user's choices go against past experience, the system gives a warning and asks for confirmation. This scheme enables the user to share responsibilities with the system; thus, it allows the synergy of human intuition with automatic control.

Presently, only an experienced user can take part in the top-level control, as she has to learn the structure of description and representation spaces, and keep track of the system's decisions. Related open problems include developing a better interface, providing advanced tools for user participation, and optimizing the division of responsibilities between the human operator and the system.

11.3.6 Expanding the description space

The mechanism for guiding the application of changer operators is an integral part of the SHAPER system, and it plays a significant role in the overall performance; however, we have done little work on the automation of this task, and the implemented control procedure relies on the user-specified rules. If the human operator does not provide rules for delaying the change of descriptions, then the system expands the entire description space before solving any problems.

We constructed basic tools for use of suspension and cancellation rules (Section 11.1.1), and designed a statistical procedure for evaluating changer operators (Section 11.1.2). We have assumed that all changer algorithms terminate in feasible time, and that they do *not* generate invalid domain descriptions; that is, the resulting description never leads to incorrect solutions or execution errors. Furthermore, the control mechanism is effective only if the problem-solving time is *not* negligible in comparison with the running time of changer operators.

Related problems include extending the rule-based control mechanism and developing advanced tools for the construction of new rules. We also plan to design a repository of advanced domain-independent heuristics for guiding the expansion of description spaces. A more general problem is the development of a statistical-learning technique for selecting changer operators, which would replace the system's dependency on the human user.

11.3.7 Summary of limitations

We emphasized that the top-level control does not rely on specifics of the PRODIGY problem-solving architecture; however, we used some general properties of PRODIGY in making assumptions about solver and changer algorithms. We rely on these assumptions, which limit the generality of the control mechanism. We have already discussed most of the PRODIGY-related assumptions, and now give their summary.

- Problem solvers run much longer than statistical computations and, hence, we may neglect the time of statistical analysis
- Description changers always terminate in reasonable time and, hence, the system does not have to interrupt their execution
- An input language for describing domains and problems is the same for all solver and changer algorithms

- The system does *not* incrementally improve representations in the process of problem solving; in particular, if it uses learning to generate a new description, the learning stage is separated from the utilization of the learned knowledge in problem solving

- In most domains, the system solves a large number of problems and accumulates enough data for statistical selection; if some domains include only a few problems, we need to provide effective preference rules for these domains

11.3.8 Future research directions

We have summarized the main contributions and reviewed open problems related to specific contributions. We now discuss some other directions for future work.

We have assumed that the execution of solvers and changers is sequential, and that the system gets one problem at a time. The use of parallelism is an important research direction, which may require significant extensions to the utility model, structure of description and representation spaces, and selection mechanisms.

If the SHAPER system gets several problems at once, it has to decide on the order of solving them. If SHAPER does not have time to solve all input problems, it has to identify the ones that give large gains and skip low-gain problems. These decisions require an enhancement of the statistical evaluation.

Even though all solver and changer algorithms in SHAPER are domain-independent, the top-level control allows the use of domain-specific algorithms. We expect that a large library of specialized problem solvers and description changers would give much better results than a small collection of general algorithms. We conjecture that SHAPER's control is scalable to a large library of algorithms; however, it may require an extension to the preference-rule mechanism and development of other mechanisms to supplement statistical selection.

We have proposed a formal model of problem solving with multiple representations. The future challenges include further investigation of the role of representation, as well as studies of formal properties of representation changes, such as soundness, completeness, and admissibility. We aim to develop a unified theory that will subsume our current formalism, Korf's model of representation transformations [Korf, 1980], and Simon's view of cognitive representations [Larkin and Simon, 1987; Kaplan and Simon, 1990; Tabachneck-Schijf *et al.*, 1997], as well as theories of abstraction [Knoblock, 1993; Giunchiglia and Walsh, 1992] and macro operators [Korf, 1985b; Cohen, 1992; Tadepalli and Natarajan, 1996].

Part IV

Empirical results

I used to be solely responsible for the quality of my work, but now my computer shares the responsibility with me.
— Herbert A. Simon, personal communications.

The experiments with the main components of SHAPER have confirmed effectiveness of the described techniques. We have tested primary effects (Sections 3.6 and 3.7), abstractions (Sections 4.4 and 5.2), and goal-specific descriptions (Section 5.4), and demonstrated that they help to reduce search times and solution costs. We have also experimented with the statistical selection mechanism, which has proved effective for a variety of distributions (Sections 8.6, 8.7, and 9.2).

The subject of the next four chapters is an empirical evaluation of the integrated system. We apply SHAPER to the Machining problems (Chapter 12), Sokoban puzzle (Chapter 13), STRIPS world (Chapter 14), and Logistics tasks (Chapter 15). The experiments do *not* include the Robot Domain, because it contains too few problems and, hence, does not allow for statistical learning. We consider a variety of representation spaces and gain functions, and demonstrate the system's ability to identify appropriate search strategies.

We composed solver libraries from three versions of the PRODIGY search engine. Specifically, the libraries include SAVTA and SABA, as well as the LINEAR search engine, whose behavior is similar to PRODIGY2 (see Section 2.2.5). We constructed solver algorithms by combining these three engines with cost-bound heuristics, which limit the search depth.

For each domain and gain function, we plot SHAPER's gains on a series of problems, and show that the system's performance gradually improves with experience. Note that the order of problems in the series is always *random;* we do not arrange them by difficulty, or by any other criterion.

The gain measurements account not only for the search time, but also for the cost of processing statistical data and selecting a solver algorithm. That is, the system measures the *overall* time for choosing and applying a solver; however, the selection time has been negligibly small in *all* experiments. In fact, if we ignored it and re-plotted the graphs, the shift of the gain curves would be smaller than their thickness.

The empirical results have re-confirmed the power of the developed approach and showed that the utility of the control mechanism does *not* depend on specific properties of domains, solver algorithms, or gain functions. The SHAPER system has correctly identified the appropriate representations and time bounds in almost all cases.

Note that we did *not* use the experimental domains during the development and implementation of SHAPER, nor "fine-tuned" the system for any of them. Since the developed techniques have proved effective in all four domains, we expect them to be equally effective for other problem-solving tasks.

The presentation of empirical results consists of four chapters, which correspond to the four test domains. In Chapter 12, we explain the design of experiments and give the results of applying SHAPER to the Machining Domain. Then, we describe similar tests in the other three domains (Chapters 13–15); the reader may browse the report of these tests in an arbitrary order.

Chapter 12

Machining Domain

We first apply $\mathcal{S}$HAPER to a simple model of a machine shop, described in Section 3.6.2. All problems in this domain are solvable, and they never cause a failure of PRODIGY search. Moreover, if we do not set a tight cost bound, then PRODIGY finds a solution in feasible time; that is, there is no risk of infinite looping.

12.1 Selection among domain descriptions

Suppose that the system includes only one search engine, SAVTA, and it runs without cost bounds. We may apply changer algorithms to construct new descriptions of the Machining Domain, and then $\mathcal{S}$HAPER will select the description that maximizes SAVTA's expected gains.

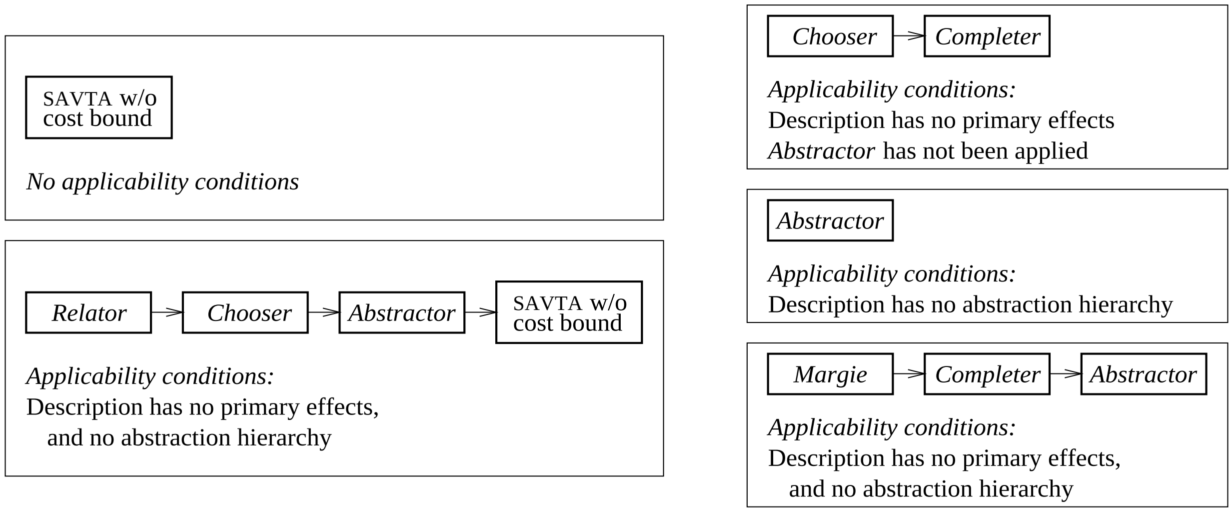
In Figure 12.1, we show a collection of solver and changer operators constructed from the available algorithms. Recall that a *solver operator* consists of a search algorithm and applicability conditions (see Section 7.2.1). It may also include procedures for problem-specific description improvements. On the other hand, a *changer operator* is a sequence of problem-independent changers, along with conditions for their applicability.

The control module applies changer operators to the initial encoding of the Machining Domain and constructs two new descriptions, as shown in Figure 12.2. The first description is based on a selection of primary effects, given in Figure 3.33 (page 133), and the second one is a combination of primary effects with abstraction (see Figure 5.5 in page 5.5). Then, the system pairs the descriptions with solver operators, thus producing four representations. We describe the results of statistical selection among them, for several utility functions.

Linear gain functions

We begin with simple gain functions, which linearly depend on three factors: the number of goals in a problem, denoted *n-goals*, search time, and solution cost. Since machining problems never cause failures, we do not distinguish between failure and interrupt outcomes in defining the gains.

First, we experiment with a function that decreases with the running time, and does not



(a) Solver operators. (b) Changer operators.

Figure 12.1: Solver and changer operators in the experiments on choosing a description. We show the sequences of algorithms that comprise these operators, and their applicability conditions.

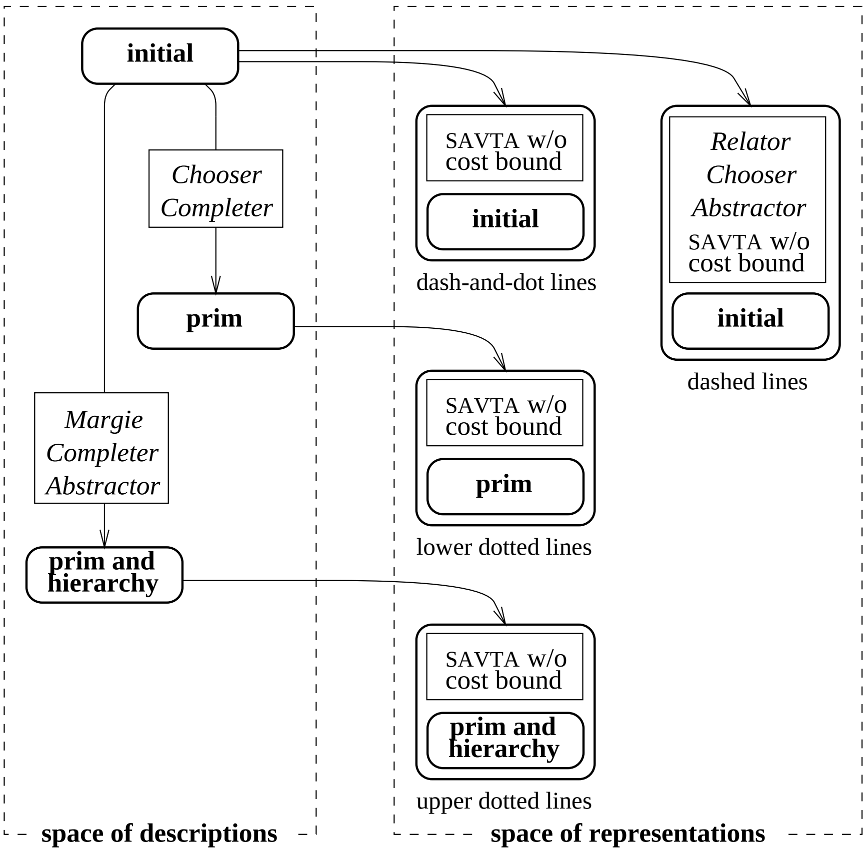


Figure 12.2: Description and representation space in the Machining Domain. The system uses the operators in Figure 12.1 to expand this space. Note that the *Abstractor* operator does *not* produce new descriptions, because the hierarchy collapses to a single level. The small-font subscriptions refer to the legend in Figures 12.3–12.6, which show the performance of each representation.

account for the other factors:

$$gain = \begin{cases} 0.5 - time, & \text{if success} \\ -time, & \text{if failure or interrupt} \end{cases} \tag{12.1}$$

Second, we consider the situation when the gain mostly depends on the number of goals and solution cost, whereas its dependency on the running time is negligible:

$$gain = \begin{cases} 2 \cdot n\text{-}goals - cost - 0.01 \cdot time, & \text{if success and } cost < 2 \cdot n\text{-}goals \\ -0.01 \cdot time, & \text{otherwise} \end{cases} \tag{12.2}$$

Finally, we give the results of using a gain function that accounts for all three factors:

$$gain = \begin{cases} 4 \cdot n\text{-}goals - cost - 50 \cdot time, & \text{if success and } cost < 4 \cdot n\text{-}goals \\ -50 \cdot time, & \text{otherwise} \end{cases} \tag{12.3}$$

We performed two experiments with each of these functions. First, the $\mathcal{S}$HAPER system was applied to sequences of fifty problems; then, it ran on ten-times longer sequences. In each case, the system started without prior performance data, and gradually identified an appropriate representation and time bound. We present a graphical summary of these tests (Figures 12.3 and Figure 12.4), and then compare the system's cumulative gains with the optimal gains (Table 12.1).

Short problem sequences

We give the results of fifty-problem experiments in Figure 12.3, where the top row of graphs shows the system's behavior with Function 12.1, the middle row contains the results of using Function 12.2, and the bottom row is for Function 12.3. The horizontal axes of these graphs show the number of a problem in a sequence, from 1 to 50, and the vertical axes are the gain values.

For every function, we give the raw empirical data in the left-hand graph, which shows the gain on each problem, and provide alternative views of the same data in the other two graphs. The solid line in the middle graph is the result of smoothing the raw curve, that is, averaging the gain values over five-problem intervals: we plot the mean gain for problems 1–5, the mean for 6–10, and so on.

The solid curve on the right is analogous to the artificial-test graphs in Section 8.7: it shows the *cumulative* per-problem gain, obtained up to the current problem. For instance, the cumulative value for 10 is the mean gain on the first ten problems, and the value for 50 is the mean on all fifty problems.

The broken lines illustrate the performance of the four available representations, without time bounds. For each representation, we plot the results of applying it to all fifty problems. In particular, the dashed curves show the behavior of the goal-specific abstraction, with unlimited search time, which is the *optimal* strategy for all three gain functions.

The results confirm that $\mathcal{S}$HAPER's performance gradually improves with experience; however, it does not converge to the optimal strategy, because the learning sequences are not sufficiently long. We next present experiments with larger problem samples, which enable $\mathcal{S}$HAPER to identify the best representation and time bound.

Long problem sequences

In Figure 12.4, we illustrate the results of applying SHAPER to 500-problem sequences, and compare them with the performance of each representation. The left-hand graphs contain smoothed gain curves, which are similar to that in Figure 12.3; however, we now average the gains over ten-problem intervals. The curves on the right show the cumulative per-problem gains.

The system converged to the optimal choice of a representation in all three cases, after processing about a hundred problems. After finding the best strategy, SHAPER occasionally experimented with less effective representations (see Section 8.4), which caused minor deviations from optimal performance.

Discontinuous gain functions

We next give results of choosing representations for three nonlinear functions, and demonstrate that complex functions do not confuse the selection mechanism. In the first series of tests, SHAPER earns a reward only for optimal solutions:

$$gain = \begin{cases} 0.5 - time, & \text{if the solver finds an optimal solution} \\ -time, & \text{otherwise} \end{cases} \tag{12.4}$$

In the second series, we reward the system for solving a given problem by a deadline, and penalize for missing the deadline:

$$gain = \begin{cases} 1.0, & \text{if success and } time \leq 0.5 \\ -1.0, & \text{otherwise} \end{cases} \tag{12.5}$$

Finally, SHAPER has to select appropriate representations for a complex discontinuous function, which depends on the problem difficulty, search time, and solution quality:

$$gain = \begin{cases} 4 \cdot n\text{-}goals, & \text{if success and } time \leq 0.3 \\ 4 \cdot n\text{-}goals - 50 \cdot time, & \text{if success and } 0.3 < time \leq 0.5 \\ 4 \cdot n\text{-}goals - cost - 50 \cdot time, & \text{if success, } time > 0.5\text{, and } cost < 4 \cdot n\text{-}goals \\ -50 \cdot time, & \text{otherwise} \end{cases} \tag{12.6}$$

The system successfully identified the optimal strategy for each of these functions. We summarize the results of processing fifty-problem sequences in Figure 12.5, and the results for longer sequences in Figure 12.6. Observe that the nonlinear functions did *not* confuse SHAPER; in fact, the choice of the right strategy required less data than in the experiments with linear gains.

Summary

We have demonstrated SHAPER's ability to choose appropriate domain descriptions and time bounds in the Machining Domain. The system usually converges to the right strategy after processing fifty to a hundred problems; however, it may suffer significant losses in the beginning, when trying alternative representations.

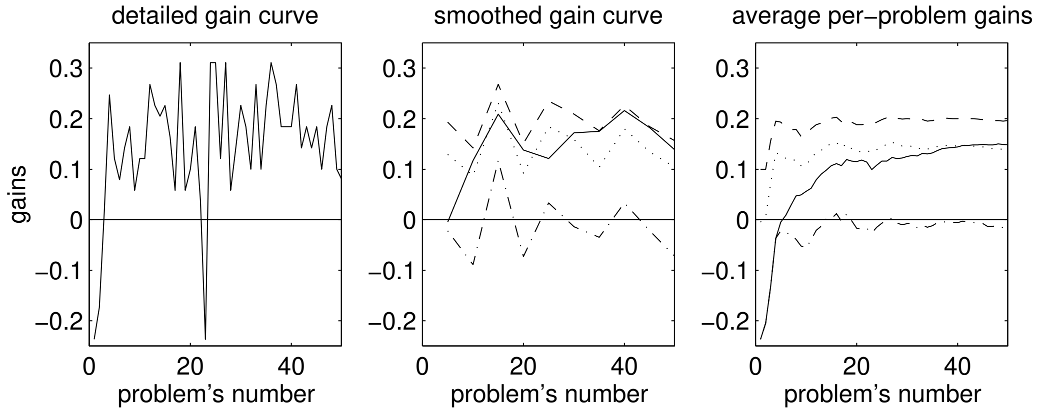
	problem sequences				optimal
	short		long		gain
Function 12.1	0.148	(80%)	0.177	(96%)	0.184
Function 12.2	3.91	(55%)	6.67	(93%)	7.17
Function 12.3	10.3	(53%)	17.1	(88%)	19.4
Function 12.4	0.115	(63%)	0.176	(96%)	0.184
Function 12.5	0.680	(68%)	0.956	(96%)	1.000
Function 12.6	38.6	(85%)	44.6	(99%)	45.2

Table 12.1: Cumulative per-problem gains and their comparison with the optimal values. We list the average gains on the short problem sequences (Figures 12.3 and 12.5), and on the long sequences (Figures 12.4 and 12.6). For each gain value, we give the corresponding percentage of the optimal gain.

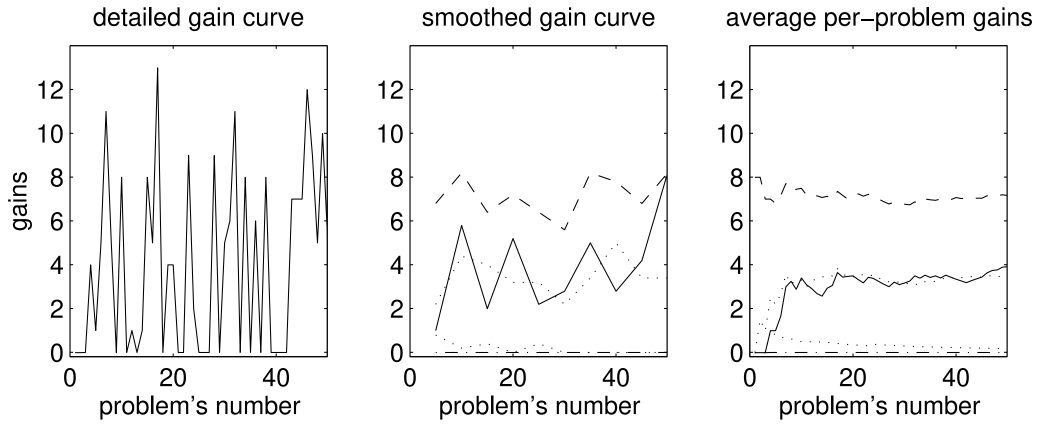
In addition, SHAPER may incur minor losses in later stages, during its occasional attempts to employ ineffective representations. The purpose of these attempts is to verify the accuracy of the available data; their frequency gradually diminishes with the accumulation of additional data (see Section 8.4).

The learning mechanism proved equally effective for a variety of gain functions; in fact, we were not able to devise a function that would cause significantly worse performance. The behavior of the learner on Machining problems was very similar to the artificial tests of Section 8.7.

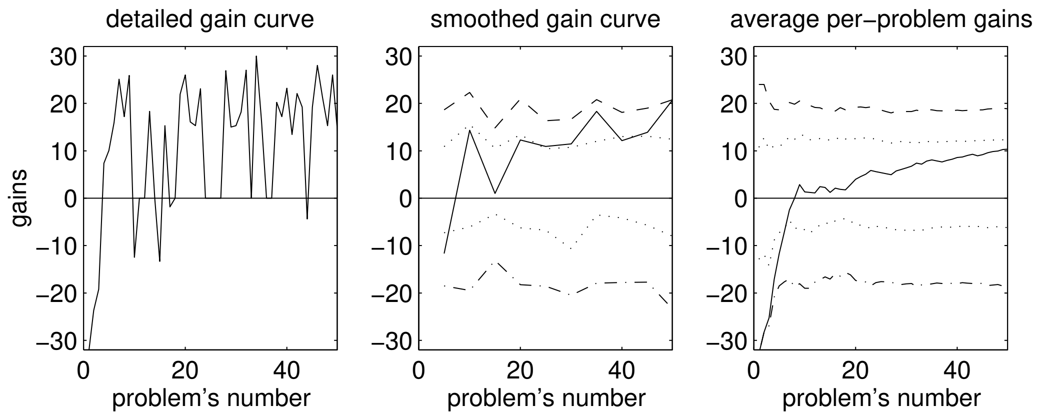
In Table 12.1, we summarize the cumulative per-problem gains, obtained on short and long sequences. If the system had a *prior* knowledge of the optimal strategy for each utility function, then it would earn the gain values given in the rightmost column. The cumulative gains on the fifty-problem sequences range from 53% to 85% of the optimal values, whereas the gains on the long sequences are between 88% and 99% of optimal.



(a) Gain linearly decreases with the running time (Function 12.1).



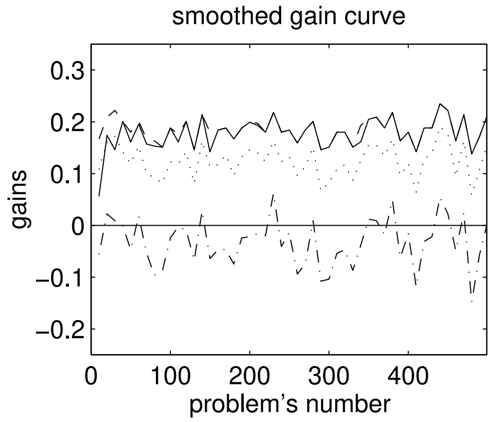
(b) Gain mainly depends on the solution quality (Function 12.2).



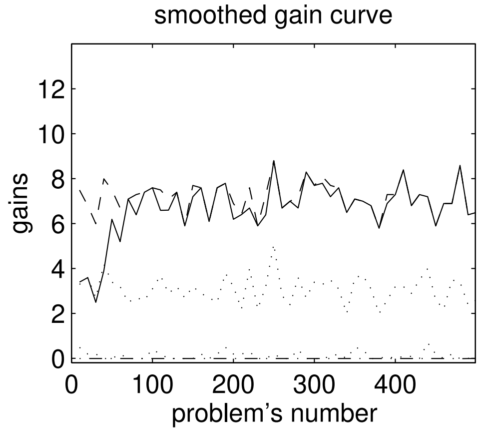
(c) Gain depends on both running time and solution quality (Function 12.3).

Figure 12.3: Performance of SHAPER on sequences of fifty Machining problems, with linear gain functions. The system chooses among four representations, illustrated in Figure 12.2, and determines appropriate time bounds. We give the gains on each of the fifty problems (left), as well as smoothed gain curves (middle) and cumulative per-problem gains (right).

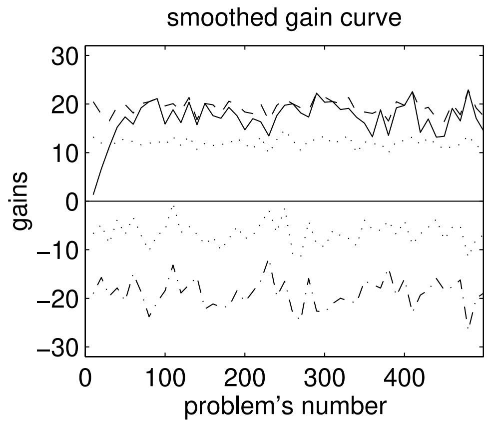
We plot not only the system's gains (solid lines), but also the performance of each available representation with no time bound. Specifically, the graphs illustrate the behavior of standard PRODIGY search (dash-and-dot lines), primary effects (lower dotted lines), abstraction search (upper dotted lines), and goal-specific abstraction (dashed lines). We do not show the lower dotted curves for Function 12.1, because they completely coincide with the dash-and-dot curves.



(a) Gain linearly decreases with the running time (Function 12.1).

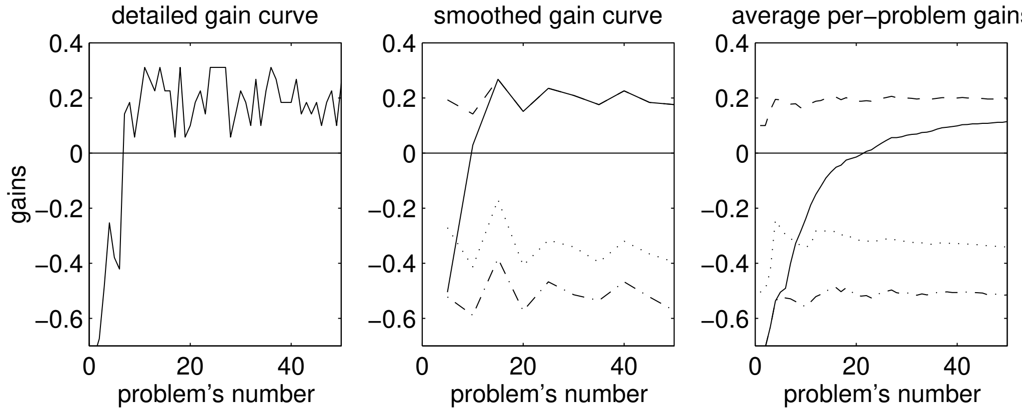


(b) Gain mainly depends on the solution quality (Function 12.2).

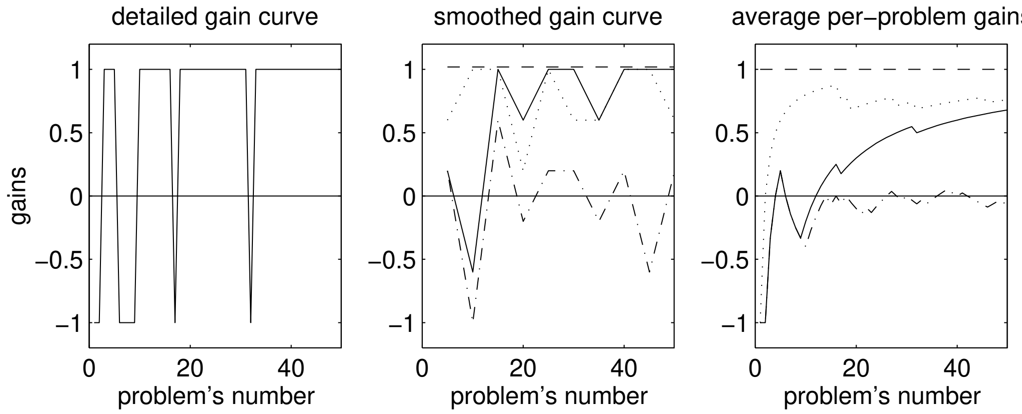


(c) Gain depends on both running time and solution quality (Function 12.3).

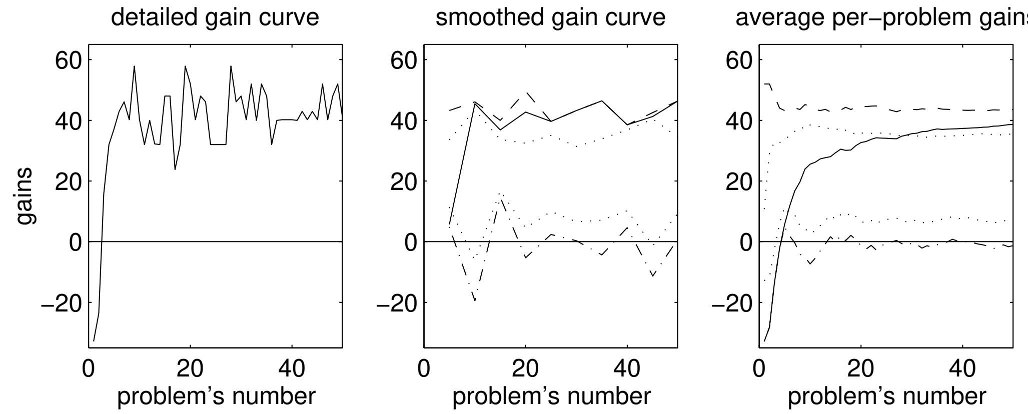
Figure 12.4: Performance on 500-problem sequences in the Machining Domain, with linear gain functions. For every function, we give the smoothed gain curve (solid lines, left) and cumulative per-problem gains (solid lines, right). In addition, we show the performance of each representation without time bounds (broken lines), using the legend of Figure 12.3.



(a) System gets a reward only for the optimal solution (Function 12.4).

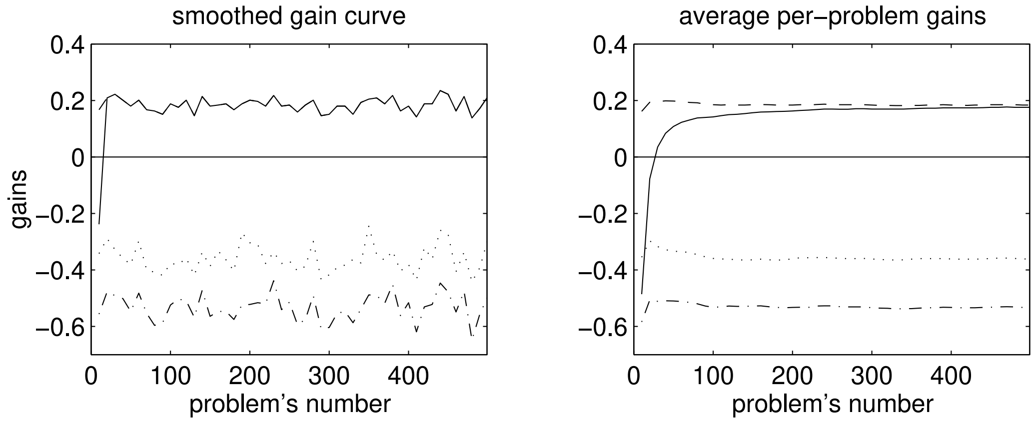


(b) Gain is a discontinuous function of running time (Function 12.5):
It is 1.0 if a problem is solved by the deadline, and −1.0 otherwise.

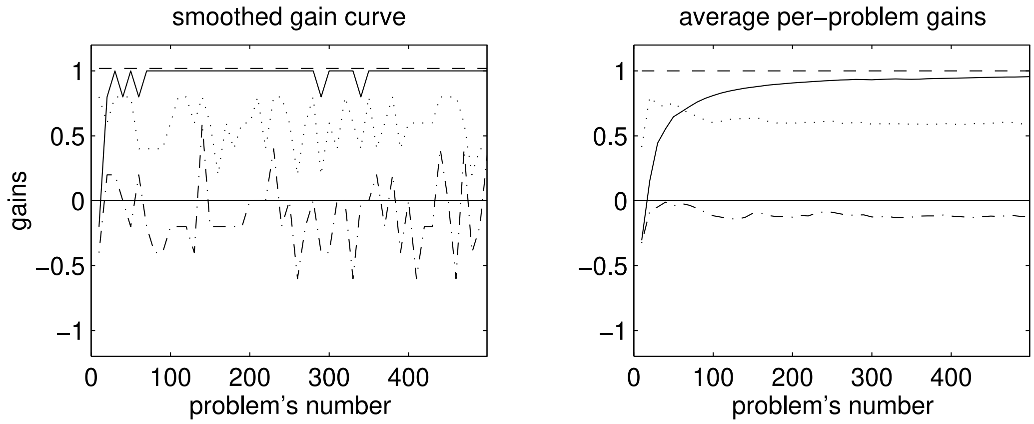


(c) Gain is a complex discontinuous function (Function 12.6), which *cannot* be decomposed into the payment for running time and the reward for solving a problem.

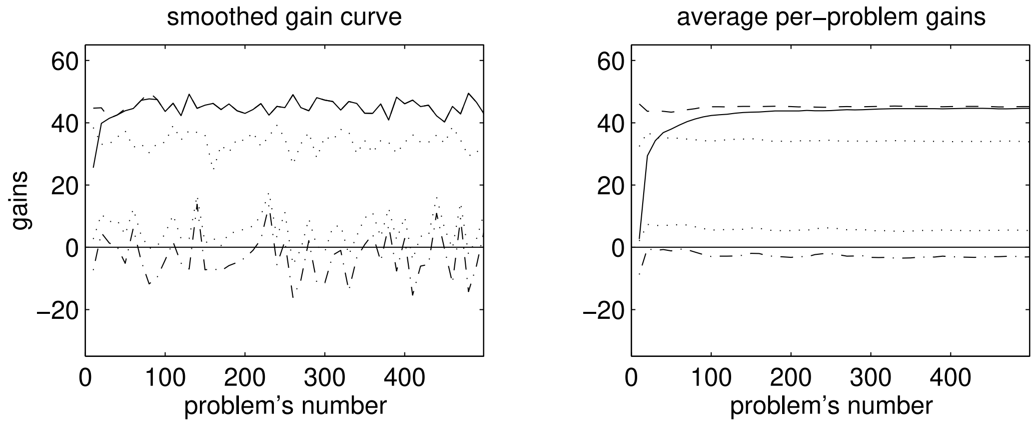
Figure 12.5: Results of applying $\mathcal{S}$HAPER to fifty-problem sequences, with discontinuous gain functions. The graphs include the raw gain curves (left), smoothed curves (middle), and cumulative gains (right); the legend is the same as in Figure 12.3. We do *not* show the lower dotted lines (search with primary effects) for Functions 12.1 and 12.1, because these lines are identical to the dash-and-dot curves (standard PRODIGY search).



(a) System gets a reward only for the optimal solution (Function 12.4).



(b) Gain is a discontinuous function of running time (Function 12.5).



(c) Gain is a complex discontinuous function (Function 12.6).

Figure 12.6: Results for sequences of 500 problems, with discontinuous gain functions. The graphs include the smoothed gain curves (left) and cumulative per-problem gains (right).

12.2 Selection among problem solvers

We next present experiments with a library of four problem solvers (see Figure 12.8), which are based on the SAVTA and SABA search engines. Two solvers perform depth-first search without a cost bound, whereas the other two utilize a *loose bound*, which approximates the doubled cost of an optimal solution. We did not include the LINEAR version of PRODIGY, because its behavior in the Machining Domain proved identical to the behavior of SAVTA.

The heuristic for computing a cost bound is a function of the goal statement; it accounts for the number of goals and for the required quality of machining operations. Recall that the system may choose between rough and fine operations (see Figure 3.33 on page 133); a rough operator incurs a cost of 1, whereas a fine operator is twice as expensive.

We divide the goal literals into two groups: effects of rough operators, such as drilled and polished, and fine effects, such as finely-polished. If the number of literals in the first group is *n-rough*, and that in the other group is *n-fine*, then the loose cost bound is

$$2 \cdot \textit{n-rough} + 4 \cdot \textit{n-fine}.$$

Four solver operators

If the $\mathcal{S}$HAPER system runs with the four solver operators in Figure 12.7, and no description changers, then it generates the four representations illustrated in Figure 12.8. We evaluated the system's ability to choose among them, for Functions 12.1, 12.2, and 12.6.

We give the results in Figures 12.11 and 12.12, where solid lines mark the system's performance, and broken lines show the behavior of each representation. The gains of SABA without cost bounds turned out to be identical to its gains with loose bounds (dashed lines); however, $\mathcal{S}$HAPER had no prior knowledge of this identity, and needed to test both strategies.

The best choice for Functions 12.1 and 12.6 is the SABA algorithm. On the other hand, if the system applies Function 12.2 to compute the gains, it should not solve any problems, because all available algorithms give negative results, regardless of the time bound. The system adopts the right strategy in all three cases, after processing about twenty problems.

Sixteen representations

We now consider a larger library of solver operators, illustrated in Figure 12.9, which includes not only the four search algorithms, but also their synergy with problem-specific changers. In addition, $\mathcal{S}$HAPER utilizes the changer library in Figure 12.1(b), which leads to generating sixteen representations (see Figure 12.10).

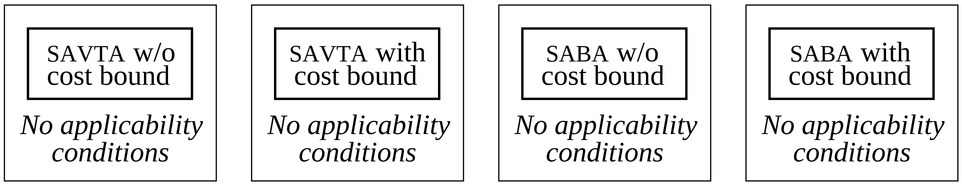


Figure 12.7: Basic solver operators, which are applicable with every domain description. We constructed them from the two main versions of the PRODIGY4 search engine, called SAVTA and SABA.

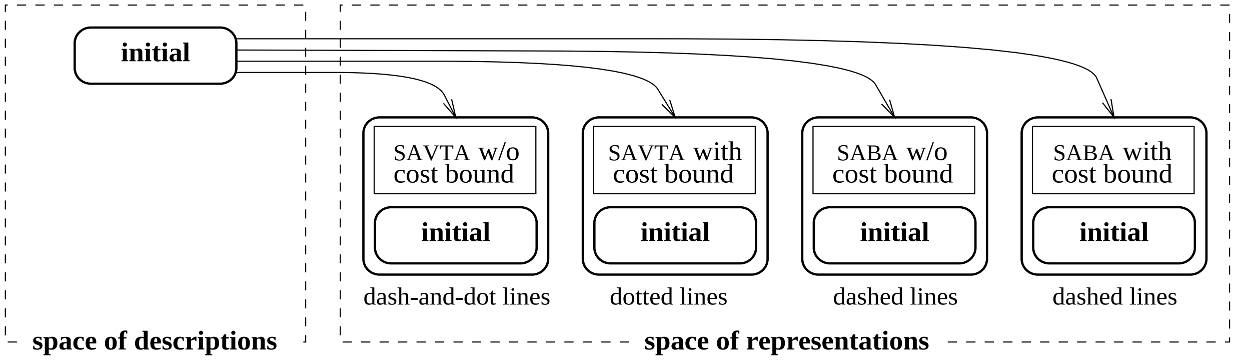


Figure 12.8: Representations constructed without changer operators. The system pairs the solvers (Figure 12.7) with the initial description, which does not include primary effects or abstraction. The small-font subscriptions refer to the corresponding gain curves in Figures 12.11 and 12.12.

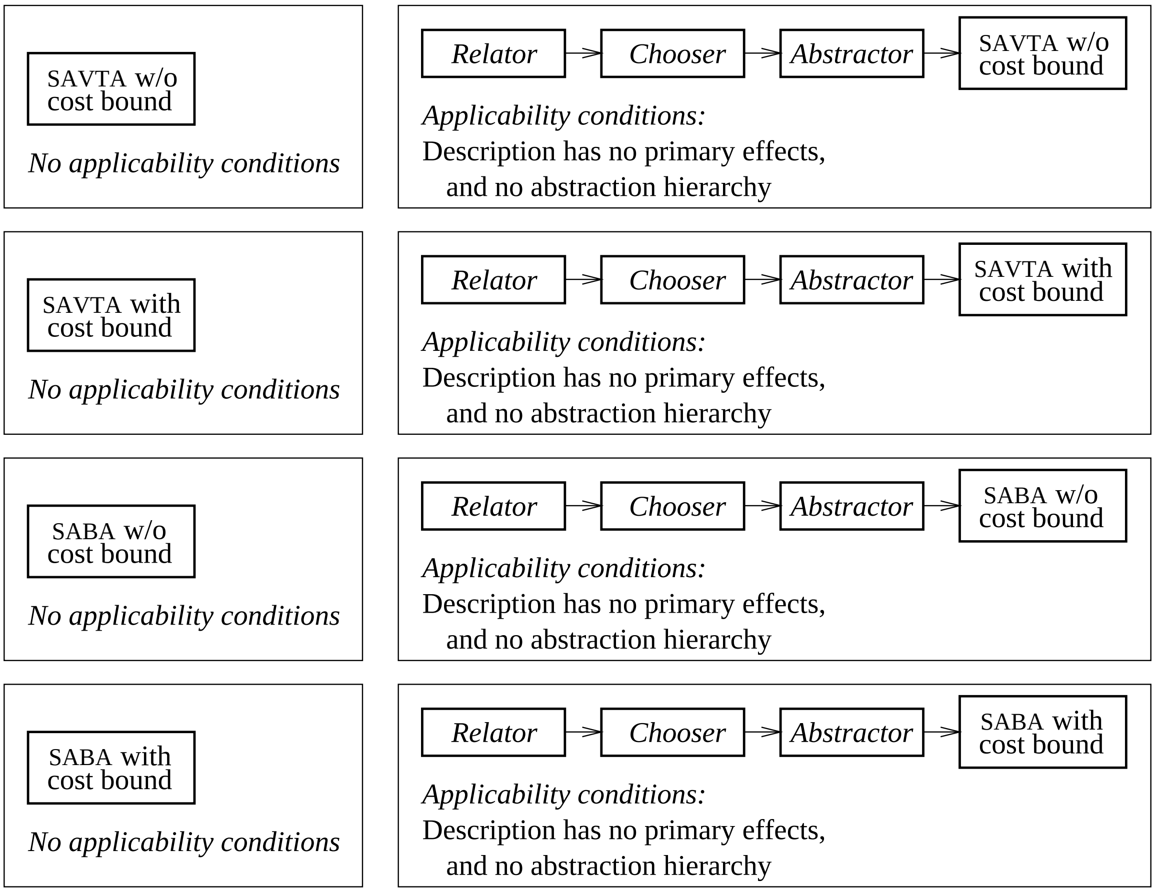


Figure 12.9: Extended set of solver operators for the Machining Domain. We use this set with the library of description changers given in Figure 12.1(b).

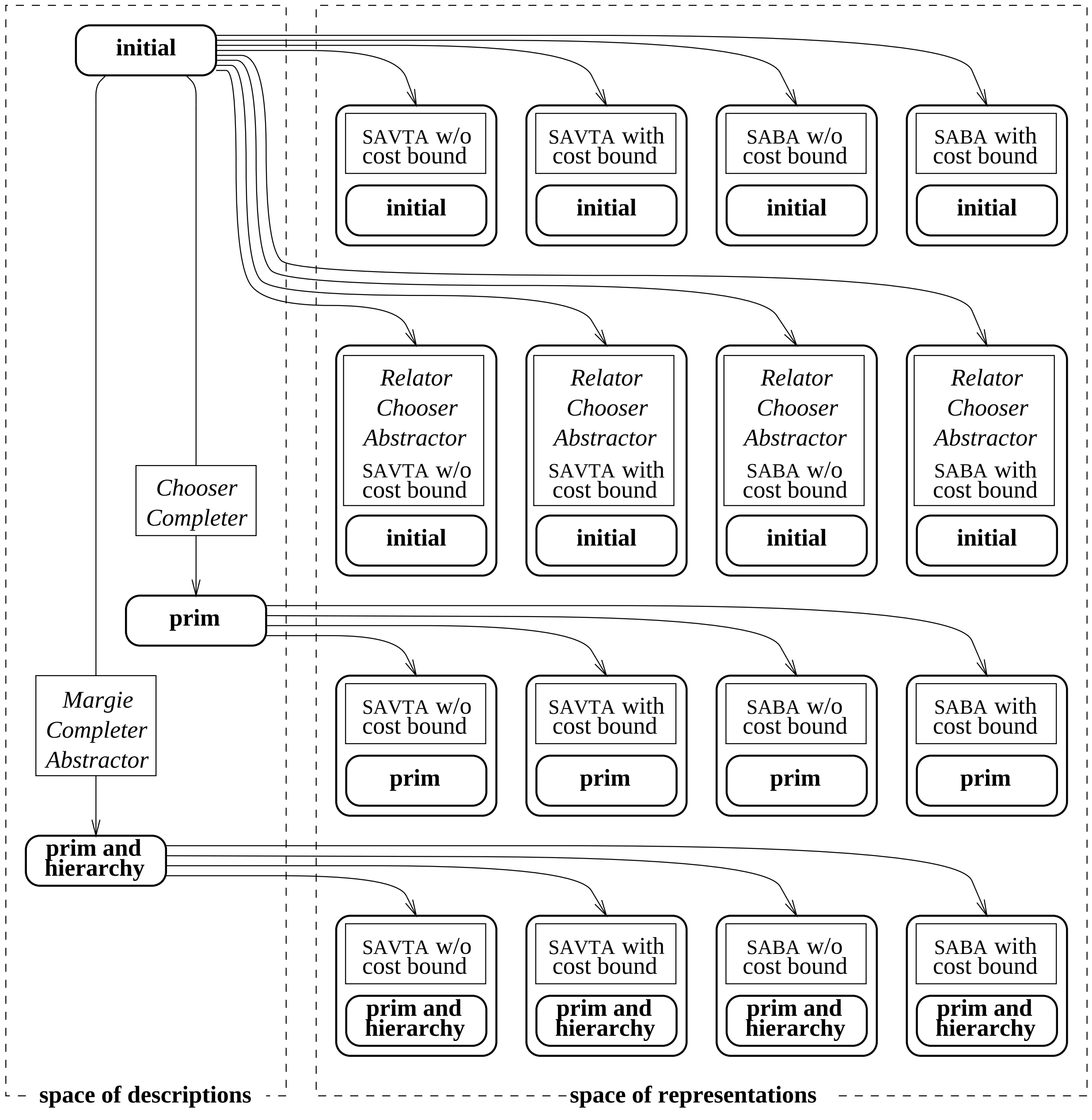


Figure 12.10: Large representation space for the Machining Domain. The system utilizes all solvers in Figure 12.9 and changers in Figure 12.1(b), which give rise to sixteen alternative representations. We give the results of incremental selection among them in Figures 12.13 and 12.14.

	problem sequences				optimal
	short		long		gain
Function 12.1	0.112	(61%)	0.176	(96%)	0.184
Function 12.2	−0.0044	—	−0.0004	—	0
Function 12.6	36.8	(82%)	44.5	(99%)	45.1

Table 12.2: Average per-problem gains in the solver-selection experiments (Figures 12.11 and 12.12). We summarize the results of processing short and long problem sequences, and convert each result into a percentage of the optimal gain.

	representation space				optimal
	large		small		gain
	short problem sequences				
Function 12.1	0.009	(5%)	0.148	(80%)	0.196
Function 12.2	5.11	(71%)	3.91	(55%)	7.17
Function 12.6	33.8	(75%)	38.6	(85%)	45.2
	long problem sequences				
Function 12.1	0.158	(86%)	0.177	(96%)	0.184
Function 12.2	6.93	(97%)	6.67	(93%)	7.17
Function 12.6	44.1	(98%)	44.6	(99%)	45.2

Table 12.3: Average gains in the experiments with the large space, which comprises sixteen representations (Figures 12.13 and 12.14). We also give the results of using a smaller space, with only four representations (Figures 12.3–12.6), and the performance of the best available strategy.

We plot the system's gains in Figures 12.13 and 12.14 (solid lines), and compare them with the results of choosing among four domain descriptions (dash-and-dot lines) and among four problem solvers (dotted lines). We also show the utility of SAVTA with problem-specific descriptions and no cost bounds, which has proved the best strategy.

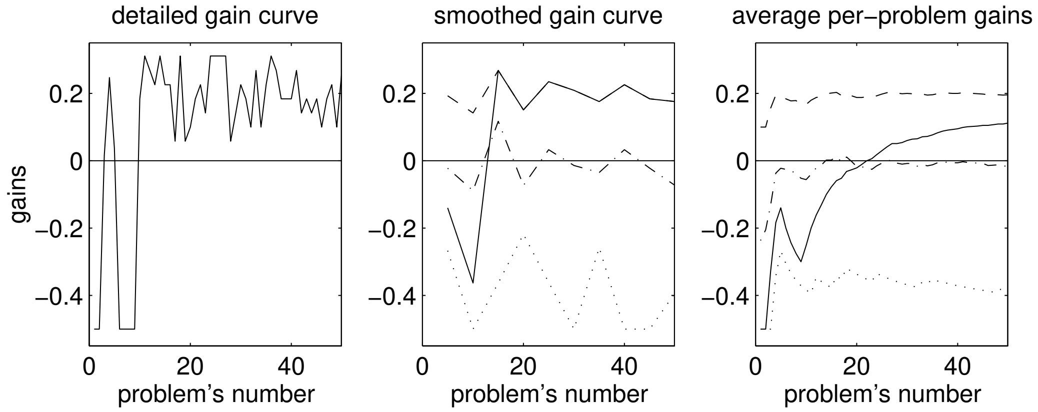
When the system selects among the sixteen representations, or among the four descriptions, it identifies this strategy. On the other hand, if $\mathcal{S}$HAPER employs the four solver algorithms without changers, then it cannot improve the description and, hence, converges to a less effective strategy (dotted lines).

Summary

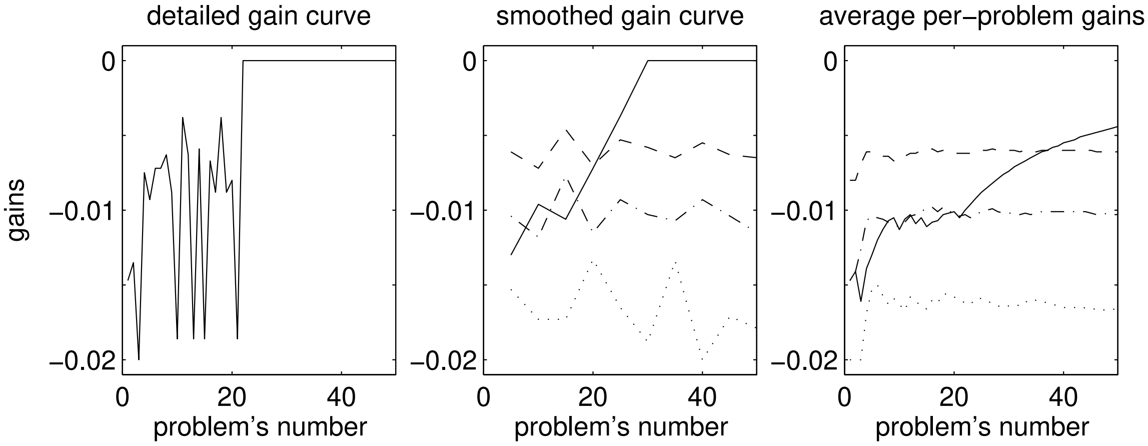
The tests with multiple solvers have re-confirmed the system's ability to identify the right strategy, with moderate initial losses. In Table 12.2, we list the cumulative per-problem gains in the solver-selection experiments. The performance is similar to the results of choosing a description (see Table 12.1), and to the controlled experiments of Section 8.7.

In Table 12.3, we summarize the gains obtained with a space of sixteen representations, and compare them with the results of selecting among four descriptions. When $\mathcal{S}$HAPER expands the larger space, it evaluates more representations and incurs greater initial losses.

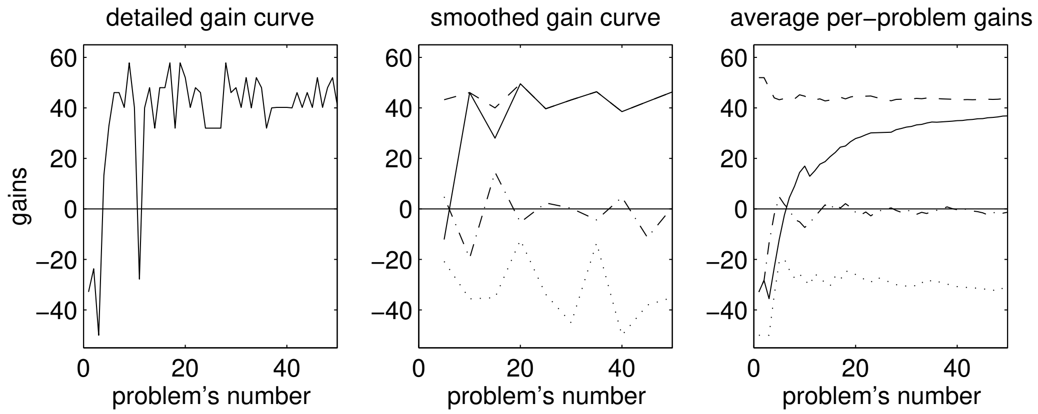
This increase in losses, however, was surprisingly modest. The system rejected most of the representations in the very beginning of the learning sequence. It then performed a more thorough comparison of near-optimal strategies, which did not cause significant deviations from ideal performance.



(a) Gain linearly decreases with the running time (Function 12.1).



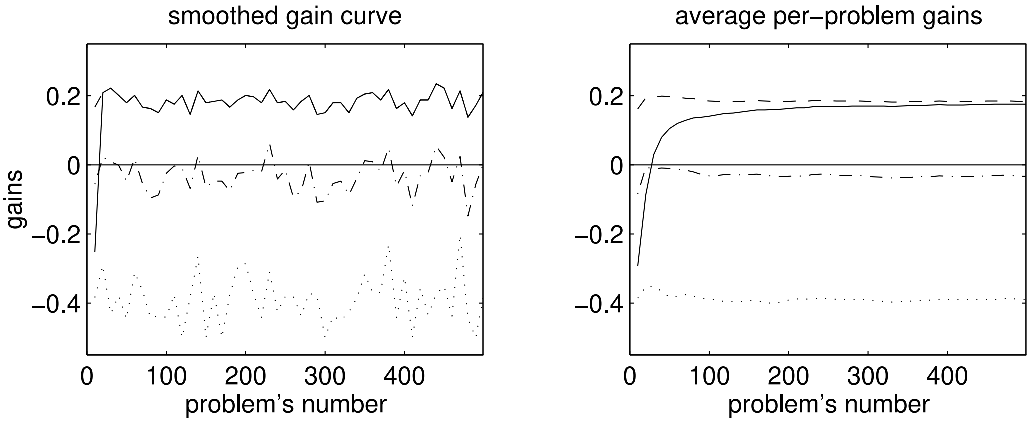
(b) Gain mainly depends on the solution quality (Function 12.2).



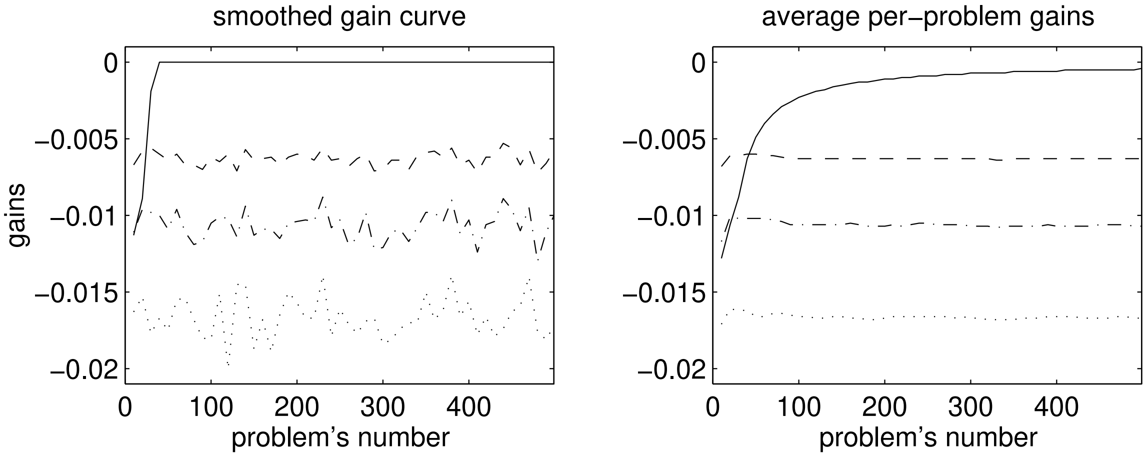
(c) Gain is a discontinuous function of time and solution quality (Function 12.6).

Figure 12.11: Selection among four problem solvers, shown in Figure 12.7. We apply the system to fifty-problem sequences, with no prior performance data. The graphs include the raw results (left), smoothed curves (solid lines, middle), and cumulative per-problem gains (solid lines, right).

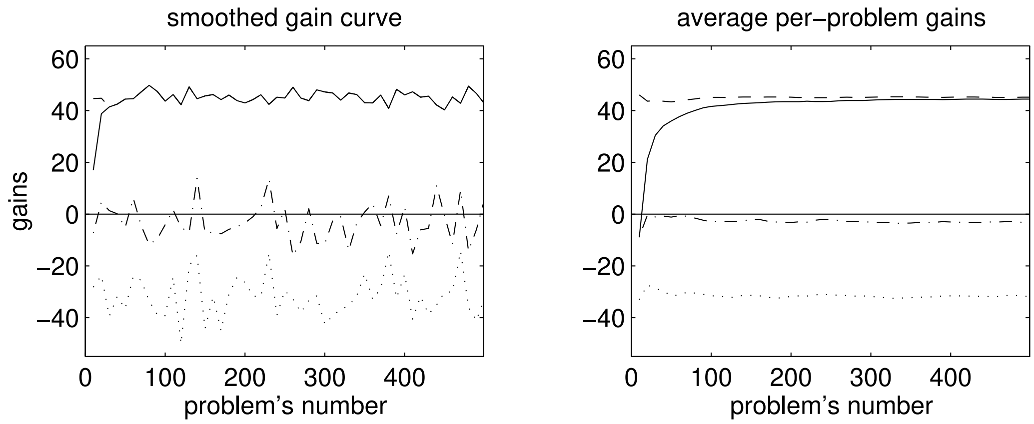
Furthermore, we give the performance of every solver, with no time limit. Specifically, we plot the results of applying SAVTA without cost bounds (dash-and-dot lines), SAVTA with loose cost bounds (dotted lines), and SABA (dashed lines). The behavior of the SABA algorithm without cost bounds is identical to that with loose bounds.



(a) Gain linearly decreases with the running time (Function 12.1).

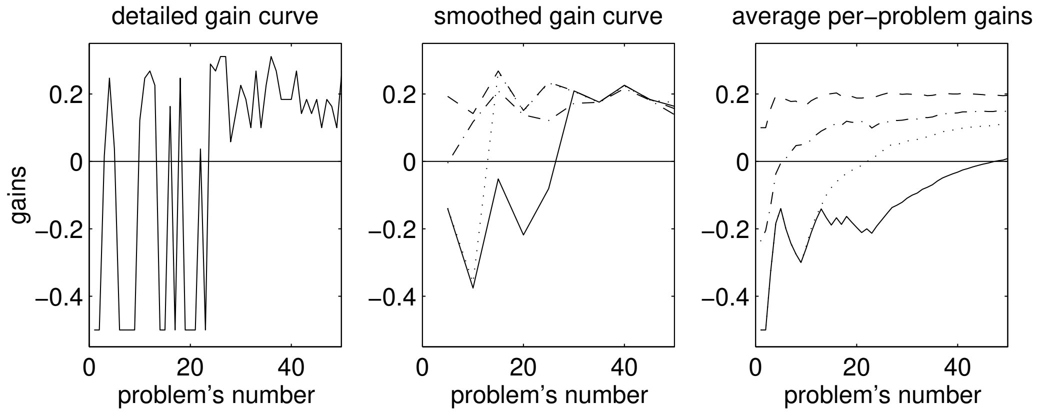


(b) Gain mainly depends on the solution quality (Function 12.2).

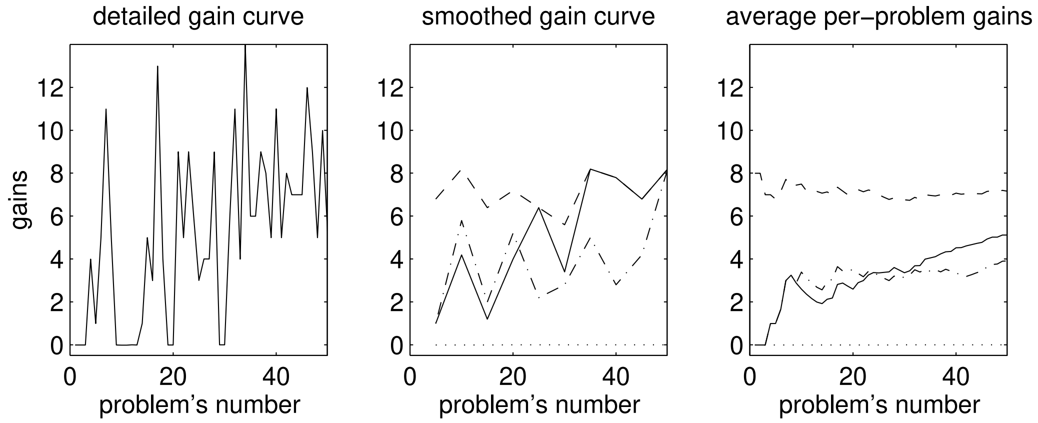


(c) Gain is a discontinuous function of time and solution quality (Function 12.6).

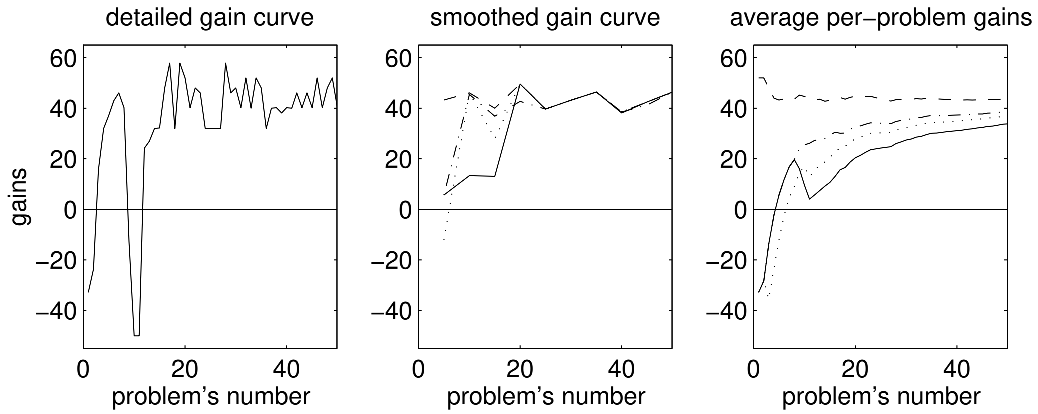
Figure 12.12: Selection among the four solver operators, on sequences of 500 problems. We give the smoothed gain curves (solid lines, left) and cumulative gains (solid lines, right), as well as the performance of each available representation without time bounds (broken lines).



(a) Gain linearly decreases with the running time (Function 12.1).



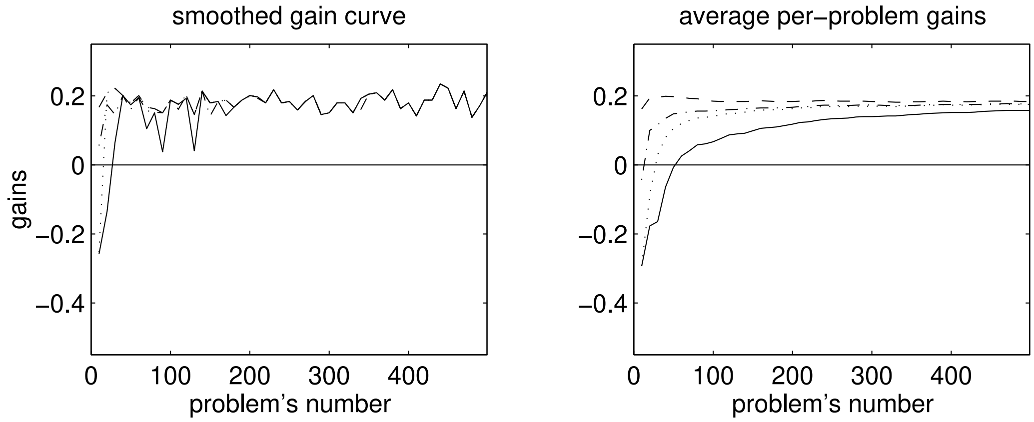
(b) Gain mainly depends on the solution quality (Function 12.2).



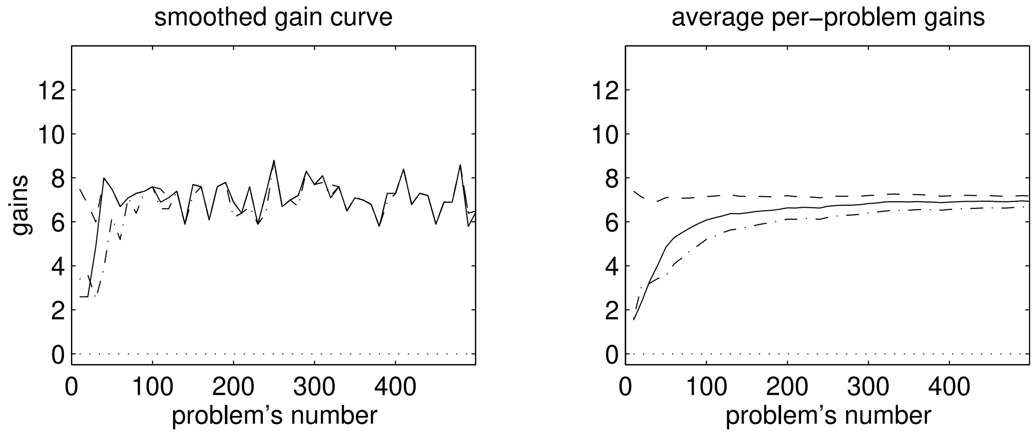
(c) Gain is a discontinuous function of time and solution quality (Function 12.6).

Figure 12.13: Problem solving with sixteen alternative representations (see Figure 12.10). The system processes a set of fifty problems, and gradually selects appropriate representations and bounds. The graphs contain the raw gain data (left), smoothed gains (solid lines, middle), and cumulative results (solid lines, right).

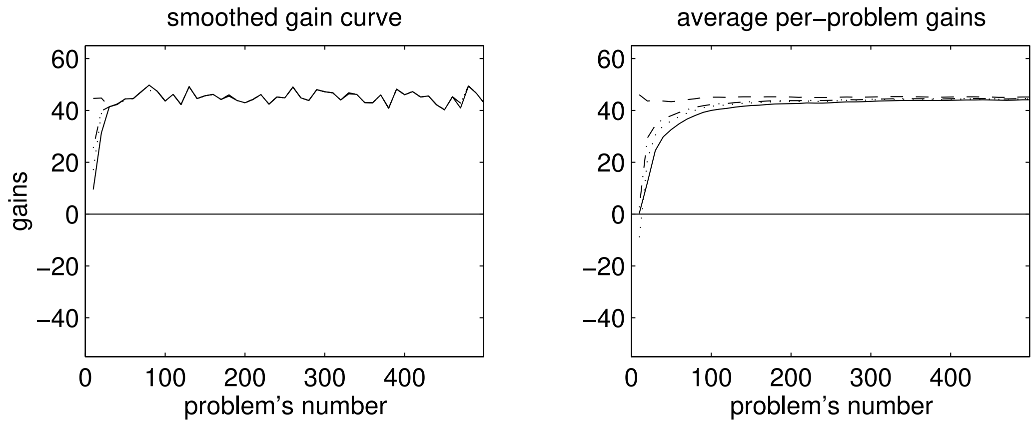
We compare these data with the results on two smaller-scale tasks: choice among the four representations in Figure 12.2 (dash-and-dot lines), and selection among the solvers in Figure 12.7 (dotted lines). We also show the performance of SAVTA with goal-specific abstraction and no time bounds (dashed lines), which is the best strategy for all three gain functions.



(a) Gain linearly decreases with the running time (Function 12.1).



(b) Gain mainly depends on the solution quality (Function 12.2).



(c) Gain is a discontinuous function of time and solution quality (Function 12.6).

Figure 12.14: Applying the system to 500-problem sequences, with the large representation space (solid lines). We also plot the performance of the best available strategy (dashed lines), and the results of using the smaller spaces (the other broken lines); the legend is the same as in Figure 12.13.

12.3 Different time bounds

When $\mathcal{S}$HAPER tests the available representations, it sets larger-than-optimal time bounds, which facilitate exploration. The system includes a knob parameter for controlling the exploratory bounds. We review the role of this parameter and evaluate the utility of different knob settings.

Exploration knob

In Section 8.4.2, we have described a three-step mechanism for choosing a time bound. First, the statistical module estimates the optimal bound and computes the expected gain $g_{\max}$, as well as its standard deviation $\sigma_{\max}$. Second, it considers larger bounds, and estimates the gain g and deviation σ for each bound. Third, it identifies the maximal bound whose gain is "not much different" from optimal. By default, $\mathcal{S}$HAPER considers g sufficiently close to $g_{\max}$ if $\frac{g_{\max}-g}{\sqrt{\sigma_{\max}^2+\sigma^2}} \leq 0.1$; thus, it selects the largest time bound that satisfies this condition.

The human user has the option of changing the limit on the $\frac{g_{\max}-g}{\sqrt{\sigma_{\max}^2+\sigma^2}}$ ratio. If this limit is less than 0.1, then $\mathcal{S}$HAPER chooses bounds that are closer to the estimated optimum; hence, it incurs smaller losses during the early stages of learning, but may fail to identify the globally optimal bound. This strategy is effective for short problem sequences, which do not allow the amortization of losses. On the other hand, a large limit causes a more thorough exploration, at the expense of greater initial losses.

We tried several settings of this exploration knob and found out that the overall performance is surprisingly insensitive to changes in the knob value. The optimal setting varies across domains, gain functions, and problem sequences; however, the default always gives near-optimal results.

Small and large bounds

First, we re-ran the sixteen-representation experiments (see Figure 12.8) for two smaller values of the exploration knob, 0.02 and 0.05. In Figures 12.15 and 12.16, we plot the *differences* between the resulting gains and the default-knob gains. The performance was very similar to the default case, and the changes in the cumulative gains were negligibly small.

Second, we tested two larger knob values, 0.2 and 0.5, and again observed a close-to-default behavior (see Figures 12.17 and 12.18). Since PRODIGY was able to solve all machining problems in feasible time, large exploratory bounds did *not* cause high initial losses. The experiments with other domains have revealed a more significant impact of the exploration knob on $\mathcal{S}$HAPER's gains (see Sections 13.3, 14.3, and 15.3).

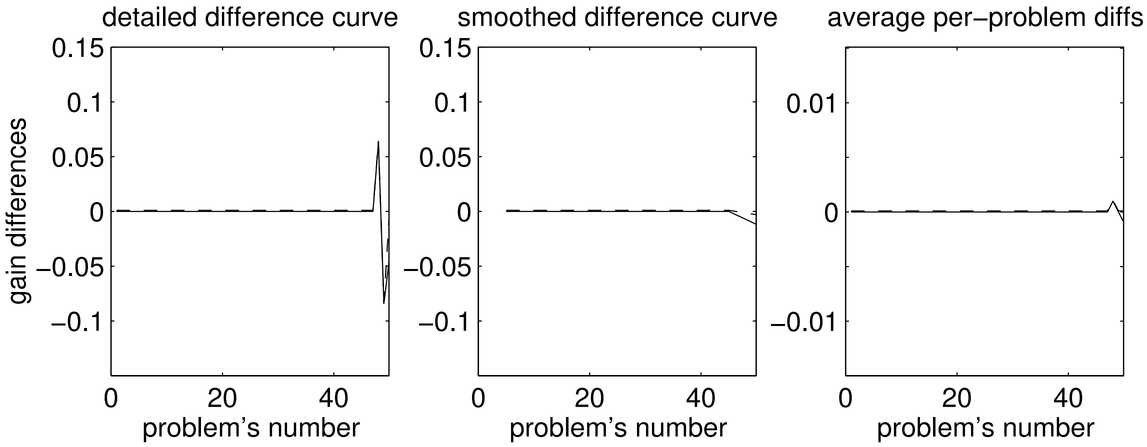
Summary

In Table 12.4, we sum up the dependency of average gains on the exploration knob, and convert the results of various settings into the percentages of the default gains. In most cases, the changes in the knob value affected the cumulative gain by less than a percent.

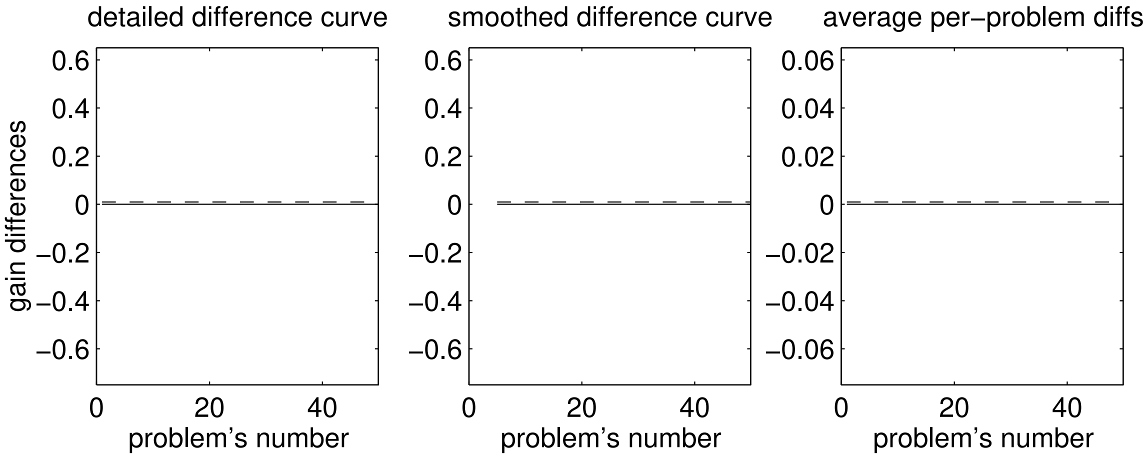
	small knob values				default	large knob values			
	0.02		0.05		0.1	0.2		0.5	
	short problem sequences								
Function 12.1	0.008	(89%)	0.009	(100%)	0.009	0.011	(122%)	0.008	(89%)
Function 12.2	5.113	(100%)	5.113	(100%)	5.113	5.113	(100%)	5.113	(100%)
Function 12.6	33.50	(99%)	33.84	(100%)	33.80	33.84	(100%)	33.76	(100%)
	long problem sequences								
Function 12.1	0.159	(101%)	0.161	(102%)	0.158	0.156	(99%)	0.160	(101%)
Function 12.2	6.963	(100%)	6.961	(100%)	6.931	6.957	(100%)	6.943	(100%)
Function 12.6	43.97	(100%)	43.84	(99%)	44.13	44.13	(100%)	44.07	(100%)

Table 12.4: Dependency of the cumulative per-problem gains on the exploration knob. We list the average gains for different knob values, and give the corresponding percentages of the default-strategy gains, rounded to whole percents.

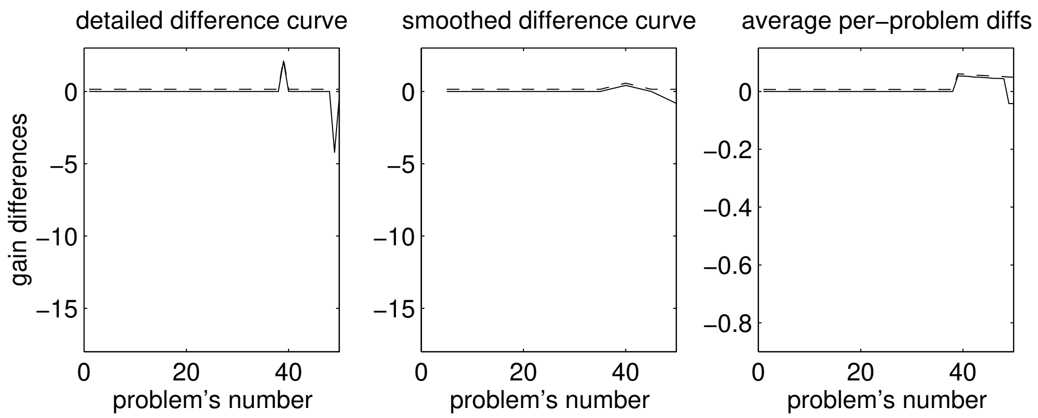
The only notable exception was the fifty-problem experiment with Function 12.1, which revealed a possibility of 22% improvement on the default behavior (first line of Table 12.4); however, note that the absolute gain difference was *not* significant. Since the cumulative gain was close to zero, due to initial losses, the small absolute improvement translated into a large percentage. If SHAPER processes more problems, it accumulates a larger gain, and the relative improvement becomes small (see the results for 500 problems).



(a) Gain linearly decreases with the running time (Function 12.1).

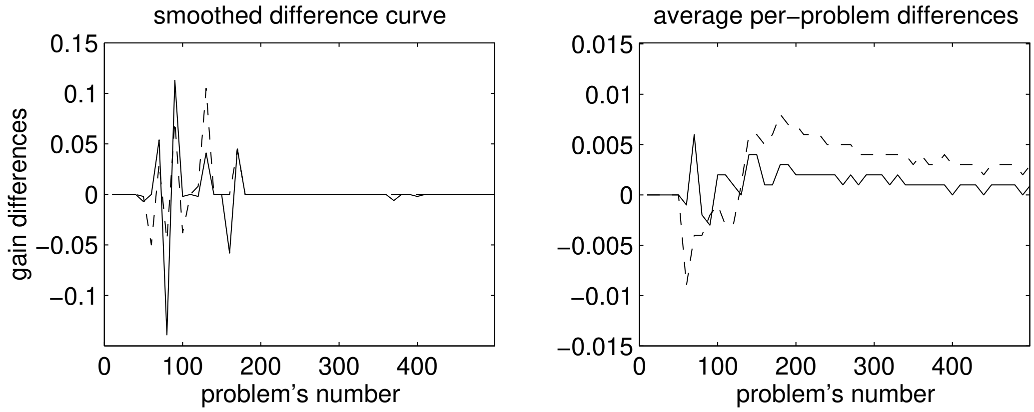


(b) Gain mainly depends on the solution quality (Function 12.2).

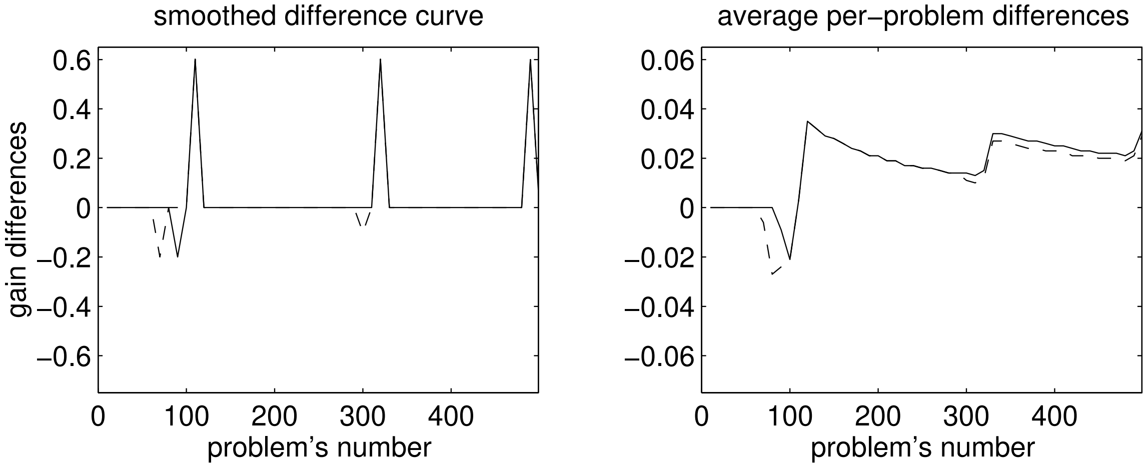


(c) Gain is a discontinuous function of time and solution quality (Function 12.6).

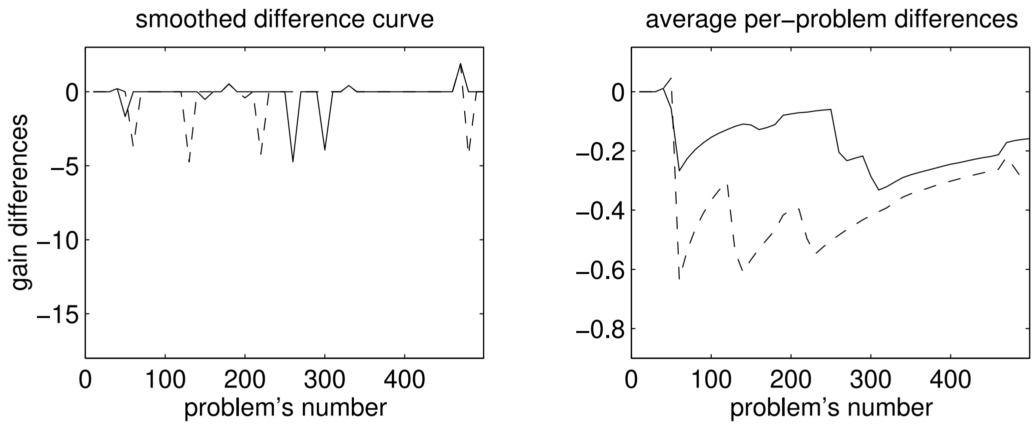
Figure 12.15: Performance with small values of the exploration knob, on sequences of fifty problems. The solid lines show the *differences* between the gains with the knob value 0.02 and that with the value 0.1. Similarly, the dashed lines mark the differences between the 0.05-knob gains and the 0.1-knob gains. We plot the difference for each of the fifty problems (left), as well as smoothed curves (middle) and cumulative per-problem differences (right).



(a) Gain linearly decreases with the running time (Function 12.1).

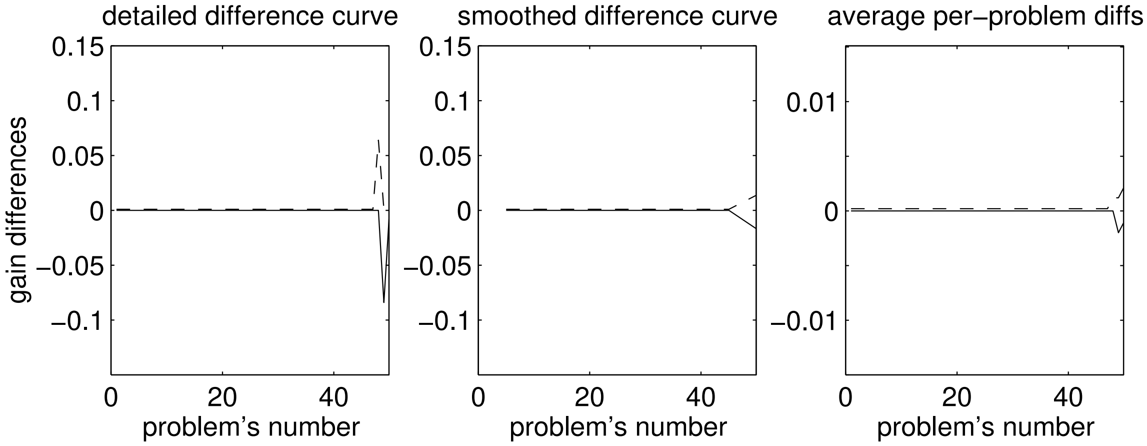


(b) Gain mainly depends on the solution quality (Function 12.2).

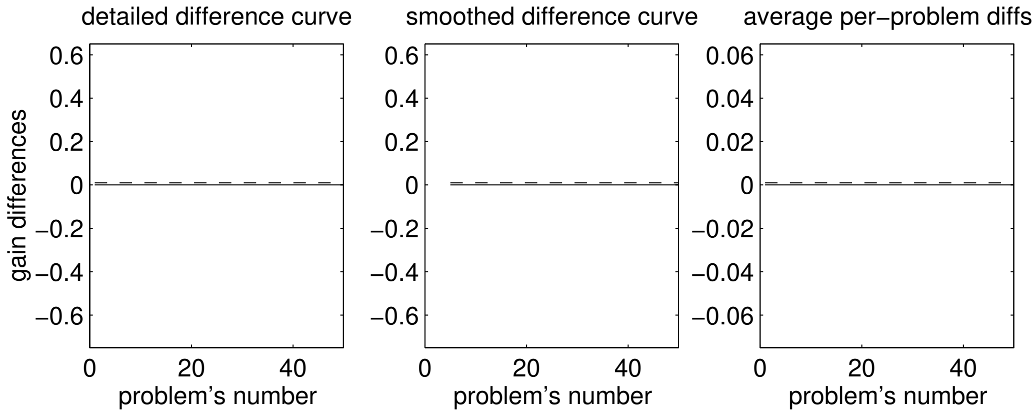


(c) Gain is a discontinuous function of time and solution quality (Function 12.6).

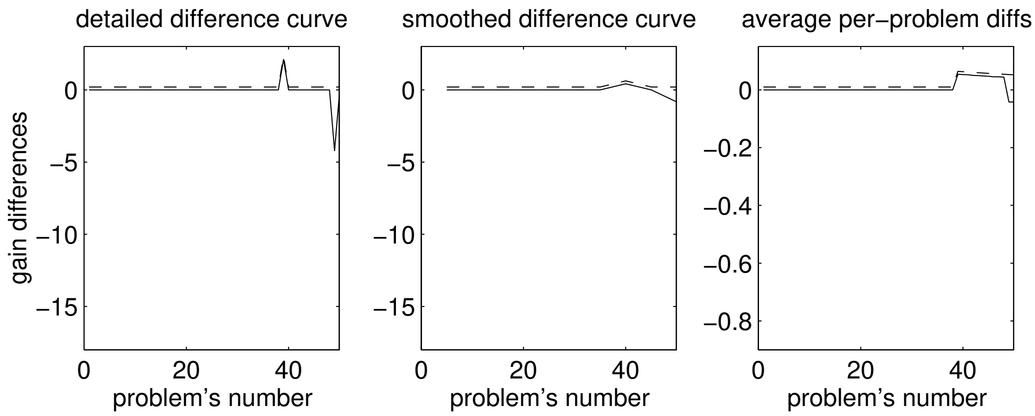
Figure 12.16: Performance with the knob values 0.02 and 0.05, on sequences of 500 problems. We plot the smoothed gain-difference curves (left) and cumulative per-problem differences (right), using the legend of Figure 12.15.



(a) Gain linearly decreases with the running time (Function 12.1).

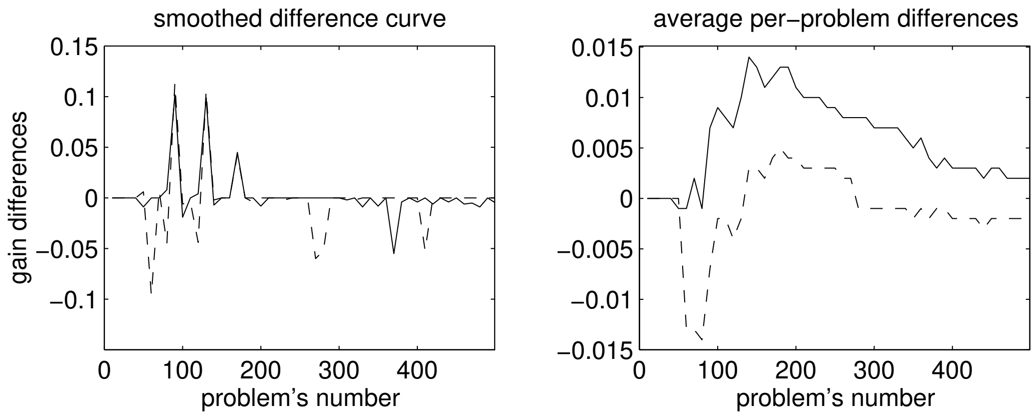


(b) Gain mainly depends on the solution quality (Function 12.2).

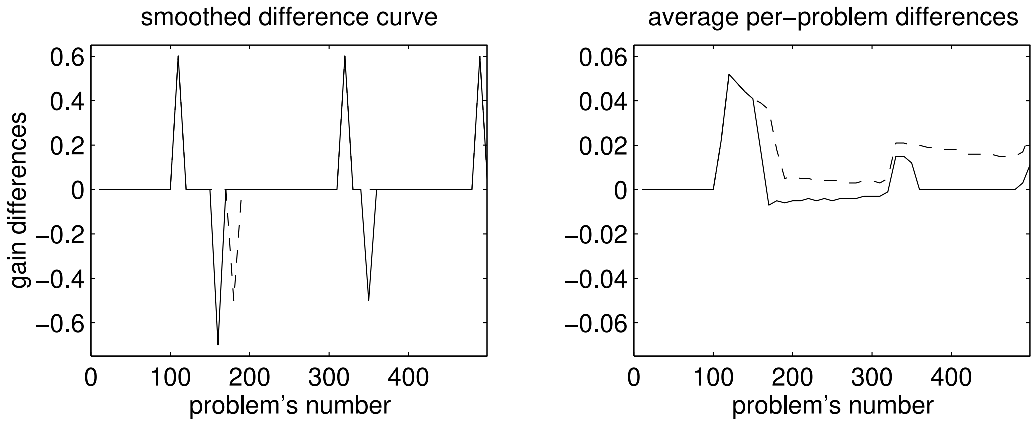


(c) Gain is a discontinuous function of time and solution quality (Function 12.6).

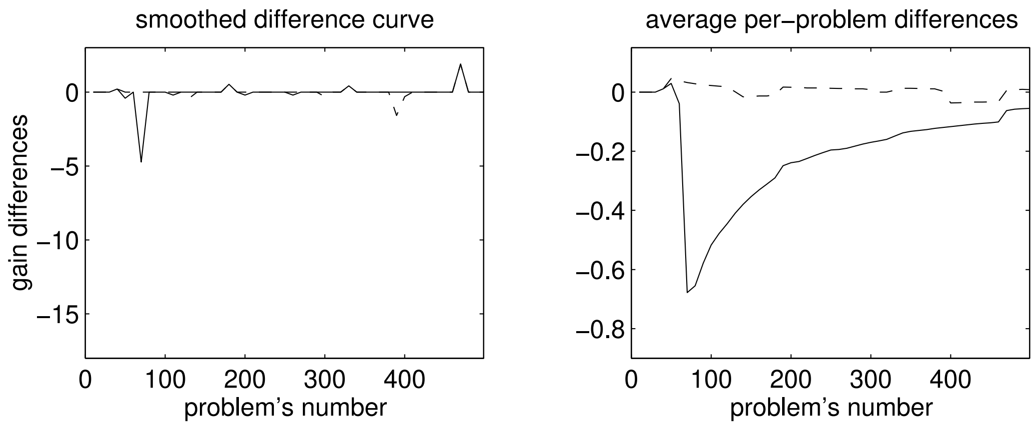
Figure 12.17: Processing fifty-problem sequences with large values of the exploration knob. The solid lines show the differences between the 0.5-knob gains and the default gains. Similarly, the dashed curves illustrate the difference between the knob value 0.2 and the default setting.



(a) Gain linearly decreases with the running time (Function 12.1).



(b) Gain mainly depends on the solution quality (Function 12.2).



(c) Gain is a discontinuous function of time and solution quality (Function 12.6).

Figure 12.18: Results for sequences of 500 problems, with large knob values. We plot the gain-difference curves for the values 0.5 (solid lines) and 0.2 (dashed lines).

Chapter 13

Sokoban Domain

The Sokoban puzzle (see Section 3.7.2) is harder than the Machining Domain and causes the expansion of larger search spaces. It includes problems that require impractically long search, as well as some unsolvable problems. Furthermore, its encoding is based on functional types, which occasionally lead to incompleteness of PRODIGY search.

In most cases, PRODIGY cannot find near-optimal solutions to Sokoban problems, and the enforcement of cost bounds may result in gross inefficiency; hence, we apply the available search engines *without* cost limits. The use of primary effects and abstraction helps to reduce the search time (see Sections 3.7.2 and 4.4). On the other hand, goal-specific description changes have proved useless, and we have *not* utilized the *Relator* algorithm in the Sokoban experiments.

13.1 Choice among three alternatives

We begin with small-scale selection tasks, which require the evaluation of three alternative representations. The $\mathcal{S}$HAPER system has to find appropriate search strategies for the gain functions in Figure 13.1, which do *not* account for the solution quality.

First, we consider a linear dependency of gain on running time, with a fixed reward for solving a problem:

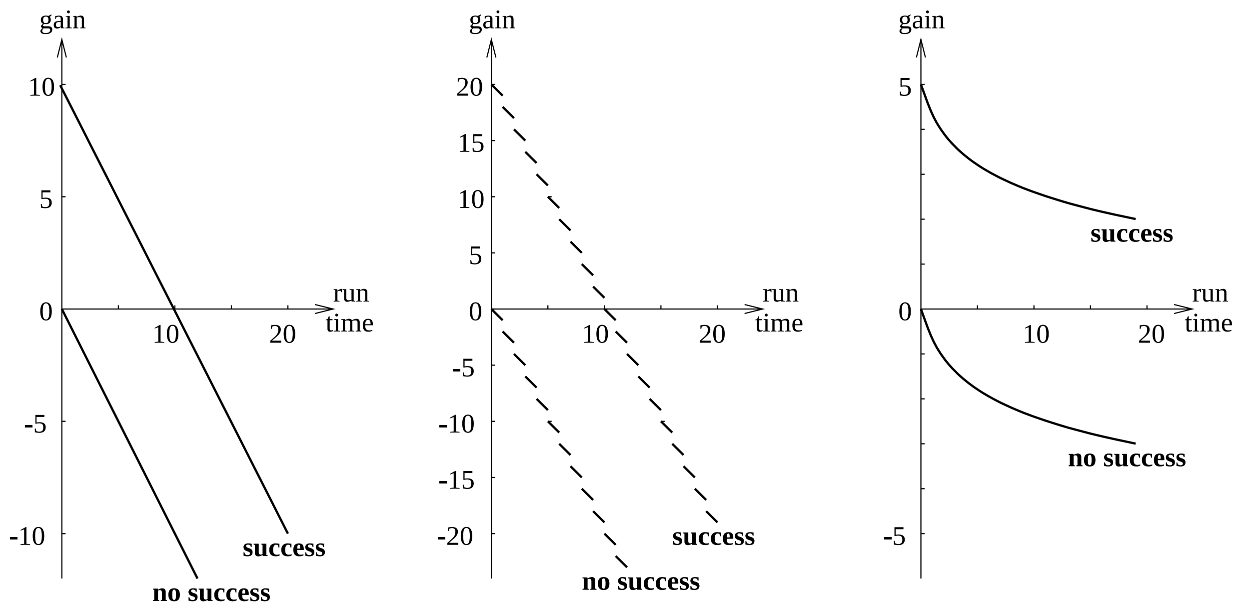
$$gain = \begin{cases} 10 - time, & \text{if success} \\ -time, & \text{if failure or interrupt} \end{cases} \tag{13.1}$$

Then, we experiment with a discontinuous gain function, and show that this discontinuity does not have a negative effect on the learning process:

$$gain = \begin{cases} 20 - time - \lfloor time \rfloor, & \text{if success} \\ -time - \lfloor time \rfloor, & \text{if failure or interrupt} \end{cases} \tag{13.2}$$

Finally, $\mathcal{S}$HAPER needs to identify the right strategy for a linear dependency of gain on the logarithm of running time:

$$gain = \begin{cases} 5 - \ln(time + 1), & \text{if success} \\ -\ln(time + 1), & \text{if failure or interrupt} \end{cases} \tag{13.3}$$



(a) Function 13.1 (b) Function 13.2 (c) Function 13.3

Figure 13.1: Gain functions in the Sokoban experiments. Note that we do not account for the solution quality, and do not distinguish between failure terminations and interrupts.

Selecting a domain description

The first experiment involves the LINEAR search engine and three changer operators, as shown in Figure 13.2(a,b). The system inputs the domain description given in Figure 3.36 (page 137), which has *no* primary effects or abstraction, and constructs two new descriptions (see Figure 13.2c). One of them is based on the primary effects given in Figure 3.37 (page 139), and the other is the abstraction in Figure 4.32 (page 185).

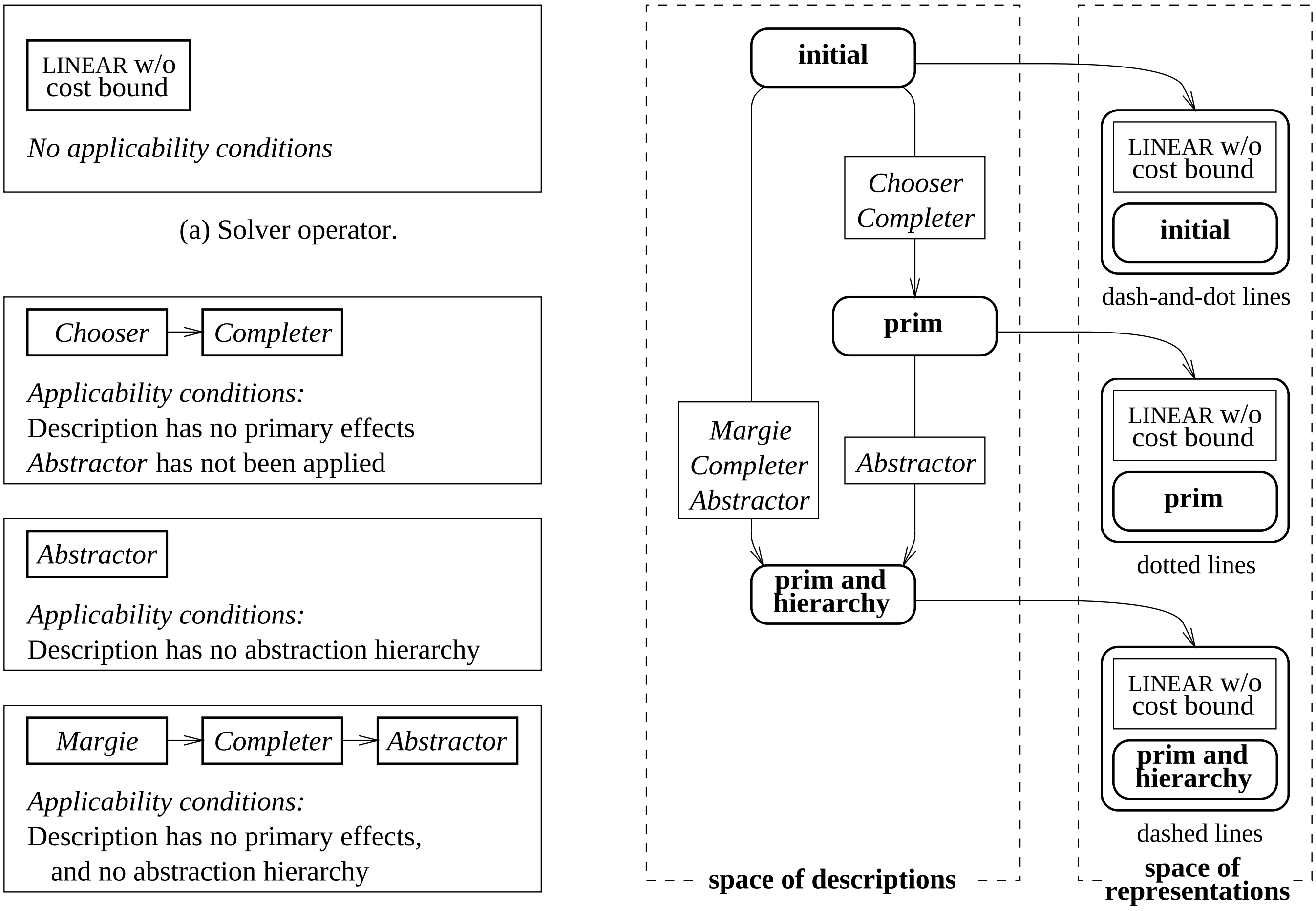
The graphs in Figures 13.4 and 13.5 comprise the results of selecting a representation and time bound for each of the gain functions (solid lines). We also show the behavior of the three available representations, with the *optimally* chosen bounds (broken lines).

The system identified the right description and time limit in all three cases. The local maxima of gains between problems 50 and 100 (see the right-hand graphs in Figure 13.5) were due to a random fluctuation in problem difficulty: $\mathcal{S}$HAPER encountered several easy tasks, which resulted in unusually high gains.

Selecting a problem solver

The next experiment was based on the algorithm library illustrated in Figure 13.3(a), which included three search engines and no changer operators. We provided a domain encoding with both primary effects and abstraction, and $\mathcal{S}$HAPER paired it with the three available engines, thus obtaining three representations (see Figure 13.3b). The results of choosing among them are summarized in Figures 13.6 and 13.7.

The system found appropriate solvers and time bounds for Functions 13.2 and 13.3; however, it made a wrong choice for Function 13.1. When $\mathcal{S}$HAPER ran with this function,



(a) Solver operator.

(b) Changer operators.

(c) Expanded spaces.

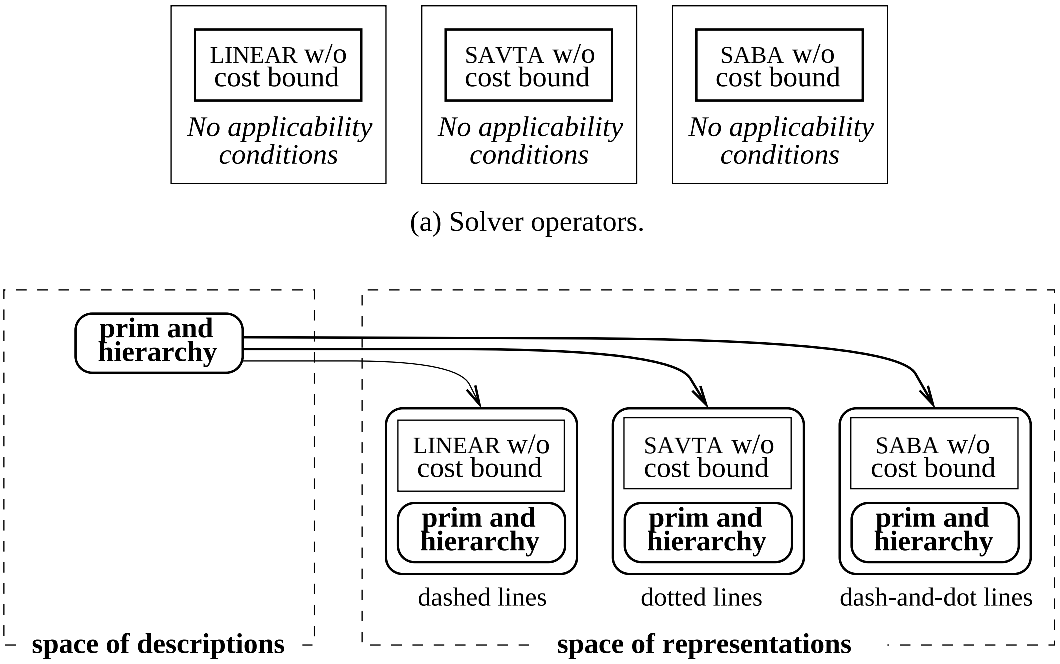
Figure 13.2: Solver and changer operators in the experiments on choosing a domain description, and the resulting space of representations. Note that the application of *Abstractor* to the initial description does *not* lead to a multi-level hierarchy. On the other hand, when the system invokes *Abstractor* after selecting primary effects, it produces the same hierarchy as *Margie*. The subscriptions specify the corresponding gain curves in Figures 13.4 and 13.5.

it correctly identified LINEAR as the best solver, but converged to a suboptimal time limit (1.62 seconds), which led to noticeably smaller gains than the optimal bound (3.02 seconds).

Summary

The system selected the right representation in all experiments, and found an appropriate time bound in all but one case. The learning curves are similar to that in the Machining Domain. In particular, the speed of converging to an effective strategy is the same in both domains: $\mathcal{S}$HAPER usually adopts the right behavior after solving fifty to a hundred problems.

In Table 12.1, we list the average per-problem gains for each of the learning sequences, and compare them with the results of applying the optimal search strategies. The percentage values of cumulative gains are smaller than in the Machining Domain (see Table 12.1),



(a) Solver operators.

(b) Expanded spaces.

Figure 13.3: Solver operators for the Sokoban Domain, which correspond to three search engines in the PRODIGY architecture. The system pairs them with a given domain description, which includes primary effects and abstraction, and thus expands a space of three representations. The small-font subscriptions refer to the gain curves in Figures 13.6 and 13.7.

	choice among descriptions		choice among solvers		optimal gain
	short problem sequences				
Function 13.1	0.625	(37%)	0.245	(14%)	1.711
Function 13.2	2.989	(99%)	0.886	(29%)	3.007
Function 13.3	0.689	(56%)	0.249	(20%)	1.240
	long problem sequences				
Function 13.1	0.610	(76%)	0.421	(52%)	0.802
Function 13.2	1.940	(88%)	1.939	(88%)	2.213
Function 13.3	0.477	(85%)	0.371	(66%)	0.563

Table 13.1: Cumulative per-problem gains in the experiments with small representation spaces. We summarize the results of choosing among three descriptions (Figures 13.4 and 13.5), and among three search engines (Figures 13.6 and 13.7). For every cumulative value, we show the corresponding percentage of the optimal gain.

	primary effects	primary effects and abstraction
search reduction	> 500	> 5000
	gain-increase factor	
Function 13.1	1.1	1.8
Function 13.2	1.7	2.3
Function 13.3	1.1	2.7

Table 13.2: Time savings due to primary effects and abstraction, and the respective growth of gains. Note that a huge search-reduction factor translates into a modest increase in cumulative gains.

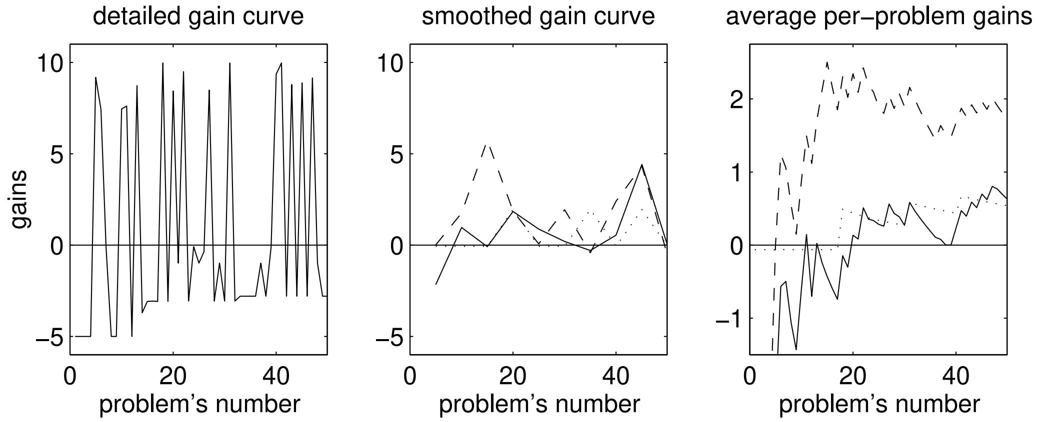
due to greater initial losses. Since many Sokoban problems require infeasibly long search, experimentation with large time bounds incurs high losses during the early stages of learning.

Running time versus gains

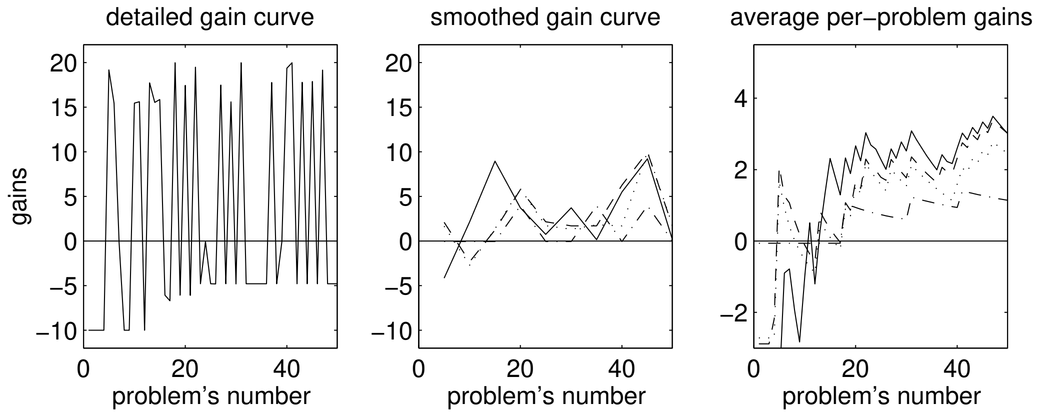
The experiments have shown that a drastic reduction of search time may *not* lead to a proportional upsurge of expected gain. Recall that primary effects increase the speed of solving Sokoban problems by three orders of magnitude (Section 3.7.2), and their synergy with abstraction gives an even greater improvement (Section 4.4). On the other hand, the resulting gain increase is relatively modest: the improvement factor ranges from 1.1 to 2.7 (see Table 13.2).

Intuitively, a much faster procedure is *not* necessarily much better in practice. For example, a software company usually cannot achieve a sharp growth of profits by purchasing faster machines. In many cases, a speed increase gives little practical advantage. For instance, the user of a word processor would hardly benefit from a fast computer.

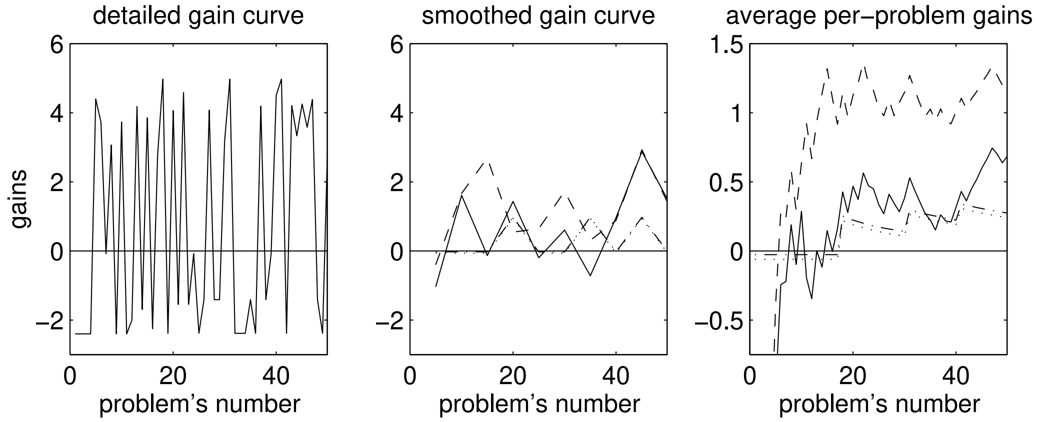
The results of applying SHAPER to other domains have supported this intuition: the gain-increase factor has always been less than 5. Moreover, we have not been able to design any utility function that would cause a much larger increase in cumulative gain.



(a) Gain linearly decreases with the running time (Function 13.1).

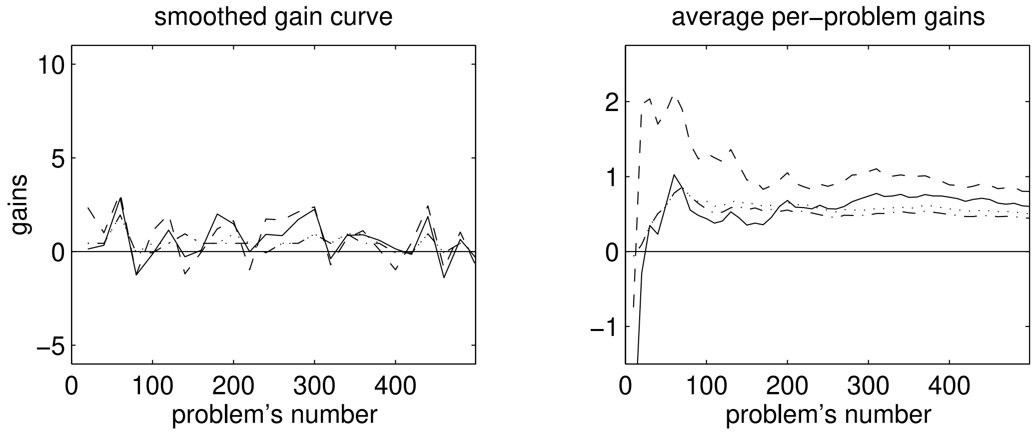


(b) Gain is a discontinuous function of time (Function 13.2).

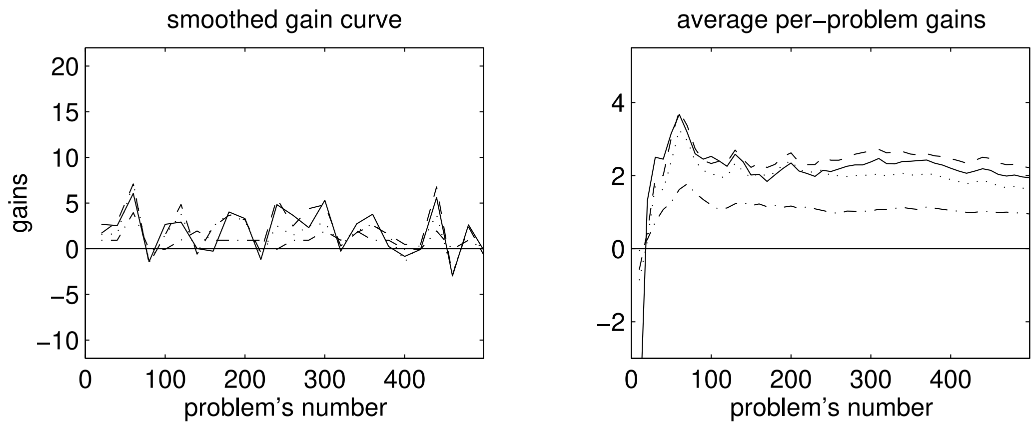


(c) Gain linearly decreases with the logarithm of time (Function 13.3).

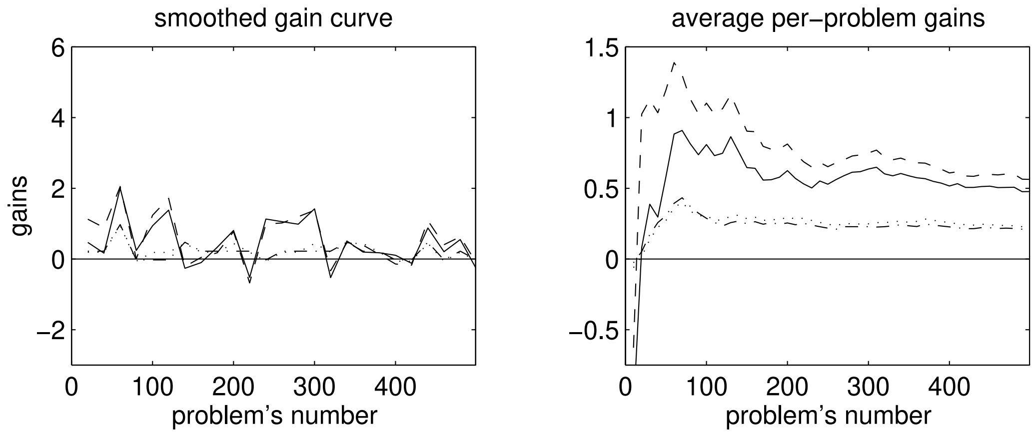
Figure 13.4: Selection among three domain descriptions (see Figure 13.2). We plot the raw results for fifty-problem sequences (left), as well as smoothed gain curves (solid lines, middle) and cumulative gains (solid lines, right). In addition, the graphs show the optimal performance of the available representations, which include the standard PRODIGY search (dash-and-dot lines), primary effects (dots), and abstraction (dashes). Note that we do *not* show the dash-and-dot curves for Function 13.1, because they are identical to the dotted lines.



(a) Gain linearly decreases with the running time (Function 13.1).

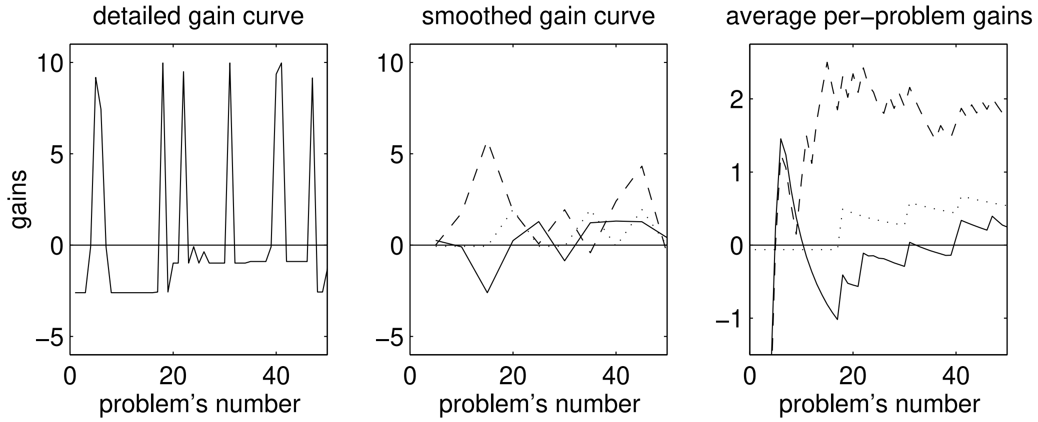


(b) Gain is a discontinuous function of time (Function 13.2).

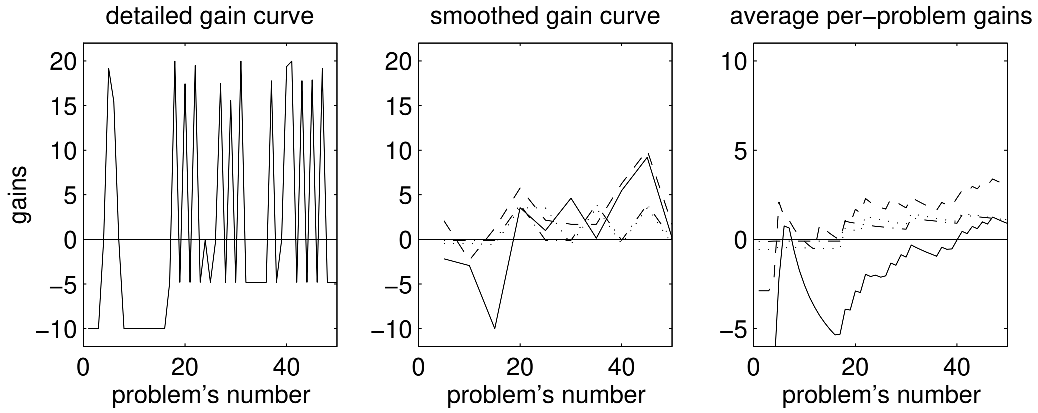


(c) Gain linearly decreases with the logarithm of time (Function 13.3).

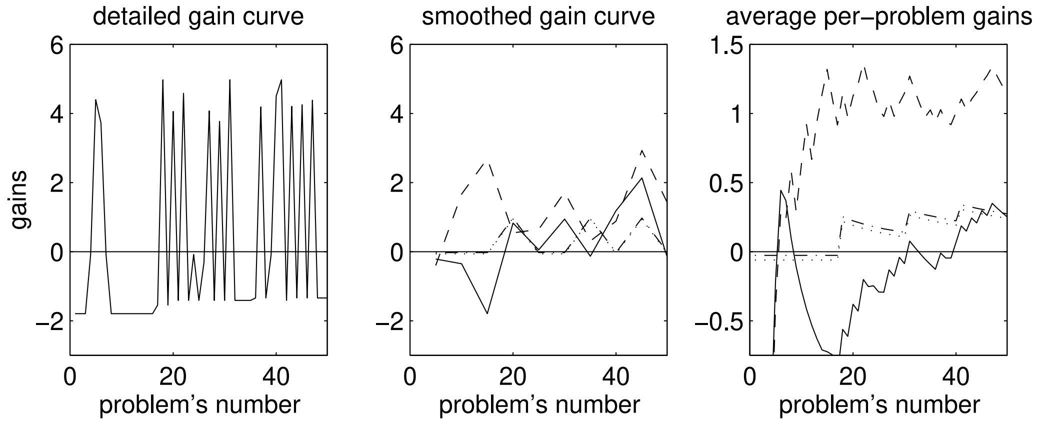
Figure 13.5: Performance on 500-problem sequences, with three alternative descriptions. We give the smoothed gain curves (solid lines, left) and cumulative per-problem gains (solid lines, right), as well as the optimal behavior of each description (broken lines).



(a) Gain linearly decreases with the running time (Function 13.1).

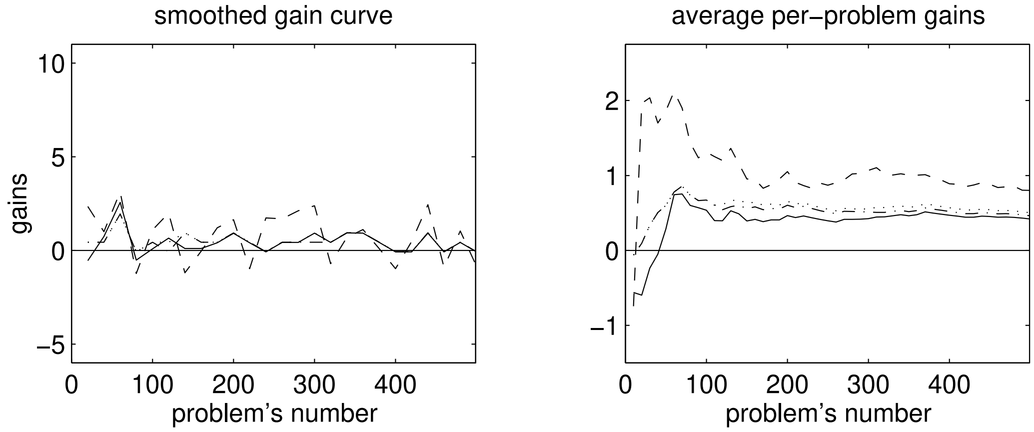


(b) Gain is a discontinuous function of time (Function 13.2).

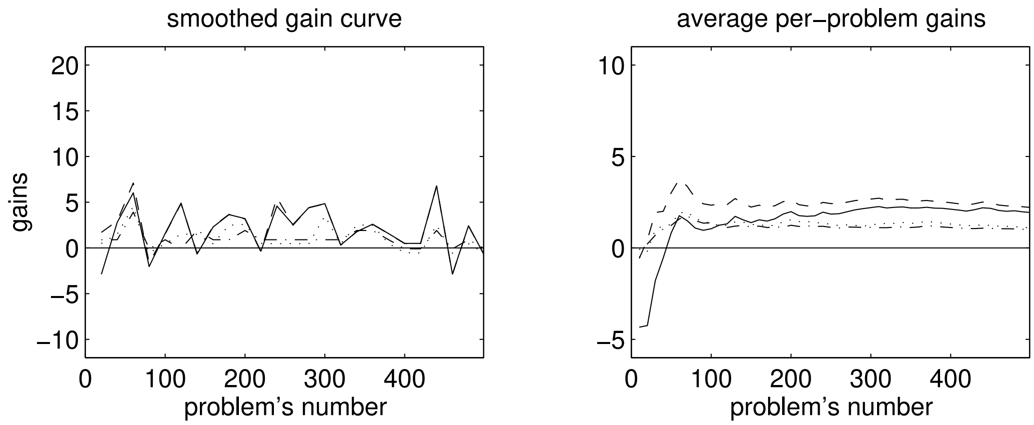


(c) Gain linearly decreases with the logarithm of time (Function 13.3).

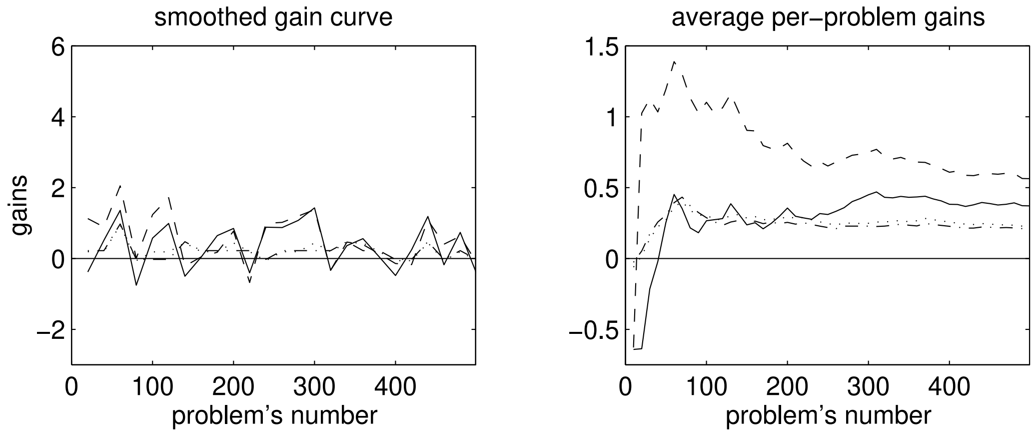
Figure 13.6: Processing fifty-problem sequences and choosing among three search engines (see Figure 13.3). The graphs include the raw gains (left), smoothed curves (middle), and cumulative results (right). We also show the optimal performance of LINEAR (dashed lines), SAVTA (dotted lines), and SABA (dash-and-dot lines). We do *not* plot the dash-and-dot curves for Function 13.1, because they completely coincide with the dotted curves.



(a) Gain linearly decreases with the running time (Function 13.1).



(b) Gain is a discontinuous function of time (Function 13.2).



(c) Gain linearly decreases with the logarithm of time (Function 13.3).

Figure 13.7: Results for sequences of 500 problems, with a library of three search engines. We give the smoothed curves (left) and cumulative per-problem gains (right).

13.2 Larger representation space

If $\mathcal{S}$HAPER utilizes the changer operators in Figure 13.2(b) along with the solvers in Figure 13.3(a), then it generates nine alternative representations, illustrated in Figure 13.8. We give the results of choosing among them, and compare the system's gains with the performance on the two smaller-scale tasks.

Simple gain functions

The $\mathcal{S}$HAPER system found the right representation and time bound for all three gain functions described in Section 13.1. In Figures 13.9 and 13.10, we present the resulting gains (solid lines), and compare them with the outcomes of choosing among three descriptions (dash-and-dot lines) and among three search engines (dotted lines). The graphs also show the behavior of the optimal strategies (dashed lines), which are based on the LINEAR algorithm with primary effects and abstraction.

Note that, when $\mathcal{S}$HAPER utilizes nine representations, it converges to the optimal strategy after processing 150 to 200 problems, whereas the choice among three alternatives requires at most a hundred problems. Moreover, the larger selection task causes higher initial losses, because the system begins by testing all representations with large time bounds.

Distinguishing failures from interrupts

We next consider more complex functions, which differentiate failure terminations from interrupts. The first of them is a linear dependency on search time, with a partial reward for a failure termination:

$$gain = \begin{cases} 10 - \mathit{time}, & \text{if success} \\ 5 - \mathit{time}, & \text{if failure} \\ -\mathit{time}, & \text{if interrupt} \end{cases} \tag{13.4}$$

The second test involves a variable reward, which is proportional to the area of the Sokoban grid. Thus, the system gets more points for solving harder problems:

$$gain = \begin{cases} \mathit{area} - \mathit{time}, & \text{if success} \\ 0.5 \cdot \mathit{area} - \mathit{time}, & \text{if failure} \\ -\mathit{time}, & \text{if interrupt} \end{cases} \tag{13.5}$$

Finally, the third gain function is the product of the grid size and linear dependency on time. In this case, a large-scale Sokoban problem may lead not only to a big reward, but also to a significant loss:

$$gain = \begin{cases} \mathit{area} \cdot (10 - \mathit{time}), & \text{if success} \\ \mathit{area} \cdot (5 - \mathit{time}), & \text{if failure} \\ -\mathit{area} \cdot \mathit{time}, & \text{if interrupt} \end{cases} \tag{13.6}$$

The system found the right strategy for each of the three functions (see Figures 13.11 and 13.12), and its behavior proved similar to that with the simpler gain functions. The results re-confirm that the large representation space causes slower convergence and greater initial losses than the two smaller spaces.

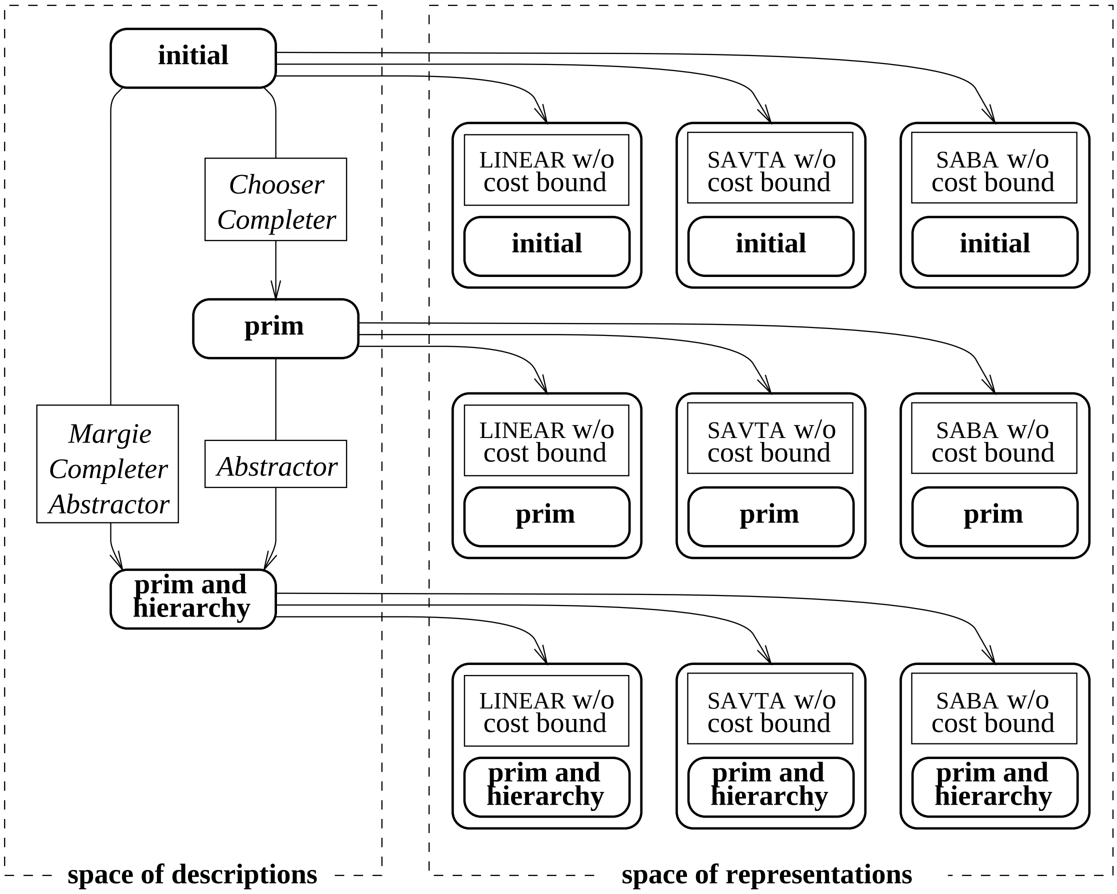


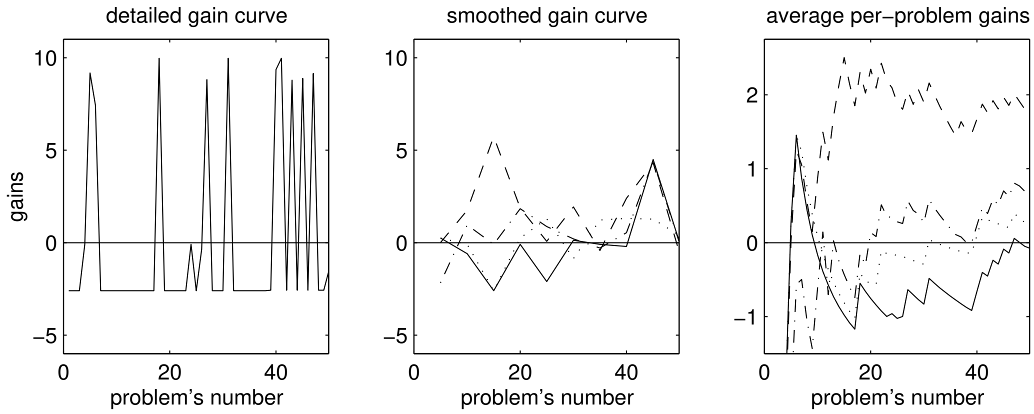
Figure 13.8: Space of nine representations in the Sokoban Domain. The system applies the changer algorithms given in Figure 13.2(b), and then pairs the domain descriptions with the solvers in Figure 13.3(b). We show the results of choosing among these representations in Figures 13.9–13.12.

Summary

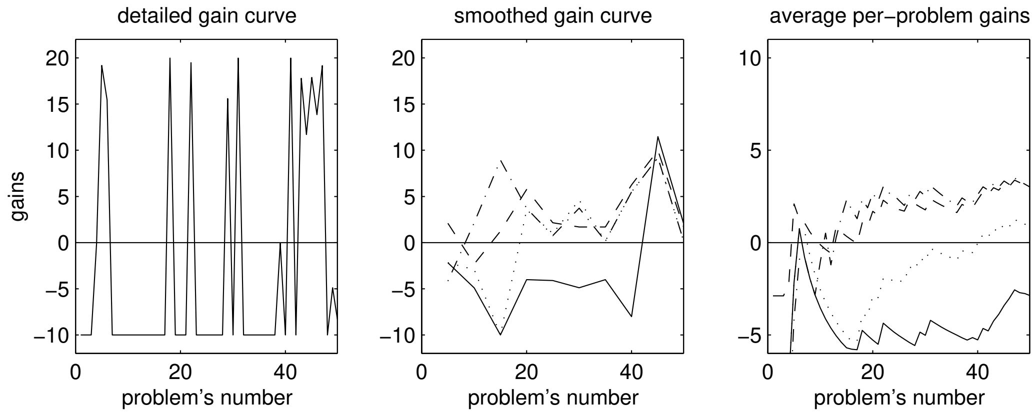
The system converged to the right choice of a representation and time bound in all six cases, and its performance proved stable across different gain functions. In Table 13.3, we summarize the cumulative gains and compare them with the analogous data for smaller representation spaces. Observe that the learning behavior is similar to that in the Machining Domain (Section 12.2) and to the artificial tests (Section 8.7).

	selection among nine representations (solid lines)		three descriptions (dash-and-dot lines)		three solvers (dotted lines)		optimal gain (dashed lines)
	short problem sequences						
Function 13.1	−0.077	—	0.625	(37%)	0.245	(14%)	1.711
Function 13.2	−2.876	—	2.989	(99%)	0.886	(29%)	3.007
Function 13.3	−0.762	—	0.689	(56%)	0.249	(20%)	1.240
Function 13.4	0.172	(10%)	0.699	(41%)	1.361	(80%)	1.711
Function 13.5	3.70	(32%)	6.43	(56%)	4.17	(36%)	11.43
Function 13.6	25.5	(97%)	25.5	(97%)	25.7	(98%)	26.2
	long problem sequences						
Function 13.1	0.330	(41%)	0.610	(76%)	0.421	(52%)	0.802
Function 13.2	1.169	(53%)	1.940	(88%)	1.939	(88%)	2.213
Function 13.3	0.334	(59%)	0.477	(85%)	0.371	(66%)	0.563
Function 13.4	0.382	(48%)	0.407	(51%)	0.747	(93%)	0.802
Function 13.5	6.82	(77%)	7.50	(85%)	7.59	(86%)	8.78
Function 13.6	15.1	(83%)	17.8	(98%)	17.8	(98%)	18.1

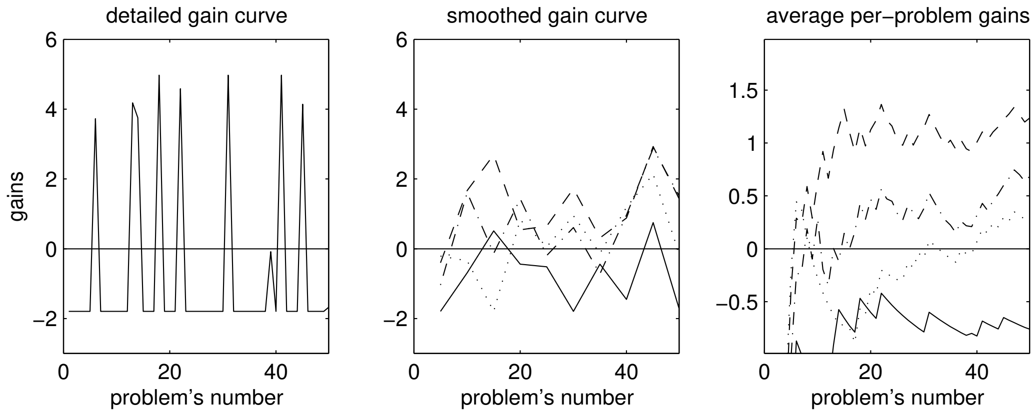
Table 13.3: Average per-problem gains in the experiments with the space of nine representations (see Figures 13.9–13.12). We compare them with the analogous data for the two smaller spaces, and convert each gain value into a percentage of the optimal gain.



(a) Gain linearly decreases with the running time (Function 13.1).

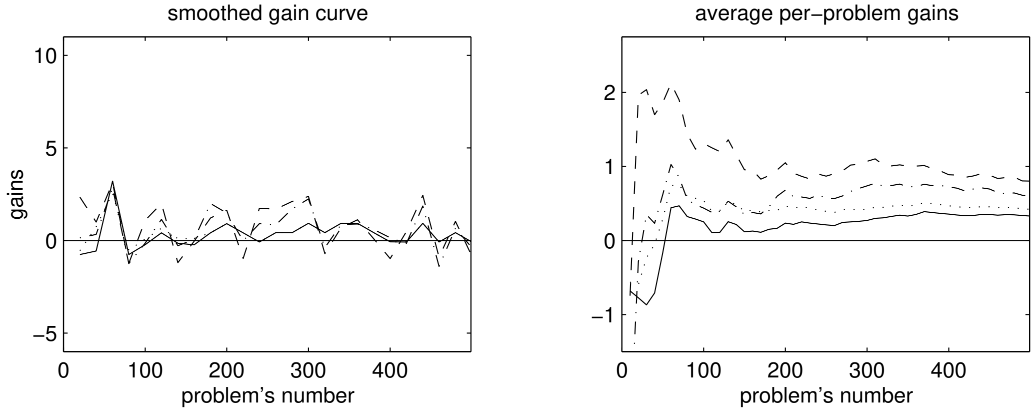


(b) Gain is a discontinuous function of time (Function 13.2).

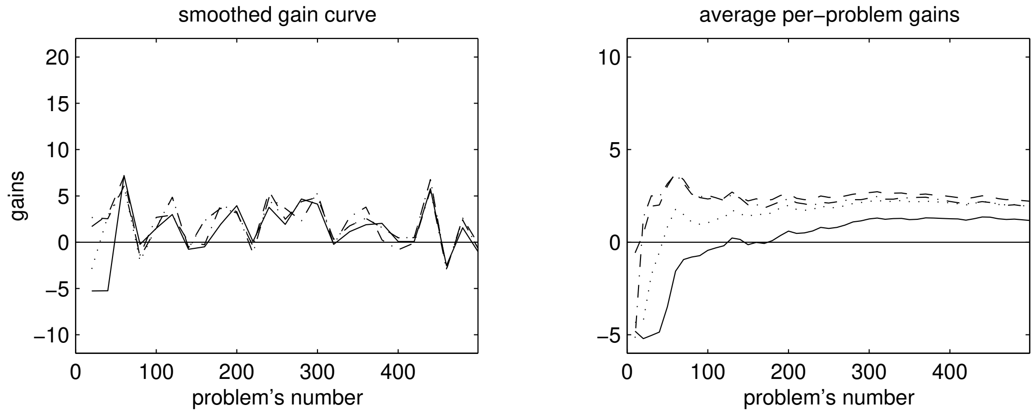


(c) Gain linearly decreases with the logarithm of time (Function 13.3).

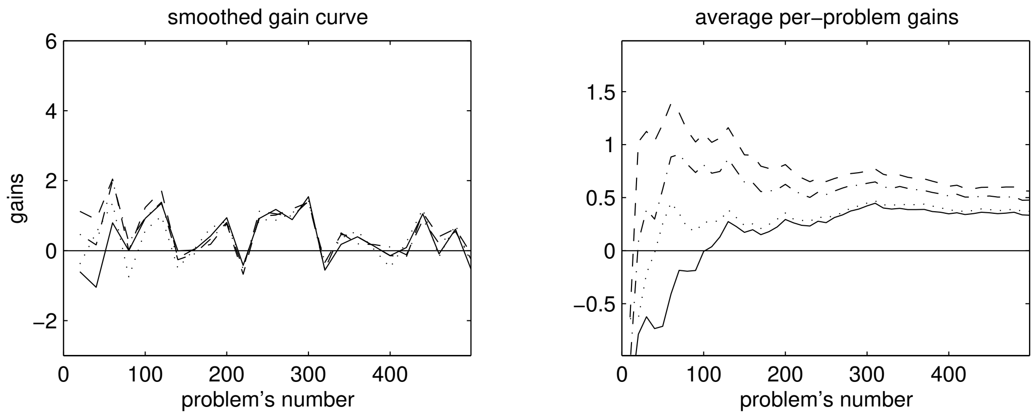
Figure 13.9: Choosing a representation for simple gain functions, illustrated in Figure 13.1. The system generates nine representations (see Figure 13.8), and gradually identifies the most effective among them. We plot $\mathcal{S}$HAPER's gains (solid lines) and compare them with the results on two smaller tasks: selection among three descriptions (dash-and-dot lines), and choice of a solver (dotted lines). In addition, the graphs show the optimal performance of LINEAR search with abstraction (dashed lines), which is the best available strategy.



(a) Gain linearly decreases with the running time (Function 13.1).

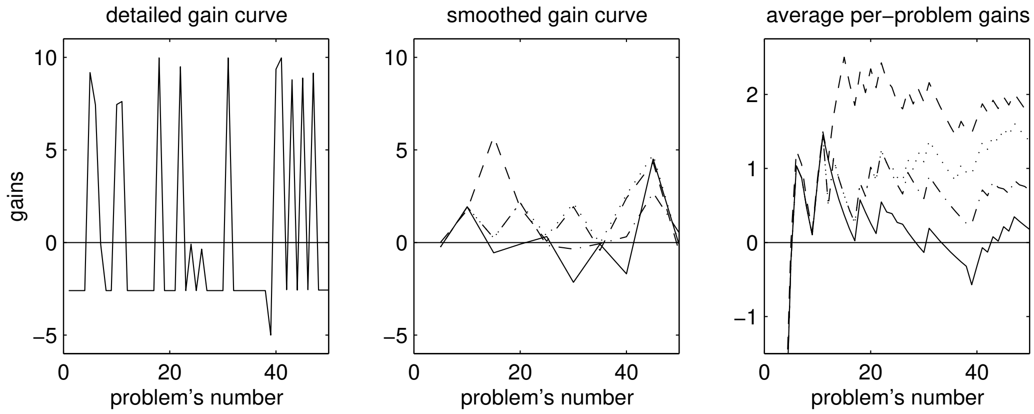


(b) Gain is a discontinuous function of time (Function 13.2).

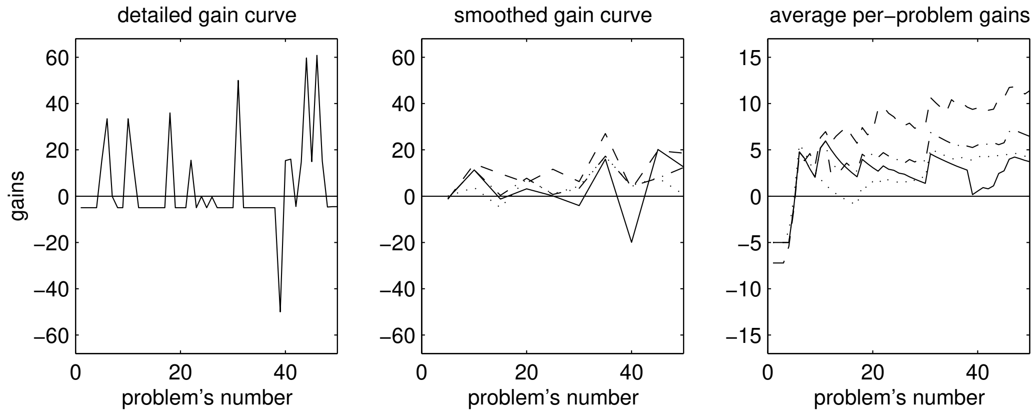


(c) Gain linearly decreases with the logarithm of time (Function 13.3).

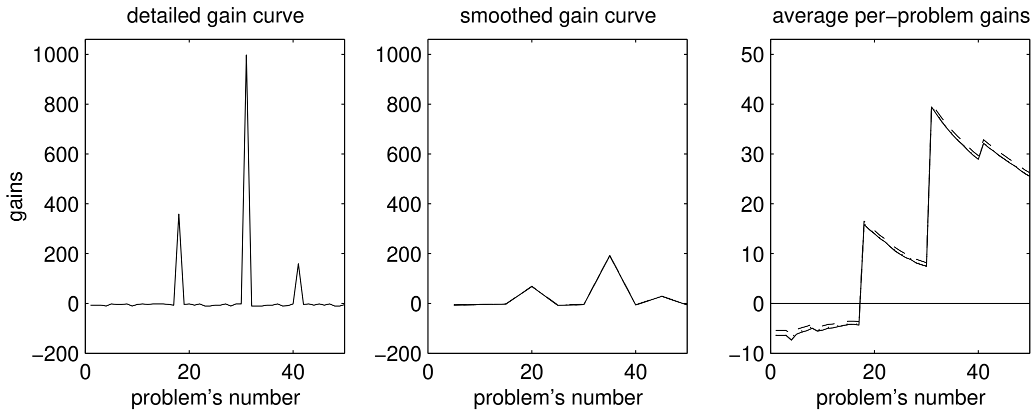
Figure 13.10: Processing sequences of 500 problems, with the space of nine alternative representations (solid lines). We compare the resulting gains with the analogous data for the smaller spaces (dotted and dash-and-dot lines), and with the behavior of the most effective representation (dashes).



(a) Gain is a linear function of problem-solving time,
with a partial reward for a failure (Function 13.4).

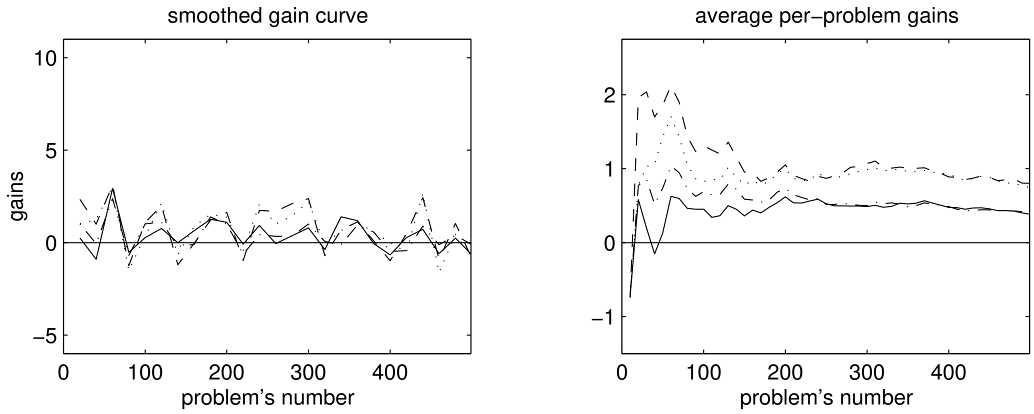


(b) Reward is proportional to the size of the Sokoban grid (Function 13.5).

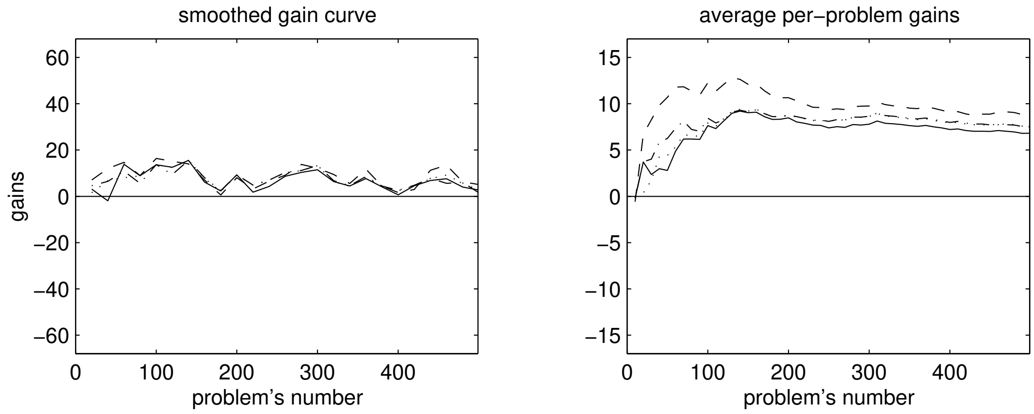


(c) Gain is a complex function of time and grid size (Function 13.6).

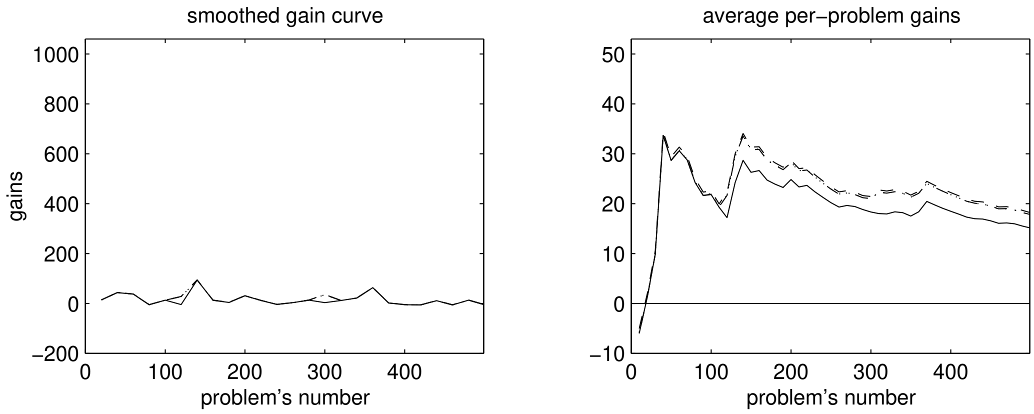
Figure 13.11: Identifying an appropriate representation for Functions 13.4–13.6. The graphs include the results of utilizing all nine representations (solid lines), as well as the performance with the two smaller spaces (dotted and dash-and-dot lines).



(a) System gets a partial reward for a failure (Function 13.4).



(b) Reward is proportional to the size of the Sokoban grid (Function 13.5).



(c) Gain is a complex function of time and grid size (Function 13.6).

Figure 13.12: Processing 500-problem sequences with Functions 13.4–13.6. We show the results for the large representation space (solid lines) and two small ones (dotted and dash-and-dot lines).

13.3 Different time bounds

The last Sokoban experiment involves several alternative values of the exploration knob, which controls the computation of time bounds (see Section 12.3). Recall that an increase of the knob value leads to a more thorough evaluation of the available search strategies, for the expense of larger time losses.

We have analyzed the system's behavior with the value 0.1 of the exploration knob (Section 13.2), and now give the results of performing the same tasks with other knob settings. The tests involve two simple dependencies of gain on the search time (Functions 13.1 and 13.3), as well as a more complex utility model (Function 13.6).

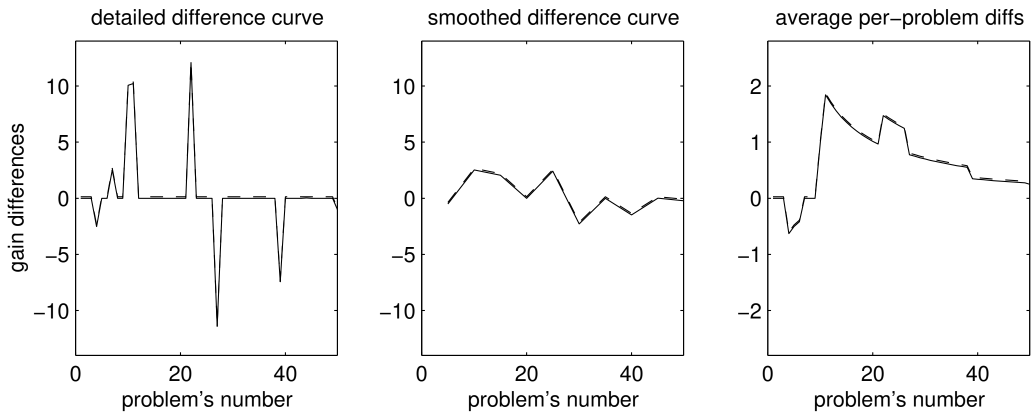
The graphs in Figures 13.13 and 13.14 show the outcome of running $\mathcal{S}$HAPER with small knob values, 0.02 and 0.05. Specifically, we plot the *differences* between the resulting gains and the default performance. In Figures 13.15 and 13.16, we give similar difference curves for the tests with large knob values, 0.2 and 0.5. The spikes of the solid curves in Figures 13.13(a), 13.14(c), 13.15(a), and 13.16(b) are due to random fluctuations in problem difficulty.

The experiments have shown that the value 0.1, which is the default setting, usually ensures a near-optimal behavior of the learning mechanism (see Table 13.4). The only exception was a fifty-problem experiment with Function 13.1, when the default yielded worse results than all other settings.

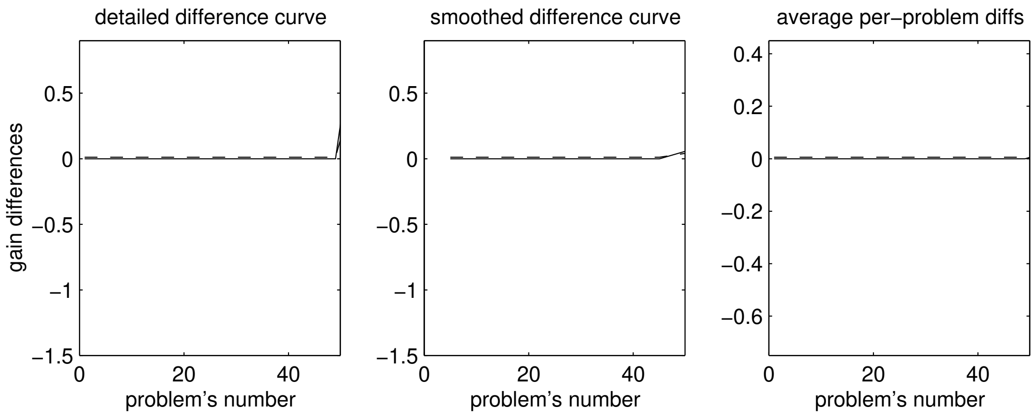
Observe that, when the system ran with the values 0.02 and 0.05, it failed to find an effective strategy for Function 13.3. The control module picked the right representation, but then chose a too small time bound (0.03 seconds), which proved much less effective than the optimal limit (7.21 seconds). This observation illustrates the "danger" of small knob values, which may hinder the exploration.

	small knob values				default	large knob values			
	0.02		0.05		0.1	0.2		0.5	
	short problem sequences								
Function 13.1	0.171	—	0.171	—	−0.077	0.171	—	0.023	—
Function 13.3	−0.756	—	−0.759	—	−0.762	-0.762	—	−0.762	—
Function 13.6	25.46	(100%)	25.46	(100%)	25.46	25.46	(100%)	25.46	(100%)
	long problem sequences								
Function 13.1	0.370	(112%)	0.323	(98%)	0.330	0.422	(128%)	0.262	(79%)
Function 13.3	0.085	(25%)	0.079	(24%)	0.334	0.334	(100%)	0.334	(100%)
Function 13.6	14.67	(97%)	15.13	(100%)	15.12	15.12	(100%)	15.12	(100%)

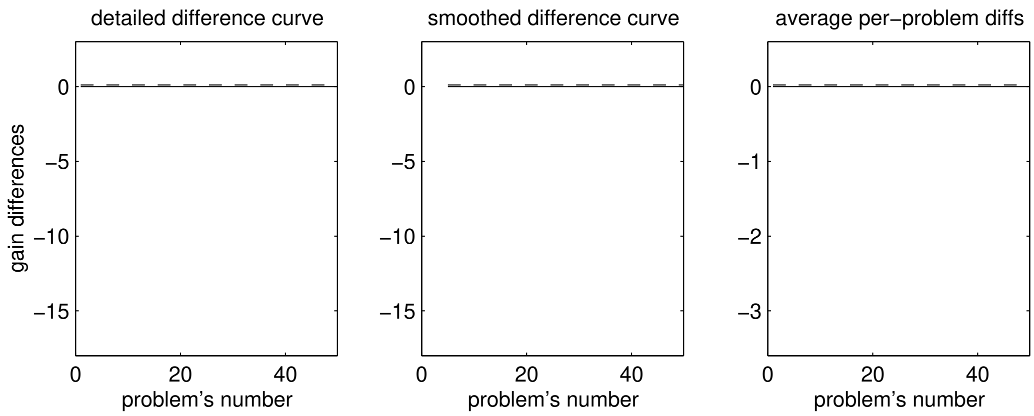
Table 13.4: Summary of the experiments with different values of the exploration knob. We give the cumulative per-problem gains and the respective percentages of the default-strategy results.



(a) Gain linearly decreases with the running time (Function 13.1).

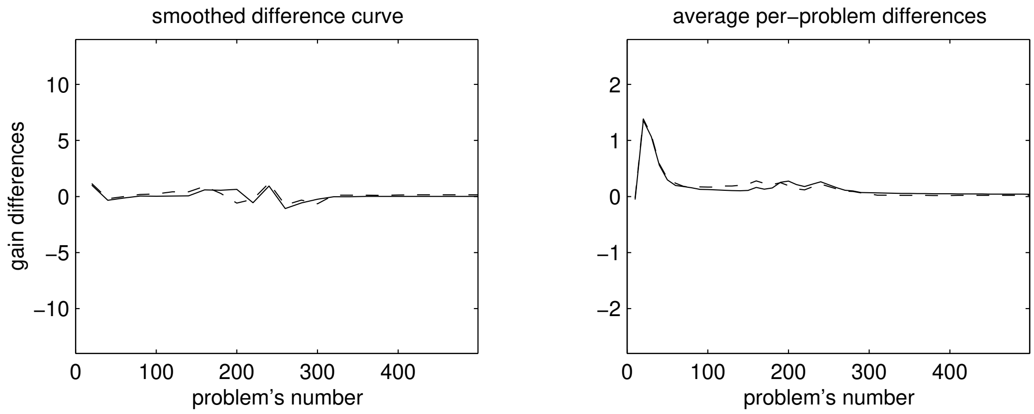


(b) Gain linearly decreases with the logarithm of time (Function 13.3).

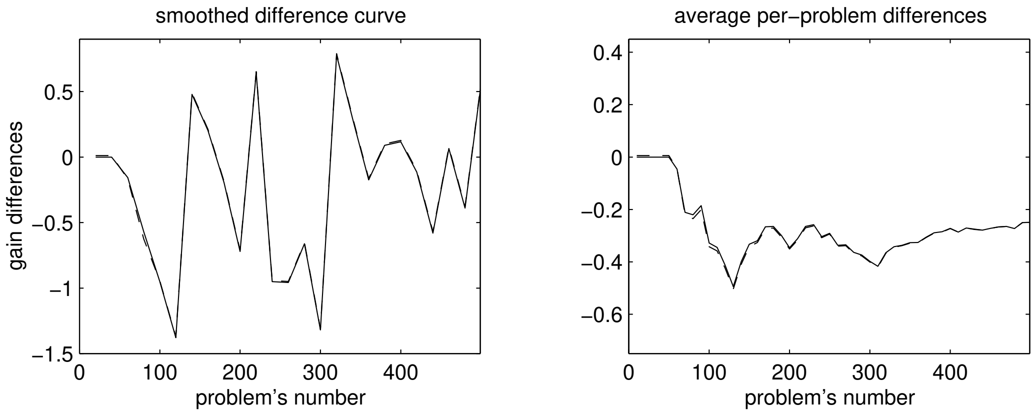


(c) Gain is a complex function of time and grid size (Function 13.6).

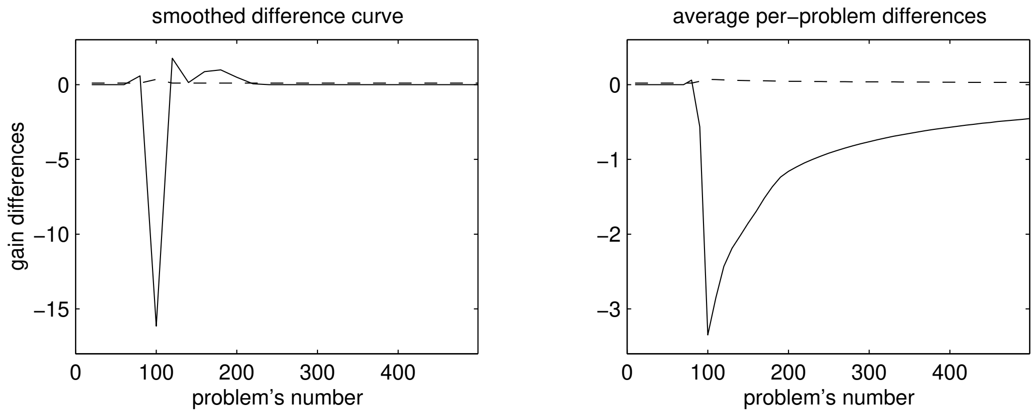
Figure 13.13: Statistical learning with small values of the exploration knob. The solid curves represent the *differences* between the results with the knob value 0.02 and that with the value 0.1. The dashed curves show the differences between the 0.05-knob gains and the 0.1-knob gains.



(a) Gain linearly decreases with the running time (Function 13.1).

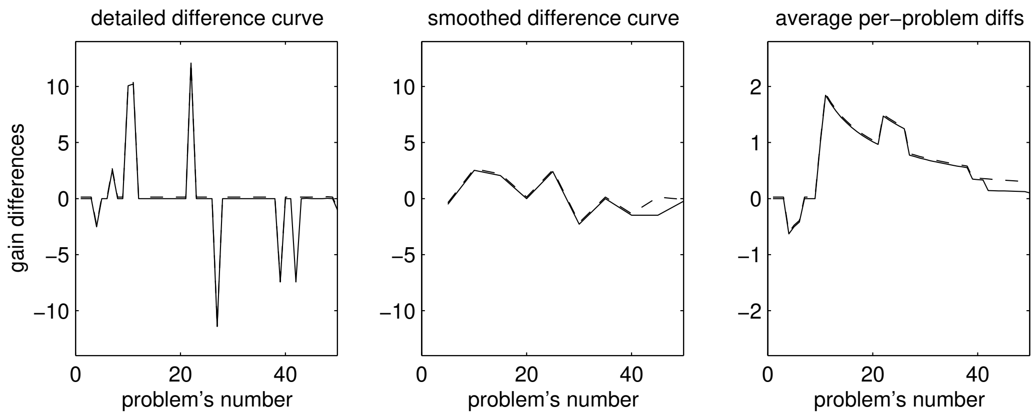


(b) Gain linearly decreases with the logarithm of time (Function 13.3).

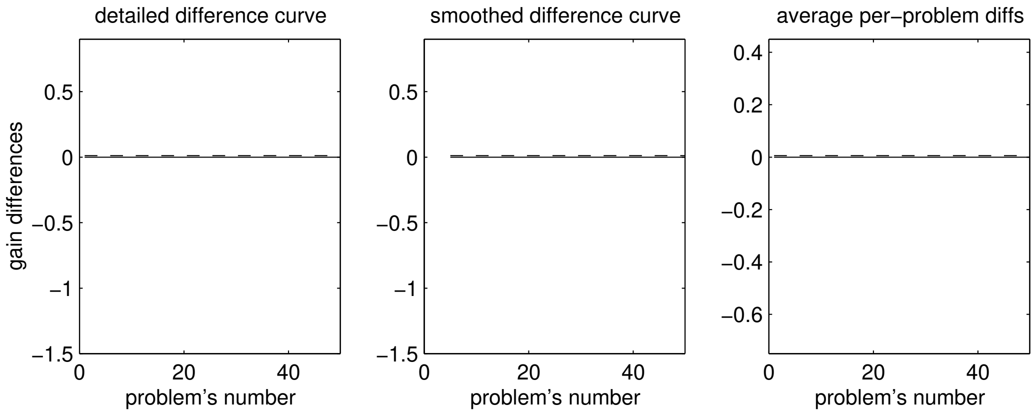


(c) Gain is a complex function of time and grid size (Function 13.6).

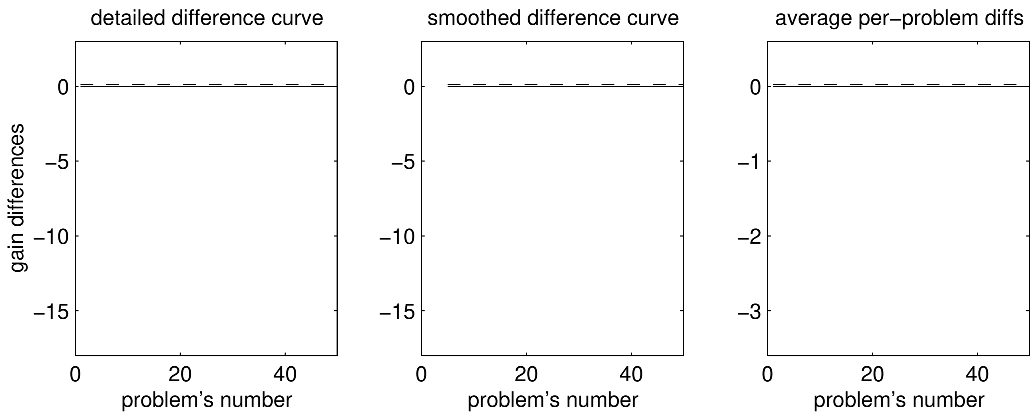
Figure 13.14: Results of processing 500 problems with small values of the exploration knob. The graphs include the smoothed gain-difference curves (left) and cumulative differences (right).



(a) Gain linearly decreases with the running time (Function 13.1).

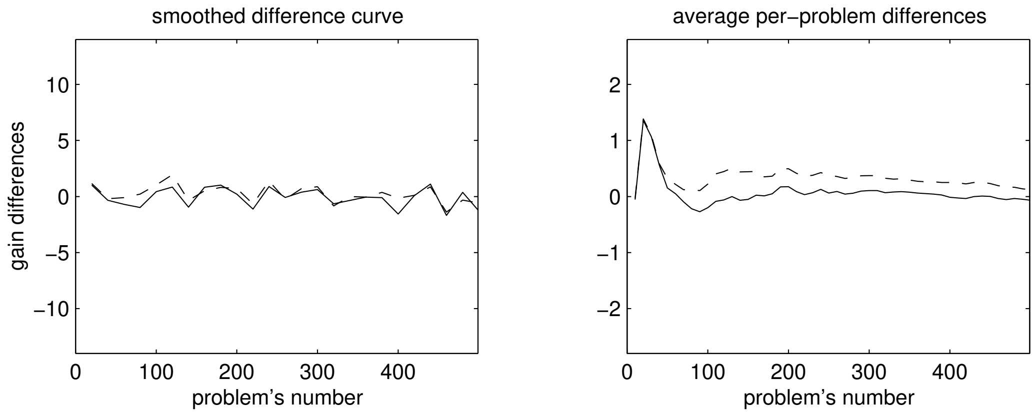


(b) Gain linearly decreases with the logarithm of time (Function 13.3).

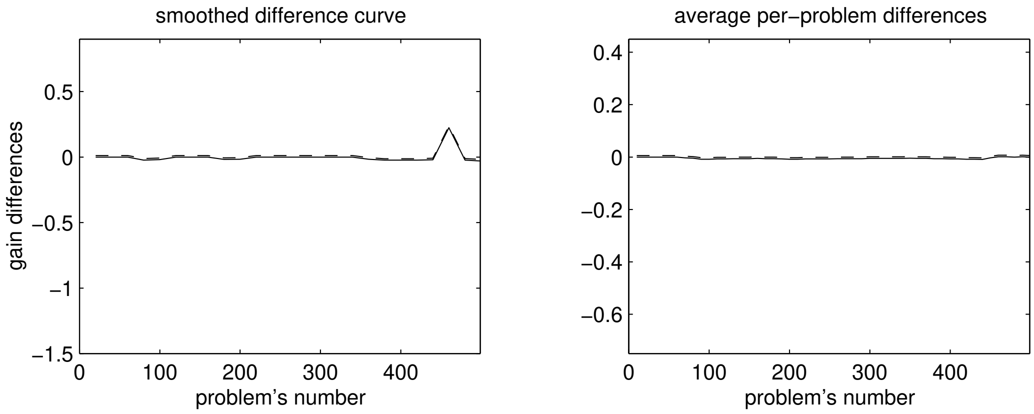


(c) Gain is a complex function of time and grid size (Function 13.6).

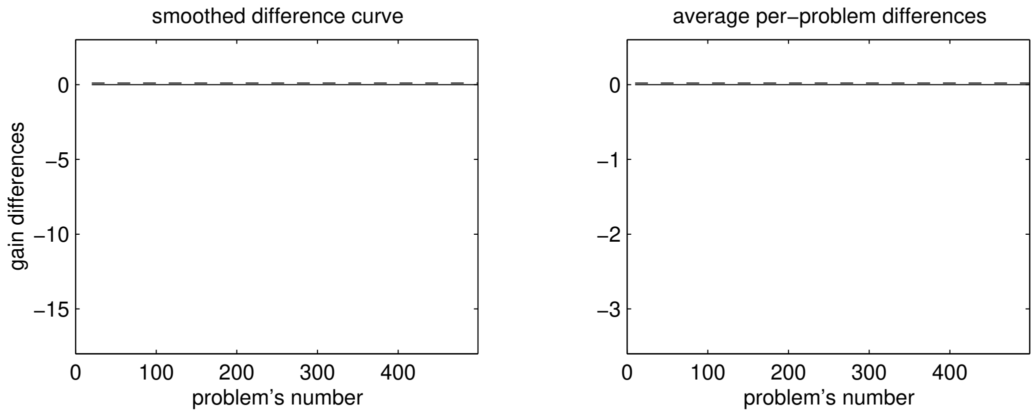
Figure 13.15: Performance with large knob values. We show the differences between the 0.5-knob gains and the default-knob gains (solid lines), as well as the analogous differences for the 0.2-knob experiments (dashed lines).



(a) Gain linearly decreases with the running time (Function 13.1).



(b) Gain linearly decreases with the logarithm of time (Function 13.2).



(c) Gain is a complex function of time and grid size (Function 13.6).

Figure 13.16: Processing long problem sequences, with large values of the exploration knob. Specifically, we give the difference curves for the knob values 0.5 (solid lines) and 0.2 (dashed curves).

Chapter 14

Extended Strips Domain

The STRIPS world is larger than the other test domains: it includes ten object types and twenty-three operators (see Figures 3.39–3.42 on pages 141–3.42). Its problem library contains a variety of search tasks, which range from very simple to unsolvable.

We tested the $\mathcal{S}$HAPER system with two linear utility functions (see Figure 14.1a,b), and with a nonlinear dependency of gain on the running time and solution quality (Figure 14.1c). The changer library was the same as in the other domains: it included algorithms for choosing primary effects and abstraction, illustrated in Figure 14.2(b). The solver library comprised the LINEAR, SAVTA, and SABA algorithms, combined with alternative cost-bound heuristics and problem-specific description changers.

14.1 Small-scale selection tasks

We consider limited libraries of solver and changer operators, which give rise to small representation spaces. The system has to choose among domain descriptions (Figure 14.2), search algorithms (Figure 14.3b), and cost-bound heuristics (Figure 14.3c).

Four descriptions

The first solver library comprises the LINEAR algorithm and its synergy with problem-specific changers (see Figure 14.2a). The system employs these two solvers, along with the standard library of changers (Figure 14.2b), and produces four representations (Figure 14.2c). The *Chooser* and *Completer* algorithms select the primary effects shown in Figures 3.41 and 3.42 (pages 143 and 144), and *Margie* generates the description illustrated Figure 5.8 and 5.9 (pages 199 and 200).

The results of processing fifty-problem sequences are summarized in Figure 14.4, and the results for longer sequences are in Figure 14.5. The graphs include the learning curves (solid lines), as well as the behavior of three fixed strategies: search with primary effects (dots), abstraction problem solving (dashes), and goal-specific abstraction (dash-and-dot lines). Note that we have tested the fixed strategies with the *optimal* time bounds.

The search without primary effects yields *negative* gains, for all three utility functions, and we do not show its behavior in the graphs. Thus, improvements to the initial domain

(a) Gain function is a linear dependency on problem-solving time:

$$gain = \begin{cases} 1 - time, & \text{if success} \\ -time, & \text{if failure or interrupt} \end{cases} \tag{14.1}$$

(b) Gain linearly decreases with running time and solution cost:

$$gain = \begin{cases} 100 - cost - 50 \cdot time, & \text{if success and } cost < 100 \\ -50 \cdot time, & \text{otherwise} \end{cases} \tag{14.2}$$

(c) Gain is a nonlinear function of time and solution cost, with a partial reward for a failure termination:

$$gain = \begin{cases} 100 - cost \cdot time, & \text{if success and } cost < 50 \\ 100 - 50 \cdot time, & \text{if success and } cost \geq 50 \\ 50 - 50 \cdot time, & \text{if failure} \\ -50 \cdot time, & \text{if interrupt} \end{cases} \tag{14.3}$$

Figure 14.1: Utility computation in the Extended STRIPS domain. We first tested the system with linear gain functions (a,b), and then with a continuous nonlinear function (c).

description are essential for obtaining positive results.

The problem-independent abstraction proved more effective than the other descriptions, for all three gain functions. When the system ran with the linear functions, it converged to using the abstraction with appropriate time bounds. On the other hand, when we tested SHAPER with Function 14.3, it chose the second best description, which caused a suboptimal performance.

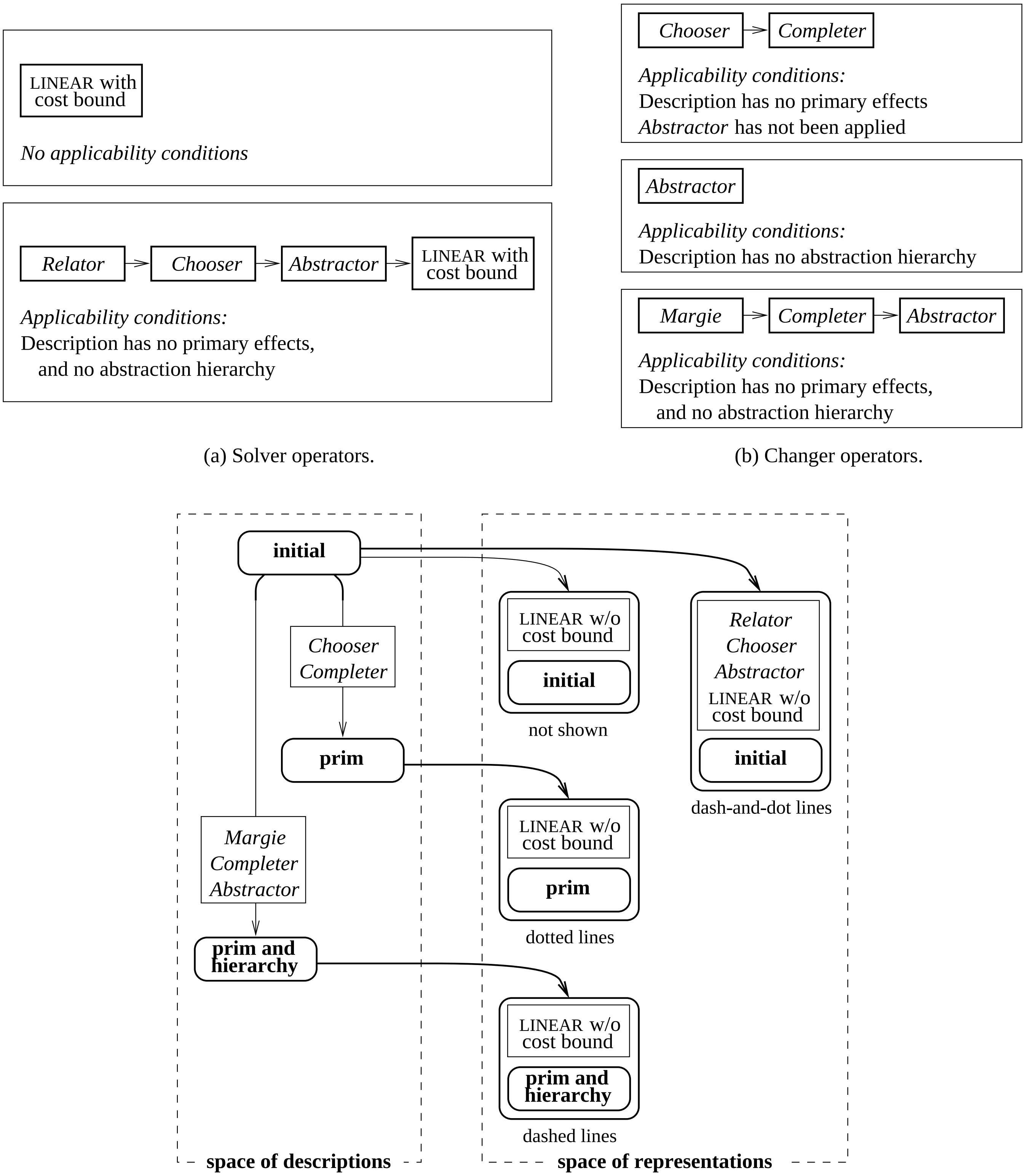
Three search engines

The second solver library consists of the LINEAR, SAVTA, and SABA algorithms, without cost bounds. The system inputs a description with primary effects and abstraction, and pairs it with the solvers, thus expanding a space of three representations (see Figure 14.3a).

The resulting gains are summarized in Figures 14.12 and 14.13: we show the learning curves (solid lines), as well as the performance of each solver with the optimal time bound (broken lines). The outcome is similar to the previous experiment: SHAPER finds the right strategy for Functions 14.1 and 14.2, but chooses the second best solver for Function 14.3.

Three cost bounds

Next, we experiment with three versions of the LINEAR search engine. The first version runs without cost bounds; the second one utilizes *loose bounds,* which are twice larger than the optimal-cost estimates; and the third employs *tight bounds,* which closely approximate the optimal costs. The control module uses these solvers with the input description, which includes primary effects and abstraction (see Figure 14.3b).

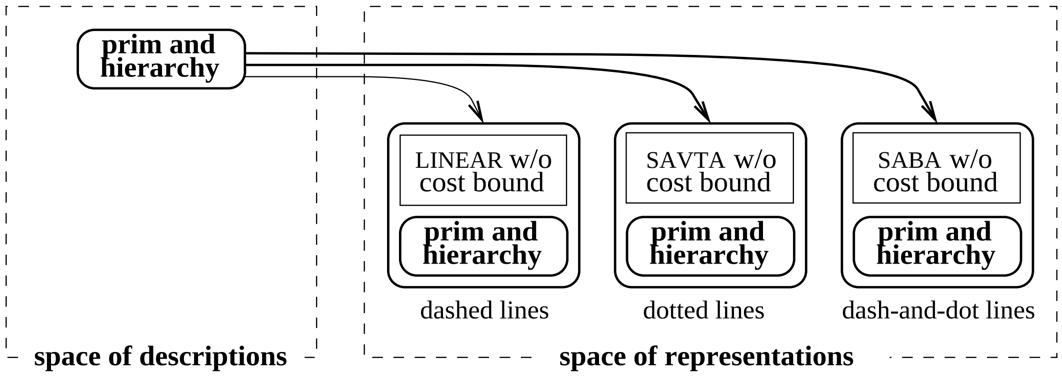


(a) Solver operators.

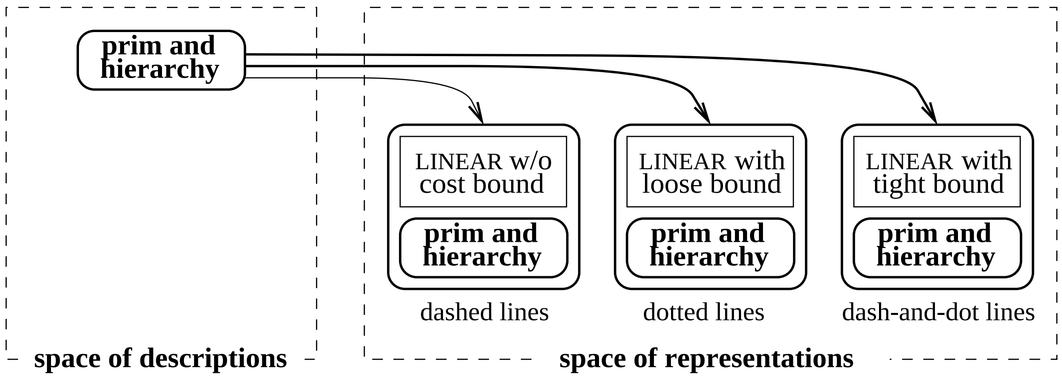
(b) Changer operators.

(c) Expanded spaces.

Figure 14.2: Description space in the STRIPS Domain and the result of pairing the descriptions with the LINEAR search engine. The system employs two solver operators (a), along with the standard changer library (b), and generates four representations (c). Note that the goal-independent version of *Abstractor* fails to generate a multi-level hierarchy. The subscriptions in the representation space specify the corresponding curves in Figures 14.4 and 14.5.



(a) Alternative search engines, without cost bounds (subscriptions refer to Figures 14.6 and 14.7).



(b) LINEAR engine, with alternative cost bounds (subscriptions refer to Figures 14.8 and 14.9).

Figure 14.3: Experiments without description changers. The first task is to identify the most effective search engine (a), and the other is to choose a technique for limiting the search depth (b). The system pairs the available solver operators with a given domain description, which includes primary effects and abstraction, and evaluates the resulting representations.

	choice among						optimal
	descriptions		solvers		cost bounds		gain
	short problem sequences						
Function 14.1	0.048	(17%)	0.144	(50%)	0.162	(56%)	0.288
Function 14.2	11.1	(50%)	18.7	(84%)	8.8	(39%)	22.3
Function 14.3	34.5	(55%)	52.1	(83%)	44.5	(71%)	62.6
	long problem sequences						
Function 14.1	0.164	(68%)	0.182	(76%)	0.214	(89%)	0.240
Function 14.2	14.4	(71%)	18.4	(91%)	18.3	(91%)	20.2
Function 14.3	40.6	(69%)	50.8	(87%)	54.5	(93%)	58.6

Table 14.1: Summary of experiments with small spaces, which involve choices among four domain descriptions (Figures 14.4 and 14.5), three solver algorithms (Figures 14.6 and 14.7), and three heuristics for setting cost bounds (Figures 14.8 and 14.9). We list the average per-problem gains and the respective percentages of the optimal-strategy gains.

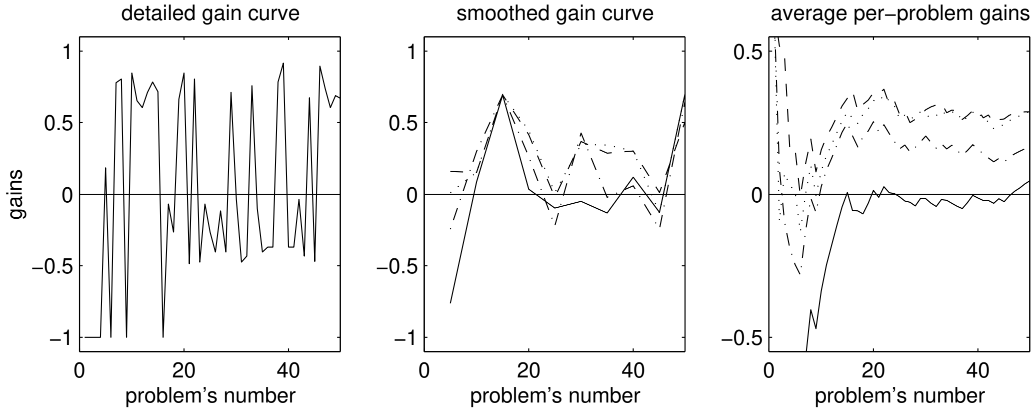
We present the learning results in Figures 14.8 and 14.9 (solid lines), and compare them with the optimal performance of each algorithm (broken lines). The system converges to the right choice of a solver and time bound in all three cases.

Summary

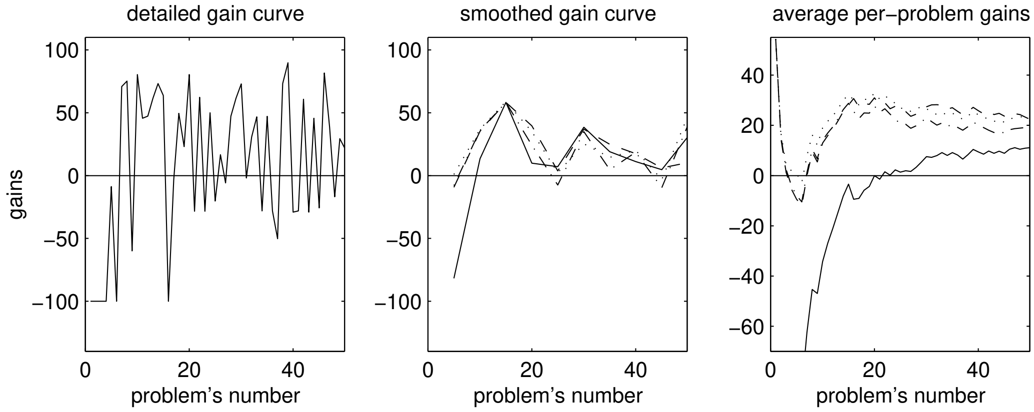
The control module found appropriate strategies for the linear gain functions; however, it was confused by the more complex function, and made a suboptimal choice in two out of three cases. This outcome was surprising, since the system successfully dealt with nonlinear utility models in other domains. We have not studied functions that confuse the statistical mechanism, and plan to analyze their properties as a part of the future work.

The learning behavior was similar to the results in the Machining and Sokoban domain: SHAPER incurred major losses in the very beginning of the learning process, and then performed a more thorough evaluation of near-optimal strategies, which caused little deviation from ideal performance. The system converged to its final choice of a strategy after solving 50 to 200 problems.

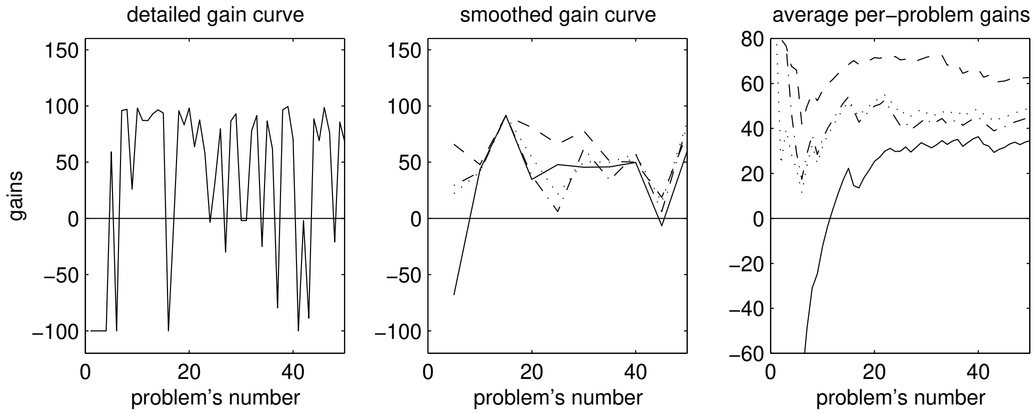
In Table 14.1, we summarize the cumulative gains and indicate the respective percentages of the the optimal-strategy gains. The percentage values for short problem sequences range from 17% to 84%, and the long-sequence values are between 68% and 93%, which is similar to the Sokoban results.



(a) Gain linearly decreases with the running time (Function 14.1).



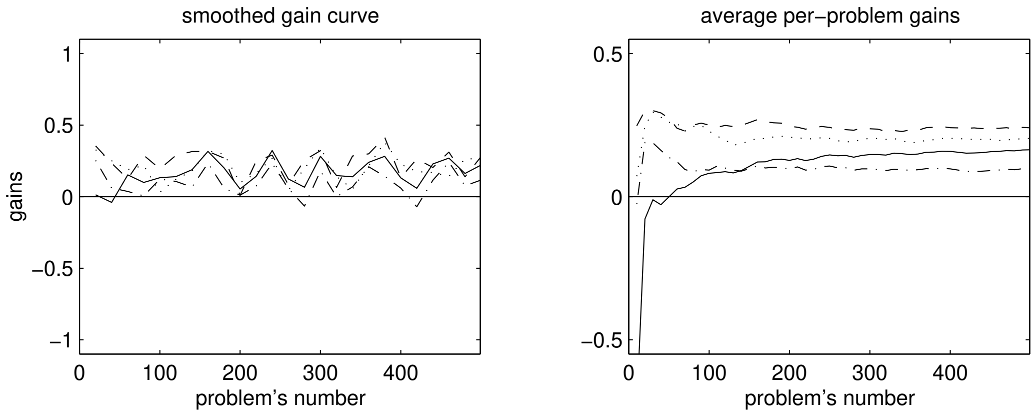
(b) Gain decreases with the time and solution cost (Function 14.2).



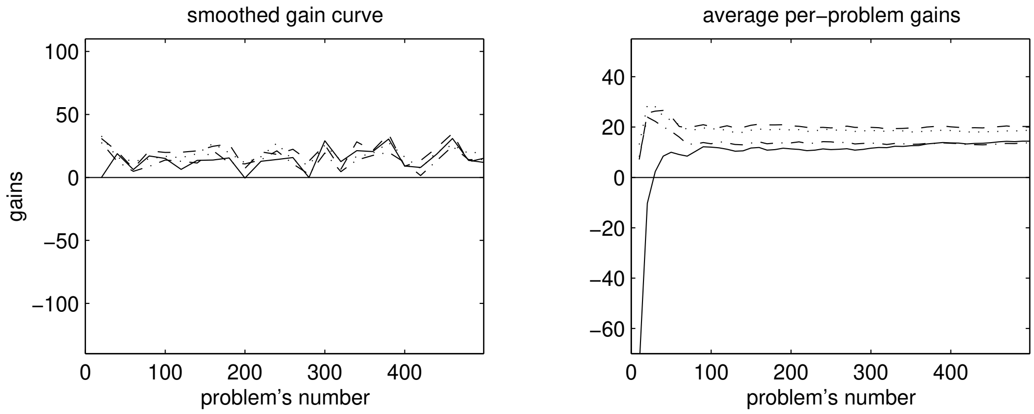
(c) Gain is a nonlinear function of time and cost (Function 14.3).

Figure 14.4: Incremental selection of an effective domain description, for the LINEAR algorithm without a cost bound. The graphs comprise the raw learning curves for fifty-problem sequences (left), as well as the smoothed curves (middle) and cumulative gains (right).

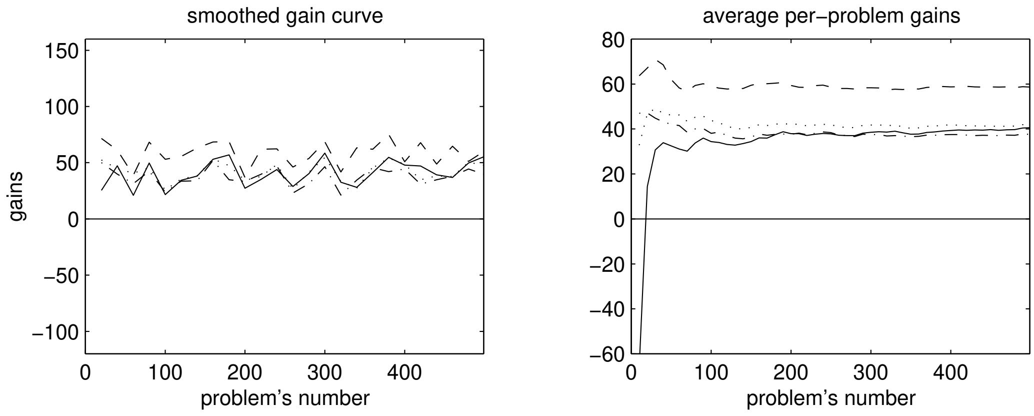
In addition, they include the performance of three fixed strategies with the optimal time limits: search with primary effects (dotted lines), abstraction (dashed lines), and goal-specific descriptions (dash-and-dot lines). We do *not* show the results of search with the initial domain description, which would lead to negative "gains" regardless of the time bound.



(a) Gain linearly decreases with the running time (Function 14.1).

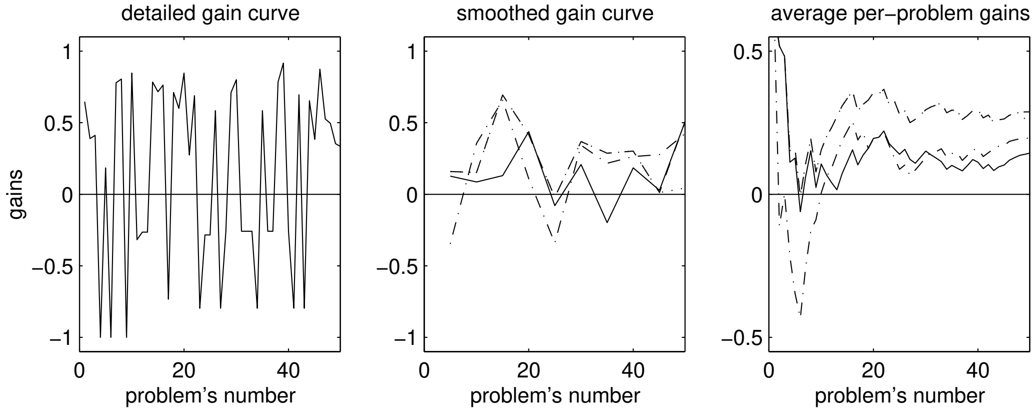


(b) Gain decreases with the time and solution cost (Function 14.2).

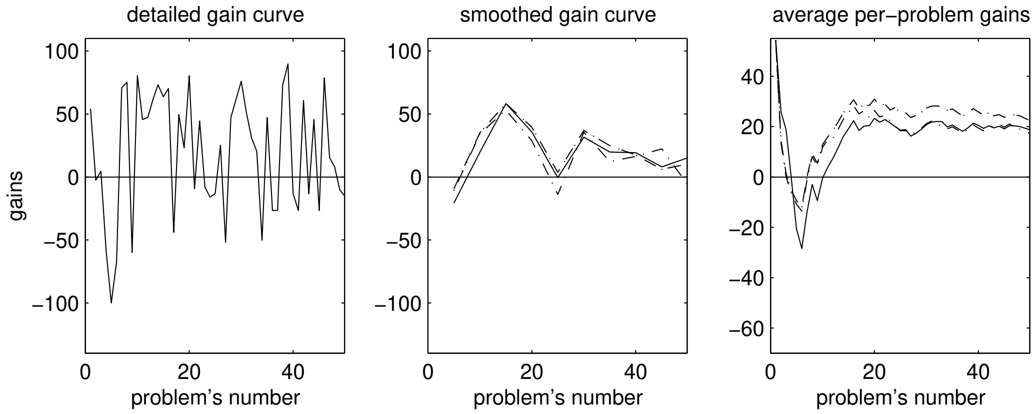


(c) Gain is a nonlinear function of time and cost (Function 14.3).

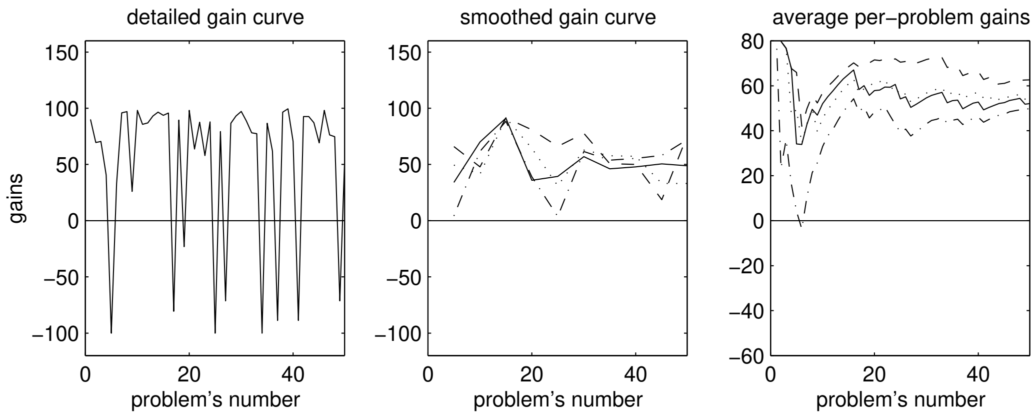
Figure 14.5: Behavior on long problem sequences, with four alternative descriptions. We plot the smoothed curves (left) and cumulative per-problem gains (right), using the legend of Figure 14.4.



(a) Gain linearly decreases with the running time (Function 14.1).

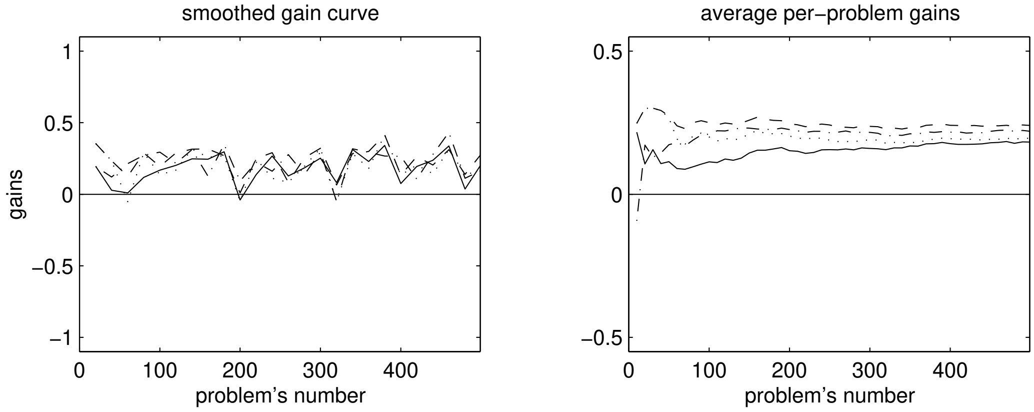


(b) Gain decreases with the time and solution cost (Function 14.2).

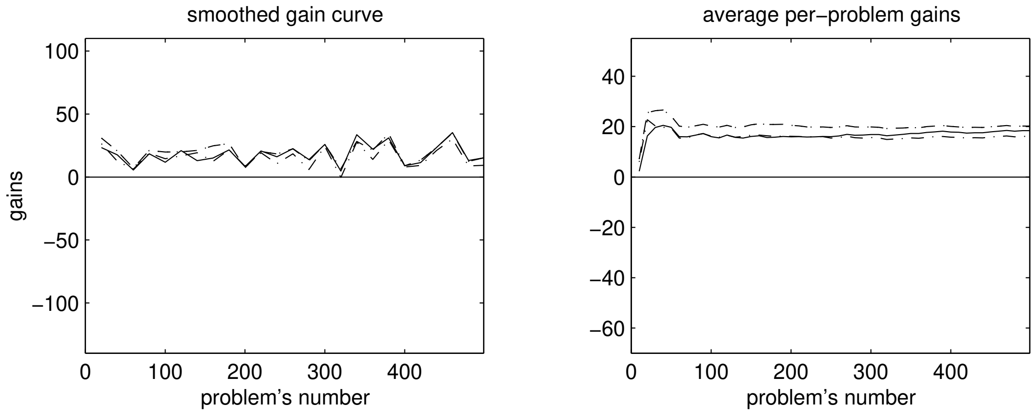


(c) Gain is a nonlinear function of time and cost (Function 14.3).

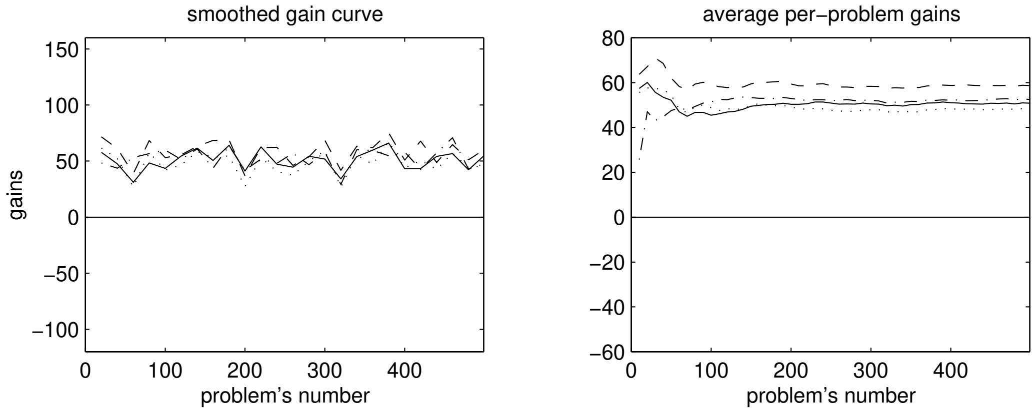
Figure 14.6: Choosing among the three search engines, without cost bounds. We compare the learning results (solid lines) with the optimal performance of LINEAR (dashes), SAVTA (dots), and SABA (dash-and-dot lines).



(a) Gain linearly decreases with the running time (Function 14.1).

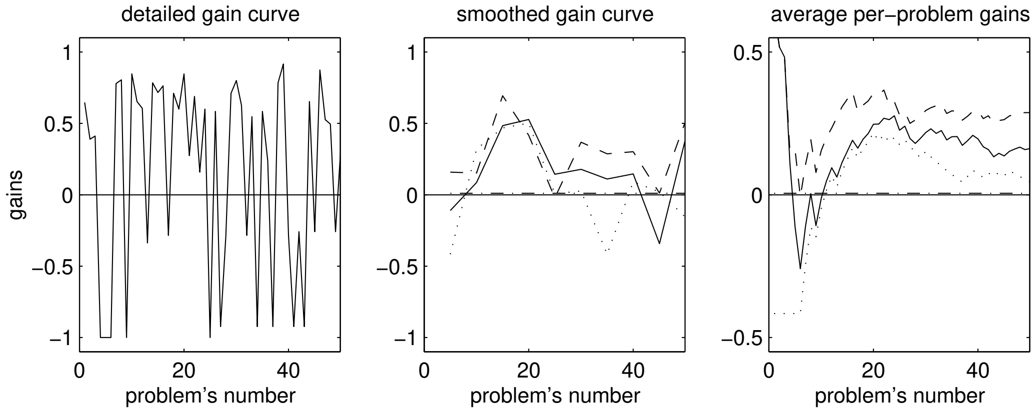


(b) Gain decreases with the time and solution cost (Function 14.2).

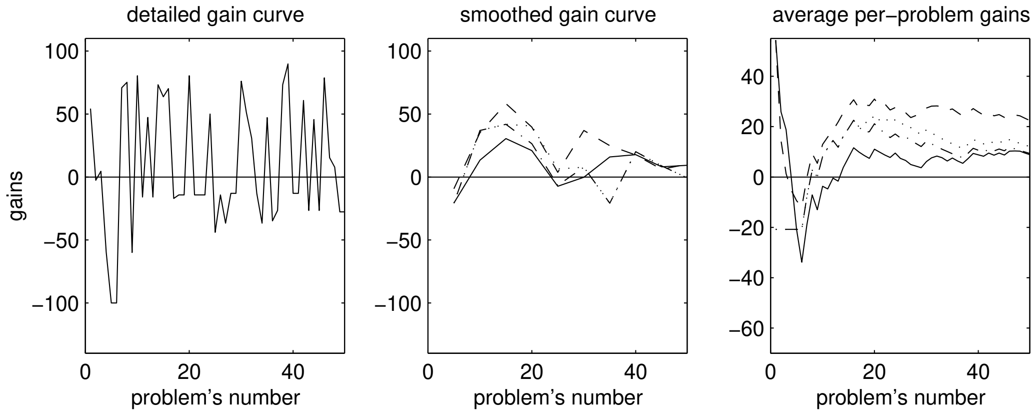


(c) Gain is a nonlinear function of time and cost (Function 14.3).

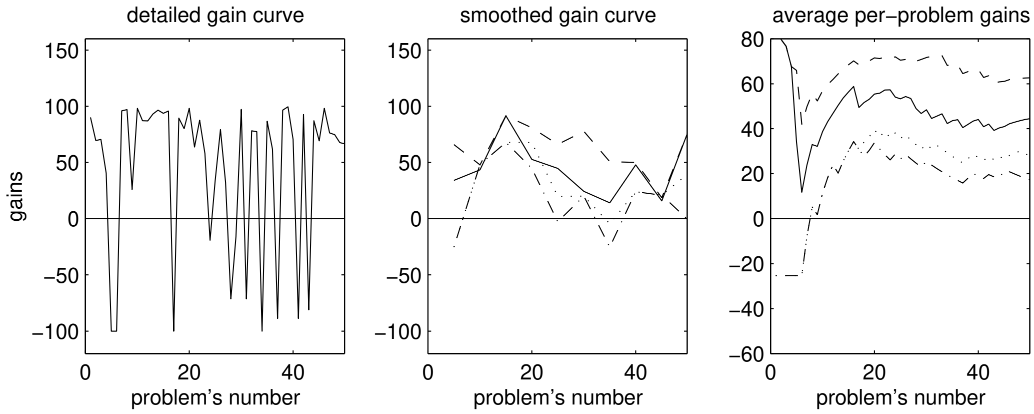
Figure 14.7: Results of applying $\mathcal{S}$HAPER to long problem sequences, with the library of three search engines. We show the system's gains (solid lines) and the optimal performance of each available engine (broken lines), using the legend of Figure 14.6.



(a) Gain linearly decreases with the running time (Function 14.1).

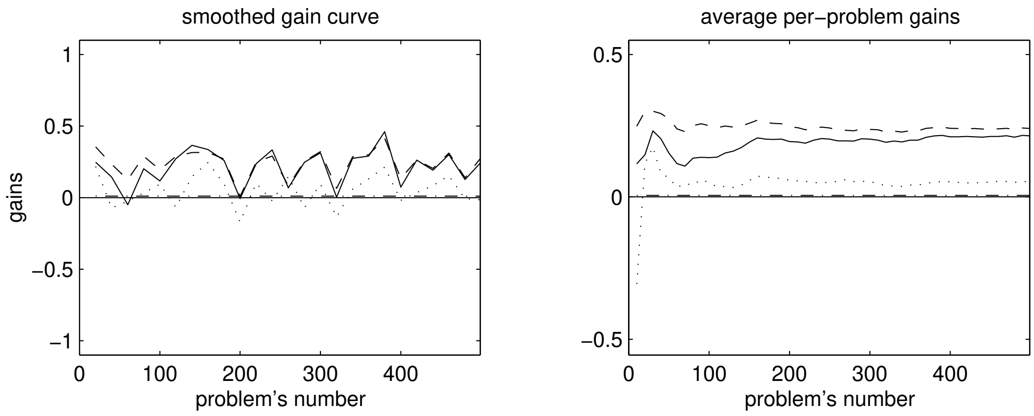


(b) Gain decreases with the time and solution cost (Function 14.2).

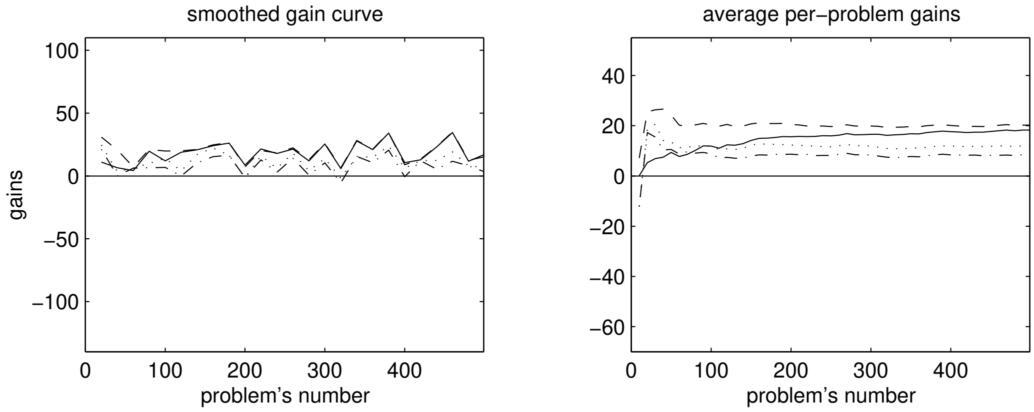


(c) Gain is a nonlinear function of time and cost (Function 14.3).

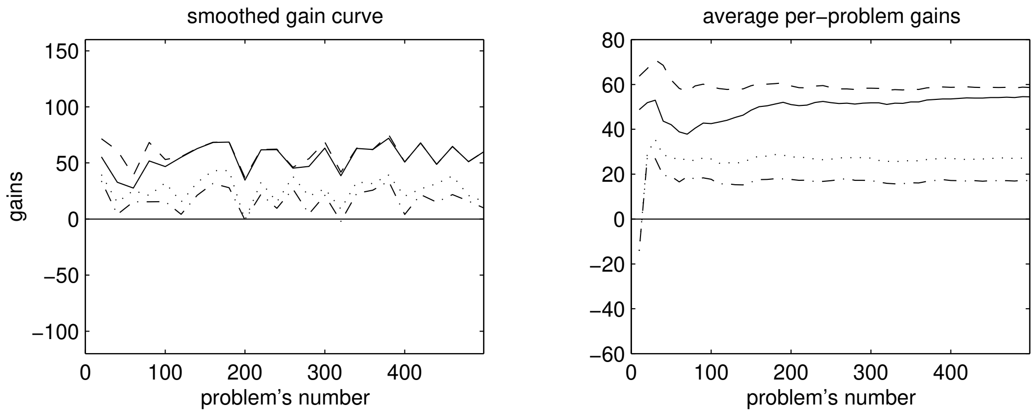
Figure 14.8: Selection among three heuristics for controlling the search depth of the LINEAR algorithm. These heuristics include unbounded search (dashed lines), loose cost bounds (dotted lines), and tight bounds (dash-and-dot lines).



(a) Gain linearly decreases with the running time (Function 14.1).



(b) Gain decreases with the time and solution cost (Function 14.2).



(c) Gain is a nonlinear function of time and cost (Function 14.3).

Figure 14.9: Incremental selection of a cost-bound heuristic for the LINEAR search engine, on long problem sequences; the legend corresponds to that in Figure 14.8.

14.2 Large-scale tasks

We now describe two experiments with larger spaces, which demonstrate scalability of the selection technique. First, the system runs with six solver operators and the standard changers. Then, it employs a still bigger library, which requires choosing among thirty-six alternatives.

Twelve representations

If SHAPER runs with the solver operators in Figure 14.10 and the changer library in Figure 14.2(b), then it expands a space of twelve representations (Figure 14.11). The control module identifies the right representation and time bound for each gain function, after processing 150 to 300 problems (see the dotted lines in Figures 14.12 and 14.13).

We compare the learning curves with a smaller-scale experiment (dash-and-dot lines), which involves the four representations in Figure 14.2, and with the best available strategy (dashed lines), which is based on the LINEAR algorithm with goal-independent abstraction.

Thirty-six representations

The larger library comprised eighteen solver operators: it included not only the operators in Figure 14.10, but also their combinations with the heuristic for setting loose cost bounds, and with the tight-bound heuristic. The control module paired the initial description with all eighteen operators, and the other two descriptions with nine solvers, thus obtaining thirty-six representations.

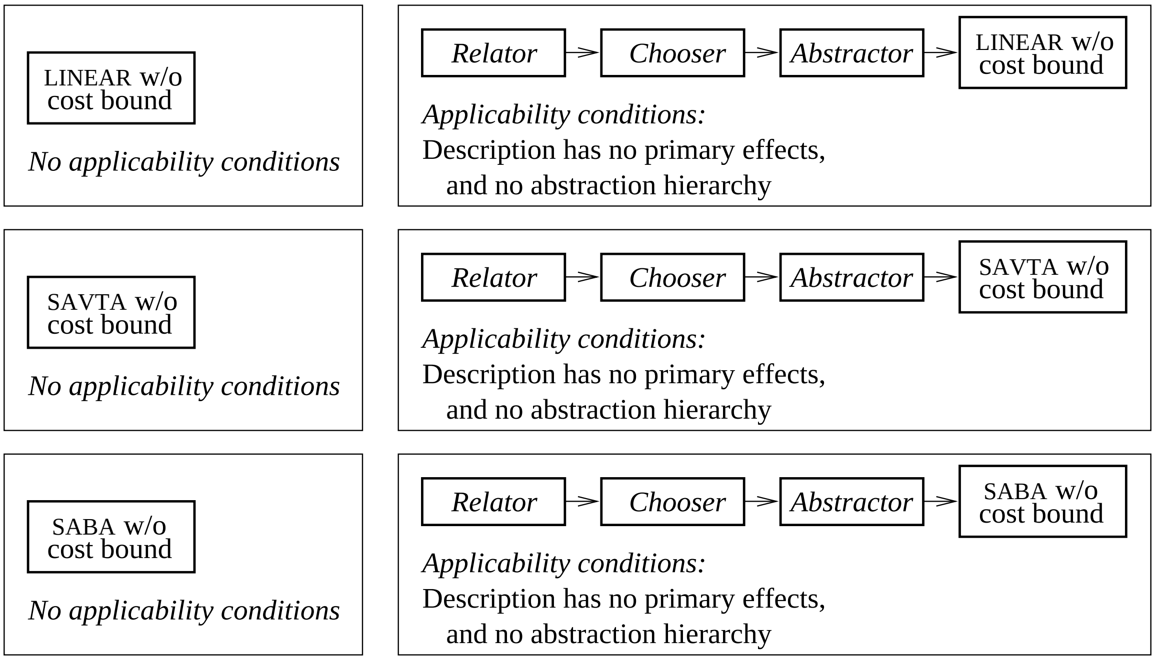


Figure 14.10: Library of six solver operators for the STRIPS domain, which includes the three search engines without cost bounds, as well as their synergy with goal-specific description changers.

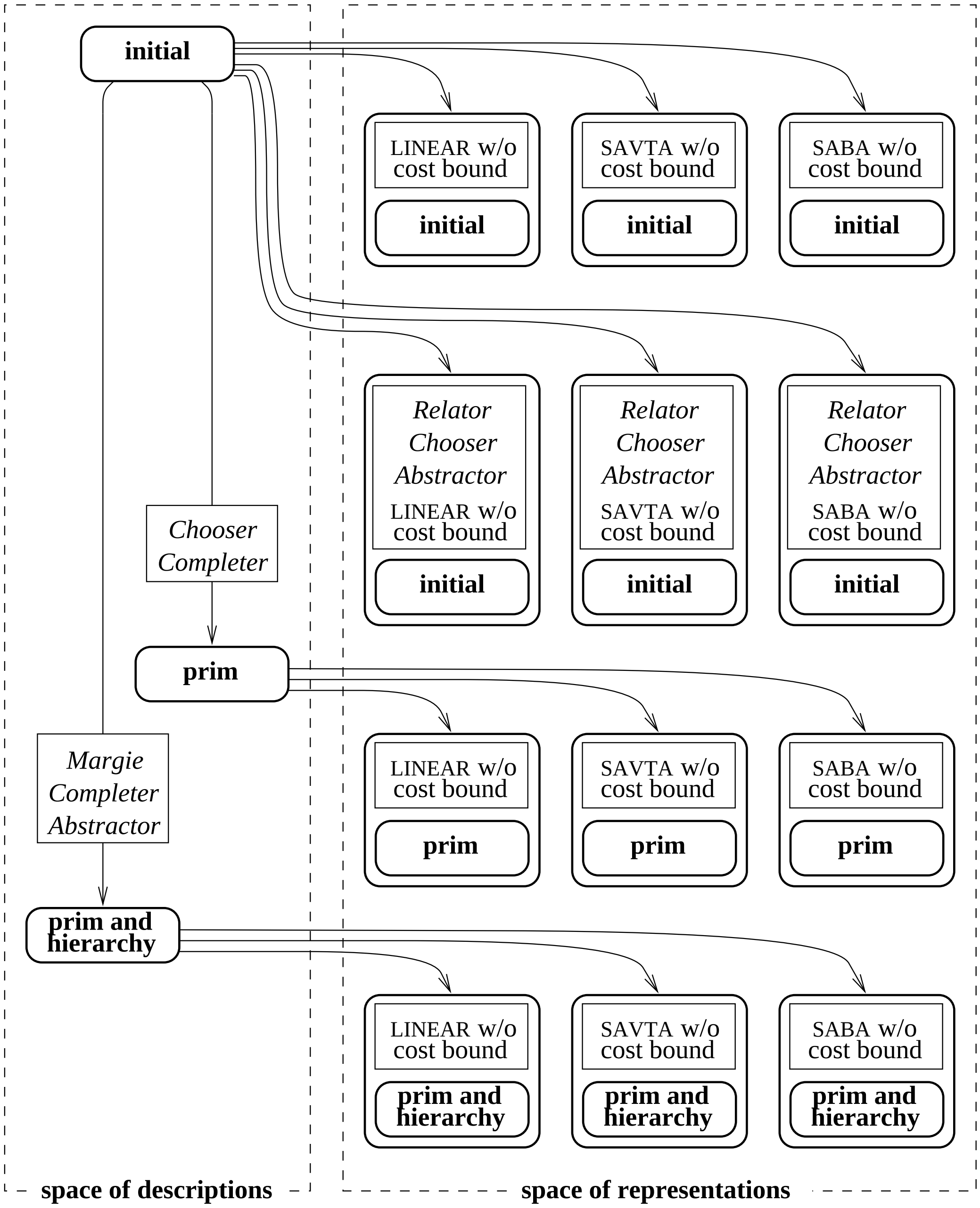


Figure 14.11: Space of twelve representations in the STRIPS domain. The system utilizes the three solver operators in Figure 14.10, along with the three changer operators in Figure 14.2(b).

	selection among thirty-six representations (solid lines)		twelve reps w/o cost bounds (dotted lines)		four descriptions (dash-and-dot lines)		optimal gain (dashed lines)
	short problem sequences						
Function 14.1	−0.128	—	−0.104	—	0.048	(17%)	0.288
Function 14.2	−29.5	—	−17.4	—	11.1	(50%)	22.3
Function 14.3	−6.3	—	10.6	(17%)	34.5	(55%)	62.6
	long problem sequences						
Function 14.1	0.108	(45%)	0.145	(60%)	0.164	(68%)	0.240
Function 14.2	11.7	(58%)	14.8	(73%)	14.4	(71%)	20.2
Function 14.3	38.5	(66%)	43.9	(75%)	40.6	(69%)	58.6

Table 14.2: Cumulative per-problem gains in the experiments with the large representation space, and the analogous results for two smaller-scale tasks. The parenthesized notes in the column headings refer to the curves in Figures 14.12 and 14.13.

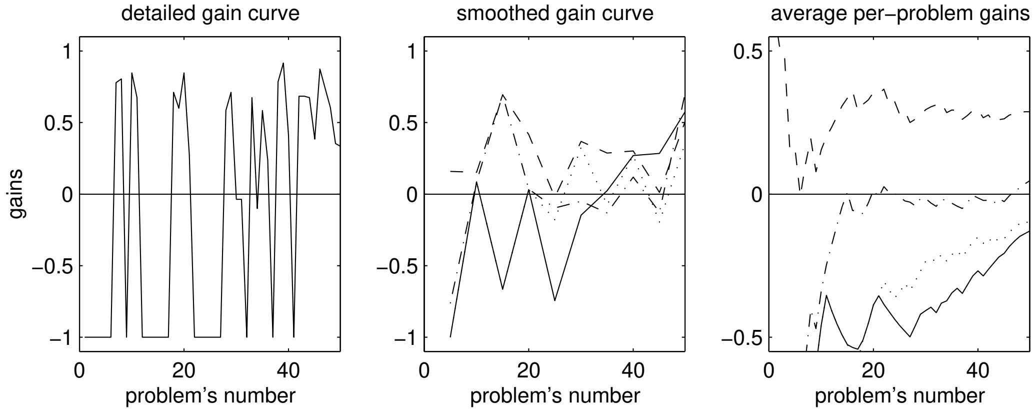
The system correctly determined that the LINEAR solver with abstraction and no cost bounds was the best choice for all three gain functions, and found the right time limits for the linear functions. When $\mathcal{S}$HAPER ran with the nonlinear utility model, it chose a wrong time limit (0.92 seconds), which was smaller than optimal (1.55 seconds).

In Figures 14.12 and 14.13, we give the learning curves (solid), and compare them with the optimal strategy (dashes) and with two smaller selection tasks (dotted and dash-and-dot lines). In Figures 14.14 and 14.15, we compare the same curves with the results of Section 14.1. Observe that the large representation space causes greater initial losses and slower convergence: the system processes 300 to 500 problems before choosing its final strategy.

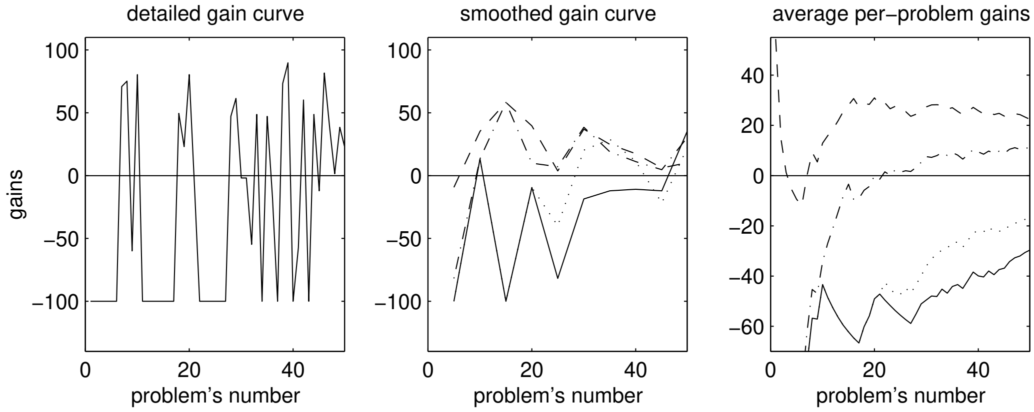
Summary

The tests confirmed $\mathcal{S}$HAPER's ability to choose from a sizable suite of candidate strategies. The learning behavior was similar to large-scale experiments in other domains: the control module discarded most representations in the beginning of a learning sequence, and then gradually selected among near-optimal strategies, which did not cause significant losses.

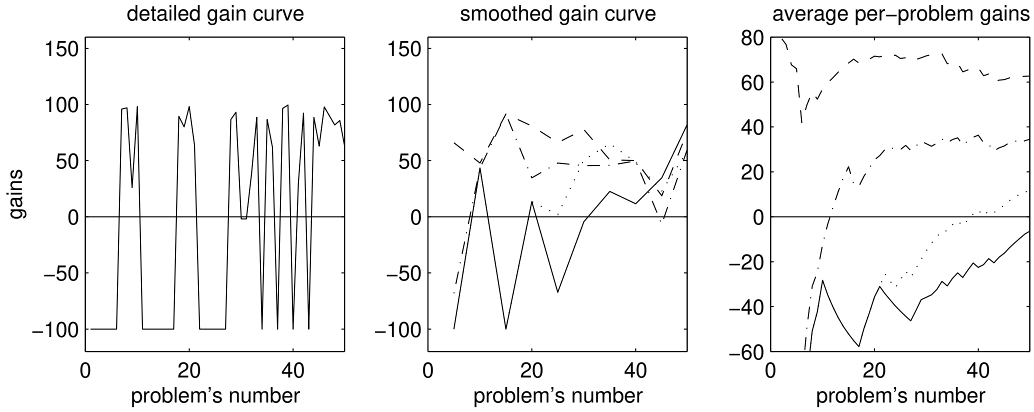
In Table 14.2, we list the cumulative gains obtained with the space of thirty-six representations, and compare them with the results for smaller spaces. Observe that the large space caused significant losses on short sequences of problems; however, when $\mathcal{S}$HAPER processed longer sequences, it amortized the initial losses and obtained good overall results.



(a) Gain linearly decreases with the running time (Function 14.1).

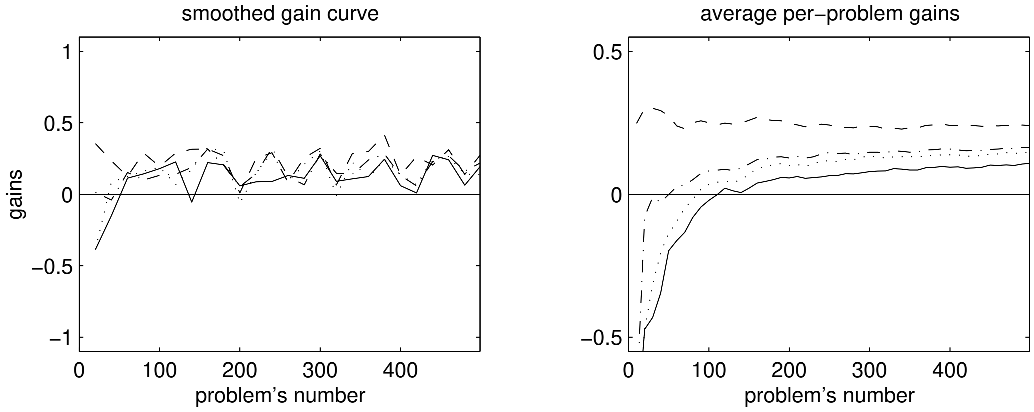


(b) Gain decreases with the time and solution cost (Function 14.2).

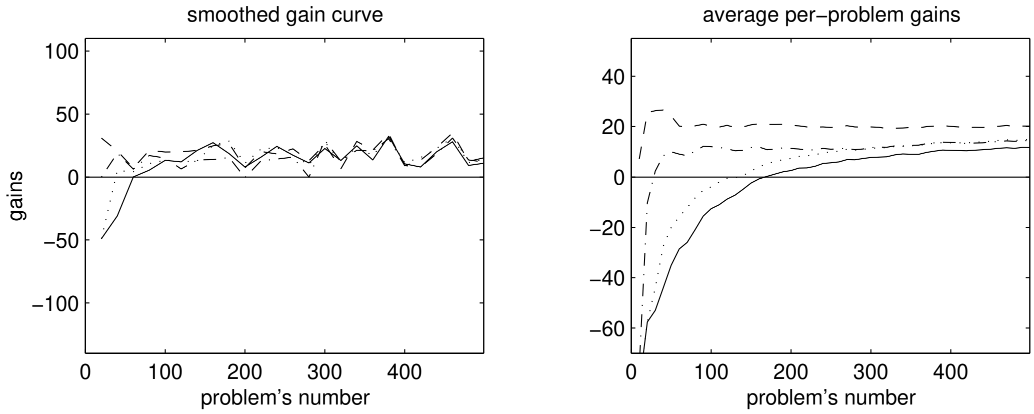


(c) Gain is a nonlinear function of time and cost (Function 14.3).

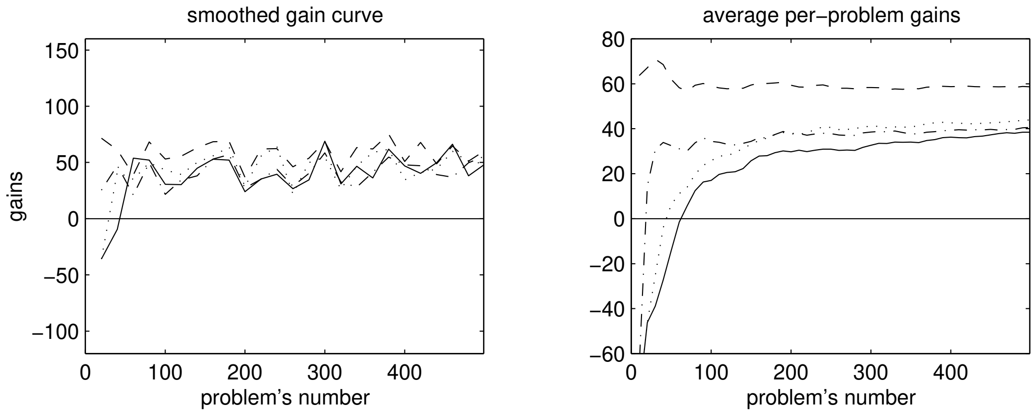
Figure 14.12: Comparison of three incremental-learning tasks, which include selection among all thirty-six representations (solid), among the twelve alternatives in Figure 14.11 (dotted), and among the four domain descriptions in Figure 14.2 (dash-and-dot). In addition, the graphs show the behavior of the best available representation, with the optimal time bound (dashes).



(a) Gain linearly decreases with the running time (Function 14.1).

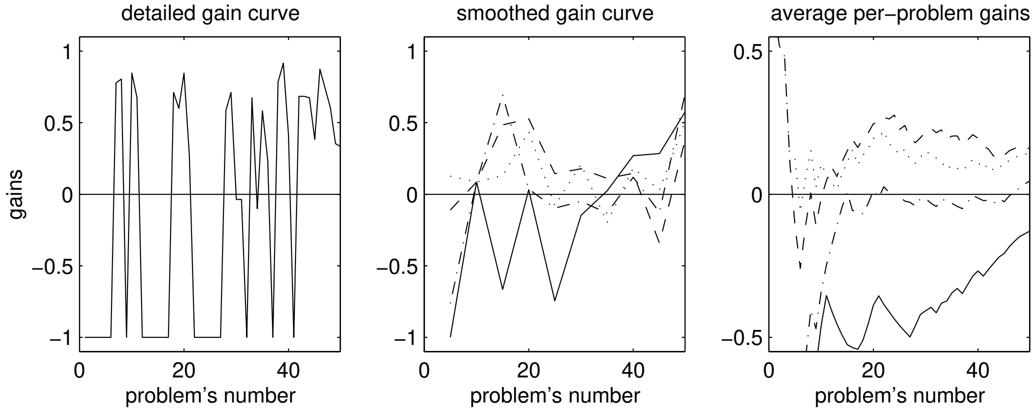


(b) Gain decreases with the time and solution cost (Function 14.2).

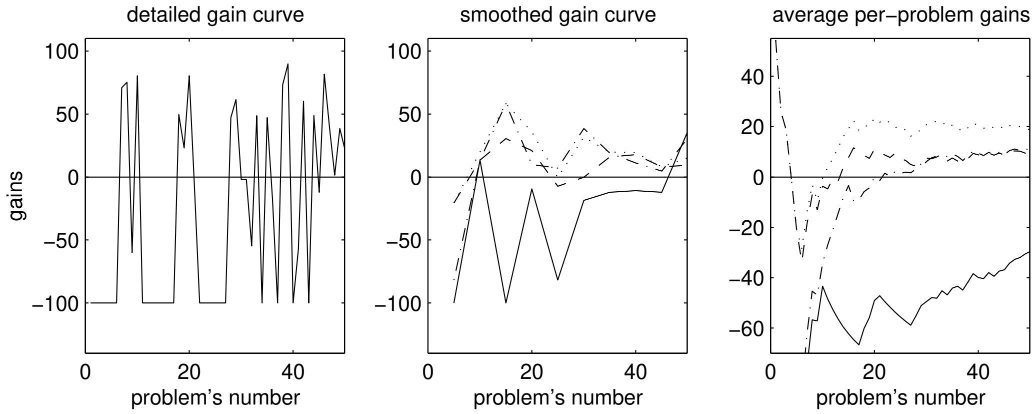


(c) Gain is a nonlinear function of time and cost (Function 14.3).

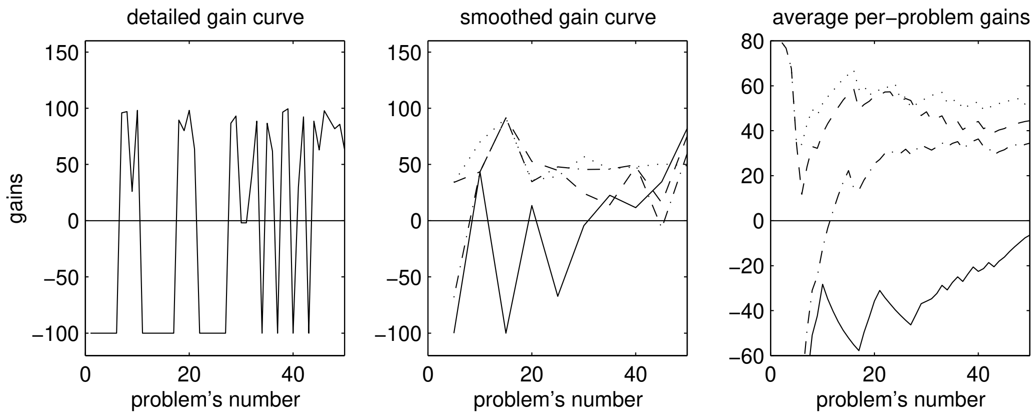
Figure 14.13: Comparing the three incremental-selection tasks of different scale (see Figure 14.12), on longer problem sequences. We plot the results of utilizing large representation space (solid lines), two smaller spaces (dotted and dash-and-dot lines), and optimal representation (dashed lines).



(a) Gain linearly decreases with the running time (Function 14.1).

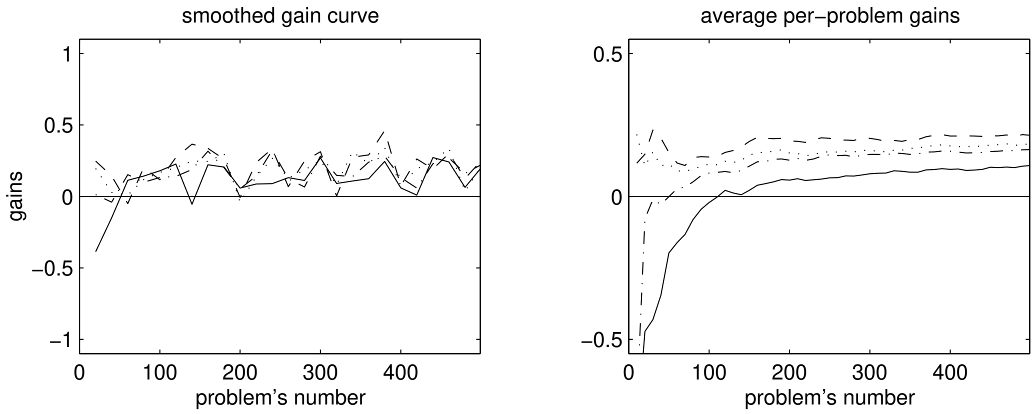


(b) Gain decreases with the time and solution cost (Function 14.2).

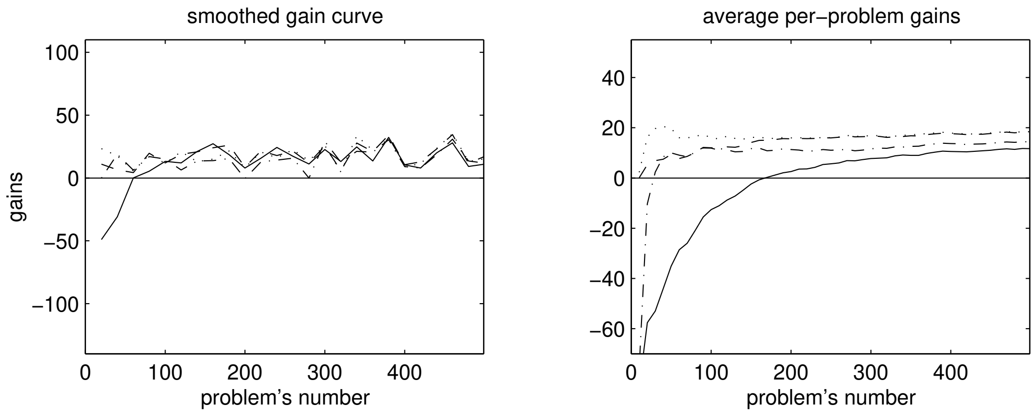


(c) Gain is a nonlinear function of time and cost (Function 14.3).

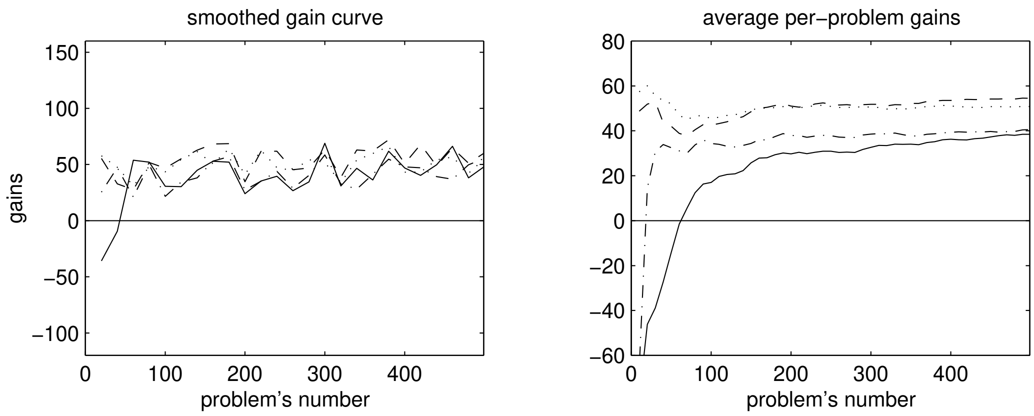
Figure 14.14: Processing short problem sequences with a space of thirty-six representations. We plot the learning curves (solid) and the analogous data for three small-scale tasks: choice among four descriptions (dash-and-dot), three search engines (dots), and three time-bound heuristics (dashes).



(a) Gain linearly decreases with the running time (Function 14.1).



(b) Gain decreases with the time and solution cost (Function 14.2).



(c) Gain is a nonlinear function of time and cost (Function 14.3).

Figure 14.15: Running $\mathcal{S}$HAPER on long sequences of problems, with thirty-six representations (solid lines) and with smaller spaces (broken lines); the legend is the same as in Figure 14.14.

14.3 Different time bounds

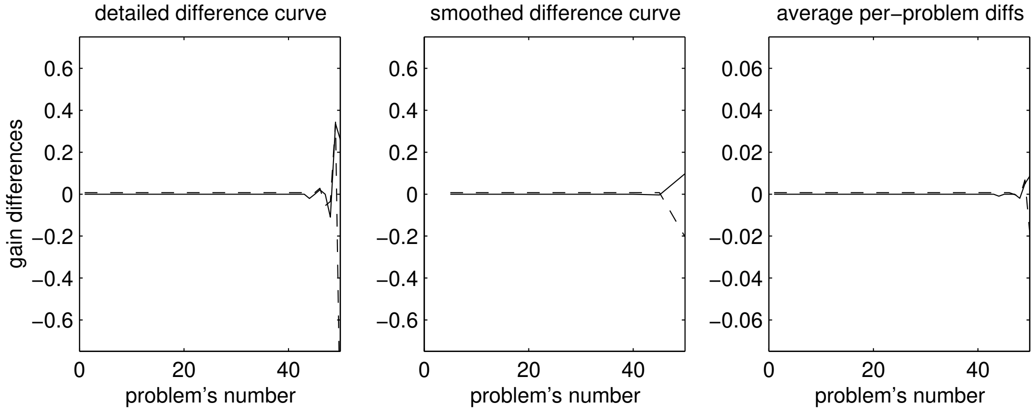
We re-ran the large-scale tests with different settings of the exploration knob, using the same experimental setup as in the other domains (see Section 12.3). The results of using small knob values are given in Figures 14.16 and 14.17, and the data for large values are in Figures 14.18 and 14.19. For each knob setting, the graphs show the difference between the resulting performance and the default behavior.

The tests confirmed that the default setting gives near-optimal results (see Table 14.3). The largest deviation from the optimum was in the experiment with Function 14.3, when a larger knob value lead to a 6% increase in the total gains (see the bottom row of Table 14.3).

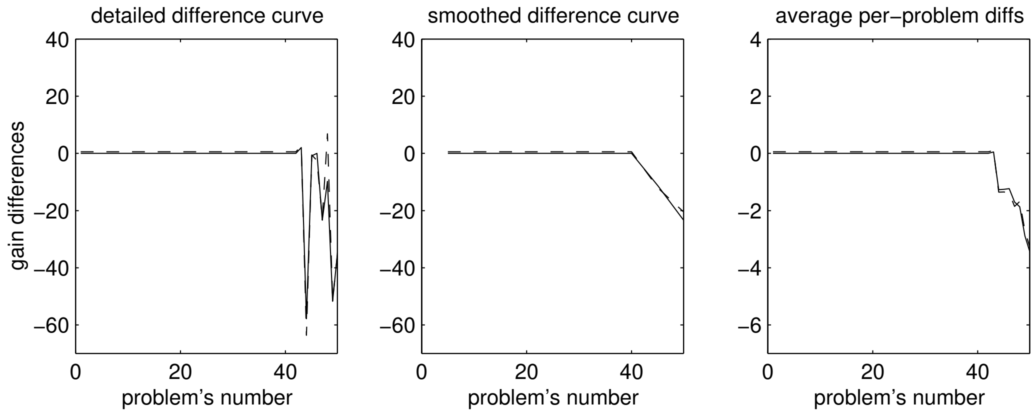
Note that the exploration knob does *not* affect the system's behavior in the beginning of a learning sequence (see the left-hand graphs in Figures 14.16 and 14.18). The control module initially employs a heuristic for choosing time bounds, which does not depend on a knob setting (see Section 8.5.1), and switches to the statistical computation only after accumulating preliminary data.

	small knob values				default	large knob values			
	0.02		0.05		0.1	0.2		0.5	
					short problem sequences				
Function 14.1	−0.119	—	−0.150	—	−0.128	−0.119	—	−0.124	—
Function 14.2	−33.0	—	−32.9	—	−29.5	−29.1	—	−29.7	—
Function 14.3	−9.68	—	−10.46	—	−6.27	−7.14	—	−6.35	—
					long problem sequences				
Function 14.1	0.106	(98%)	0.088	(81%)	0.108	.110	(102%)	0.096	(89%)
Function 14.2	9.3	(79%)	10.1	(86%)	11.7	10.5	(90%)	8.9	(76%)
Function 14.3	33.5	(87%)	37.6	(98%)	38.5	40.9	(106%)	39.2	(102%)

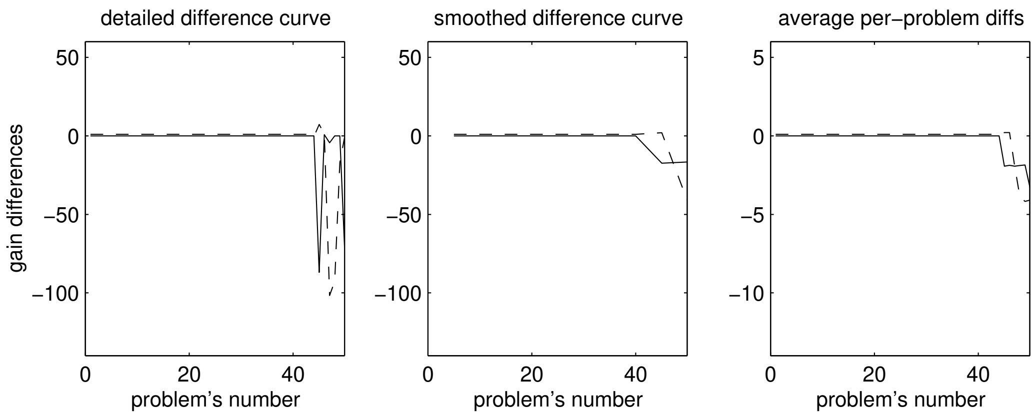
Table 14.3: Average per-problem gains for five different values of the exploration knob. For each positive gain, we indicate the corresponding percentage of the default-strategy gain.



(a) Gain linearly decreases with the running time (Function 14.1).

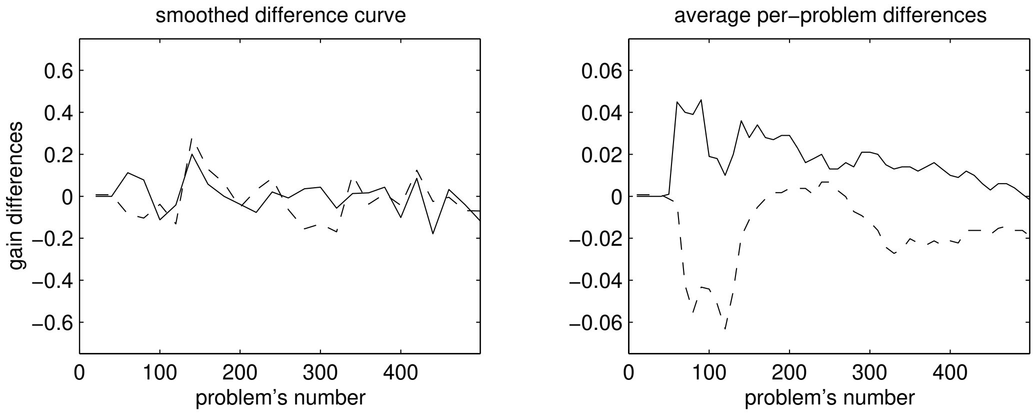


(b) Gain decreases with the time and solution cost (Function 14.2).

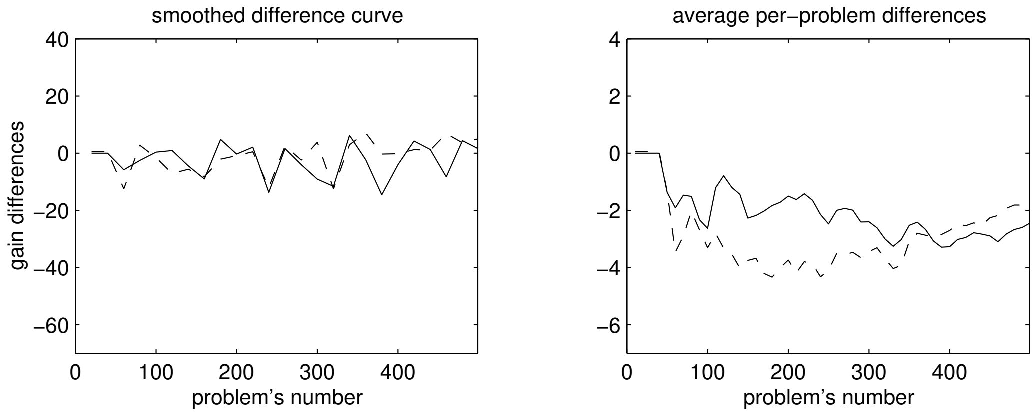


(c) Gain is a nonlinear function of time and cost (Function 14.3).

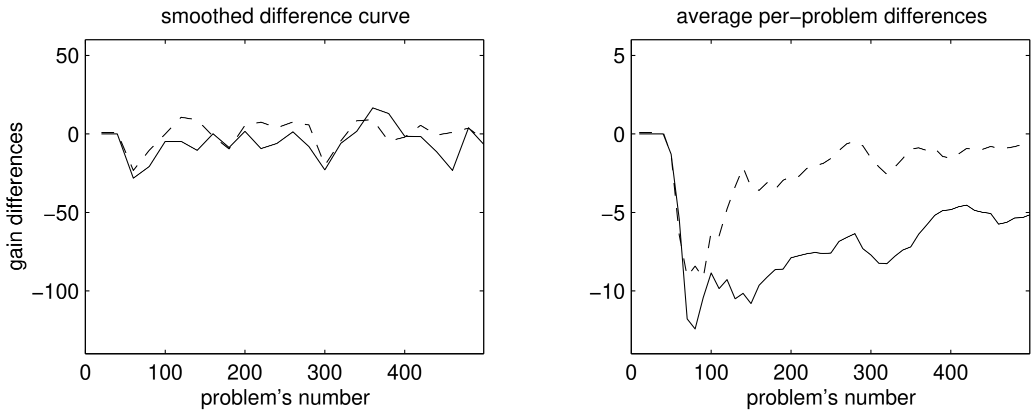
Figure 14.16: Experiments with small values of the exploration knob. The solid lines represent the differences between the 0.02-knob results and 0.1-knob results. Similarly, the dashed lines mark the differences between the 0.05-knob and 0.1-knob gains.



(a) Gain linearly decreases with the running time (Function 14.1).

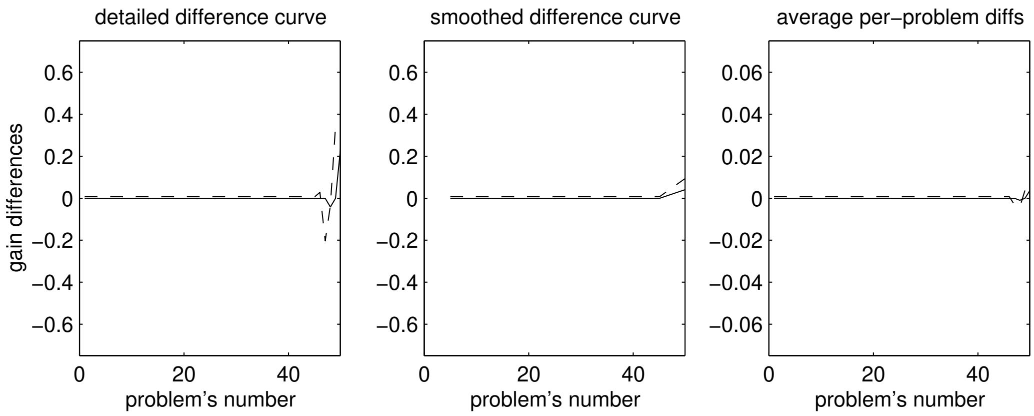


(b) Gain decreases with the time and solution cost (Function 14.2).

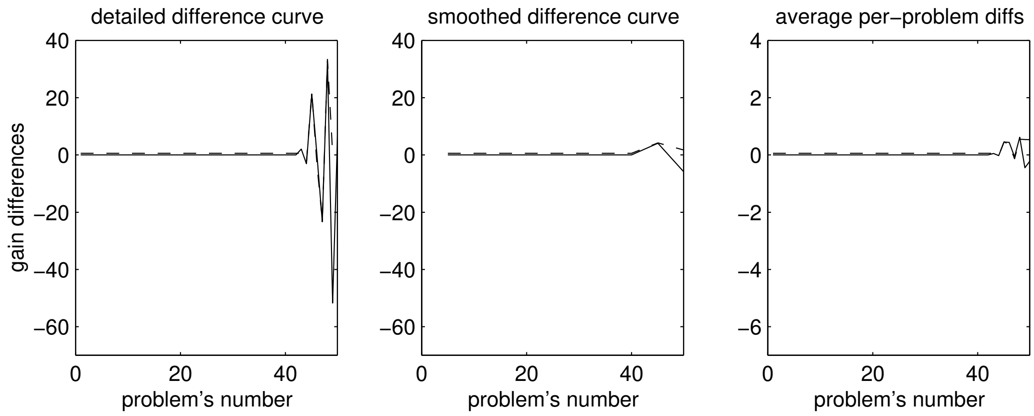


(c) Gain is a nonlinear function of time and cost (Function 14.3).

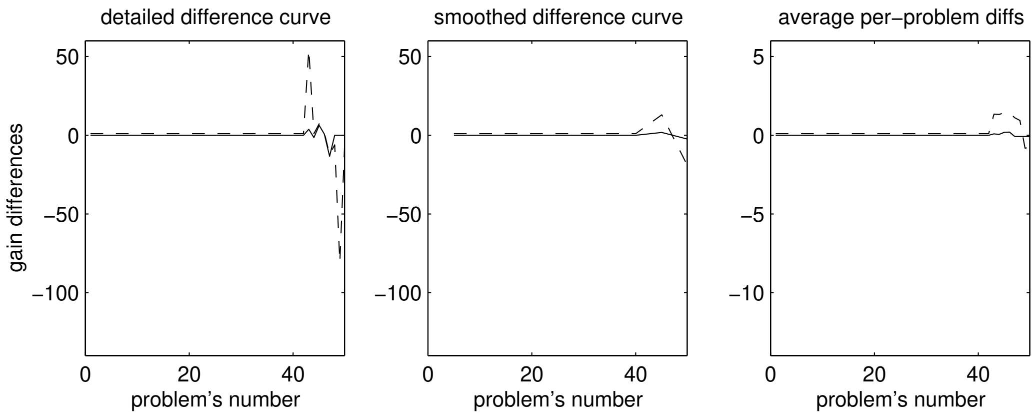
Figure 14.17: Results of processing long problem sequences with the knob values 0.02 and 0.05.



(a) Gain linearly decreases with the running time (Function 14.1).

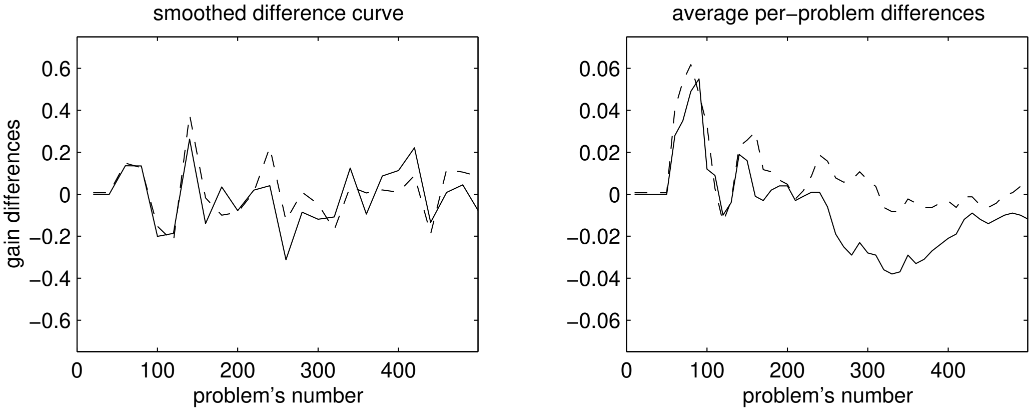


(b) Gain decreases with the time and solution cost (Function 14.2).

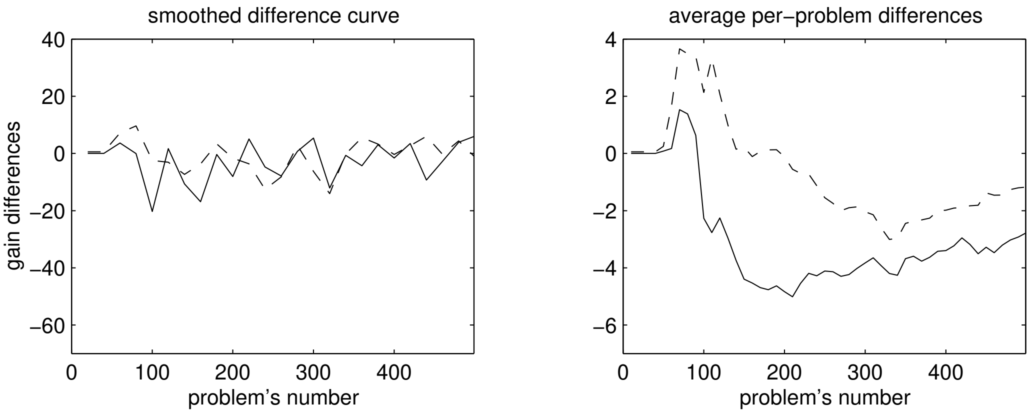


(c) Gain is a nonlinear function of time and cost (Function 14.3).

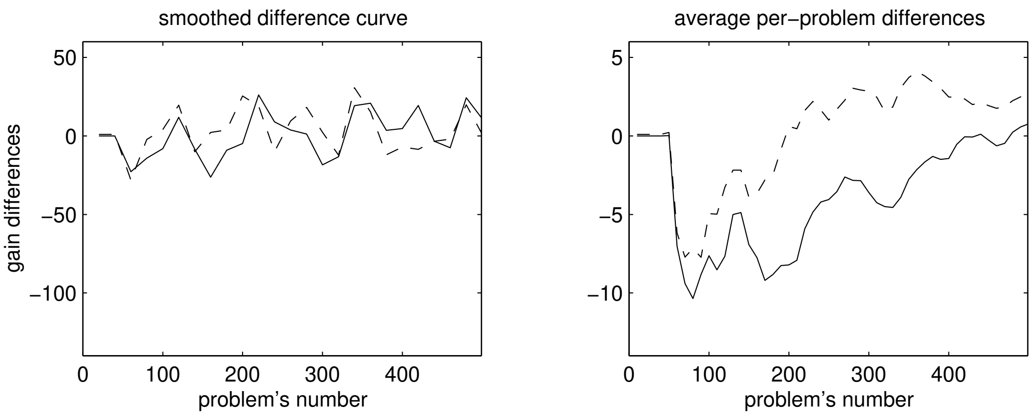
Figure 14.18: Testing large knob values on fifty-problem sequences. We plot the differences between the 0.5-knob gains and default-knob gains (solid lines), and similar difference curves for the 0.2-knob tests (dashed lines).



(a) Gain linearly decreases with the running time (Function 14.1).



(b) Gain decreases with the time and solution cost (Function 14.2).



(c) Gain is a nonlinear function of time and cost (Function 14.3).

Figure 14.19: Applying $\mathcal{S}$HAPER with the knob values 0.5 and 0.2 to long sequences of problems.

Chapter 15

Logistics Domain

The last series of experiments is based on the PRODIGY Logistics domain [Veloso, 1994], which comprises eight object types, six operators, and two inference rules (see Figure 3.47 on page 147). Recall that Logistics problems involve delivery of packages, by vans and airplanes; all of them are solvable, but some require hours of PRODIGY search.

First, we experimented with linear dependencies of gain on the running time, solution quality, and problem size (see Figure 15.1a–c). Then, we tested the system's ability to deal with nonlinear utility models, using the "unnatural" gain functions in Figure 15.1(d–f).

15.1 Choosing a description and solver

We conducted small-scale experiments with two representation spaces, illustrated in Figures 15.2 and 15.3. First, $\mathcal{S}$HAPER ran with the LINEAR solver and standard library of changers (see Figure 15.2a), and expanded a space of two descriptions (Figure 15.2b). The abstraction hierarchy in the new description was as shown in Figure 4.34 (page 186). We tested the selection mechanism with the linear gain functions, and it chose the right description and time bound for each function (see Figures 15.4 and 15.6).

The second experiment involved a library of three search engines, without cost bounds. We added the abstraction hierarchy to the domain encoding and ran the system *without* changer operators, thus obtaining the representations in Figure 15.3. The control module correctly determined that LINEAR was the best solver, and found a proper time limit for each utility function (see Figures 15.6 and 15.7).

In Table 15.1, we summarize the cumulative gains and compare them with the optimal-strategy results. The gains on the short problem sequences varied from 53% to 84% of optimal, whereas the long-sequence results were at least 84%. The system converged to the right strategy after processing thirty to a hundred problems, which was similar to its behavior in other domains.

The Logistics tests have confirmed that a significant efficiency improvement usually does *not* translate into a proportional gain increase. The use of abstraction resulted in ten-fold to hundred-fold search reduction on most problems (see Section 4.4), but the respective growth of the optimal-strategy gains was much smaller (see Table 15.2).

Linear gain functions

(a) Gain linearly decreases with the problem-solving time:

$$gain = \begin{cases} 1 - time, & \text{if success} \\ -time, & \text{if failure or interrupt} \end{cases} \quad (15.1)$$

(b) Gain depends on the search time and solution cost:

$$gain = \begin{cases} 100 - cost - 50 \cdot time, & \text{if success and } cost < 100 \\ -50 \cdot time, & \text{otherwise} \end{cases} \quad (15.2)$$

(c) Reward is proportional to the number of delivered packages:

$$gain = \begin{cases} 50 \cdot n\text{-}packs - cost - 50 \cdot time, & \text{if success and } cost < 50 \cdot n\text{-}packs \\ -50 \cdot time, & \text{otherwise} \end{cases} \quad (15.3)$$

Artificial functions

(d) Gain linearly decreases with the cube of the running time:

$$gain = \begin{cases} 1 - time^3, & \text{if success} \\ -time^3, & \text{if failure or interrupt} \end{cases} \quad (15.4)$$

(e) SHAPER has to find a solution with cost less than 50:

$$gain = \begin{cases} 2 - time, & \text{if success and } cost < 50 \\ -time, & \text{otherwise} \end{cases} \quad (15.5)$$

(f) Payment for the unit time is proportional to the number of packages:

$$gain = \begin{cases} 4 - n\text{-}packs \cdot time, & \text{if success} \\ -n\text{-}packs \cdot time, & \text{if failure or interrupt} \end{cases} \quad (15.6)$$

Figure 15.1: Utility computation in the Logistics Domain. The small-scale selection tasks (see Section 15.1) are based on three linear functions for calculating gains. The experiments with a larger space (see Section 15.2) involve not only linear dependencies, but also artificial functions, which test SHAPER's capability of adapting to complex utility models.

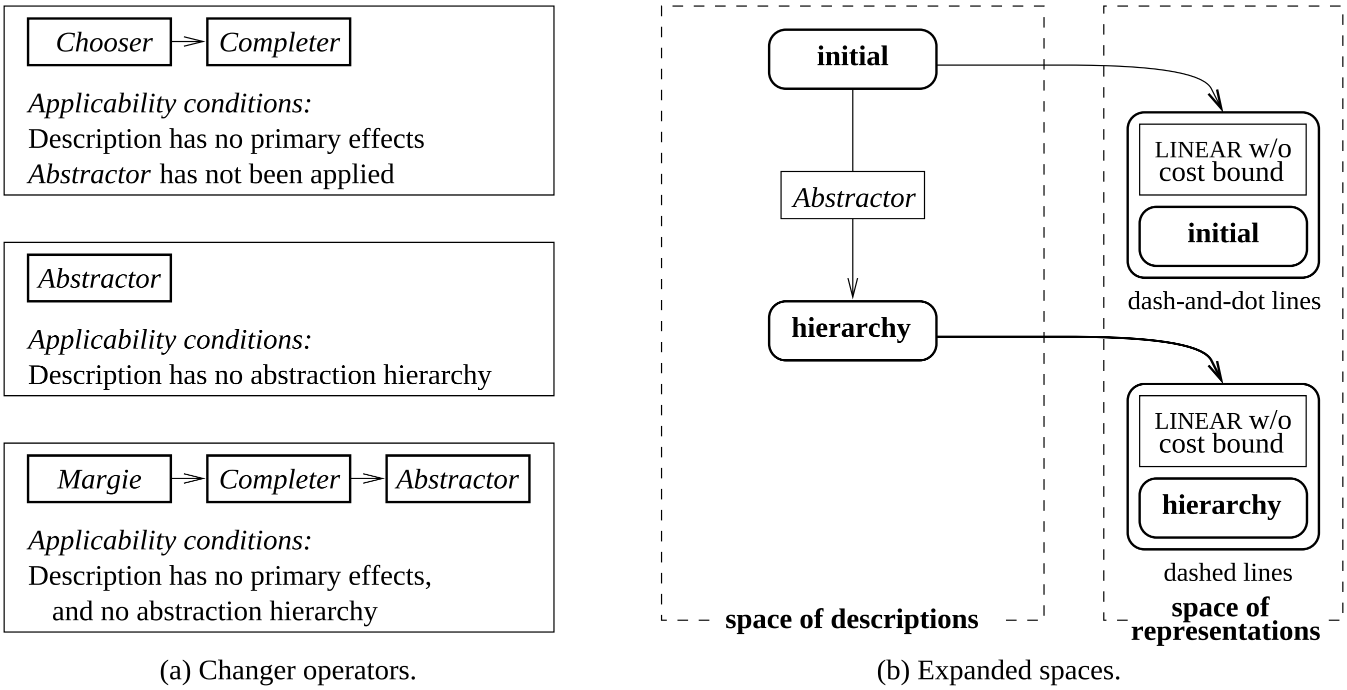


Figure 15.2: Changer operators in the experiments on selecting a domain description (a), and the expanded representation space (b). The *Abstractor* algorithm builds a four-level hierarchy, whereas the other changers fail to generate new descriptions.

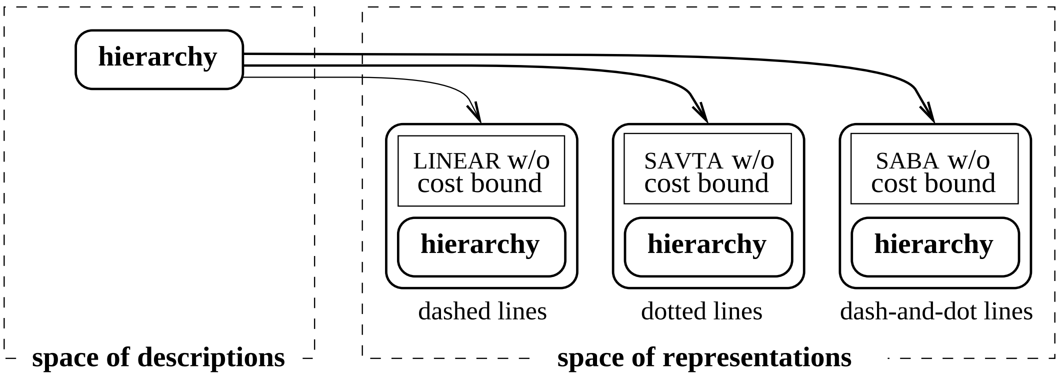


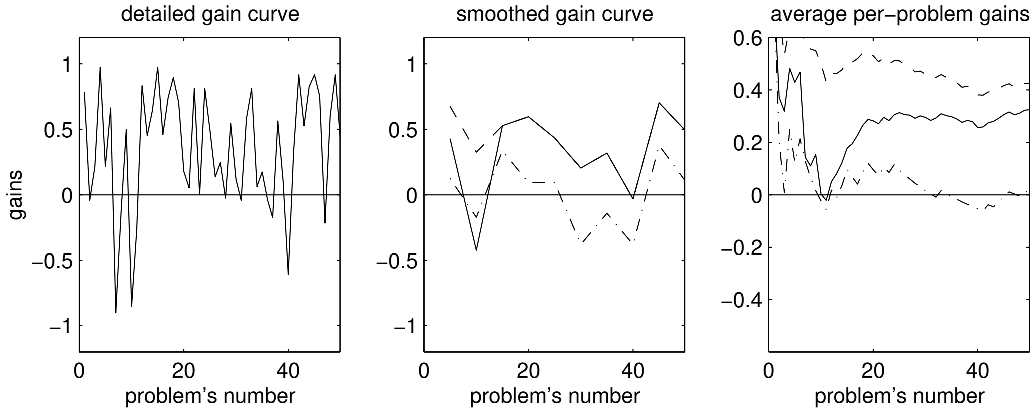
Figure 15.3: Running the system without changer operators. The control mechanism combines the three search engines with a given domain description, which includes an abstraction hierarchy.

	choice among				optimal
	descriptions		solvers		gain
	short problem sequences				
Function 15.1	0.325	(77%)	0.225	(53%)	0.424
Function 15.2	13.6	(75%)	13.9	(77%)	18.1
Function 15.3	55.4	(84%)	54.6	(83%)	65.7
	long problem sequences				
Function 15.1	0.397	(95%)	0.360	(87%)	0.416
Function 15.2	21.6	(95%)	19.1	(84%)	22.8
Function 15.3	58.3	(97%)	54.2	(90%)	60.1

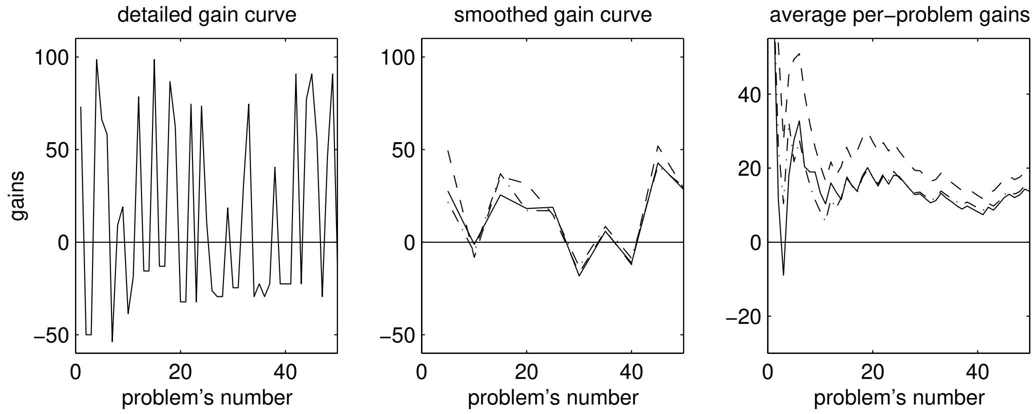
Table 15.1: Average per-problem gains in the small-scale selection experiments (Figures 15.4–15.7), and the corresponding percentages of the optimal-strategy results.

	gain increase
Function 15.1	4.2
Function 15.2	1.3
Function 15.3	1.8

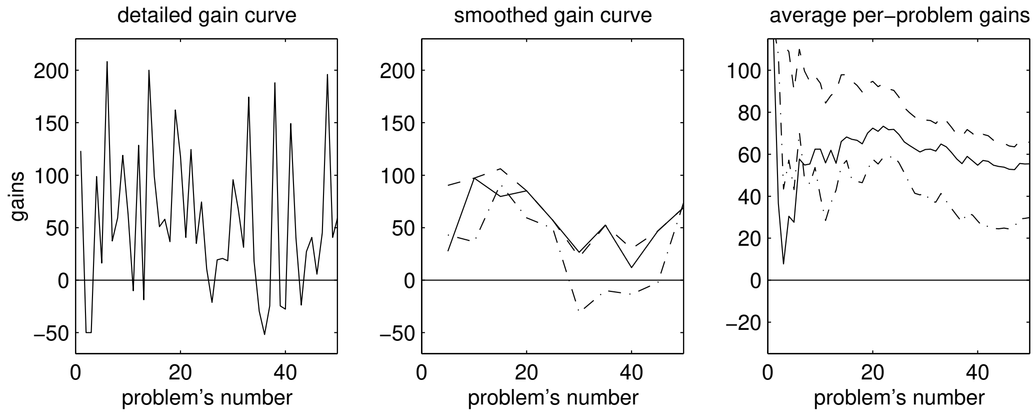
Table 15.2: Increase in the cumulative gains due to abstraction. We specify the increase factor for each of the three utility functions. Observe that the growth of gains is much smaller than the time-saving factor, which is usually between 10 and 100.



(a) Gain linearly decreases with the running time (Function 15.1).

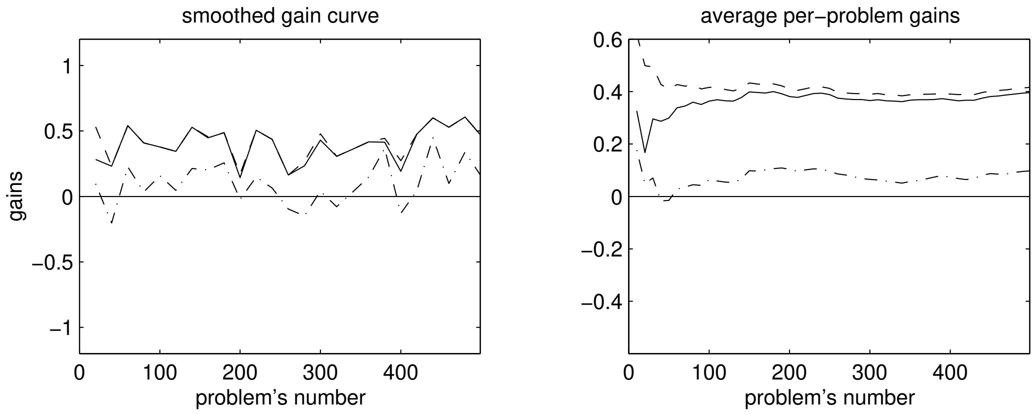


(b) Gain decreases with the time and solution cost (Function 15.2).

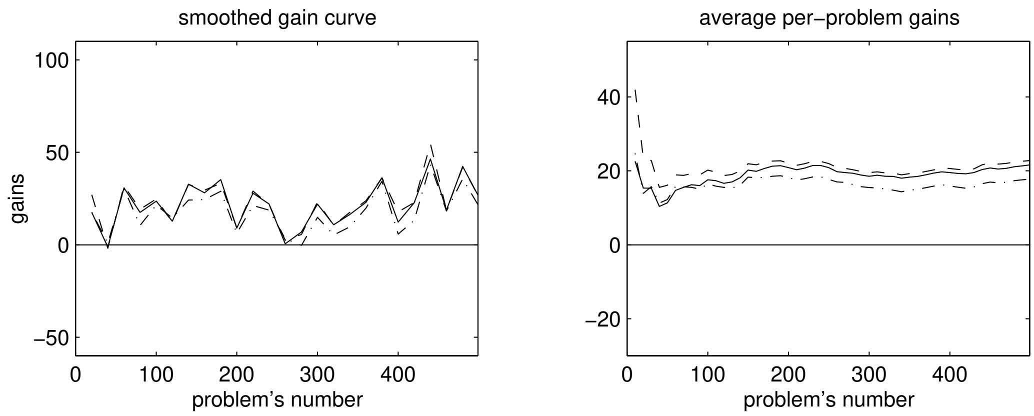


(c) Reward depends on the number of packages (Function 15.3).

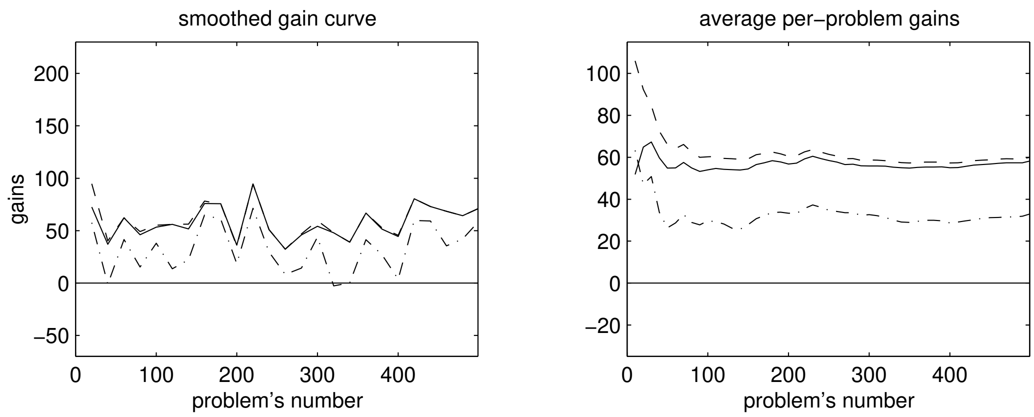
Figure 15.4: Choosing between the initial description and abstraction. The graphs include the learning curves (solid), the results of using the initial description with the optimal time bound (dash-and-dot), and the optimal behavior of abstraction search (dashes).



(a) Gain linearly decreases with the running time (Function 15.1).

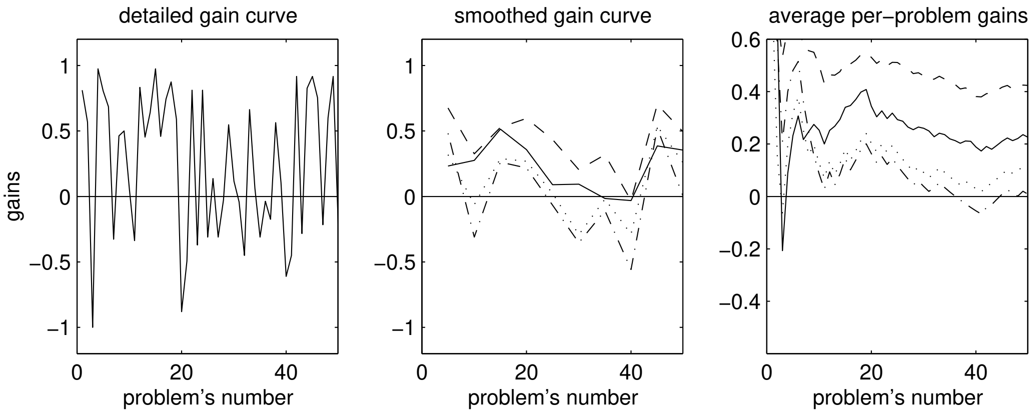


(b) Gain decreases with the time and solution cost (Function 15.2).

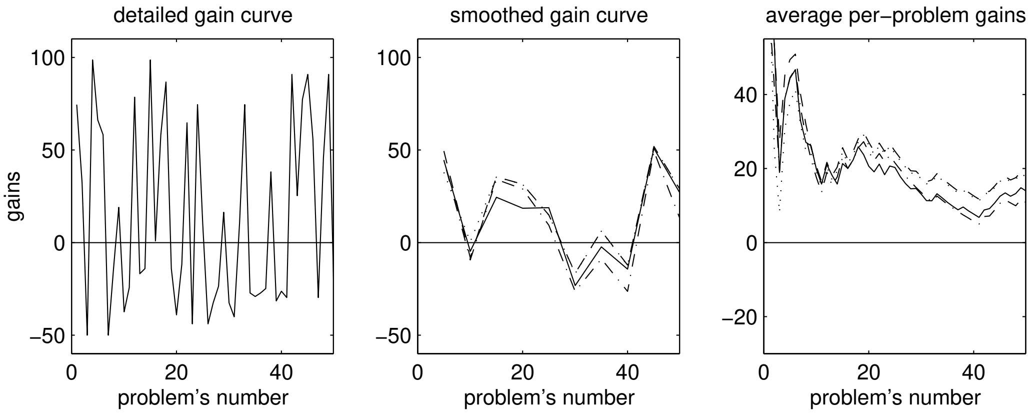


(c) Reward depends on the number of packages (Function 15.3).

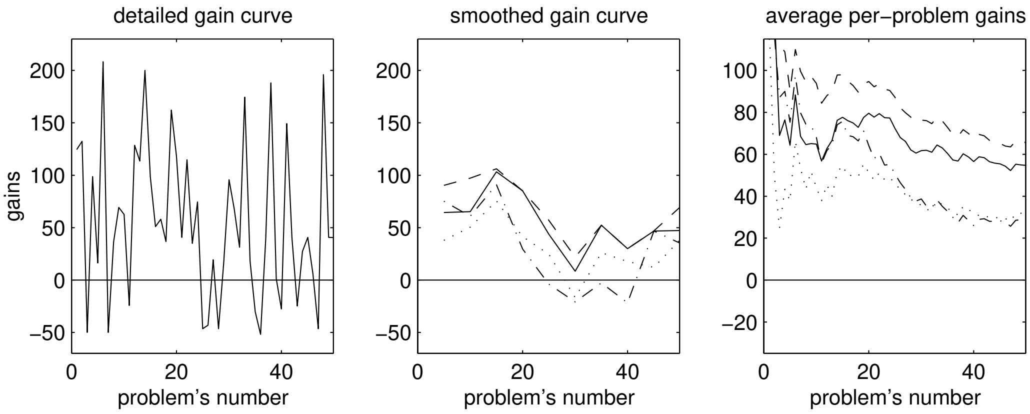
Figure 15.5: Processing long problem sequences, with the two alternative descriptions. We plot the smoothed gain curves (left) and cumulative gains (right), using the legend of Figure 15.4.



(a) Gain linearly decreases with the running time (Function 15.1).

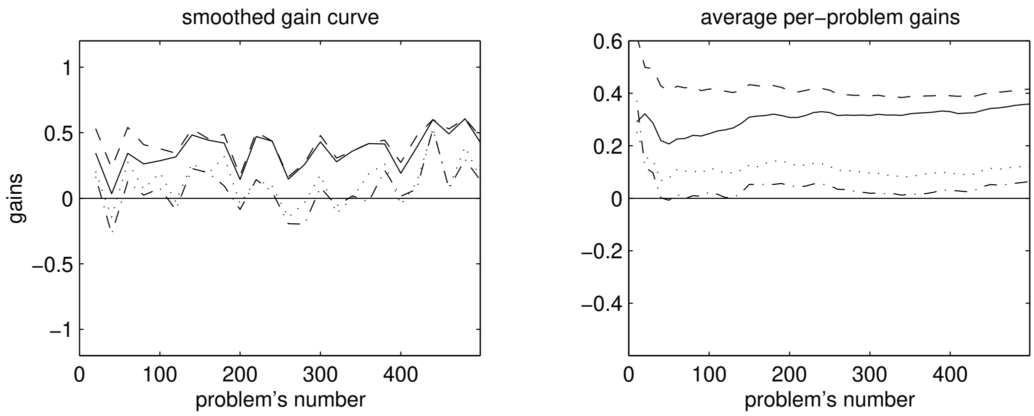


(b) Gain decreases with the time and solution cost (Function 15.2).

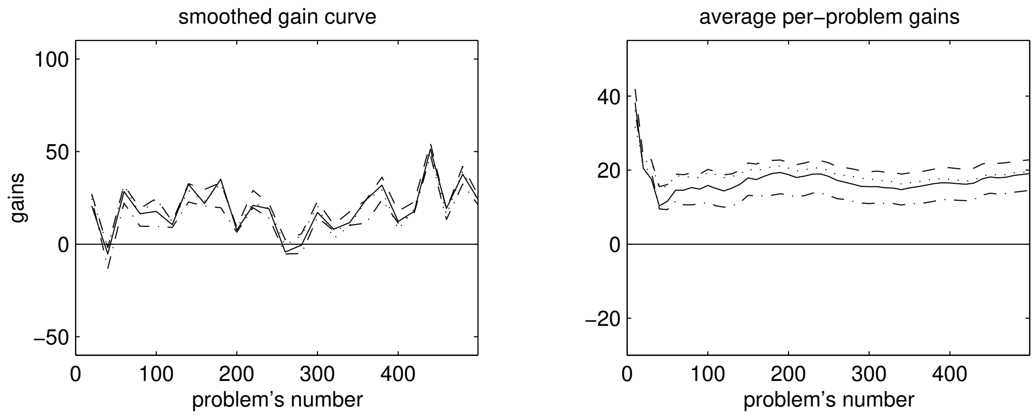


(c) Reward depends on the number of packages (Function 15.3).

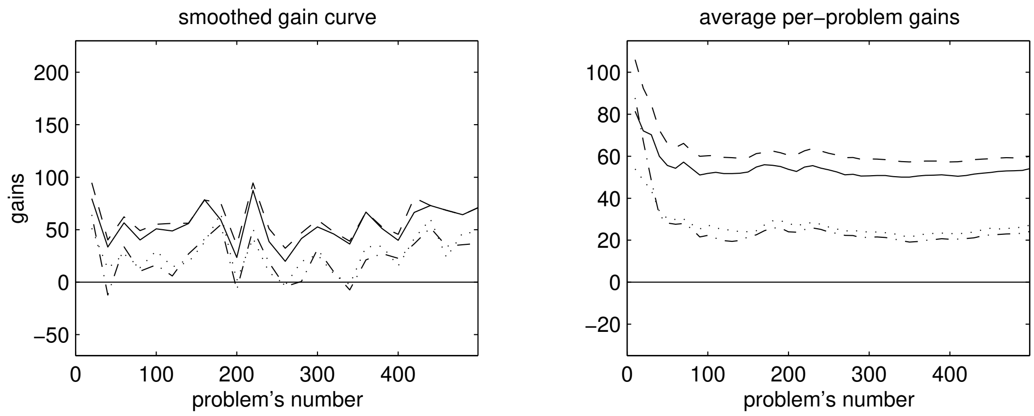
Figure 15.6: Choosing among the search engines, without cost bounds. We give the learning curves (solid), and the optimal performance of LINEAR (dashes), SAVTA (dots), and SABA (dash-and-dot).



(a) Gain linearly decreases with the running time (Function 15.1).



(b) Gain decreases with the time and solution cost (Function 15.2).



(c) Reward depends on the number of packages (Function 15.3).

Figure 15.7: Applying $\mathcal{S}$HAPER to long problem sequences, with a library of three search engines. The graphs comprise the results of incremental selection (solid lines), as well as the optimal behavior of each search algorithm (broken lines); the legend is the same as in Figure 15.6.

15.2 Space of twelve representations

We now consider a library of six solvers, which include not only the basic search engines, but also their synergy with a heuristic for computing *loose cost bounds.* The heuristic procedure estimates the cost of an optimal solution and sets a twice larger bound, thus limiting the depth of PRODIGY search. The system employs the six solvers along with the changers in Figure 15.2(a), and constructs the twelve representations in Figure 15.8.

We first experimented with the linear gain functions (Figures 15.9 and 15.10), and then with the artificial functions (Figures 15.11 and 15.12). The LINEAR algorithm with abstraction and no cost bounds proved more effective than the other representations, and SHAPER chose it in all cases. Furthermore, the system found proper time bounds for Functions 15.1–15.5; however, it underestimated the optimal bound for Function 15.6, and the selected time limit (0.99 seconds) gave noticeably worse results than optimal (1.15 seconds).

The convergence was slower than in the other domains: SHAPER found the right strategy after processing 300 to 500 problems; however, the percentage values of the cumulative gains (see Table 15.3) were no smaller than in the Sokoban and STRIPS domain. The slow convergence was due to several close-to-optimal representations, which caused the control module to "hesitate" among them. Since the system discarded ineffective strategies during the early stages of learning, its later hesitation had little effect on performance.

	selection among						optimal
	twelve representations (solid lines)		two descriptions (dash-and-dot lines)		three solvers (dotted lines)		gain (dashed lines)
	short problem sequences						
Function 15.1	−0.314	—	0.325	(77%)	0.225	(53%)	0.424
Function 15.2	3.4	(25%)	13.6	(75%)	13.9	(77%)	18.1
Function 15.3	12.3	(22%)	55.4	(84%)	54.6	(83%)	65.7
Function 15.4	0.071	(16%)	0.339	(75%)	0.432	(95%)	0.454
Function 15.5	0.074	(15%)	0.274	(55%)	0.398	(79%)	0.501
Function 15.6	0.083	(5%)	1.388	(84%)	1.262	(77%)	1.646
	long problem sequences						
Function 15.1	0.268	(64%)	0.397	(95%)	0.360	(87%)	0.416
Function 15.2	14.1	(62%)	21.6	(95%)	19.1	(84%)	22.8
Function 15.3	37.7	(63%)	58.3	(97%)	54.2	(90%)	60.1
Function 15.4	0.365	(70%)	0.454	(87%)	0.498	(96%)	0.521
Function 15.5	0.423	(70%)	0.385	(64%)	0.582	(96%)	0.605
Function 15.6	0.712	(44%)	1.511	(92%)	1.529	(94%)	1.634

Table 15.3: Summary of the incremental-learning results in the Logistics Domain. We list the average per-problem gains and the respective percentages of the optimal gains. The parenthetical notes in the column headings refer to Figures 14.14–14.13.

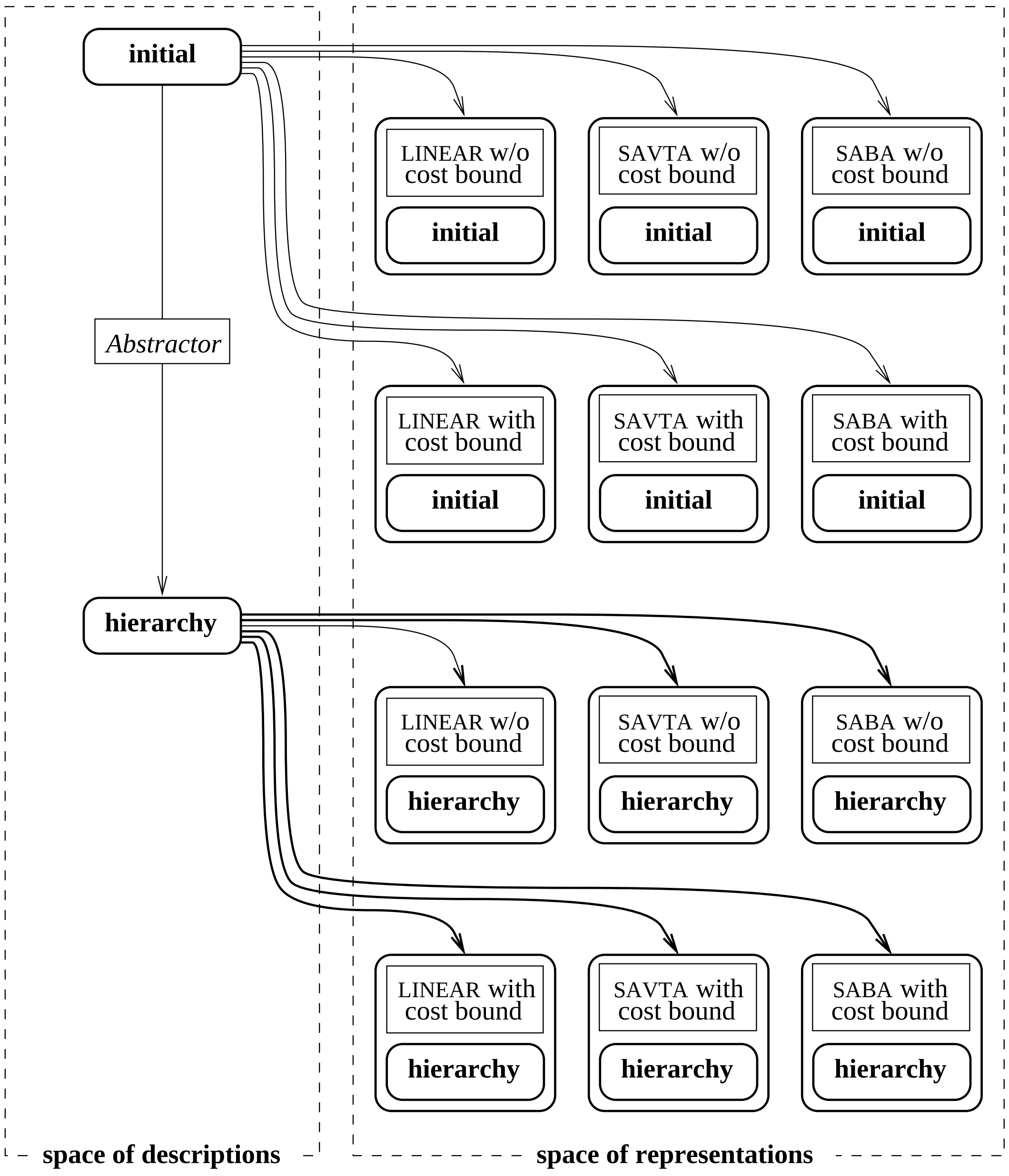
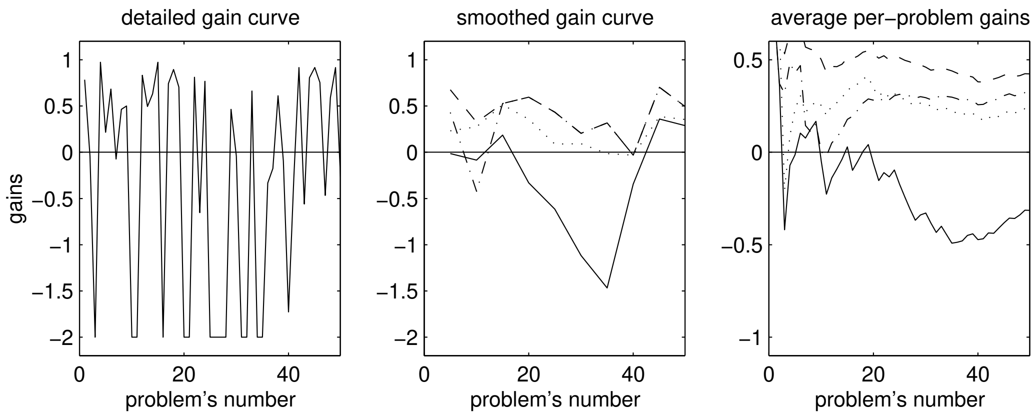
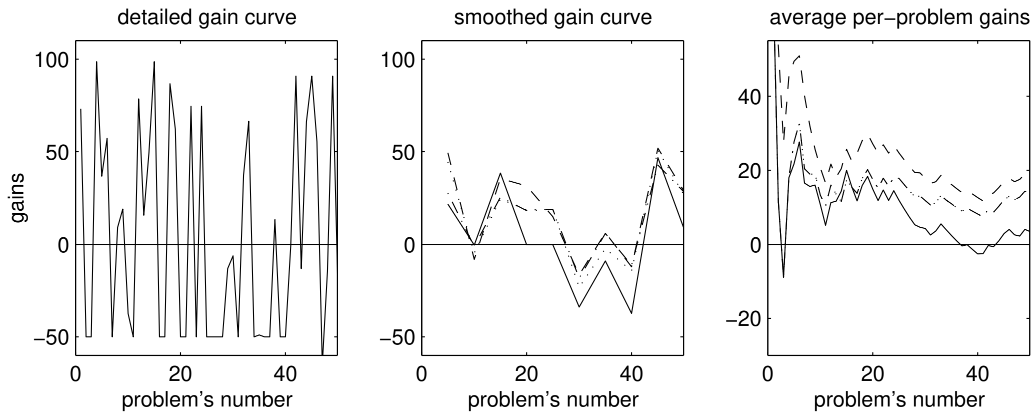


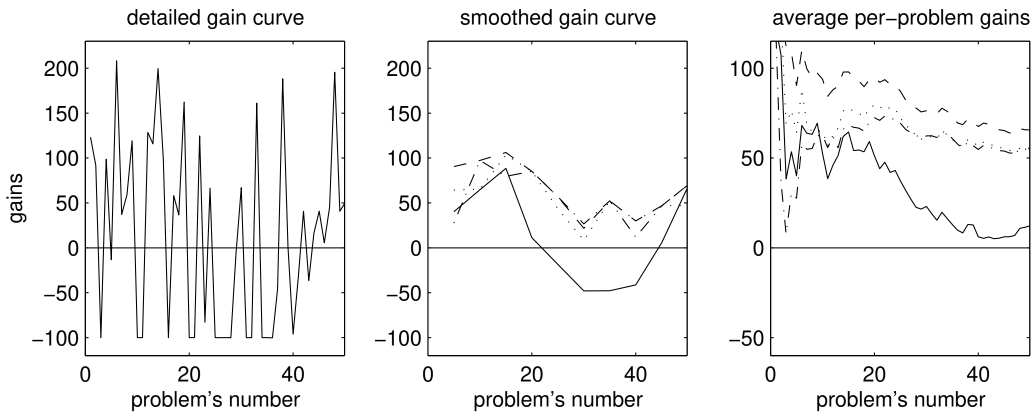
Figure 15.8: Large representation space in the Logistics Domain. The system generates an abstraction hierarchy, and then combines the initial and new description with six solver operators.



(a) Gain linearly decreases with the running time (Function 15.1).

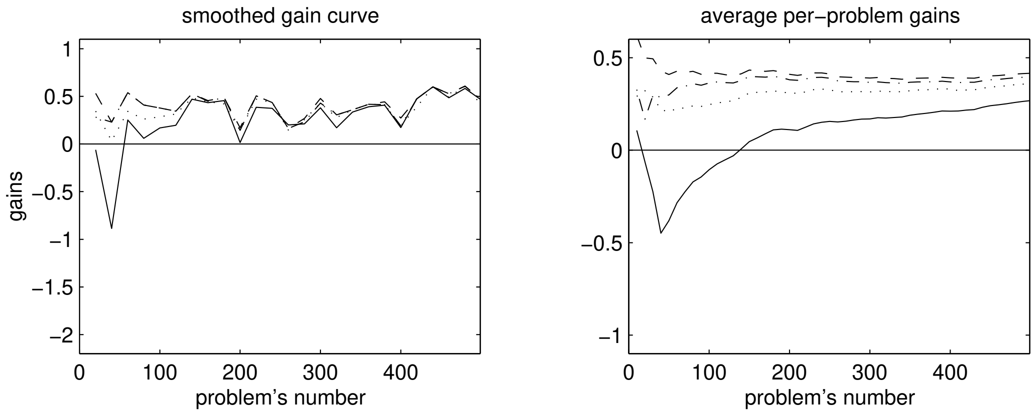


(b) Gain decreases with the time and solution cost (Function 15.2).

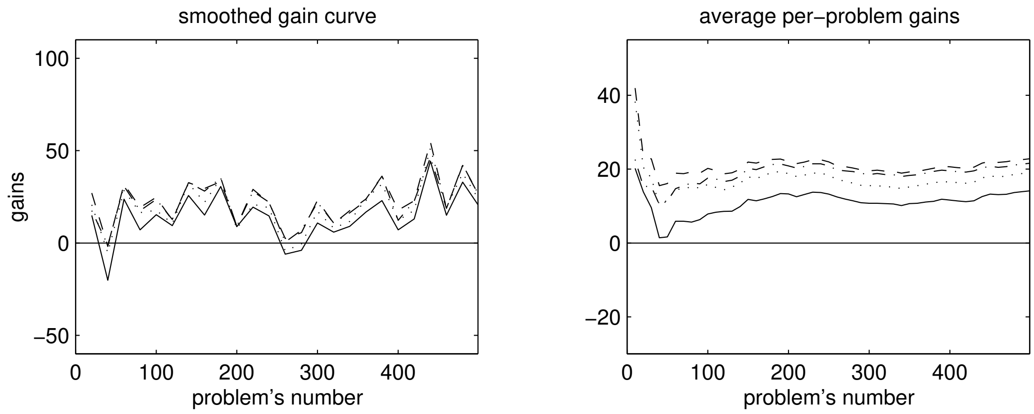


(c) Reward depends on the number of packages (Function 15.3).

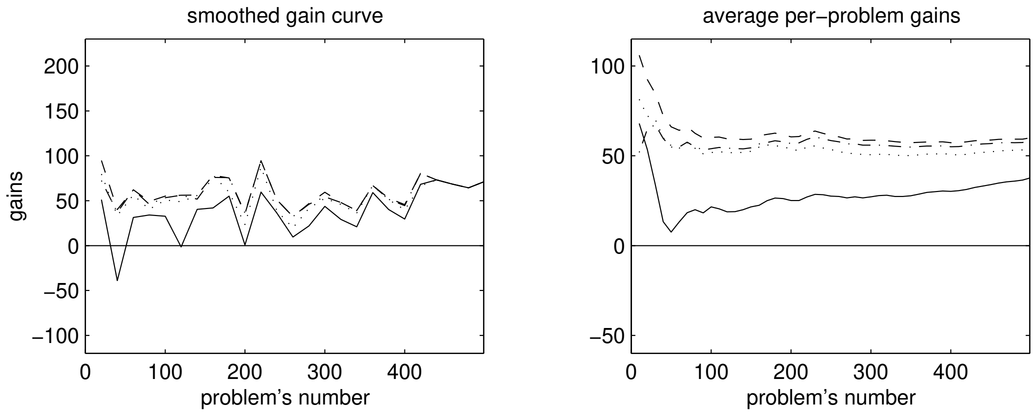
Figure 15.9: Selection of search strategies for the linear gain functions. The graphs show $\mathcal{S}$HAPER's behavior with a space of twelve representations (solid lines), as well the results of choosing among two descriptions (dash-and-dot lines) and among three search engines (dotted lines). In addition, we plot the performance of the optimal strategy, which is based on the LINEAR algorithm, with abstraction and no cost bounds.



(a) Gain linearly decreases with the running time (Function 15.1).

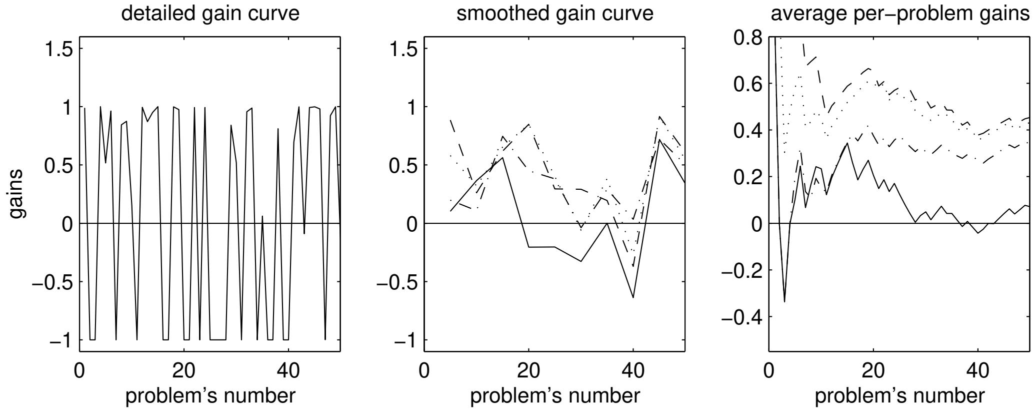


(b) Gain decreases with the time and solution cost (Function 15.2).

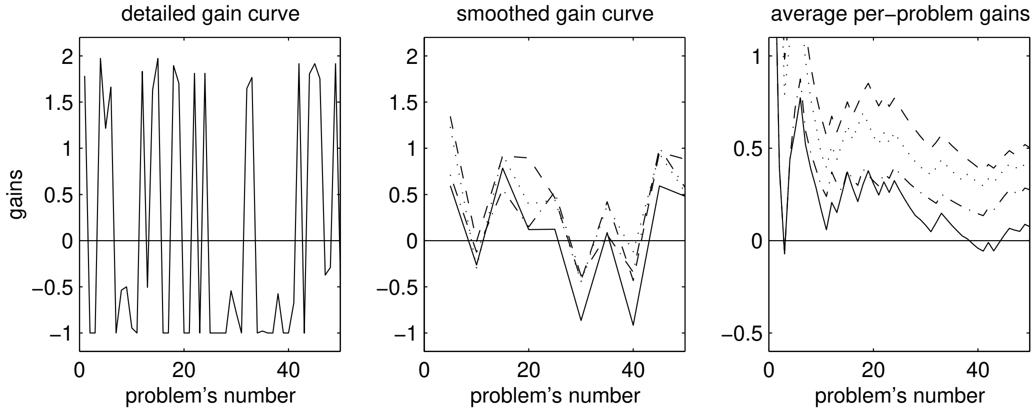


(c) Reward depends on the number of packages (Function 15.3).

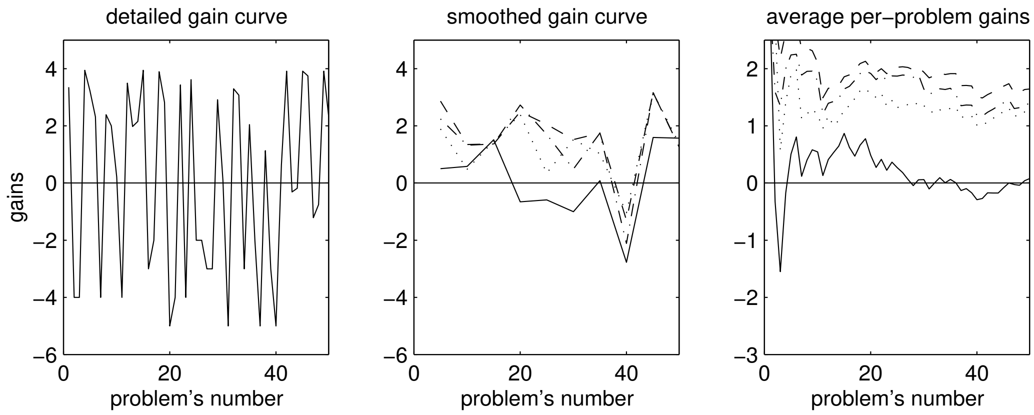
Figure 15.10: Processing long problem sequences with the large representation space (solid lines), and the analogous data for the two small spaces (dotted and dash-and-dot lines).



(a) Gain linearly decreases with the cube of time (Function 15.4).

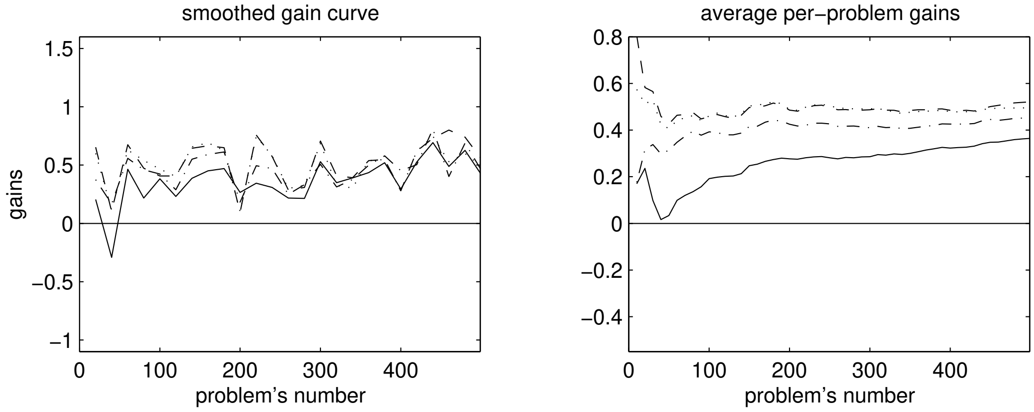


(b) System gets a reward only for a low-cost solution (Function 15.5).

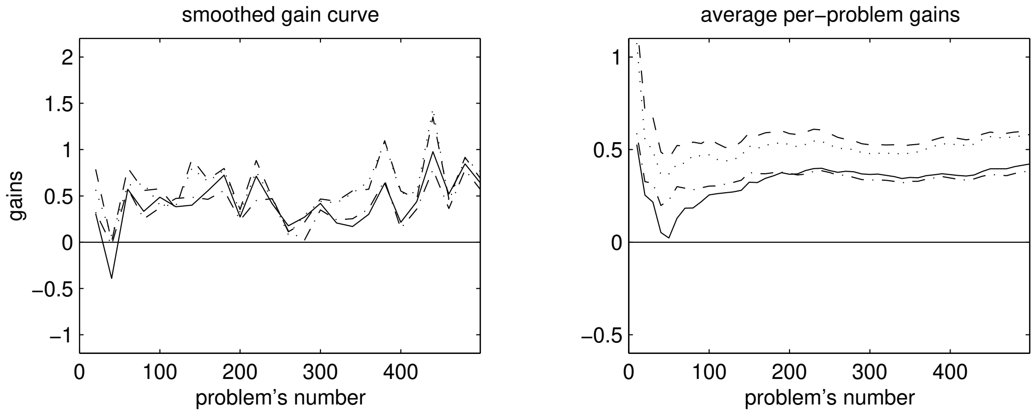


(c) Gain depends on the search time and problem size (Function 15.6).

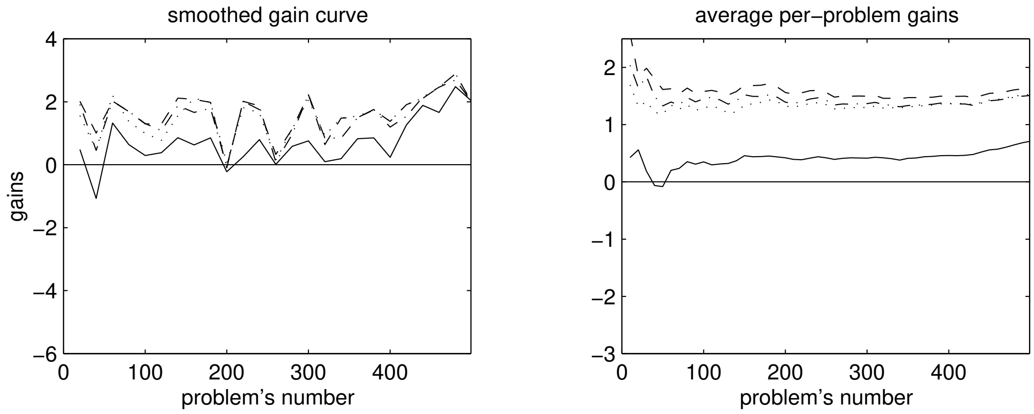
Figure 15.11: Choosing representations for the artificial utility functions. We present the results of utilizing the space of twelve representations (solid lines), the system's behavior with the smaller spaces (dotted and dash-and-dot lines), and the performance of the best available strategy (dashes); the legend corresponds to that in Figure 15.10.



(a) Gain linearly decreases with the cube of time (Function 15.4).



(b) System gets a reward only for a low-cost solution (Function 15.5).



(c) Gain depends on the search time and problem size (Function 15.6).

Figure 15.12: Results of processing long problem sequences, with the artificial gain functions. We give the learning curves (solid, dotted, and dash-and-dot lines), as well as the performance of the most effective representation (dashed lines), using the same legend as in Figures 15.9–15.11.

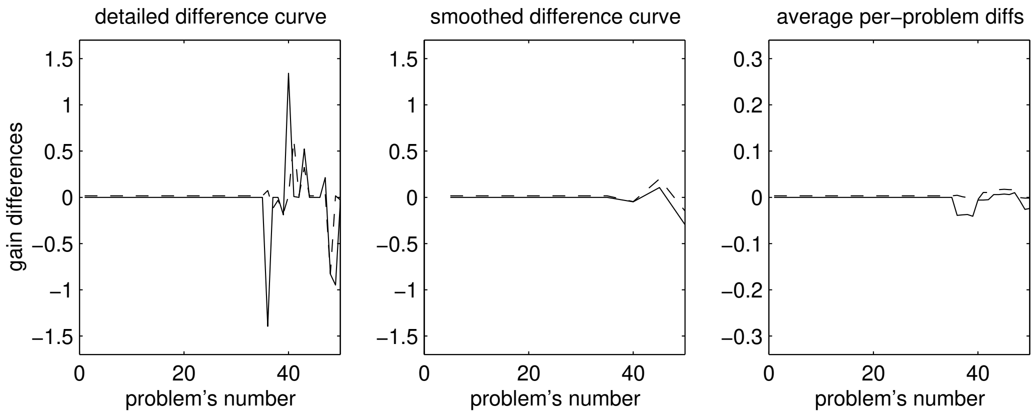
15.3 Different time bounds

Experiments with alternative values of the exploration knob confirmed that the default setting usually ensures near-optimal performance. For each value, we re-ran the tests of Section 15.2 with Functions 15.1, 15.3, and 15.6. We present the difference curves for small knob values in Figures 15.13 and 15.14, similar curves for large values in Figures 15.15 and 15.16, and the cumulative gains in Table 15.4.

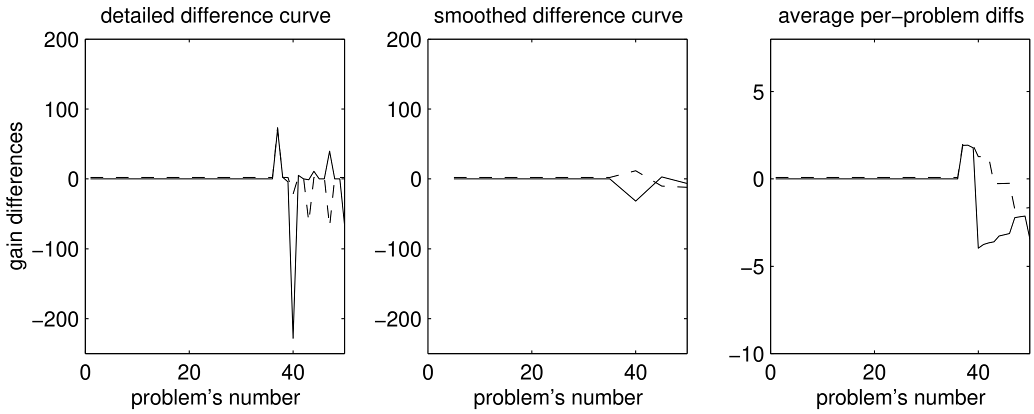
The optimal choice of a knob value depended on a gain function and problem sequence; however, the default setting usually gave satisfactory results. The main exception was the short-sequence test with Function 15.6, when the default gains were almost thrice smaller than the results with other knob settings; however, the absolute difference in gains was *not* significant. Since the total gain was close to zero, this small difference appeared as a large relative improvement over the default behavior.

	small knob values				default	large knob values			
	0.02		0.05		0.1	0.2		0.5	
	short problem sequences								
Function 15.1	−0.338	—	−0.319	—	−0.314	−0.323	—	−0.373	—
Function 15.3	14.3	(80%)	16.1	(90%)	17.8	10.2	(57%)	13.9	(78%)
Function 15.6	0.090	(108%)	0.201	(242%)	0.083	0.221	(266%)	0.233	(281%)
	long problem sequences								
Function 15.1	0.050	(19%)	0.256	(96%)	0.268	0.227	(85%)	0.173	(65%)
Function 15.3	42.0	(91%)	41.5	(90%)	46.2	44.8	(97%)	44.7	(97%)
Function 15.6	0.491	(69%)	0.653	(92%)	0.712	0.802	(113%)	0.918	(129%)

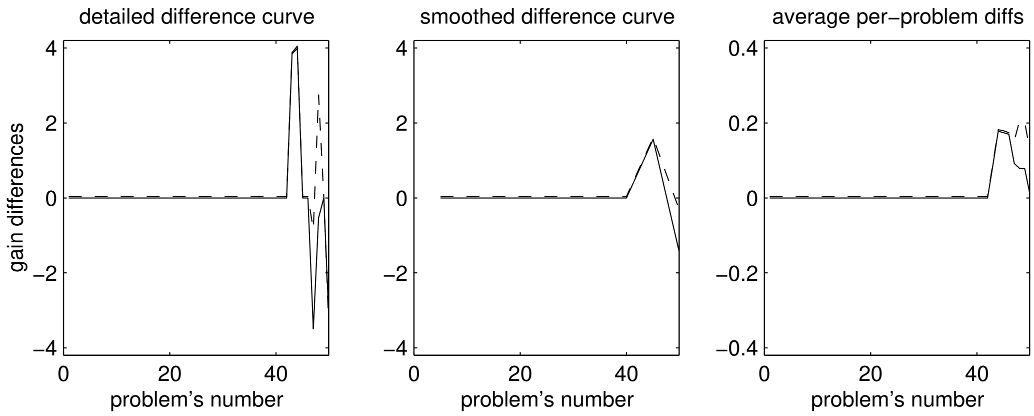
Table 15.4: Summary of tests with five alternative settings of the exploration knob. For each knob value, we list the resulting average gains and the respective percentages of the default-setting gains.



(a) Gain linearly decreases with the running time (Function 15.1).

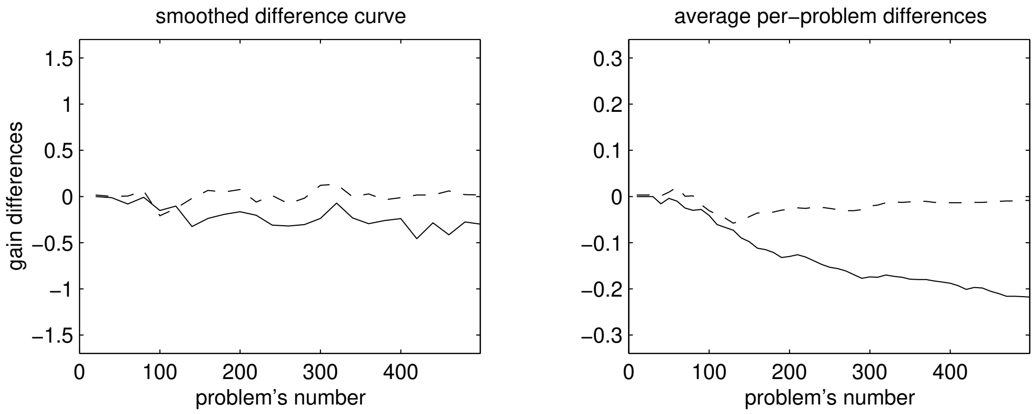


(b) Reward depends on the number of packages (Function 15.3).

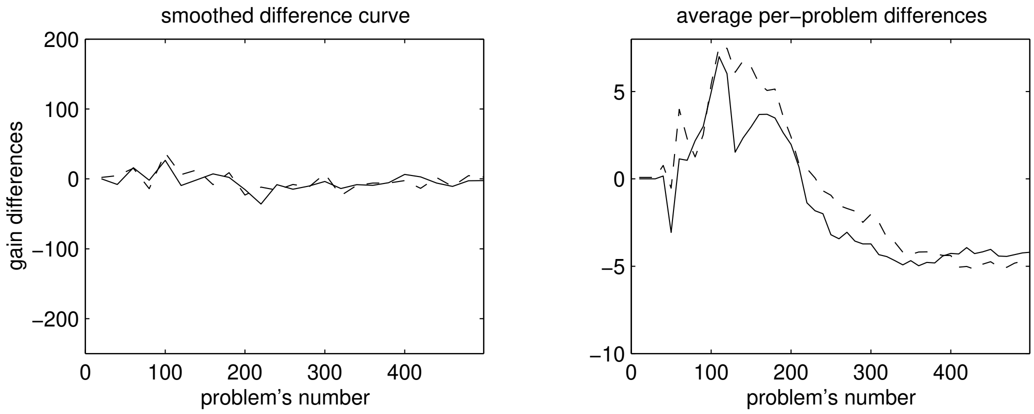


(c) Gain depends on the search time and problem size (Function 15.6).

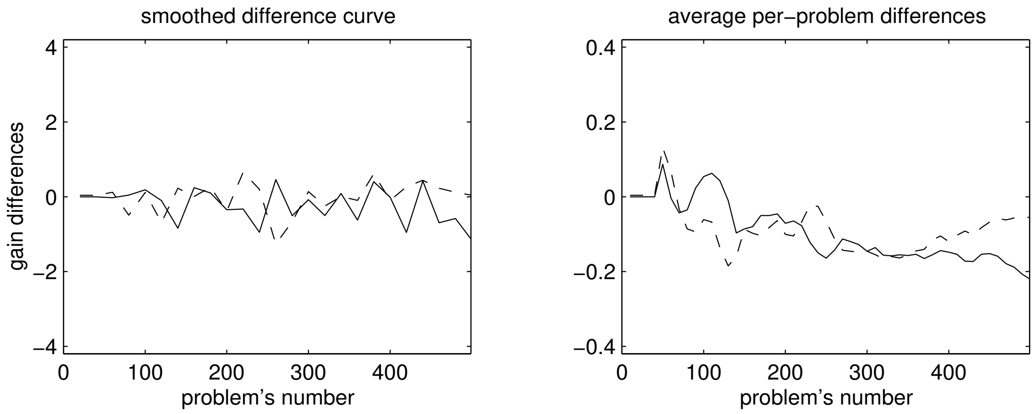
Figure 15.13: Tests with small values of the exploration knob, on fifty-problem sequences. The graphs show the differences between the gains with the knob value 0.02 and that with the value 0.1 (solid lines), as well as the differences between the 0.05-knob and 0.1-knob gains (dashed lines).



(a) Gain linearly decreases with the running time (Function 15.1).

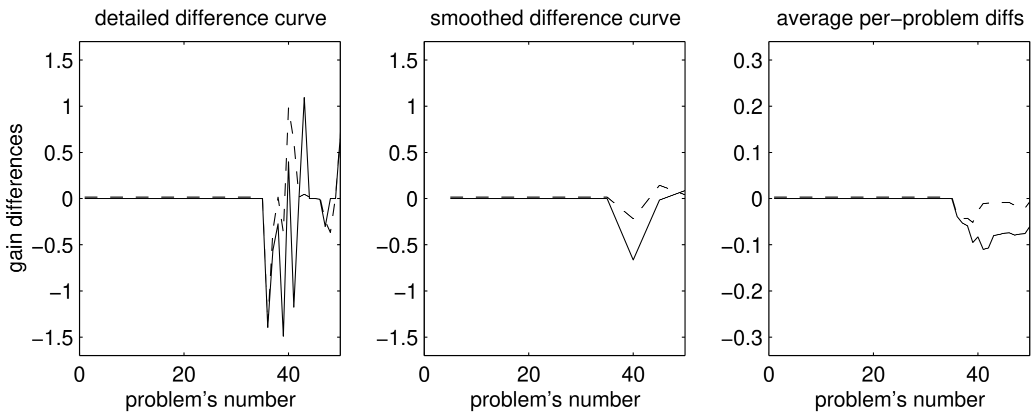


(b) Reward depends on the number of packages (Function 15.3).

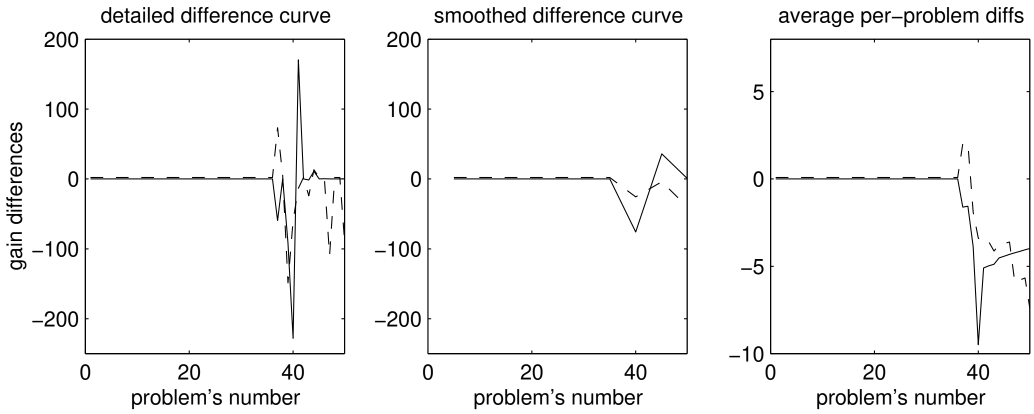


(c) Gain depends on the search time and problem size (Function 15.6).

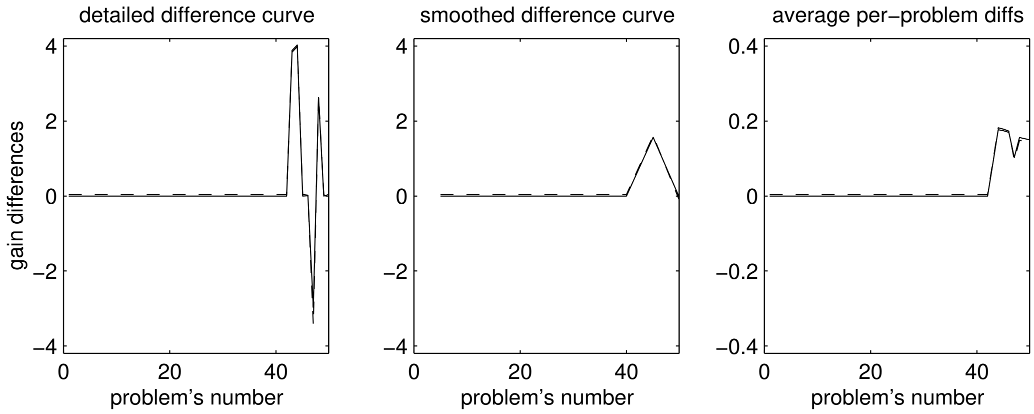
Figure 15.14: Processing 500-problem sequences with the small knob values. We give the difference curves for the knob value 0.02 (solid), and the analogous curves for the value 0.05 (dashes).



(a) Gain linearly decreases with the running time (Function 13.1).

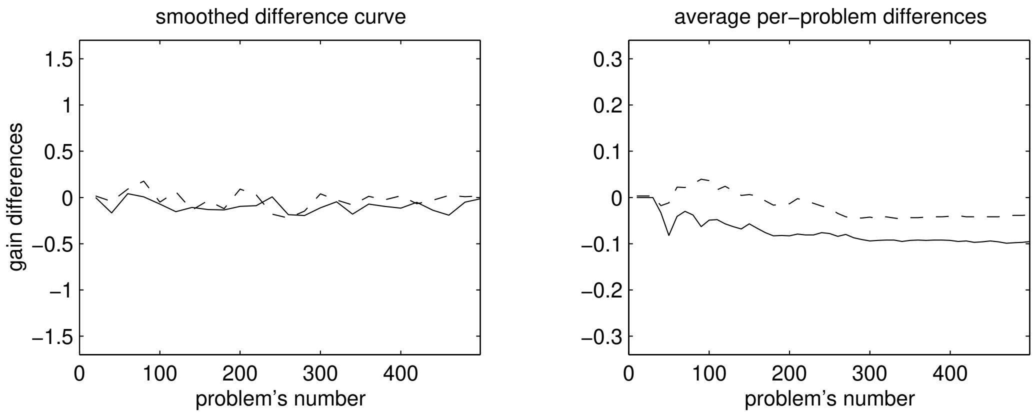


(b) Reward depends on the number of packages (Function 15.3).

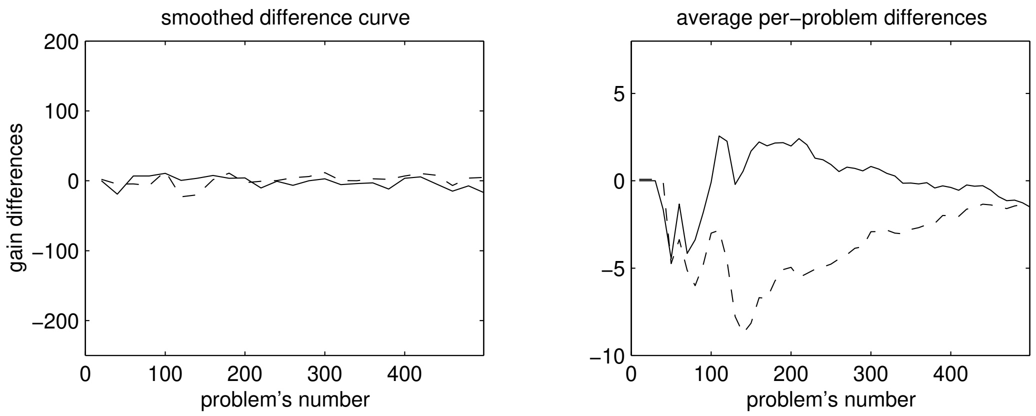


(c) Gain depends on the search time and problem size (Function 15.6).

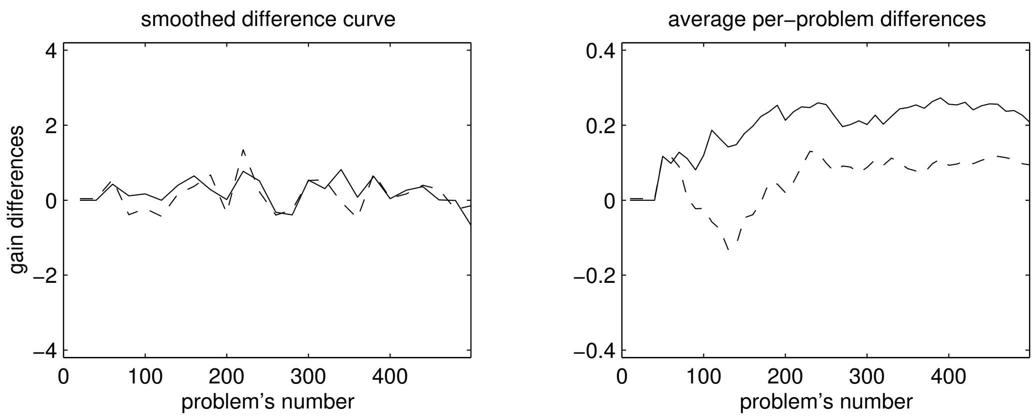
Figure 15.15: Experiments with large knob values, on short sequences of problems. We plot the differences between the 0.5-knob and 0.1-knob gains (solid), as well as the differences between the 0.2-knob and 0.1-knob results (dashes).



(a) Gain linearly decreases with the running time (Function 15.1).



(b) Reward depends on the number of packages (Function 15.2).



(c) Gain depends on the search time and problem size (Function 15.6).

Figure 15.16: Applying $\mathcal{S}$HAPER with the knob values 0.5 (solid) and 0.2 (dashes) to long sequences.

Concluding remarks

A book is never finished, it is only published; therefore, I am solely responsible for all errors and omissions.
— Derick Wood [1993], *Data Structures, Algorithms, and Performance.*

The most important contribution of the reported work is a general-purpose architecture for improving and evaluating representations in AI problem solving. This work has been a combination of theoretical and empirical investigations: it has involved mathematical analysis, design of learning and search algorithms, pilot experiments with alternative heuristics, implementation of the SHAPER system, and empirical evaluation of SHAPER's main components.

The developed system is based on two general principles, proposed by Simon during his investigations of human problem solving:

1. "A representation consists of both data structures and programs operating on them" [Larkin and Simon, 1987]. Thus, a representation change may involve not only modification of the data structures that describe a problem, but also selection of an appropriate search procedure. Finding the right match of a domain description and search algorithm is essential for efficient problem solving.

2. "The same processes that are ordinarily used to search *within* problem space can be used to search *for* problem space (problem representation)." When a system operates with multiple representations, its behavior "could be divided into two segments, alternating between search for a solution in some problem space and search for a new problem space" [Kaplan and Simon, 1990].

We have implemented SHAPER in Allegro Common Lisp and integrated it with the PRODIGY4 system. The overall size of the code is about 60,000 lines, which includes the PRODIGY search algorithms (20,000 lines), description changes (15,000), and control module (25,000).

Architectural decisions

The key architectural decisions underlying SHAPER follow from Simon's two principles. We defined a representation as a combination of a domain description and solver procedure, and subdivided the process of representation change into the generation of a new description and selection of a solver. The system explores the space of different domain representations, using changer algorithms as operators for the expansion of this space. It alternates between the construction of new representations and their application to problem solving.

The evaluation of the available representations is based on a general utility model, which accounts for the percentage of solved problems, efficiency of search, and quality of the resulting solutions. We used this model in developing a statistical technique for automated selection of effective representations. Furthermore, we integrated the statistical procedure with several heuristic mechanisms that guide the search in the representation space.

The work on the $\mathcal{S}$HAPER system has led to the following results, which correspond to the main components of the system:

- Automatic choice and utilization of primary effects in AI problem solving
- Abstraction mechanism for the full PRODIGY language, and its synergy with the algorithms for choosing primary effects
- Tools for the centralized access to problem solvers, changer algorithms, and domain descriptions; these tools form the system's "control center," which maintains the space of representations
- Statistical analysis of the past performance and estimation of expected search time; the estimate procedure accounts for problem sizes and similarity among problems
- Automated exploration of the representation space and selection of effective representations; the exploration module utilizes not only statistical estimates, but also several types of user-specified heuristic rules

Advantages and limitations

The control architecture does not rely on properties of specific problem solvers and description changers. We may use it with any collection of learning and search algorithms that have a common input language and satisfy certain general assumptions, summarized in Section 7.4.1.

On the other hand, the current implementation of this architecture is based on the PRODIGY data structures; hence, we cannot port it to other problem-solving systems without rewriting the code. The construction of a portable, system-independent version of the control mechanism is an important engineering problem.

The generation of new representations involves search at three different levels. The description changers form the base of this three-level hierarchy. Every changer algorithm searches in its own limited space of description improvements, and chooses potentially good descriptions from this space. For example, *Chooser* and *Completer* explore a space of alternative selections of primary effects, *Abstractor* searches for a fine-grained ordered hierarchy, and *Refiner* operates with alternative partial instantiations of predicates.

The second level is the expansion of the global description space. The control module invokes changer algorithms to generate new nodes of this space, and employs heuristic rules to guide the expansion order. Finally, the third level involves selecting solver algorithms and pairing them with appropriate domain descriptions. This hierarchical structure of the

SHAPER system limits the size of description and representation spaces. It prevents the combinatorial explosion in the number of candidate representations, reported by Korf.

The problem-solving power of the developed architecture is limited by the capabilities of the embedded solver and changer algorithms. The control module learns to make the best use of the available algorithms, but it cannot go beyond their potential. For example, SHAPER is unable to learn macro operators or control rules, because it does not include appropriate description changers. Similarly, it cannot outperform the fastest of the available problem solvers.

To our knowledge, the only system without similar limitations is Korf's [1980] universal engine for generating new representations, which systematically expands the huge space of *all* isomorphic and homomorphic transformations of a given problem, and can potentially perform any representation improvement.

The human operator has a number of options for extending the SHAPER system. In particular, she may add new algorithms to the system's library of solvers and changers, define applicability conditions for the available algorithms, hand-code domain descriptions, provide heuristic rules for guiding the search in the representation space, implement procedures for estimating problem difficulty, and supply training problems.

The system supports a wide range of utility functions for evaluating representations, and the user may construct complex gain functions that encode her value judgments. The top-level control procedure searches for representations that maximize the expected gains. The human operator has an option to take part in the search process and guide some of the top-level decisions.

The control module has several knob parameters, which affect the behavior of the statistical learner, utilization of preference rules, decisions in the absence of relevant past data, and resolution of conflicts among different mechanisms for selecting representations. We defined default values of the knob variables, and implemented an interface procedure for inspecting and adjusting the knobs, which enables the human operator to tune the system; however, experiments have shown that SHAPER's behavior is surprisingly insensitive to most knob values, as long as they are within appropriate ranges.

Empirical evaluation

We have evaluated the effectiveness of SHAPER in several different domains, with a wide variety of problems and gain functions. The empirical results have confirmed the feasibility of our approach to changing representations, and demonstrated the main properties of the developed algorithms:

- Primary effects and ordered abstraction are powerful tools for reducing search, but their impact varies across domains, ranging from a thousand-fold speed-up to a substantial slow-down; thus, the system's performance depends on the choice among the implemented search-reduction tools

- The statistical learner always chooses an optimal or near-optimal combination of a domain description, solver algorithm, and time bound; the effectiveness of the learning procedure does *not* depend on the properties of specific solvers, domains, or utility functions

- Control heuristics enhance the system's performance in the initial stages of statistical learning, when the accumulated performance data are insufficient for an accurate selection of representations

The $\mathcal{S}$HAPER system is *less* effective than human problem solvers in most large-scale domains; it outperforms people only on some puzzles, such as Tower of Hanoi. This limitation is due to the state of the art: general-purpose search systems do not scale to complex reasoning tasks, and PRODIGY is no exception. Since $\mathcal{S}$HAPER is limited by the capabilities of PRODIGY search algorithms, it cannot compete with human subjects.

On the other hand, a separate evaluation of the control module has shown that it is *more* effective than human experts. In particular, we have compared the system's selections of algorithms and time bounds to manual choices by PRODIGY researchers. The automated control has led to greater cumulative gains, especially for complex utility functions.

The evaluation of the control architecture has two major limitations, which restrict the generality of the empirical results. First, we have tested this architecture only in the PRODIGY system. The control techniques have proved insensitive to the behavior of particular PRODIGY representations, and we hypothesize that they will work equally well with other AI systems. Preliminary support for this hypothesis comes from the controlled experiments with artificially generated values of problem-solving times. The artificial tests have demonstrated that the statistical learner is effective for a wide range of search-time distributions.

Second, we have constructed only a small library of solvers and changers; hence, $\mathcal{S}$HAPER usually expands a small representation space. The number of candidate representations in the described experiments has varied from two to thirty-six. Thus, the empirical evaluation has provided little data on the scalability of the control techniques.

Applying the developed techniques to a larger library of AI algorithms is an important research direction. In particular, we intend to test $\mathcal{S}$HAPER's control module in the Multiagent Planning Architecture [Wilkins and Myers, 1998], which supports centralized access to a large collection of diverse search engines.

Future challenges

The described results is one of the first steps in exploring two broad subareas of artificial intelligence: (1) automated improvement of problem representations and (2) coordination of multiple AI algorithms. We have shown a close relationship between these subareas and proposed an approach to addressing them; however, this work is only a beginning, which leaves many unanswered questions.

The main open problems include implementing a portable version of the control architecture, improving and extending control mechanisms, and building an extensive library of learning and search engines, which may include both general and domain-specific algorithms. The system's performance depends on the availability of appropriate algorithms; in particular, a large library of diverse description changers is essential for the automatic adaptation to new domains.

Note that we have *not* tried to construct a universal changer engine, which would explore all types of feasible description improvements. We conjecture that search for effective representations is an inherently complex problem, which cannot be solved by a single all-purpose algorithm.

A grand challenge is to build a software architecture that integrates thousands of AI engines and domain descriptions, much in the same way as an operating system integrates file-processing programs. It must provide standard protocols for the communication among learning and search procedures, and support routine inclusion of new domains, AI algorithms, control techniques, and interface tools.

Wilkins and Myers [1998] have recently made a major step toward addressing this grand problem. They have built a software architecture that supports the synergetic use of diverse search procedures, collected from different AI systems. On the negative side, their architecture does not allow inclusion of description changers and has no general-purpose control mechanisms. Thus, the human operator must optimize the domain descriptions and implement specialized control procedures. We plan to integrate this architecture with Shaper's control algorithms, which may be another important step toward a large-scale system for improving representations.

Other major challenges include developing a unified theory of search with multiple representations, automating the synthesis of specialized description-improving procedures, and investigating the representation changes performed by human problem solvers.

We hypothesize that the cognitive architecture for constructing new representations is similar to the Shaper system; that is, the human mind includes solver and changer algorithms, and techniques for choosing among them. We also conjecture that the human problem solver employs a wide variety of highly specialized changer algorithms, arranged by areas of knowledge, as well as some more general algorithms. The related studies by Simon and his colleagues provide preliminary support for these conjectures.

Bibliography

[Aho *et al.*, 1974] Alfred A. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishers, Reading, MA, 1974.

[Allen and Minton, 1996] John A. Allen and Steven Minton. Selecting the right heuristic algorithm: Runtime performance predictors. In Gordon McCalla, editor, *Advances in Artificial Intelligence: The Eleventh Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 41–53, Berlin, Germany, 1996. Springer.

[Allen *et al.*, 1992] John A. Allen, Pat Langley, and Stan Matwin. Knowledge and regularity in planning. In *Proceedings of the AAAI 1992 Spring Symposium on Computational Considerations in Supporting Incremental Modification and Reuse*, pages 7–12, 1992.

[Amarel, 1961] Saul Amarel. An approach to automatic theory formation. In Heinz M. Von Foerster, editor, *Principles of Self-Organiation: Transactions.* Pergamon Press, New York, NY, 1961.

[Amarel, 1965] Saul Amarel. Problem solving procedures for efficient syntactic analysis. In *ACM Twentieth National Conference*, 1965.

[Amarel, 1968] Saul Amarel. On representations of problems of reasoning about actions. In Donald Michie, editor, *Machine Intelligence 3*, pages 131–171. American Elsevier Publishers, New York, NY, 1968.

[Amarel, 1971] Saul Amarel. Representations and modeling in problems of program formation. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 6*, pages 411–466. American Elsevier Publishers, New York, NY, 1971.

[Anthony and Biggs, 1992] Martin Anthony and Norman Biggs. *Computational Learning Theory.* Cambridge University Press, 1992.

[Bacchus and Yang, 1991] Fahiem Bacchus and Qiang Yang. The downward refinement property. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 286–291, 1991.

[Bacchus and Yang, 1992] Fahiem Bacchus and Qiang Yang. The expected value of hierarchical problem-solving. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992.

[Bacchus and Yang, 1994] Fahiem Bacchus and Qiang Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 71(1):43–100, 1994.

[Bäckström and Jonsson, 1995] Christer Bäckström and Peter Jonsson. Planning with abstraction hierarchies can be exponentially less efficient. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1599–1604, 1995.

[Barrett and Weld, 1994] Anthony Barrett and Dan Weld. Partial order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67(1):71–112, 1994.

[Blum and Furst, 1997] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):281–300, 1997.

[Blumer *et al.*, 1987] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Occam's razor. *Information Processing Letters*, 24:377–380, 1987.

[Blythe and Reilly, 1993a] Jim Blythe and W. Scott Reilly. Integrating reactive and deliberative planning in a household robot. In *AAAI Fall Symposium on Instantiating Real-World Agents*, 1993.

[Blythe and Reilly, 1993b] Jim Blythe and W. Scott Reilly. Integrating reactive and deliberative planning for agents. Technical Report CMU-CS-93-155, School of Computer Science, Carnegie Mellon University, 1993.

[Blythe and Veloso, 1992] Jim Blythe and Manuela M. Veloso. An analysis of search techniques for a totally-ordered nonlinear planner. In *Proceedings of the First International Conference on AI Planning Systems*, pages 13–19, 1992.

[Boehm-Davis *et al.*, 1989] D. A. Boehm-Davis, R. W. Holt, M. Koll, G. Yastrop, and R. Peters. Effects of different database formats on information retrieval. *Human Factors*, 31:579–592, 1989.

[Borrajo and Veloso, 1996] Daniel Borrajo and Manuela Veloso. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *Artificial Initelligence Review*, 10:1–34, 1996.

[Breese and Horvitz, 1990] John S. Breese and Eric J. Horvitz. Ideal reformulation of belief networks. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence*, pages 64–72, 1990.

[Carbonell and Gil, 1990] Jaime G. Carbonell and Yolanda Gil. Learning by experimentation: The operator refinement method. In R. S. Michalski and Y. Kodratoff, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 191–213. Morgan Kaufmann, Palo Alto, CA, 1990.

[Carbonell *et al.*, 1990] Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton. PRODIGY: An integrated architecture for planning and learning. In Kurt VanLehn, editor, *Architectures for Intelligence*. Erlbaum, Hillside, NJ, 1990.

[Carbonell *et al.*, 1992] Jaime G. Carbonell, Jim Blythe, Oren Etzioni, Yolanda Gil, Robert Joseph, Dan Kahn, Craig A. Knoblock, Steven Minton, Alicia Pérez, Scott Reilly, Manuela M. Veloso, and Xuemei Wang. PRODIGY4.0: The manual and tutorial. Technical Report CMU-CS-92-150, School of Computer Science, Carnegie Mellon University, 1992.

[Carbonell, 1983] Jaime G. Carbonell. Learning by analogy: Formulating and generalizing plans from past experience. In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*. Tioga Publishers, Palo Alto, CA, 1983.

[Carbonell, 1990] Jaime G. Carbonell, editor. *Machine Learning: Paradigms and Methods*. MIT Press, Boston, MA, 1990.

[Carre, 1971] B. A. Carre. An algebra for network routing problems. *Journal of the Institute of Mathematics and Its Applications*, 7:273–294, 1971.

[Chapman, 1987] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[Cheng and Carbonell, 1986] Patricia W. Cheng and Jaime G. Carbonell. The FERMI system: Inducing iterative macro-operators from experience. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 490–495, 1986.

[Christensen, 1990] Jens Christensen. A hierarchical planner that generates its own abstraction hierarchies. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 1004–1009, 1990.

[Cohen *et al.*, 1994] William W. Cohen, Russell Greiner, and Dale Schuurmans. Probabilistic hill-climbing. In S.J. Hanson, T. Petsche, M. Kearns, and R.L. Rivest, editors, *Computational Learning Theory and Natural Learning Systems*, volume II, pages 171–181. MIT Press, Boston, MA, 1994.

[Cohen, 1992] William W. Cohen. Using distribution-free learning theory to analyze solution-path caching mechanisms. *Computational Intelligence*, 8(2):336–375, 1992.

[Cohen, 1995] Paul R. Cohen. *Empirical Methods for Artificial Intelligence*. MIT Press, Cambridge, MA, 1995.

[Cormen *et al.*, 1990] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[Cox and Veloso, 1997a] Michael T. Cox and Manuela M. Veloso. Controlling for unexpected goals when planning in a mixed-initiative setting. In E. Costa and A. Cardoso, editors, *Progress in Artificial Intelligence: Eighth Portuguese Conference on Artificial Intelligence*, pages 309–318. Springer-Verlag, Berlin, Germany, 1997.

[Cox and Veloso, 1997b] Michael T. Cox and Manuela M. Veloso. Supporting combined human and machine planning: An interface for planning by analogical reasoning. In D. B. Leake and E. Plaza, editors, *Case-Based Reasoning Research and Development: Second International Conference on Case-Based Reasoning*, pages 531–540. Springer-Verlag, Berlin, Germany, 1997.

[Drastal *et al.*, 1994 1997] George A. Drastal, Russell Greiner, Stephen J. Hanson, Michael Kearns, Thomas Petsche, Ronald L. Rivest, and Jude W. Shavlik, editors. *Computational Learning Theory and Natural Learning Systems*, volume I–IV. MIT Press, Boston, MA, 1994–1997.

[Driskill and Carbonell, 1996] Robert Driskill and Jaime G. Carbonell. Search control in problem solving: A gapped macro operator approach. Unpublished Manuscript, 1996.

[Duncker, 1945] K. Duncker. On problem solving. *Psychological Monographs*, 58:1–113, 1945.

[Ellman and Giunchiglia, 1998] Tom Ellman and Fausto Giunchiglia, editors. *Proceedings of the Symposium of Abstraction, Reformulation and Approximation*, 1998.

[Ernst and Goldstein, 1982] George M. Ernst and Michael M. Goldstein. Mechanical discovery of classes of problem-solving strategies. *Journal of the American Association for Computing Machinery*, 29(1):1–23, 1982.

[Erol *et al.*, 1994] Kutluhan Erol, James Handler, and Dana S. Nau. Htn planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1123–1128, 1994.

[Etzioni and Minton, 1992] Oren Etzioni and Steven Minton. Why EBL produces overly-specific knowledge: A critique of the PRODIGY approaches. In *Proceedings of the Ninth International Workshop on Machine Learning*, pages 137–143, 1992.

[Etzioni, 1990] Oren Etzioni. *A Structural Theory of Explanation-Based Learning.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1990. Technical Report CMU-CS-90-185.

[Etzioni, 1992] Oren Etzioni. An asymptotic analysis of speedup learning. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 137–143, 1992.

[Etzioni, 1993] Oren Etzioni. Acquiring search control knowledge via static analysis. *Artificial Intelligence*, 62(2):255–301, 1993.

[Fikes and Nilsson, 1971] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Fikes and Nilsson, 1993] Richard E. Fikes and Nils J. Nilsson. STRIPS, a retrospective. *Artificial Intelligence*, 2:227–232, 1993.

[Fikes *et al.*, 1972] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288, 1972.

[Fink and Blythe, 1998] Eugene Fink and Jim Blythe. A complete bidirectional planner. In *Proceedings of the Fourth International Conference on AI Planning Systems*, pages 78–84, 1998.

[Fink and Veloso, 1996] Eugene Fink and Manuela M. Veloso. Formalizing the PRODIGY planning algorithm. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 261–271. IOS Press, Amsterdam, Netherlands, 1996.

[Fink and Yang, 1992a] Eugene Fink and Qiang Yang. Automatically abstracting effects of operators. In *Proceedings of the First International Conference on AI Planning Systems*, pages 243–251, 1992.

[Fink and Yang, 1992b] Eugene Fink and Qiang Yang. Formalizing plan justifications. In *Proceedings of the Ninth Conference of the Canadian Society for Computational Studies of Intelligence*, pages 9–14, 1992.

[Fink and Yang, 1993] Eugene Fink and Qiang Yang. Forbidding preconditions and ordered abstraction hierarchies. In *Proceedings of the AAAI 1993 Spring Symposium on Foundations of Automatic Planning*, pages 34–38, 1993.

[Fink and Yang, 1995] Eugene Fink and Qiang Yang. Planning with primary effects: Experiments and analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1606–1611, 1995.

[Fink and Yang, 1997] Eugene Fink and Qiang Yang. Automatically selecting and using primary effects in planning: Theory and experiments. *Artificial Intelligence*, 89:285–315, 1997.

[Foulser *et al.*, 1992] D. E. Foulser, Ming Li, and Qiang Yang. Theory and algorithms for plan merging. *Artificial Intelligence*, 57:143–182, 1992.

[Gentner and Stevens, 1983] Dedre Gentner and Albert L. Stevens, editors. *Mental Models*, Hillside, NJ, 1983. Lawrence Erlbaum Associates.

[Gil and Pérez, 1994] Yolanda Gil and Alicia Pérez. Applying a general-purpose planning and learning architectures to process planning. In *Proceedigs of the AAAI 1994 Fall Symposium on Planning and Learning*, pages 48–52, 1994.

[Gil, 1991] Yolanda Gil. A specification of process planning for PRODIGY. Technical Report CMU-CS-91-179, School of Computer Science, Carnegie Mellon University, 1991.

[Gil, 1992] Yolanda Gil. *Acquiring Domain Knowledge for Planning by Experimentation.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1992. Technical Report CMU-CS-92-175.

[Giunchiglia and Walsh, 1992] Fausto Giunchiglia and Toby Walsh. A theory of abstraction. *Artificial Intelligence*, 57:323–389, 1992.

[Golding *et al.*, 1987] Andrew G. Golding, Paul S. Rosenbloom, and John E. Laird. Learning general search control from outside guidance. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 334–337, 1987.

[Goldstein, 1978] Michael M. Goldstein. *The Mechanical Discovery of Problem-Solving Strategies*. PhD thesis, Computer Engineering Department, Case Western Reserve University, 1978.

[Ha and Haddawy, 1997] Vu Ha and Peter Haddawy. Problem-focused incremental elicitation of multi-attribute utility models. In *Proceedings of the Thriteenth Conference on Uncertainty in Artificial Intelligence*, pages 215–222, 1997.

[Haigh and Veloso, 1996] Karen Zita Haigh and Manuela M. Veloso. Interleaving planning and robot execution for asynchronous user requests. In *Proceedings of the International Conference on Intelligent Robots and Systems*, 1996.

[Haigh and Veloso, 1997a] Karen Zita Haigh and Manuela M. Veloso. High-level planning and low-level execution: Towards a complete robotic agent. *Autonomous Agents*, 1997.

[Haigh and Veloso, 1997b] Karen Zita Haigh and Manuela M. Veloso. Interleaving planning and robot execution for asynchronous user requests. *Autonomous Robots*, 5(1):79–95, 1997.

[Haigh and Veloso, 1998] Karen Zita Haigh and Manuela M. Veloso. Planning, execution and learning in a robotic agent. In *Proceedings of the Fourth International Conference on AI Planning Systems*, pages 120–127, 1998.

[Haigh, 1998] Karen Zita Haigh. *Situation-Dependent Learning for Interleaved Planning and Robot Execution*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1998. Technical Report CMU-CS-98-108.

[Hall, 1987] Rogers P. Hall. Understanding analogical reasoning: Computational approaches. *Artificial Intelligence*, 39:39–120, 1987.

[Hansen and Zilberstein, 1996] Eric A. Hansen and Shlomo Zilberstein. Monitoring the progress of anytime problem-solving. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 1229–1234, 1996.

[Hansson and Mayer, 1989] Othar Hansson and Andrew Mayer. Heuristic search and evidential reasoning. In *Proceedings of the Fifth Workshop on Uncertainty in Artificial Intelligence*, pages 152–161, 1989.

[Haussler, 1988] David Haussler. Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, 36:177–221, 1988.

[Hayes and Simon, 1974] John R. Hayes and Herbert A. Simon. Understanding written problem instructions. In L. W. Gregg, editor, *Knowledge and Cognition*, pages 167–200. Lawrence Erlbaum Associates, Potomac, MD, 1974.

[Hayes and Simon, 1976] John R. Hayes and Herbert A. Simon. The understanding process: Problem isomorphs. *Cognitive Psychology*, 8:165–190, 1976.

[Hayes and Simon, 1977] John R. Hayes and Herbert A. Simon. Psychological difference among problem isomorphs. In N. J. Castellan, D. B. Pisoni, and G. R. Potts, editors, *Cognitive Theory*. Lawrence Erlbaum Associates, Hillside, NJ, 1977.

[Hibler, 1994] David Hibler. Implicit abstraction by thought experiments. In *Proceedings of the Workshop on Theory Reformulation and Abstraction*, pages 9–26, 1994.

[Holte *et al.*, 1994] Robert C. Holte, C. Drummond, M. B. Perez, Robert M. Zimmer, and Alan J. MacDonald. Searching with abstractions: A unifying framework and new high-performance algorithm. In *Proceedings of the Tenth Conference of the Canadian Society for Computational Studies of Intelligence*, pages 263–270, 1994.

[Holte *et al.*, 1996a] Robert C. Holte, T. Mkadmi, Robert M. Zimmer, and Alan J. MacDonald. Speeding up problem solving by abstraction: A graph-oriented approach. *Artificial Intelligence*, 85:321–361, 1996.

[Holte *et al.*, 1996b] Robert C. Holte, M. B. Perez, Robert M. Zimmer, and Alan J. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 530–535, 1996.

[Holte, 1988] Robert C. Holte. *An Analytical Framework for Learning Systems.* PhD thesis, Artificial Intelligence Laboratory, University of Texas at Austin, 1988. Technical Report AI88-72.

[Horvitz, 1988] Eric J. Horvitz. Reasoning under varying and uncertain resource constraints. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 111–116, 1988.

[Hull, 1997] John C. Hull. *Options, Futures, and Other Derivatives Securities.* Prentice Hall, Upper Saddle River, NJ, third edition, 1997.

[Jones and Schkade, 1995] Donald R. Jones and David A. Schkade. Choosing and translating between problem representations. *Organizational Behavior and Human Decision Processes*, 61(2):214–223, 1995.

[Joseph, 1992] Robert L. Joseph. *Graphical Knowledge Acquisition for Visually-Oriented Domains.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1992. Technical Report CMU-CS-92-188.

[Junghanns and Schaeffer, 1992] Andreas Junghanns and Jonathan Schaeffer. Sokoban: Improving the search with relevance cuts. *Journal of Theoretical Computing Science*, 1992. To appear.

[Junghanns and Schaeffer, 1998] Andreas Junghanns and Jonathan Schaeffer. Single-agent search in the presence of deadlocks. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 419–424, 1998.

[Junghanns and Schaeffer, 1999] Andreas Junghanns and Jonathan Schaeffer. Domain-dependent single-agent search enhancements. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, 1999.

[Kambhampati and Srivastava, 1996a] Subbarao Kambhampati and Biplav Srivastava. Unifying classical planning approaches. Technical Report 96-006, Deptartment of Computer Science, Arizona State University, 1996.

[Kambhampati and Srivastava, 1996b] Subbarao Kambhampati and Biplav Srivastava. Universal classical planner: An algorithm for unifying state-space and plan-space planning. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 261–271. IOS Press, Amsterdam, Netherlands, 1996.

[Kaplan and Simon, 1990] Craig A. Kaplan and Herbert A. Simon. In search of insight. *Cognitive Psychology*, 22:374–419, 1990.

[Kaplan, 1989] Craig A. Kaplan. SWITCH: A simulation of representational change in the Mutilated Checkerboard problem. Technical Report C.I.P. 477, Department of Psychology, Carnegie Mellon University, 1989.

[Knoblock and Yang, 1994] Craig A. Knoblock and Qiang Yang. Evaluating the trade-offs in partial-order planning algorithms. In *Proceedings of the Tenth Conference of the Canadian Society for Computational Studies of Intelligence*, pages 279–286, 1994.

[Knoblock and Yang, 1995] Craig A. Knoblock and Qiang Yang. Relating the performance of partial-order planning algorithms to domain features. *SIGART Bulletin*, 6(1), 1995.

[Knoblock *et al.*, 1991a] Craig A. Knoblock, Steven Minton, and Oren Etzioni. Integrating abstraction and explanation-based learning in PRODIGY. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 541–546, 1991.

[Knoblock *et al.*, 1991b] Craig A. Knoblock, Josh Tenenberg, and Qiang Yang. Characterizing abstraction hierarchies for planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 692–697, 1991.

[Knoblock, 1990] Craig A. Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 923–928, 1990.

[Knoblock, 1991] Craig A. Knoblock. Search reduction in hierarchical problem solving. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 686–691, 1991.

[Knoblock, 1992] Craig A. Knoblock. An analysis of ABSTRIPS. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, 1992.

[Knoblock, 1993] Craig A. Knoblock. *Generating Abstraction Hierarchies: An Automated Approach to Reducing Search in Planning.* Kluwer Academic Publishers, Boston, MA, 1993.

[Knoblock, 1994] Craig A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68:243–302, 1994.

[Koenig, 1997] Sven Koenig. *Goal-Directed Acting with Incomplete Information.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1997. Technical Report CMU-CS-97-199.

[Kolodner, 1984] Janet L. Kolodner. *Retrieval and Organization Strategies in Conceptual Memory: A Computer Model.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1984.

[Kook and Novak, 1991] Hyung Joon Kook and Gordon S. Novak. Representation of models for expert problem solving in physics. *IEEE Transactions on Software Engineering*, 3(1):48–54, 1991.

[Korf, 1980] Richard E. Korf. Toward a model of representation changes. *Artificial Intelligence*, 14:41–78, 1980.

[Korf, 1985a] Richard E. Korf. *Learning to Solve Problems by Searching for Macro-Operators.* Putnam Publishing Inc., Boston, MA, 1985.

[Korf, 1985b] Richard E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.

[Korf, 1987] Richard E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33:65–88, 1987.

[Kuokka, 1990] Daniel R. Kuokka. *The Deliberative Integration of Planning, Execution, and Learning.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1990. Technical Report CMU-CS-90-135.

[Laird *et al.*, 1986] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[Laird *et al.*, 1987] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[Langley, 1983] Pat Langley. Learning effective search heuristics. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 419–421, 1983.

[Larkin and Simon, 1981] Jill Larkin and Herbert A. Simon. Learning through growth of skill in mental modeling. In *Proceedings of the Third Annual Conference of the Cognitive Science Society*, 1981.

[Larkin and Simon, 1987] Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11:65–99, 1987.

[Larkin *et al.*, 1988] Jill H. Larkin, Frederick Reif, Jaime G. Carbonell, and Angela Gugliotta. FERMI: A flexible expert reasoner with multi-domain inferencing. *Cognitive Psychology*, 12:101–138, 1988.

[Lehmann, 1977] D. J. Lehmann. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4:59–76, 1977.

[Levy and Nayak, 1995] Alon Y. Levy and P. Pandurang Nayak, editors. *Proceedings of the Symposium of Abstraction, Reformulation and Approximation*, 1995.

[Lowry, 1992] Michael R. Lowry, editor. *Proceedings of the Workshop on Change of Representation and Problem Reformulation*. NASA Ames Research Center, 1992. Technical Report FIA-92-06.

[Van Baalen, 1989] Jeffrey Van Baalen. *Toward a Theory of Representation Design*. PhD thesis, Artificial Intelligence Laboratory, Masschusetts Institute of Technology, 1989. Technical Report 1128.

[Van Baalen, 1994] Jeffrey Van Baalen, editor. *Proceedings of the Workshop on Theory Reformulation and Abstraction*. Computer Science Department, University of Wyoming, 1994. Technical Report 123.

[McAllester and Rosenblitt, 1991] David A. McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 634–639, 1991.

[Mendenhall, 1987] William Mendenhall. *Introduction to Probability and Statistics*. Duxbury Press, Boston, MA, seventh edition, 1987.

[Minton *et al.*, 1989a] Steven Minton, Jaime G. Carbonell, Craig A. Knoblock, Dan R. Kuokka, Oren Etzioni, and Yolanda Gil. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence*, 40:63–118, 1989.

[Minton *et al.*, 1989b] Steven Minton, Dan R. Kuokka, Yolanda Gil, Robert L. Joseph, and Jaime G. Carbonell. PRODIGY2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, 1989.

[Minton *et al.*, 1991] Steven Minton, John Bresina, and Mark Drummond. Commitment strategies in planning: A comparative analysis. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 259–261, 1991.

[Minton *et al.*, 1994] Steven Minton, John Bresina, and Mark Drummond. Total-order and partial-order planning: A comparative analysis. *Journal of Artificial Intelligence Research*, 2:227–262, 1994.

[Minton, 1985] Steven Minton. Selectively generalizing plans for problem-solving. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 596–599, 1985.

[Minton, 1988] Steven Minton. *Learning Search Control Knowledge: An Explanation-Based Approach.* Kluwer Academic Publishers, Boston, MA, 1988.

[Minton, 1990] Steven Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence Journal*, 42:363–391, 1990.

[Minton, 1993a] Steven Minton. An analytical learning system for specialized heuristics. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, 1993.

[Minton, 1993b] Steven Minton. Integrating heuristics for constraint satisfaction problems: A case study. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 1993.

[Minton, 1996] Steven Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints: An International Journal*, 1:7–43, 1996.

[Mitchell *et al.*, 1983] Tom M. Mitchell, Paul E. Utgoff, and Ranan B. Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristics. In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 163–190. Tioga Publishers, Palo Alto, CA, 1983.

[Mooney, 1988] Raymond J. Mooney. Generalizing the order of operators in macro-operators. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 270–283, San Mateo, CA, 1988. Morgan Kaufmann.

[Mouaddib and Zilberstein, 1995] Abdelillah Mouaddib and Shlomo Zilberstein. Knowledge-based anytime computation. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 775–781, 1995.

[Mühlpfordt and Schmid, 1998] Martin Mühlpfordt and Ute Schmid. Synthesis of recursive functions with interdependent parameters. In *Proceedings of the Workshop on Applied Learning Theory*, pages 132–139, Kaiserslautern, Germany, 1998.

[Natarajan, 1991] Balas K. Natarajan. *Machine Learning: A Theoretical Approach.* Morgan Kaufmann, San Mateo, CA, 1991.

[Newell and Simon, 1961] Allen Newell and Herbert A. Simon. GPS, a program that simulates human thought. In H. Billing, editor, *Lernende Automaten*, pages 109–124. R. Oldenbourg, Munich, Germany, 1961.

[Newell and Simon, 1972] Allen Newell and Herbert A. Simon. *Human Problem Solving.* Prentice Hall, Englewood Cliffs, NJ, 1972.

[Newell *et al.*, 1960] Allen Newell, J. C. Shaw, and Herbert A. Simon. A variety of intelligent learning in a general problem solver. In Marshall C. Yovits, editor, *International Tracts in Computer Science and Technology and Their Applications*, volume 2: Self-Organizing Systems, pages 153–189. Pergamon Press, New York, NY, 1960.

[Newell, 1965] Allen Newell. Limitations of the current stock of ideas about problem solving. In A. Kent and O. Tualbee, editors, *Electronic Information Handling*. Spartan Books, Washington, DC, 1965.

[Newell, 1966] Allen Newell. On the representations of problems. In *Computer Science Research Reviews*. Carnegie Institute of Technology, Pittsburgh, PA, 1966.

[Newell, 1992] Allen Newell. Unified theories of cognition and the role of Soar. In J. A. Michon and A. Akyürek, editors, *Soar: A Cognitive Architecture in Perspective*, pages 25–79. Kluwer Academic Publishers, Netherlands, 1992.

[Nilsson, 1971] Nils J. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, New York, NY, 1971.

[Nilsson, 1980] Nils J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA, 1980.

[Novak, 1995] Gordon S. Novak. Creation of views for reuse of software with different data representations. *IEEE Transactions on Software Engineering*, 21(12):993–1005, 1995.

[Ohlsson, 1984] S. Ohlsson. Restructuring revisited i: Summary and critique of the gestalt theory of problem solving. *Scandinavian Journal of Psychology*, 25:65–78, 1984.

[Paige and Simon, 1966] Jeffery M. Paige and Herbert A. Simon. Cognitive processes in solving algebra word problems. In B. Kleinmuntz, editor, *Problem Solving*. John Wiley & Sons, New York, NY, 1966.

[Pednault, 1988a] Edwin P. D. Pednault. Extending conventional planning tecnhiques to handle actions with context-dependent effects. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 55–59, 1988.

[Pednault, 1988b] Edwin P. D. Pednault. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence*, 4:356–372, 1988.

[Penberthy and Weld, 1992] J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial-order planner for ADL. In *Proceedings of the Third International Conference on Knowledge Representation in Reasoning*, pages 103–114, 1992.

[Peot and Smith, 1993] Mark A. Peot and David E. Smith. Threat-removal strategies for partial-order planning. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 492–499, 1993.

[Pérez and Carbonell, 1993] M. Alicia Pérez and Jaime G. Carbonell. Automated acquisition of control knowledge to improve the quality of plans. Technical Report Technical Report CMU-CS-93-142, School of Computer Science, Carnegie Mellon University, 1993.

[Pérez and Etzioni, 1992] M. Alicia Pérez and Oren Etzioni. DYNAMIC: A new role for training problems in EBL. In D. Sleeman and P. Edwards, editors, *Proceedings of the Ninth International Conference on Machine Learning*, San Mateo, CA, 1992. Morgan Kaufmann.

[Pérez, 1995] M. Alicia Pérez. *Learning Search Control Knowledge to Improve Plan Quality.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1995. Technical Report CMU-CS-95-175.

[Peterson, 1994] Donald Peterson. Re-representation and emergent information in three cases of problem solving. In *Artificial Intelligence and Creativity*, pages 81–92. Kluwer Academic Publishers, Dordrecht, Netherlands, 1994.

[Peterson, 1996] Donald Peterson, editor. *Forms of Representation*, Exeter, United Kingdom, 1996. Intellect Books.

[Pohl, 1971] Ira Pohl. Bi-directional search. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 6*, pages 127–140. American Elsevier Publishers, New York, NY, 1971.

[Polya, 1957] George Polya. *How to Solve It.* Doubleday, Garden City, NY, second edition, 1957.

[Qin and Simon, 1992] Yulin Qin and Herbert A. Simon. Imagery and mental models in problem solving. In N. Hari Narayanan, editor, *Proceedings of the AAAI 1992 Spring Symposium on Reasoning with Diagrammatic Representations*, Palo Alto, CA, 1992. Stanford University.

[Rich and Knight, 1991] Elain Rich and Kevin Knight. *Artificial Intelligence.* McGraw-Hill, New York, second edition, 1991.

[Russell and Norvig, 1995] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[Russell *et al.*, 1993] Stuart J. Russell, Devika Subramanian, and Ronald Parr. Provably bounded optimal agents. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 338–344, 1993.

[Russell, 1990] Stuart J. Russell. Fine-grained decision-theoretic search control. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence*, pages 436–442, 1990.

[Sacerdoti, 1974] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[Sacerdoti, 1977] Earl D. Sacerdoti. *A Structure for Plans and Behavior.* Elsevier Publishers, Amsterdam, Netherlands, 1977.

[Schank *et al.*, 1975] Roger C. Schank, Neil M. Goldman, Charles J. Rieger III, and Christopher K. Riesbeck. Inference and paraphrase by computer. *Journal of the Association for Computing Machinery*, 22(3):309–328, 1975.

[Schmid and Wysotzki, 1996] Ute Schmid and Fritz Wysotzki. Induction of recursive program schemes. In *Proceedings of the Tenth European Conference on Machine Learning*, pages 214–226, 1996.

[Shell and Carbonell, 1989] Peter Shell and Jaime G. Carbonell. Towards a general framework for composing disjunctive and iterative macro-operators. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.

[Simon *et al.*, 1985] Herbert A. Simon, K. Kotovsky, and J. R. Hayes. Why are some problems hard? Evidence from the Tower of Hanoi. *Cognitive Psychology*, 17:248–294, 1985.

[Simon, 1975] Herbert A. Simon. The functional equivalence of problem solving skills. *Cognitive Psychology*, 7:268–288, 1975.

[Simon, 1979] Herbert A. Simon. *Models of Thought*, volume I. Yale University Press, New Haven, CT, 1979.

[Simon, 1989] Herbert A. Simon. *Models of Thought*, volume II. Yale University Press, New Haven, CT, 1989.

[Simon, 1996] Herbert A. Simon. *The Sciences of the Artificial.* MIT Press, Cambridge, MA, third edition, 1996.

[Smirnov, 1997] Yury V. Smirnov. *Hybrid Algorithms for On-Line Search and Combinatorial Optimization Problems.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1997. Technical Report CMU-CS-97-171.

[Smith and Peot, 1992] David E. Smith and Mark A. Peot. A critical look at Knoblock's hierarchy mechanism. In *Proceedings of the First International Conference on AI Planning Systems*, pages 307–308, 1992.

[Stefik, 1981] Mark Stefik. Planning with constraints (MOLGEN: Part 1). *Artificial Intelligence*, 16(2):111–140, 1981.

[Stone and Veloso, 1994] Peter Stone and Manuela M. Veloso. Learning to solve complex planning problems: Finding useful auxiliary problems. In *Proceedings of the AAAI 1994 Fall Symposium on Planning and Learning*, pages 137–141, 1994.

[Stone and Veloso, 1996] Peter Stone and Manuela M. Veloso. User-guided interleaving of planning and execution. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 103–112. IOS Press, Amsterdam, Netherlands, 1996.

[Stone *et al.*, 1994] Peter Stone, Manuela M. Veloso, and Jim Blythe. The need for different domain-independent heuristics. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 164–169, 1994.

[Tabachneck-Schijf *et al.*, 1997] Hermina J. M. Tabachneck-Schijf, Anthony M. Leonardo, and Herbert A. Simon. CaMeRa: A computational model of multiple representations. *Cognitive Science*, 21(3):305–350, 1997.

[Tabachneck, 1992] Hermina J. M. Tabachneck. *Computational Differences in Mental Representations: Effects of Mode of Data Presentation on Reasoning and Understanding.* PhD thesis, Department of Psychology, Carnegie Mellon University, 1992.

[Tadepalli and Natarajan, 1996] Prasad Tadepalli and Balas K. Natarajan. A formal framework for speedup learning from problems and solutions. *Journal of Artificial Intelligence Research*, 4:445–475, 1996.

[Tamble *et al.*, 1990] Milind Tamble, Allen Newell, and Paul S. Rosenbloom. The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning*, 5:299–348, 1990.

[Tate, 1976] Austin Tate. Project planning using a hierarchical nonlinear planner. Technical Report 25, Department of Artificial Intelligence, University of Edinburg, 1976.

[Tate, 1977] Austin Tate. Generating project networks. In *Proceedings of the Second International Joint Conference on Artificial Intelligence*, pages 888–900, 1977.

[Tenenberg, 1988] Josh D. Tenenberg. *Abstraction in Planning.* PhD thesis, Department of Computer Science, University of Rochester, 1988. Technical Report 250.

[Unruh and Rosenbloom, 1989] Amy Unruh and Paul S. Rosenbloom. Abstraction in problem solving and learning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 681–687, 1989.

[Unruh and Rosenbloom, 1990] Amy Unruh and Paul S. Rosenbloom. Two new weak method incerements for abstraction. In *Proceedings of the Workshop on Automatic Generation of Approximations and Abstractions*, 1990.

[Valiant, 1984] Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.

[Veloso and Blythe, 1994] Manuela M. Veloso and Jim Blythe. Linkability: Examining casual link commitments in partial-order planning. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 170–175, 1994.

[Veloso and Borrajo, 1994] Manuela M. Veloso and Daniel Borrajo. Learning strategy knowledge incrementally. In *Proceedings of the Sixth International Conference on Tools with Artificial Intelligence*, pages 484–490, New Orleans, LA, 1994.

[Veloso and Carbonell, 1990] Manuela M. Veloso and Jaime G. Carbonell. Integrating analogy into a general problem-solving architecture. In M. Zemankova and Z. Ras, editors, *Intelligent Systems*, pages 29–51. Ellis Horwood, Chichester, United Kingdom, 1990.

[Veloso and Carbonell, 1993a] Manuela M. Veloso and Jaime G. Carbonell. Derivational analogy in PRODIGY: Automating case acquisition, storage, and utilization. *Machine Learning*, 10:249–278, 1993.

[Veloso and Carbonell, 1993b] Manuela M. Veloso and Jaime G. Carbonell. Towards scaling up machinge learning: A case study with derivational analogy in PRODIGY. In M. Zemankova and Z. Ras, editors, *Machine Learning Methods for Planning*, pages 233–272. Morgan Kaufmann, San Mateo, CA, 1993.

[Veloso and Stone, 1995] Manuela M. Veloso and Peter Stone. FLECS: Planning with a flexible commitment strategy. *Journal of Artificial Intelligence Research*, 3:25–52, 1995.

[Veloso *et al.*, 1995] Manuela M. Veloso, Jaime G. Carbonell, M. Alicia Pérez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120, 1995.

[Veloso *et al.*, 1997] Manuela M. Veloso, Alice M. Mulvehill, and Michael T. Cox. Rationale-supported mixed-initiative case-based planning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 1072–1077, 1997.

[Veloso, 1989] Manuela M. Veloso. Nonlinear problem solving using intelligent casual commitment. Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University, 1989.

[Veloso, 1994] Manuela M. Veloso. *Planning and Learning by Analogical Reasoning.* Springer-Verlag, 1994.

[Wang, 1992] Xuemei Wang. Constraint-based efficient matching in PRODIGY. Technical Report Technical Report CMU-CS-92-128, School of Computer Science, Carnegie Mellon University, 1992.

[Wang, 1994] Xuemei Wang. Learning planning operators by observation and practice. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 335–340, 1994.

[Wang, 1996] Xuemei Wang. *Learning Planning Operators by Observation and Practice.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1996. CMU-CS-96-154.

[Warren, 1974] D. H. D. Warren. WARPLAN: A system for generating plans. Technical Report Memo 76, Department of Computational Logic, University of Edinburg, 1974.

[Weld, 1994] Daniel S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.

[Wilkins and Myers, 1995] David E. Wilkins and Karen L. Myers. A common knowledge representation for plan generation and reactive execution. *Journal of Logic and Computation*, 5(6):731–761, 1995.

[Wilkins and Myers, 1998] David E. Wilkins and Karen L. Myers. A multiagent planning architecture. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 154–162, 1998.

[Wilkins *et al.*, 1995] David E. Wilkins, Karen L. Myers, J. D. Lowarance, and L. P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):197–227, 1995.

[Wilkins, 1984] David E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22(3):269–301, 1984.

[Wilkins, 1988] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, San Mateo, CA, 1988.

[Wood, 1993] Derick Wood. *Data Structures, Algorithms, and Performance*. Addison-Wesley Publishers, Reading, MA, 1993.

[Yamada and Tsuji, 1989] Seiji Yamada and Saburo Tsuji. Selective learning of macro-operators with perfect causality. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 603–608, 1989.

[Yang and Murray, 1994] Qiang Yang and Cheryl Murray. An evaluation of the temporal coherence heuristic in partial-order planning. *Computational Intelligence*, 10(3):245–267, 1994.

[Yang and Tenenberg, 1990] Qiang Yang and Josh Tenenberg. ABTWEAK: Abstracting a non-linear, least-commitment planner. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 204–209, Boston, MA, 1990.

[Yang *et al.*, 1996] Qiang Yang, Josh Tenenberg, and Steve Woods. On the implementation and evaluation of ABTWEAK. *Computational Intelligence*, 12(2):295–318, 1996.

[Yang *et al.*, 1998] Qiang Yang, Philip Fong, and Edward Kim. Design patterns for planning systems. In *Proceedings of the 1998 AIPS Workshop on Knowledge Engineering and Acquisition for Planning: Bridging Theory and Practice*, pages 104–112, 1998.

[Yang, 1997] Qiang Yang. *Intelligent Planning: A Decomposition and Abstraction Based Approach*. Springer-Verlag, Berlin, Germany, 1997.