Automated Acquisition of Control Knowledge to Improve the Quality of Plans

M. Alicia Pérez and Jaime G. Carbonell [1]

April 1993
CMU-CS-93-142

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Most of the work to date on automated control-knowledge acquisition has been aimed at improving the *efficiency of planning*; this work has been termed "speed-up learning". The work presented here focuses on the automated acquisition of control knowledge to guide a planner towards better solutions, i.e. to improve the *quality of plans* produced by the planner, as its problem solving experience increases. To date no work has focused on automatically acquiring knowledge to improve plan quality in planning systems. We present a taxonomy of plan quality metrics and a first prototype that partially automates the task of acquiring quality-enhancing control knowledge for the PRODIGY nonlinear planner. We are working on testing the effect of such control knowledge in plan quality, and developing methods to learn such control knowledge. Two complex domains, namely a transportation logistics domain, and a machining process planning domain, are being used to evaluate these ideas.

[1]This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597, and in part by a scholarship to the first author from the Ministerio de Educación y Ciencia of Spain. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government or the Spanish Government.

Keywords: plan quality, search control knowledge, machine learning, planning, knowledge acquisition

Contents

1 Introduction

The focus of this work is on the automated acquisition of search control knowledge for planning systems. Most of the work to date on automated control-knowledge acquisition has been aimed to improve the *efficiency of planning*; this work has been termed speed-up learning. Our focus is on the acquisition of control knowledge to guide the planner towards better solutions, i.e. to improve the *quality of plans* produced by the planner.

There are many variations on the notion of the quality of a plan in the context of classical planning systems [Langley and Drummond, 1990], such as:

- the length of the solution path or the total number of actions
- the execution time of the plan
- the energy, or other resources, required for the plan execution
- the robustness of the plan, or its ability to respond well under changing or uncertain conditions.

Human experts gather knowledge for producing better plans through experience. Here "better" is defined in a context-sensitive manner as a combination of plan-quality factors such as those listed above. It is precisely this experiential knowledge that we seek to capture from planning experience. We propose here to focus our attention primarily on acquiring control knowledge to guide the planner's search towards better solutions during planning, rather than post-facto plan modification. The framework for this work is the PRODIGY architecture.

The rest of this section gives an example of plan quality in a particular domain. Section 2 proposes a taxonomy of plan quality metrics. Section 3 explores the relationship between plan quality and goal interactions. Section 4 gives some background on PRODIGY, the example domains used in this work, and how PRODIGY's search decisions affect plan quality. Section 5 describes an implemented prototype for semi-automated acquisition of control rules to improve plan quality in PRODIGY. Section 6 briefly presents work in progress for learning quality-enhancing control rules. Section 7 analyzes related work, and Section 8 concludes with a summary of the expected contributions of this work [1].

1.1 An Example: Plan Quality in a Process Planning Domain

In the process planning phase of production manufacturing plan quality is crucial in order to minimize both resource consumption and execution time. The goal of process planning is to produce plans for machining parts given their specifications. Such planning requires taking into account both technological and economical considerations [Descotte and Latombe, 1985, Doyle, 1969], for instance:

- It may be advantageous to execute several cuts on the same machine with the same fixing to reduce the time spent setting up the work on the machines.
- If a hole H_1 opens into another hole H_2, then one is recommended machining H_2 before H_1 in order to avoid the risk of damaging the drill.

[1] A slightly different version of this document was submitted as a thesis proposal in the School of Computer Science, Carnegie Mellon, in December 1992.

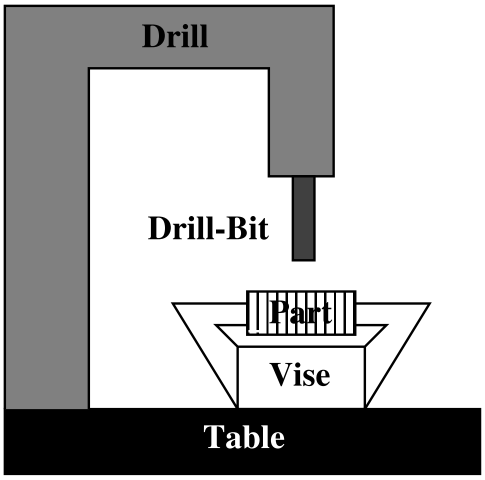

Figure 1: An Example of Set-Up in the Machining Domain (from [Joseph, 1992]). In this example the holding device is a vise, the machine a drilling machine, and the tool a drill-bit.

Most of these considerations are not pure constraints but only preferences when compromises are necessary. They often represent both the experience and the know-how of engineers, so they may differ from one company to the other.

Let us look at a concrete example of the difference in quality of plans in this domain, in particular in its implementation as one of PRODIGY's domains [Gil, 1991]. The domain concentrates on the machining, joining, and finishing steps of production manufacturing. The goal is to produce one or more parts according to certain specifications. An example of a request would be for a rectangular block of 5"x2"x1" made of aluminum and with a centered hole of diameter 1/32" running through the length of the part. In order to perform an operation on a part, the part has to be secured to the machine table with a holding device, and in many cases the part has to be clean and without burrs from preceding operations. The appropriate tool has to be selected and installed in the machine as well. As an example, Figure 1 shows a machine set-up to drill a hole in a part. Figure 2 sketches graphically the steps to produce a reamed hole. Before performing each of these steps, the appropriate tool has to be set in the machine spindle, namely a spot-drill, a high-helix-drill, and a reamer. Then some holding device (a vise in the example) has to be put on the machine, and the part has to be held by the holding device.

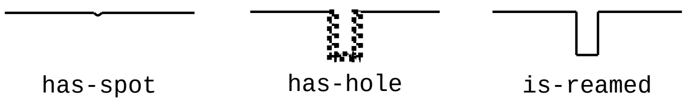

Figure 2: The Steps to Make a Reamed Hole: first a spot hole is drilled on the part, then the hole itself is made, and finally the hole is reamed. For each of these the operations the appropriate tool (respectively a spot drill, some appropriate drill bit, or a reamer) has to be installed in the drilling machine.

Suppose the planner has to build a plan to have a part with *two* reamed holes on one of its sides. If the planner works on making each hole separately, it will obtain a solution, sketched in Figure 3(a) (the operators to hold the part are omitted). This solution is not the shortest one (and in this domain a shorter solution may mean a faster and cheaper way to produce a large number of parts). Some steps may be eliminated by reordering the operations. Both holes, and spot holes for that matter, have to be in the same side and may be made with the same tools. Therefore once we have set the appropriate tool in the drill spindle and held the part on the machine table, the operations corresponding to both holes can be performed consecutively. Figure 3(b) shows a better solution to the problem. In this example the planner obtains the better solution by interleaving the problem goals. In PRODIGY this decision may be encoded in the form of a search control rule.

hole1

put tool to drill a spot hole
clean the part
drill spot hole for hole1
change tool to make hole
drill hole1
change tool to ream
clean the part
ream hole1

hole2

change tool to drill a spot hole
drill spot hole for hole2
change tool to make hole
drill hole2
change tool to ream
clean the part
ream hole2

(a)

hole1 and hole2

put tool to drill a spot hole
clean the part
drill spot hole for hole1
drill spot hole for hole2
change tool to make hole
drill hole1
drill hole2
change tool to ream
clean the part
ream hole1
clean the part
ream hole2

(b)

Figure 3: Two Plans of Different Quality to Make Two Reamed Holes on a Part. Some steps in solution (a) may be eliminated by reordering the operations, since once the corresponding tool is set, the operation may be performed for both holes consecutively. Solution (b) captures such improvement by interleaving the operations on hole1 and hole2.

2 A Taxonomy of Quality Metrics

In this section we propose a taxonomy of quality metrics for planning systems. These metrics can be classified in three large groups. The planner solutions can be compared in terms of their execution cost, their robustness or reliability under unexpected circumstances, and the satisfaction of the client with the solution itself (for example the accuracy of the result, or the comfort it provides to the user). Figure 4 presents this taxonomy, and the next subsections explore it in detail. The problem of finding good quality plans is different from that of reducing the effort required to generate plans. If only limited resources (time or space) are available for planning, the planner may have to give up trying to find a good solution and resign to any solution within the given resource bound. Section 2.6 discusses metrics of problem solving efficiency and some previous work on improving it in PRODIGY.

2.1 Minimizing Execution Cost

The quality of a plan is strongly related to the cost of executing it. Some of the factors that affect a plan's execution cost can be computed by summing over all the steps or operators in the plan, that is $C_{total} = \Sigma c_i$ where C_{total} is the total cost of executing the plan and c_i is the cost for each operator. c_i can be the operator

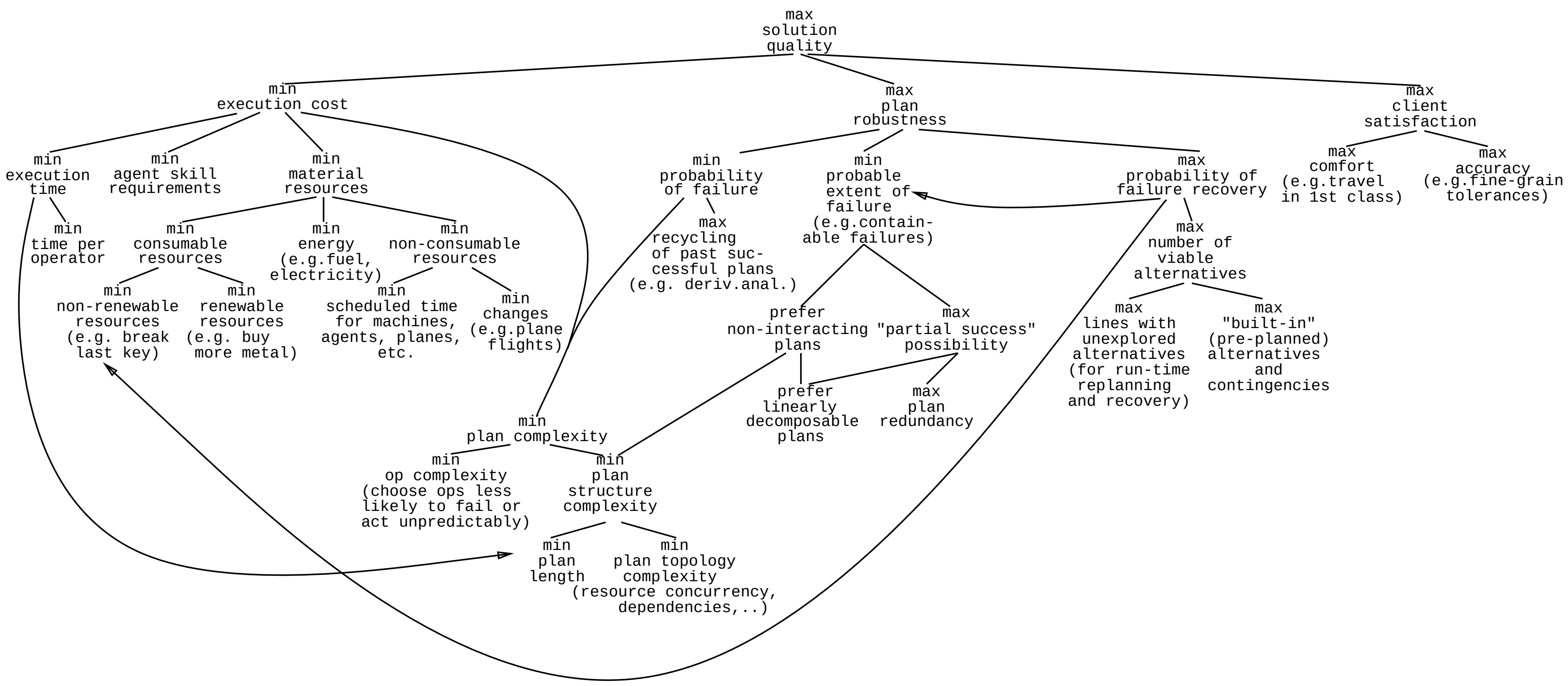

Figure 4: A Taxonomy of Quality Metrics. These metrics can be classified in three broad categories, namely those regarding execution cost, those related to robustness of the plan obtained, and those considering the degree of satisfaction of the client with the solution obtained. Note that this taxonomy does not consider planning efficiency metrics.

execution time, the cost of the resources used by the step, or 1 if the measure is simply the length of the plan or total number of actions. This section analyzes factors that influence a plan's execution cost.

Execution time: a straightforward measure of execution time is the number of operations, or length, of the plan. However different actions frequently take different times to execute and therefore considering the execution time for each operator is more significant than just the length of the plan. The plan topology affects execution time if several actions may be executed in parallel.

Material resources: a resource is something needed to perform a plan step. In a factory scheduling domain, or the process planning domain described above, resources include personnel, material, tools and machines. Each action includes which resources it requires and the characteristics that the resource must have. Resources can be classified in several categories:

- Time.
- Energy.
- *Consumable resources* are those for which an initial stockpile is available which can only be depleted by actions in the plan [Currie and Tate, 1991], for example materials. Particular instances of these resources cannot be reused by several actions. Consumable resources can be further classified as *renewable* and *non-renewable*. The agent has some way of obtaining more renewable resources if it runs out of them (for example, it can acquire more metal stock). However once it runs out of some non-renewable resource but still needs it, the agent cannot make progress in the plan execution.
- *Non-consumable* resources are those that are rendered busy for a period of time, and then released and made available to other actions. This includes the use of machines and tools. The consumption of resources is related but different from an object wearing out. For example, machines wear out, but are usually considered as non-consumable resources. The wearing is related to the cost of an operator making use of the resource, but not directly so.

The use of resources is closely related to the cost of executing a plan. Good plans try to make the *best* use of the resources available. There are many considerations about resources and resource use that influence the quality of a plan:

- Maximize the fraction of time in which a resource is actually used, and inter-operation transfer times. The planner should try to reduce the amount of time a resource remains idle but locked by some action. An example of this is reducing the time an airplane remains sitting in the airport waiting for its load to arrive. This idle time may force the planner to use a different resource for another actions in the meantime, or else to create a plan with a longer execution time. As another example, in the Hubble space telescope (HST) domain, a good schedule will try to maximize the fraction of time spent actually recording data on any instrument of the telescope, as opposed to (for example) realigning the telescope or switching instruments [Muscettola and Smith, 1990]. Scheduling systems typically address these considerations in order to generate good schedules [Fox and Smith, 1984] [2].
- Minimize the number of resource requests that will be necessary during plan execution: human experts try to reduce resource reservations, as a resource request may create waiting times until the resource becomes available, if it is being used by a different agent. Once one obtains a resource, it

[2] We make a distinction between the process planning problem and the scheduling problem. The former can be defined as selecting a sequence of operations whose execution results in the achievement of a goal, for example the completion of an order in a job shop. The latter focuses on the assignment of start and end times and resources to each of those operations.

is better to use it as much as possible before releasing it. In the process planning domain example presented in Section1.1, the better solution takes advantage of the same set-up to perform several operations, instead of setting up the machine, tool and part once for each operation to perform. This not only reduces the length of the plan but also the number of resource requests.

- Resource sharing may reduce the length and cost a plan, for example when the same truck is used to move two different packages to the same destination. Sometimes however it is better not to use a resource that is being used by other operators. This is a way to avoid resource conflicts with the plans for other agents, and therefore the resulting plans are more amenable to parallelization. We made use of this criterium in previous work on multiagent planning [Pérez, 1991]: resource preferences were encoded in control rules, mostly domain independent, and then plan topology and resource conflicts were analyzed in order to build parallel plans.

The choice of resources not only conditions the plan execution cost, but also its robustness. Many factors related to the availability of alternative resources, such as machine downtime, machine substitutability, or alternative production processes, can affect the reliability of a plan and the availability of alternative actions when using the chosen resource fails.

Agent skill requirements: they refer to the extent to which an agent can perform an action. Some examples are strength, speed, intellect, and how good it is at a particular skill. Plans with less agent skill requirements are typically less expensive.

Plan complexity: plan complexity affects both the execution cost and the robustness of a plan, and there usually exists a trade-off between these two measures of plan quality. The complexity of a plan can be seen at two levels:

- operator complexity: the choice of a particular operator is influenced by how likely it is to fail or have unpredictable results, and also by its execution cost, both in time and resources.

- complexity of the plan structure: plan length is a straightforward measure of the quality of a plan. However in some cases it is interesting to consider the plan topology as well. The dependencies among operators and their concurrency on the use of resources influence execution time, as several actions may be performed in parallel if they are independent [Pérez, 1991]. They also influence the plan robustness when several actions contend for the same resource.

2.2 Maximizing Plan Robustness

By robustness of a plan we mean its ability to respond well under changing or uncertain conditions. A plan's execution may fail because of unexpected environmental changes or events, of actions not having their intended effects (the resources are not reliable, or the operators outcome is uncertain [3]), or even of inadequacies of the planner itself. When a failure occurs, execution of a plan may go awry and produce outcomes considerably different from the desired goal. Therefore the quality of a plan depends on its reliability and its potential for recovery after a failure. Recent work with focus on execution-failure recovery uses different methods for adapting a plan upon failure so execution can continue [Howe and Cohen, 1991]. Another approach is to improve the planner's knowledge to guide its search towards more robust plans. Note that there is usually a trade-off between cost and robustness [Feldman and Sproull, 1977].

Three aspects determine a plan's robustness:

[3]However PRODIGY in particular assumes that there is not uncertainty on the operators or the world model. For a thorough investigation of inadequate operators and learning to improve their fidelity to external actions, see [Gil, 1992].

- probability of failure
- extent of the failure
- possibility of failure recovery

To reduce the probability of an execution failure the planner may reuse plans that proved successful in past similar situations and were stored then. Plans that are complex, both at the operator level and at the plan topology, are more prone to execution failures, as discussed above. Some operators may be more likely to fail or act unpredictably than others. Complex plans with many dependencies among operators and resource sharing may suffer of resource contention and therefore fail at execution time.

Another factor for plan robustness is whether the consequences of a failure during execution are localized. If all the plan actions are interacting and one of them fails, the plan may fail without making any progress towards the goal. However if two parts of a plan are non-interacting, a failure in one part will not affect the other. (We refer to these types of failures as containable failures.) In particular linearly decomposable plans are preferred, i.e. those that can be decomposed in independent subplans that achieve different subgoals. Therefore the localization of the parts of the plan that can cause execution failures increases the possibility of partial success of the plan. By analyzing those parts the planner can incorporate redundant steps that increase the probability of success of the whole plan.

Failures are often unpredictable and in spite of reducing their chances of occurring, they may eventually happen. Plan robustness includes the possibility of recovery after a failure. This possibility increases if the use of non-renewable resources in the plan is minimal. If the agent runs out of a resource of this type, no recovery that requires that same resource is possible. Recovery is facilitated when other alternatives to the faulty parts of the plan are available. These alternatives may be built in the plan by planning in advance for contingencies. The alternatives may also be stored in a library of recovery methods [Howe and Cohen, 1991]. They can also be left unexplored. In this case they can be used upon failure at running time for replanning and recovery.

2.3 Client Satisfaction

There are some other factors of plan quality that can hardly be considered in the previous categories and in some cases they are hard to quantify. For example, the human user may prefer a plan that takes him from one city to another in first class instead of second class. The process planning domain exemplifies how the degree of accuracy required in a part may influence the choice of machine or tool to perform a given operation, if the result has to satisfy a fine-grain tolerance.

In scheduling systems that try to find the *optimal* schedule, the value of the resulting state is an applicable criterium to measure plan quality. The quality of a solution can be measured by the number of the goals achieved by the solution or the value of such goals. For example in the case of scheduling the operations of the Hubble Space Telescope, the quality of the solution obtained depends on the number of proposals that the HST can accommodate [Muscettola and Smith, 1990] and how the proposals accommodated relate to the program and observation priorities, which are given as part of the problem to solve. [4]

2.4 Trade-Offs Among Quality Metrics

When deciding which plan is better one can easily run into trade-offs. This can be illustrated with an example from the process planning domain. In this domain several machines can be used to reduce the

[4] Note however that at present PRODIGY's solution to a problem achieves all the goals in the problem, or else renders the problem unsolvable. Therefore in that sense all the plans have the same value, with respect to the degree of goal satisfaction.

size of a part. The choice of a particular type of machine may depend on the degree of accuracy desired, the economy of the plan, or the time required for execution. The last two factors in turn might only be relevant depending on the number of parts on which the same operation has to be performed (i.e. how many times the same plan will be executed). Grinding a part gives better finishes and holds closer tolerances than other machines, but it may be more expensive. The shaping and planing operations, in order to reduce a part's size, are slower than milling the part, but the tools they use are less expensive and easier to sharpen. Therefore it may be necessary to consider trade-offs among the different options. Factors such as the skills available and required to operate the machine, personal likes and dislikes, and availability may also need to be considered. Most of these factors are not as basic as others but in some cases may be decisive.

Scheduling systems have to face a similar problem when the constraints encoding different quality factors conflict [Fox and Smith, 1984]. For example, removing a machine's second shift may decrease costs but may also cause an order to miss its due date. Therefore in these systems one cannot rely only on constraint propagation techniques to arrive to acceptable solutions. Rather, they choose to relax some constraints and find a solution that best satisfies the remaining constraints.

2.5 Domain-Dependent Versus Domain-Independent Metrics

In some cases the ways to measure the quality of a plan are clearly domain dependent. The goal of the scheduler presented in [Perry, 1990] is to schedule launches and terminal illuminators to maximize the depth of fire, or number of shots at incoming threats. Other criteria, such as "minimize the consumption of resources," seem obvious and applicable to every domain. However we can find cases where this is not the best thing to do. The distinction between domain-dependent and domain-independent aspects is not always clear. We take this example from [Wilkins, 1988]: consider the advice "use existing objects." This is a fairly domain-independent concept that is used by NOAH, and mentioned by Wilensky as a meta-goal for planning. However this idea still involves domain-dependent knowledge. In a house-building domain, it is desirable to use the same piece of lumber to support the roof and the sheetrock on the walls. But in other domains this may not be a good strategy. On the space shuttle, one may want different functions performed by different objects so the plan will be more robust and less vulnerable to the failure of any one object. So the "use of existing objects" idea makes assumptions about the domain that need to be stated (perhaps one wants to apply this idea only to certain portions of the domain).

2.6 Measures of Planning Cost

The quality of a planning algorithm depends both on the quality of the solutions generated and on the effort spent searching for them. If only limited computational resources are available to the planner during problem solving, the planner may have to trade off solution quality in order to find a solution at all. The planning cost depends on both the time and the space spent during problem solving. Two measures of planning cost have been used in the literature on learning and planning (for example [Minton, 1988, Pérez and Etzioni, 1992]), namely search time and number of nodes in the search tree. Figure 5 summarizes planning cost criteria. Planning time can be measured as time spent in pure planning, or amortized when planning and learning are interleaved. Planning space is usually measured as working space (for example, number of nodes expanded in search). When planning and learning are interleaved and the planner stores knowledge that will be useful later, a long-term use of space has to be considered. The stored knowledge can take the form of search control rules extracted from problem solving traces, or of cases in a case library [Veloso, 1992, Kambhampati, 1990, Hammond, 1986]. Recycling past successful experience reduces the search effort when solving new similar problems. Note that there is usually a trade-off among the amount of knowledge stored, the cost of accessing and reusing it, and the savings on search gained from it [Minton, 1988].

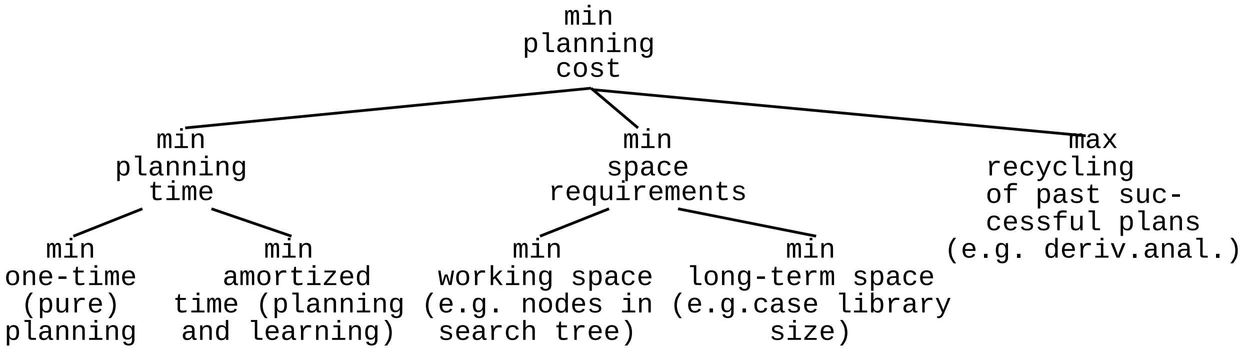

Figure 5: A Taxonomy of Metrics of Planning Cost. Most of the machine learning mechanisms designed to date to acquire control knowledge for planning systems are aimed at improving the efficiency of planning. These techniques are known as *speed-up learning*.

2.6.1 Solution Quality Versus Problem Solving Efficiency

As we have already mentioned, the problem of finding good quality plans is different from that of reducing the effort required to generate plans. In many domains finding a plan at all requires a considerable amount of search and there has been work on improving the efficiency of a problem solver with machine learning techniques [Mitchell *et al.*, 1983, Laird *et al.*, 1986, Gratch and DeJong, 1992, Korf, 1985, Veloso, 1992, Minton, 1988, Knoblock, 1991b, Etzioni, 1990]. We call this *speed-up* learning. However these mechanisms have paid none or little attention to the quality of the solutions obtained. Here we briefly present some examples of speed-up learning systems in the context of PRODIGY.

PRODIGY's explanation-based learning system (EBL) [Minton, 1988] constructs explanations from a problem solving trace and an axiomatized theory describing both the domain and relevant aspects of the problem solver's architecture. Then the resulting descriptions are expressed in control rule form, and control rules whose utility in search reduction outweighs their application overhead are retained. The system can learn from success, failure and goal interaction. These concepts are represented declaratively as target concepts. The analysis of goal interactions may lead to better plans (see Section 3) but that is not a goal of the EBL module. For the EBL module to learn control knowledge that improves the quality of the plans, it would be necessary to augment the domain theory and target concepts to be able to explain, or prove, why the solution obtained in the current problem solving episode is a good one. Something similar may be said of systems that perform static analysis on the problem space representation [Etzioni, 1990, Pérez and Etzioni, 1992].

The derivational analogy module of PRODIGY [Veloso, 1992] stores past problem-solving experiences as cases, and reuses them to solve similar problems, obtaining a considerable improvement in problem solving efficiency. The use of one or more past cases to solve the current problem may lead to shorter plans, as reported in [Veloso, 1992]. This was a surprising result but not the focus of the work. Note in fact that if the solution stored to solve a problem was not a good one, it may be reused to solve subsequent problems without trying to find a better solution.

PRODIGY's abstraction planning module [Knoblock, 1991a] divides the axiomatized domain knowledge into multiple abstraction levels. Then during problem solving, a solution is found first in the top-level space to guide the search for solutions in more detailed problem spaces. The use of abstraction hierarchies reduces the problem solving space but does not guarantee that the solution obtained is the best one (see [Carbonell *et al.*, 1992] for an example). However it may lead to produce shorter solutions since the abstractions focus the problem solver on the parts of the problem that should be solved first. [Knoblock, 1991b] presents experiments in which the use of abstractions produces solutions that are about 10% shorter than those produced by PRODIGY in certain domains. Note that the measure of plan quality used in this case is the length of the plan, and it seems that these results do not extend to other quality metrics.

In the work reported here we will focus on the part of the taxonomy related to execution cost. See Section 6 for a description of the proposed work plan.

3 Solution Quality and Goal Interactions [5]

Planning goals rarely occur in isolation. A planner must be capable of taking into account the interactions between conjunctive goals in order to produce a plan to solve the problem. There have been many research efforts addressing the issue of planning for conjunctive goals, focusing on a variety of aspects, including analyzing the complexity of this planning problem [Sussman, 1975, Chapman, 1987], designing appropriate planning algorithms [Sacerdoti, 1977, Tate, 1977, Drummond and Currie, 1987], categorizing different types of goal interactions [Wilensky, 1983], and learning control knowledge to efficiently handle the search for the interactions [Minton, 1988, Etzioni, 1990]. Our work, though built upon this previous work, goes beyond it as we aim at identifying goal interactions directly related to the quality of the plans produced.

From a practical implementation point of view we distinguish two categories of goal interactions, *explicit goal interactions* and *quality goal interactions*.

3.1 Explicit goal interactions

We include in this category the goal interactions that are explicitly represented as part of the domain knowledge in terms of preconditions and effects of the operators. A plan exhibits a goal interaction of this type when there is a goal in the plan that has been negated by a previous step in the plan [Minton, 1988, Etzioni, 1990]. These goal interactions enforce particular goal orderings in order that the planner may be able to produce a solution to the problem. In a typical example of a two-goal interaction, after one of the goals has been achieved, it is deleted by an operator that works towards achieving the other goal. [6]

Goal interactions in this category include the well-known Sussman's anomaly in the blocksworld [Sussman, 1975]. Consider also another illustrative example in a transportation domain. In this domain, packages are to be moved among different cities. Packages are carried within the same city in trucks and across cities in airplanes. Trucks and airplanes may have limited capacity. At each city there are several locations, e.g. post offices (po) and airports (ap). A package `P1` is at the Pittsburgh airport. There is only one airplane, `A1`, available also at the Pittsburgh airport. The goal consists of having both the airplane and the package at the Boston airport, and is represented by the conjunction `(and (at-airplane A1 bos-airport) (at-object P1 bos-airport)`. If the goal `(at-airplane A1 bos-airport)` is addressed first, and A1 flies from Pittsburgh to Boston, there is no way to achieve the second goal without first flying back A1 to Pittsburgh. The resulting plan involves flying A1 back and forth unnecessarily. It is conceivable to design an algorithm that fixes this kind of plans by removing unnecessary operations that reachieve the clobbered goals [Rich and Knight, 1991].

In some problems, these interactions are unavoidable and the planning system must find a solution that minimizes their effects. When this happens, search time is typically reduced and better solutions tend to be

[5] This section appears, extended, in [Pérez and Veloso, 1993]

[6] Other goal interactions in this category may be beneficial to the planning process, when solving one goal makes a second goal easier to achieve. This is generally termed goal concord and opportunistic planning takes advantage of these situations [Converse and Hammond, 1992].

found. These solutions are generally shorter in length, and more direct [Minton, 1988, Ryu and Irani, 1992, Veloso, 1992].

In least-commitment planners the critics take care of these goal interactions by establishing ordering constraints among the conflicting goals. In the case of PRODIGY, a casual-commitment planner, goal preference control knowledge is automatically acquired to deal effectively (in the sense of problem solving effort) with this kind of goal interactions by different machine learning approaches, namely explanation-based learning [Minton, 1988], static analysis [Etzioni, 1990], or derivational analogy [Veloso, 1992].

A particular problem may have many different solutions. These solutions may differ in the set of operators in the plan. If the ordering constraints between achieving two goals are explicit in the domain representation, then all the solutions to a particular problem will have the two goals interacting. On the other hand the dependencies may be the result of a particular problem solving path explored. In this case for some solutions the goals may interact and for some others they may not.

3.2 Quality goal interactions

To illustrate this difference and to motivate the quality goal interactions, we further discuss different plans with ordering constraints that are or are not explicit in the domain. In the *one-way rocket* domain [Veloso, 1989], the goals of moving two objects to a different location interact, because the rocket can only move once. This is an interaction that is represented in the domain definition as the moving operator explicitly deletes the location of the rocket. The *machine-shop scheduling* domain [Minton *et al.*, 1989] also constraints that holes in parts must be drilled before parts are polished, as the drilling operator deletes the shining effect. In this domain, the goals of polishing and making a hole in a part interact again due to the domain definition.

However, in this same machine-shop scheduling domain, when two identical machines are available to achieve two identical goals, these goals may interact, if the problem solver chooses to use just one machine to achieve both goals, as it will have to wait for the machine to be idle. If the problem solver uses the two machines instead of just one, then the goals do not interact in this particular solution.

There is a variety of equivalent examples in the logistics transportation domain. In general it is not clear what use of resources is overall the best. For example, in the logistics transportation domain, suppose that the problem solver assumes that the same truck (or airplane) must be used when moving objects from the same location into the same (or close) destiny. In this case the goals of moving the objects interact. But if different carriers are chosen, there is not such interaction. Note that the problem can become quite complicated as the domain considers other types of constraints, such as capacity for the carriers, size of the objects to be transported, distances between locations that dictate the type of carrier to use, and so on.

In the example presented in Section 1.1 the planner obtains the better solution by interleaving the problem goals. In PRODIGY this decision may be encoded in the form of search control rules. Note that if the goal interactions are not considered the planner still constructs a valid solution. It is only because of quality considerations that the interactions occur as the same set-up is used for achieving the goals for both holes. These interactions are not explicitly represented in the domain specification.

These interactions are related to plan quality as the use of resources dictates the interaction between the goals. The control knowledge that guides the planner to solve these interactions is harder to learn automatically, as the domain theory does not encode these quality criteria. Our work is a current research effort on learning control knowledge to improve the quality of the plans generated by the problem solver.

Some of the interactions between goals are due to the use of a state-space planner, as the operator ordering in the final plan is tied to the goal ordering during problem solving. By using a plan-space planner, in which actions can be inserted anywhere in the plan, some of these problems may go away. However there is still the issue of which is the appropriate control knowledge, heuristics or critics, to select the best place to insert actions into the plan. There are a few other planners that analyze the relationship between goal interactions and plan quality. Section 7.1 discusses some of this related work.

4 Background: The PRODIGY Problem Solver

PRODIGY is a domain-independent problem solver. Given an initial state and a goal expression, PRODIGY searches for a sequence of operators that will transform the initial state into a state that matches the goal expression. PRODIGY's sole problem-solving method is a form of means-ends analysis. Table 1 describes the basic search cycle of PRODIGY's nonlinear planner [Veloso, 1989]. A complete description of PRODIGY appears in [Carbonell *et al.*, 1992].

---

1. Check if the goal statement is true in the current state, or there is a reason to suspend the current search path.

   If yes, then either return the final plan or backtrack.
2. Compute the *set* of *pending goals* $\mathcal{G}$, and the set of possible *applicable operators* $\mathcal{A}$.
3. Choose a goal G from $\mathcal{G}$ or select an operator A from $\mathcal{A}$ that is directly applicable.
4. If G has been chosen, then
   - get the set $\mathcal{O}$ of *relevant operators* for the goal,
   - choose an operator O from $\mathcal{O}$,
   - get the set $\mathcal{B}$ of possible *bindings* for O,
   - choose a set B of bindings from $\mathcal{B}$,
   - go to step 1.
5. If an operator A has been selected as directly applicable, then
   - *apply* A,
   - go to step 1.

Table 1: A Skeleton of PRODIGY's Nonlinear Search Algorithm (Adapted from [Veloso, 1989]). Problem solving decisions, namely selecting which goal/subgoal to address next, which operator to apply, what bindings to select for the operator, or where to backtrack in case of failure, can be guided by control knowledge. PRODIGY's trace provides all the information about the decisions made during problem solving so it can be exploited by machine learning methods.

PRODIGY provides a rich action representation language coupled with an expressive control language. Preconditions in the operators can contain conjunctions, disjunctions, negations, and both existential and universal quantifiers with typed variables. Effects in the operators can contain conditional effects, which depend on the state in which the operator is applied. The control language allows the problem solver to represent and learn control information about the various problem solving decisions, such as selecting which goal/subgoal to address next, which operator to apply, what bindings to select for the operator or where to backtrack in case of failure. In PRODIGY, there is a clear division between the declarative domain knowledge (operators and inference rules) and the more procedural control knowledge. This simplifies both the initial specification of a domain and the incremental learning of the control knowledge.

PRODIGY is designed with a "glass-box" approach: all the steps taken, all the decisions made, and all the information consulted by the engine are available in a problem's trace. This provides an information context in which learning can take place. Previous work on PRODIGY used explanation-based learning techniques, static analysis of the domain definition [Minton, 1988, Etzioni, 1990, Pérez and Etzioni, 1992], and derivational analogy [Veloso, 1992] to acquire search control knowledge to increase problem-solving efficiency. The machine learning and knowledge acquisition work supports PRODIGY's casual-commitment method [7], as it assumes there is *intelligent* control knowledge, exterior to its search cycle, that it can rely upon to make decisions.

4.1 Example Domains

The examples presented throughout this document are extracted from two domains: a transportation logistics domain, and a machining process planning domain. These are the most complex and real-world domains to which PRODIGY has been applied up to date. In this section we describe them briefly. [8]

4.1.1 The Transportation Logistics Domain

This is a complex logistics planning domain in which packages are to be moved among different cities. Packages are carried within the same city in trucks and across cities on airplanes. Each truck operates in a single city. At each city there are several locations, e.g. post offices and airports. There are six operators in the domain namely loading and unloading trucks and airplanes, driving trucks between locations, and flying airplanes between airports. In the current version trucks and airplanes have unlimited capacity. The domain could be extended in different ways, for example to consider the capacity of the carriers and package sizes, the fuel consumption, and/or the distance between cities.

In this domain interleaving of goals and subgoals at different levels of the search is needed to find a good solution: consider for example the problem of moving two given packages from the Pittsburgh airport to the Boston airport. Accomplishing either goal individually, as a linear planner would do, would require using a different airplane (or a different trip of the same airplane) for each of the packages, which is clearly an inefficient way to solve the problem. PRODIGY's nonlinear algorithm may delay flying the airplane until both packages are loaded by means of control rules.

4.1.2 The Machining Process Planning Domain

Section 1.1 described this domain and introduced an example taken from it. This domain is to date the largest one implemented in PRODIGY. It has 75 operators and 35 inference rules. [Gil, 1991] describes it in detail. Some of the problems used to test the domain were taken from real engineers specifications as presented in [Hayes, 1990] and the number of operators (not including inference rules) in their solution ranges between 35 and 70.

4.2 PRODIGY Decisions that Affect Plan Quality

The previous section presented the types of decision points in PRODIGY's search cycle. These decisions may influence the quality of the final plan and encode available expert knowledge. Some examples follow:

- Goal ordering: in the example in Section 1.1 choosing the right goal ordering reduced plan length. In general, asymmetric goal interactions yield a preferred optimal goal ordering to minimize clobbering of previously achieved goals. Here we present another example of how goal ordering decisions influence plan quality. Suppose the problem posed to PRODIGY is to have a part with two holes, one opening into another. This can be encoded as the conjunction of two goals, one for each hole.

---

[7] In a casual-commitment strategy at each decision point the planner commits to a particular alternative, and backtracks upon failure. This is in contrast to a least-commitment strategy where decisions are deferred until all possible interactions are recognized.

[8] Throughout this document several examples are presented. In them we follow PRODIGY's standard notation: instantiated operators are enclosed in <> and literals, both state literals or goals, are enclosed in ().

PRODIGY has to start deciding on which hole to work first, i.e. which of the two goals try to achieve first. The following advice may be used to guide PRODIGY's decision:

> If a hole H_1 opens into another hole H_2, then one is recommended machining H_2 before H_1 in order to avoid the risk of damaging the drill. [Descotte and Latombe, 1985]

This advice can be translated into a search control rule. The antecedent preconditions match when one of the holes actually opens into another. The consequent recommends with which hole to start planning. It is interesting to realize that the expert advice may apply only in some circumstances. If machining the holes in the opposite order is faster, the right decision could have been different, and the rule should only fire when the risk of damaging the drill is more important than the time spent on the operations. Therefore different control rules, or control rule sets, may encode different strategies.

In some cases choosing the appropriate goal ordering is required to deal with goal interactions, both positive and negative. Section 3 presented some examples in which choosing the right goal interleaving produces better plans.

- Operator preferences: suppose PRODIGY's goal is to reduce the size of a part. Some of the candidate operators to achieve this goal are SHAPE, SHAPE-WITH-PLANER, and MILL. The following expert advice may be useful to decide which operator to try first, and can be translated into one or more control rules whose consequents propose the appropriate operator.

  > In most shaping and planning operations, cutting is done in one direction only. The return stroke represents lost time. Thus these processes are slower than milling and broaching, which cut continuously. On the other hand, shaping and planning use single-point tools that are less expensive, are easier to sharpen, and are conducive to quicker set-ups than the multiple-point tools of milling and broaching. This makes shaping or planning often economical to machine one or a few pieces of a kind. ([Doyle, 1969], p. 597).

- Binding preferences: suppose PRODIGY is asked now to solve a problem in the logistics domain. The goal is to move Package3 from the airport to the post office, and to do this two trucks, Truck1 and Truck2, are available at the airport. Truck1 has a bigger capacity, and therefore is more expensive to use, than Truck2. However Truck2's driver is known to be less reliable that Truck1's driver. To achieve the goal, PRODIGY picks operator UNLOAD-TRUCK. Then the next decision, a bindings choice, is which truck to use. If the strategy is to keep the cost low, Truck2 should be preferred, but if reliability of the plan is the major factor Truck1 should be used in spite of its greater cost. Note that the choice of bindings may influence the operators that follow. Section 5.2.1 presents this example in detail.

5 Work to Date

In order to establish the feasibility of the research proposed in this document, initial investigations were conducted on acquiring control rules to enhance plan quality. In this section we describe the prototype implemented to date. The viability of the approach and the use of the prototype are illustrated with two examples from the logistics transportation domain.

5.1 Semi-Automated (Interactive) Acquisition of Quality-Enhancement Control Rules

As mentioned in Section 2.6.1, all the methods to acquire control knowledge for PRODIGY focus on improving the planner's efficiency by reducing the search space. The work proposed here intends to acquire control knowledge that guides search in order to improve the quality of the solutions obtained by the planner. The metrics of solution quality on which we focus are the plan length and the cost of executing the plan. So far we have concentrated on the acquisition of search control rules. These rules provide guidance during search to make local decisions. However it is not clear yet whether these local decisions will be enough to lead the problem solver towards better solutions, or PRODIGY's control structure will have to be extended. The control rules encode the knowledge extracted from a domain expert: knowledge about why a solution is better than other, and about how to modify a solution to improve its quality. We do not claim that these rules, or more generally, control knowledge, will necessarily guide the planner to find optimal solutions [Simon, 1981], but that the quality of the plans will incrementally improve with experience, as the planner sees new interesting problems in the domain and interacts with the domain expert.

Table 2 shows the process we have implemented to date in order to acquire control knowledge from an expert. The next subsections describe its steps in more detail.

---

1. Run PRODIGY with the current set of control rules and obtain a solution S_p, or alternatively set S_p empty.
2. If S_p,

   Show S_p to the expert.
   Expert provides new solution S_e by modifying S_p: adding, removing, or interchanging plan operators, or modifying their bindings.

   Otherwise (no initial solution S_p is available)

   Expert provides completely new solution, S_e.
3. Test S_e. If it solves the problem, continue. Else go back to step 2.
4. Compute the partial order $\mathcal{P}$ for S_e.
5. Determine the set of decision points DP in the problem solving trace where control knowledge is required to obtain a solution S'_e that satisfies $\mathcal{P}$.
6. Run PRODIGY stopping at each of the points in DP and acquire control knowledge from the expert to make the right choice.
7. If expert still wants to improve the current solution S'_e,

   Set $S_p \leftarrow S'_e$.
   Go to step 2.

   Otherwise terminate.

Table 2: The Basic Process for Incremental Control Knowledge Acquisition. The expert provides a solution he considers good either by modifying the one proposed by the planner, or by enumerating all the operators. Then the system acquires control knowledge that will guide the planner towards the better solution.

5.1.1 Getting a Solution from the Expert

In step 2 the expert provides PRODIGY with a solution S_e that he considers better than the one PRODIGY obtains with the control knowledge currently available (we call the latter S_p). The purpose is to acquire control knowledge so that should PRODIGY see the same problem again, the better solution would be obtained. The expert may use S_p as a basis to build S_e. However in some cases having the planner find just one solution (S_p) is very expensive if control knowledge is not sufficient, and therefore we provide the capability that the expert can build a solution (S_e) from scratch to start with.

In the domains we have experimented with, solutions tend to be long and the operators include a lot of parameters. It seems useful to provide the expert with good tools to input a solution. The current interface allows him to build a solution by putting together some or all of S_p's operators and adding new operators, if needed, by typing the operator name and all its bindings. We intend to extend the interface to allow edition of the old solution, propose default values for the parameters in the new operators, and facilitate step-wise execution of the plan, among other things.

Note that it is possible that an expert would prefer to solve the problem using operators not yet in the domain specification. The APPRENTICE system [Joseph, 1992] acquires such base-level or *domain* knowledge for PRODIGY.

5.1.2 Determining Where Control Knowledge Is Needed

In step 5, the system finds the decision points where PRODIGY's current control knowledge needs to be modified so the expert's solution is found. In fact, the planner searches for *any* solution that satisfies the partial order obtained from S_e. A partial order, or partially ordered plan, encodes a set of solutions, or totally ordered plans. All of them have the same operators, and satisfy a set of ordering constraints. One step op_i precedes another step op_j in the partial order if and only if op_i adds a precondition of op_j, or op_j deletes a precondition of op_i. The partial order of a plan can be obtained efficiently [Veloso *et al.*, 1990] in negligible time compared to the time needed to generate the totally ordered plan. We are assuming that the quality of the plan is the same for all the plans encoded in the total order, since they all have the same operators. For example, if two packages have to be loaded in the same truck, and we ignore package sizes and truck capacities, the order in which the packages are loaded is irrelevant with respect to plan quality. By allowing any solution in the partial order, we reduce the amount of control knowledge that needs to be acquired. However this heuristic would not be valid in domains where constraints on the order of operators, other than the ones encoded in the partial order, influence the plan quality.

DP is the set of decision points for which control knowledge needs to be incorporated. By looking to the trace for S_p and to the expert's plan S_e, the system proposes a set of possible candidate points where a different decision should be made, and the preferred decisions themselves. PRODIGY starts to solve the problem again following the first of those recommendations. If it does not lead to S_e, it is discarded and another one tried in turn. For each recommendation this process is repeated recursively: when the planner realizes that the current path will not lead to a solution that satisfies the partial order, the path is abandoned, candidates for wrong decision points are found, and PRODIGY backtracks to one of those candidates and tries a different path. [9] This process can be seen as searching on the space of decision points in the trace, until a set, DP, is found that leads to a solution S'_e.

5.1.3 Acquiring the Relevant Knowledge

If PRODIGY runs following the recommendations in DP, it will obtain plan S'_e. Each recommendation contains a decision point, the alternative that leads to S'_e, and a reason why that alternative should be chosen (for example, an operator ordering would be violated otherwise, or a different binding would be chosen instead that the one the expert proposed). In step 6 PRODIGY restarts problem solving again stopping at each of those decision points, and requesting the expert's advice. This advice is translated into search control rules that will fire making the appropriate choices. Note that the alternative to take is known at this point and it becomes part of the rule's consequent. The rule's preconditions, or left-hand side, encode the situations in which that decision should be made. Some of the preconditions can be automatically extracted from the current meta-state. These are the current goal, and also the current operator if we are dealing with a

[9] All this process is performed through domain-independent rules specially designed for this task.

bindings decision. Also, if the reason why the system detected that a different alternative should be taken relies on other available knowledge, that knowledge is added to the rule.[10] At this point we need to acquire from the domain expert the knowledge that justifies the decision. We want to rely as less as possible on the expert's knowledge about the planner itself. In particular it is hard even for PRODIGY experts to explain, for example, why the reason for the inefficient solution is a goal ordering at a particular point of the trace. On the other hand we need to extract knowledge that can be operationalized, i.e. transformed into control rule preconditions that match the current meta-state.

The approach followed here is to get the expert's knowledge in terms of the current state, top level goals, and possibly the pending goals. To extract this knowledge, the expert is asked for the reasons as possible in terms of the proposed solution: why he/she decided to use a particular operator or to change the order of some operators with respect to their order in the planner's solution. That information is captured in the recommendation. The expert interacts by pointing to menu items.

Once the items that will become part of the rule preconditions are chosen, they need to be generalized. At this point generalization means simply to replace constants by variables. However it is not a trivial process. All the objects have types in PRODIGY and therefore new preconditions should be added that constrain the values that variables may take. In some cases the values are so specific that the constants have to remain. These constraints may be extracted from the type hierarchy, but in some cases the expert may decide to specify the set of values that the variable can take as a combination of different types (for example, any drill bit that is not specifically used to make a spot hole can be expressed in PRODIGY's language as `(comp DRILL-BIT SPOT-DRILL)`).

It is not hard to imagine that the control rules acquired from the expert in a particular problem solving context may be too general or too specific. These rules then provide inappropriate advice when the situation is slightly different.

5.1.4 Variations to the Algorithm

The algorithm in Table 2 may be modified to rely less in the domain expert. Table 3 reflects these changes. Instead of prompting the expert for another solution, PRODIGY can run in its multiple-solution mode until it finds a new solution. Then the system may decide by itself if a given solution is better than other or else it can ask the expert for his preference. Then the algorithm proceeds as before.

We have tested this variation in the transportation domain and run into practical problems: PRODIGY may require an impractically large amount of search before it finds the next substantially better solution. Before it backtracks to a point that would lead to an interestingly different solution, it tries many lesser variations of the current one, by selecting different alternatives near the leaves of the search tree. We would like to experiment with different schemas to solve this difficulty, such as imposing resource bounds (time or number of nodes), abandoning a path when the partial solution found so far looks worse than S_p [11], or trying backtracking strategies different from chronological backtracking. These heuristics seem very domain dependent and we have not explored them so far.

---

[10] A clear example of this deals with operator ordering (see example `pgh1`): if op_1 has to be applied before op_2 in the expert's solution, and PRODIGY has expanded op_1 already, op_2 cannot be expanded until op_1 has been applied, or else the required ordering cannot be satisfied. Therefore the condition that op_1 is expanded but not applied yet can be automatically added to the rule for rejecting op_2, or even to a rule for rejecting the goal for which op_2 is relevant.

[11] However we cannot decide in general that the final solution in that path is going to be worse than S_p.

1. Run PRODIGY with the current set of control rules and obtain a solution S_p. If S_p is good enough, terminate.
2. Find next solution S_n. If no one is found, terminate.
3. Decide whether S_n is better than S_p using prior knowledge or asking of the expert.
4. If S_n is not better than S_p, go back to Step 2.
5. Determine the set of decision points DP in the problem solving trace where control knowledge is required to obtain a solution S_n.
6. Run PRODIGY stopping at each of the points in DP and acquire control knowledge from the expert to make the right choice.
7. Set $S_p \leftarrow S_n$.
   Go to step 2.

Table 3: A Modified Version of the Basic Process. In this case PRODIGY runs in its multiple solutions mode and finds the better solution by itself, instead of having the human expert input it. It needs however knowledge about why a solution is better than other.

5.2 Detailed Examples

In this section we present two examples to illustrate the viability of quality solution enhancement using control rules, and the use of the method described in Section 5.1. Section 5.2.1 presents a simple example in which rules control the use of resources in the plan according to both reliability and cost. The example in Section 5.2.2 illustrates how choosing the right goal interleaving leads to better plans, in particular shorter plans.

5.2.1 A Simple Example: Choosing Different Resources

In this problem the goal is to move `package3` from the airport (`pgh-airport`) to the post office (`pgh-po`), and to do this there are two trucks available at the airport, `pt1` and `pt2`. In addition there are other facts known about the trucks and the kind of solution preferred, that are not used by the operators. Figure 6 summarizes this problem.

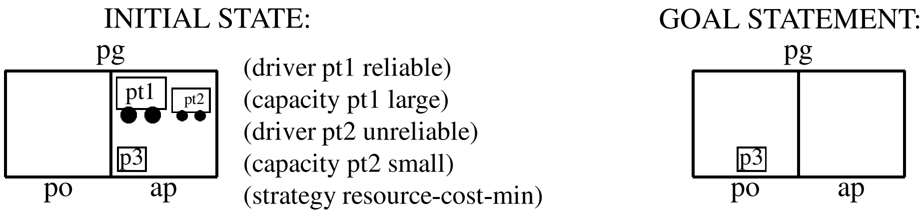

Figure 6: Initial State and Goal Statement for Problem `cap`. `p3` has to be moved from the airport `pgh-ap` to the post-office `po`. Using truck `pt2` is cheaper because its capacity is smaller, but using truck `pt1` produces a more reliable solution.

The planner automatically comes up with this solution, that uses `pt1`:

```
<load-truck package3 pt1 pgh-airport>
<drive-truck pt1 pgh-airport pgh-po>
<unload-truck package3 pt1 pgh-po>
```

but the expert prefers to use truck `pt2` to solve this problem, since the strategy is to reduce the resource cost. `pt2` is cheaper since its capacity is smaller.

A dialog is initiated once the planner solves the problem. First the new solution is built taking into account what the expert proposes, use `pt2` instead. Then the knowledge acquisition module queries the expert in a way that requires no knowledge of PRODIGY internals:

```
Why is not the solution good enough?
  1. Unnecessary steps.
  2. Resources too costly.
  3. Reliability problem.
Answer: 2

This is the initial solution obtained by the planner:

        1. <load-truck package3 pt1 pgh-airport>
        2. <drive-truck pt1 pgh-airport pgh-po>
        3. <unload-truck package3 pt1 pgh-po>

Which resource do you want to replace? pt1
New value: pt2
In which operators (optional):

Testing the expert solution...

 The plan given is a solution to the current problem.
```

Now the system has to find out why `pt2` is a better alternative:

```
These are the diffs between the two objects, pt1 and pt2:
        1. (capacity pt1 large)
        2. (driver pt1 reliable)
        3. (capacity pt2 small)
        4. (driver pt2 unreliable)
Select one or more: 1 3
```

With this information, a new control rule is built that decides about the resource to use:

```
Run problem again to acquire binding control rules.

  2 n2 (done)
  4 n4 <*finish*>
  5   n5 (at-obj package3 pgh-po)
There is a new binding decision stored at this node.

This is the rule acquired:
(CONTROL-RULE ACQ32
 (IF
  (AND (TRUE-IN-STATE (STRATEGY RESOURCE-COST-MIN))
   (TRUE-IN-STATE (CAPACITY <TRUCK1> LARGE))
   (TRUE-IN-STATE (CAPACITY <TRUCK2> SMALL))
   (DIFF <TRUCK1> <TRUCK2>)))
 (THEN PREFER BINDINGS ((<TRUCK> . <TRUCK2>)) ((<TRUCK> . <TRUCK1>))))

Testing the control rule:
The lhs would match with these bindings:
```

```
(((<TRUCK2> . #<P-O: PT2 truck>) (<TRUCK1> . #<P-O: PT1 truck>)))

and this is the result of firing the rule:

   Prefer ((<TRUCK> . #<P-O: PT2 truck>))
   over ((<TRUCK> . #<P-O: PT1 truck>)).

  7   n7 <unload-truck package3 pt2 pgh-po> [1]
  8     n8 (inside-truck package3 pt2)
 10     n10 <load-truck package3 pt2 pgh-airport> [1]
 11     n11 <LOAD-TRUCK PACKAGE3 PT2 PGH-AIRPORT> [1]
 12     n12 (at-truck pt2 pgh-po)
 14     n14 <drive-truck pt2 pgh-airport pgh-po>
 15     n15 <DRIVE-TRUCK PT2 PGH-AIRPORT PGH-PO>
 15   n16 <UNLOAD-TRUCK PACKAGE3 PT2 PGH-PO>
Achieved top-level goals.

Solution:
        <load-truck package3 pt2 pgh-airport>
        <drive-truck pt2 pgh-airport pgh-po>
        <unload-truck package3 pt2 pgh-po>
```

Note that in this example it is enough with replacing the decision at `n7`. The other binding decisions are determined by this one.

5.2.2 Finding the Right Goal Interleaving

In this problem the key to find a good solution is on interleaving the work in both top-level goals. Plan length is the metric of plan quality considered (although in this case the shorter solution makes in addition a more efficient use of the available resources). The example shows the use of the algorithm we presented in Section 5.1. The problem goal is to move a package, `p1`, from `pgh-po` to `bos-po`. In the initial state there is a truck at each post office, and two planes at `pgh-airport`. Figure 7 shows the initial state and goal statement for this problem.

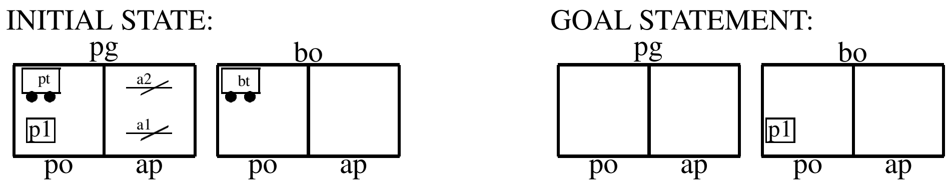

Figure 7: Initial State and Goal Statement for Problem `pgh1`. The problem goal is to move a package, `p1`, from `pgh-po` to `bos-po`. In the initial state there is a truck at each post office, and two planes at `pgh-airport`.

PRODIGY solves the problem, using some default control knowledge and obtains an inefficient solution (I will refer to this as the planner's solution):

```
1. <load-truck package1 pgh-truck pgh-po>
2. <fly-airplane airplane1 pgh-airport bos-airport>   Airplane flies unnecessarily.
3. <drive-truck pgh-truck pgh-po pgh-airport>
```

```
4. <fly-airplane airplane1 bos-airport pgh-airport>   Airplane returns.
5. <unload-truck package1 pgh-truck pgh-airport>
6. <load-airplane package1 airplane1 pgh-airport>     Once package1 is loaded,
7. <fly-airplane airplane1 pgh-airport bos-airport>   fly makes sense.
8. <unload-airplane package1 airplane1 bos-airport>   Package arrives to boston,
9. <drive-truck bos-truck bos-po bos-airport>
10. <load-truck package1 bos-truck bos-airport>       and goes by truck to bos-po.
11. <drive-truck bos-truck bos-airport bos-po>
12. <unload-truck package1 bos-truck bos-po>
```

Note that `airplane1` flies unnecessarily. Now the system presents to the expert the solution obtained by the planner and prompts him for a new and better solution. He may vary the order of the steps, remove steps, or add steps, or the solution can be completely different. In fact, the expert could provide a solution without having the planner solve the problem first. This is useful when the planner gets lost searching thousands of nodes because it lacks the appropriate control knowledge. However it is often easier to critique or modify a solution – therefore the planner usually offers one to the expert as a starting point. Once the expert gives the solution it considers better, PRODIGY first tests it to make sure that it will actually solve the given problem.

```
Enter sequence of operator numbers with the new ordering of steps:
(put a * for a diff operator) 1 3 5 6 7 8 9 10 11 12

The expert solution is:

1. #<LOAD-TRUCK [<OBJ> PACKAGE1] [<TRUCK> PGH-TRUCK] [<LOC> PGH-PO]>
3. #<DRIVE-TRUCK [<TRUCK> PGH-TRUCK] [<LOC-FROM> PGH-PO] [<LOC-TO> PGH-AIRPORT]>
5. #<UNLOAD-TRUCK [<OBJ> PACKAGE1] [<TRUCK> PGH-TRUCK] [<LOC> PGH-AIRPORT]>
6. #<LOAD-AIRPLANE [<OBJ> PACKAGE1] [<AIRPLANE> AIRPLANE1] [<LOC> PGH-AIRPORT]>
7. #<FLY-AIRPLANE [<AIRPLANE> AIRPLANE1] [<LOC-FROM> PGH-AIRPORT] [<LOC-TO> BOS-AIRPORT]>
8. #<UNLOAD-AIRPLANE [<OBJ> PACKAGE1] [<AIRPLANE> AIRPLANE1] [<LOC> BOS-AIRPORT]>
9. #<DRIVE-TRUCK [<TRUCK> BOS-TRUCK] [<LOC-FROM> BOS-PO] [<LOC-TO> BOS-AIRPORT]>
10. #<LOAD-TRUCK [<OBJ> PACKAGE1] [<TRUCK> BOS-TRUCK] [<LOC> BOS-AIRPORT]>
11. #<DRIVE-TRUCK [<TRUCK> BOS-TRUCK] [<LOC-FROM> BOS-AIRPORT] [<LOC-TO> BOS-PO]>
12. #<UNLOAD-TRUCK [<OBJ> PACKAGE1] [<TRUCK> BOS-TRUCK] [<LOC> BOS-PO]>

Testing the expert solution...

 The plan given is a solution to the current problem.
```

The system currently implemented detects the point(s) at which wrong decisions were made. It may backtrack and try different alternatives, until it solves the problem obtaining the solution proposed by the expert. Actually it obtains a solution that satisfies the partial order obtained from the expert's solution by enforcing dependencies between operator preconditions and previous operator effects. For example, it assumes that if two operators are not ordered in the partial order, their order does not matter. In this case `airplane1` flies unnecessarily due to a wrong goal interleaving after `n26`: achieving goal `(at-airplane airplane1 bos-airport)` should be postponed until the package is inside of the airplane. This was (part of) PRODIGY's trace:

```
 5   n5 (at-obj package1 bos-po)
 7   n7 <unload-truck package1 bos-truck bos-po>
 8     n8 (inside-truck package1 bos-truck)
10     n10 <load-truck package1 bos-truck bos-po> [1] ...goal loop with node 5
10     n11 <load-truck package1 bos-truck bos-airport>
11       n12 (at-obj package1 bos-airport)
13       n14 <unload-truck package1 bos-truck bos-airport> ...goal loop with node 8
13       n16 <unload-airplane package1 airplane1 bos-airport> [1]
```

```
 14        n17 (inside-airplane package1 airplane1)
 16        n19 <load-airplane package1 airplane1 pgh-airport>
 17          n20 (at-obj package1 pgh-airport)
 19          n22 <unload-truck package1 pgh-truck pgh-airport>
 20            n23 (inside-truck package1 pgh-truck)
 22            n25 <load-truck package1 pgh-truck pgh-po> [1]
 23            n26 <LOAD-TRUCK PACKAGE1 PGH-TRUCK PGH-PO> [3]
 24        n27 (at-airplane airplane1 bos-airport) [2]
 26        n29 <fly-airplane airplane1 pgh-airport bos-airport>
 27        n30 <FLY-AIRPLANE AIRPLANE1 PGH-AIRPORT BOS-AIRPORT> [2]
```

The system points the crucial decision in the current example, namely:

```
-- Wrong decision made after node
 #<APPLIED-OP-NODE 26 #<LOAD-TRUCK [<OBJ> PACKAGE1] [<TRUCK> PGH-TRUCK] [<LOC> PGH-PO]>>.

Goal #<AT-AIRPLANE AIRPLANE1 BOS-AIRPORT> was chosen.
Goal(s) #<AT-TRUCK PGH-TRUCK PGH-AIRPORT> could have been chosen instead.
```

To avoid making the wrong decision, PRODIGY has to acquire control knowledge. Our goal is to extract from the expert the relevant knowledge and express it in the form of control rules. First, this is the rule I came up with when writing by hand the control rule set:

```
(control-rule HAND-CODED  ;; reject moving the airplane if loading is needed
  (if (and (candidate-goal (at-airplane <airplane> <new-loc>))
           (known (at-airplane <airplane> <loc>))
           (expanded-operator (LOAD-AIRPLANE <obj> <airplane> <loc>))))
  (then reject goal (at-airplane <airplane> <new-loc>)))
```

We want to rely as less as possible on the expert's knowledge about PRODIGY. In particular it is hard even for PRODIGY experts to explain why the reason for the inefficient solution is a goal ordering at a particular point of the trace. The code that detects the wrong decision point would help by itself quite a lot even to PRODIGY experts.

The approach followed here is to get the expert's knowledge in terms of the current state, top level goals, and possibly the pending goals. To extract this knowledge, the expert is asked for the reasons why he decided to use a particular operator or to change the order of some operators with respect to their order in the planner's solution. This is the rule that could be learned for this problem based on that information:

```
(control-rule TEST2
  (if (and (candidate-goal (at-airplane <airplane> <dest-airport>))
           (expanded-operator (load-airplane <obj> <airplane> <orig-airport>))
           (diff <dest-airport> <orig-airport>)

           (expanded-goal (at-obj <obj> <dest-po>))

           (known (inside-truck <obj> <truck>))
           (known (at-truck <truck> <orig-po>))

           (known (at-airplane <airplane> <orig-airport>))
           (same-city <orig-airport> <orig-po>)))
  (then reject goal (at-airplane <airplane> <dest-airport>)))
```

The first three meta-predicates are obtained from the current meta-state. In particular, as the operator ordering violated at the node was that `<fly-airplane airplane1 pgh-airport bos-airport>` should be applied after `<load-airplane package1 airplane1 pgh-airport>`, once `load-airplane` has been chosen, at-airplane cannot be expanded until load-airplane is applied so the order is not violated. This gives us the first three preconditions and also allows us to write a goal reject rule (instead of a goal preference rule).

The rest of the preconditions come from the information extracted from the expert. He was presented with the current state, and the top-level goals, namely:

```
THE CURRENT STATE IS:
*(inside-truck package1 pgh-truck)
*(at-airplane airplane1 pgh-airport)    (at-airplane airplane2 pgh-airport)
*(at-truck pgh-truck pgh-po)            (at-truck bos-truck bos-po)
 (part-of bos-truck boston)             (part-of pgh-truck pittsburgh)
 (loc-at pgh-po pittsburgh)             (loc-at pgh-airport pittsburgh)
 (loc-at bos-po boston)                 (loc-at bos-airport boston)
 (same-city bos-po bos-airport)        *(same-city pgh-po pgh-airport)
 (same-city pgh-airport pgh-po)         (same-city bos-airport bos-po)

THE TOP LEVEL GOALS ARE:
*(at-obj package1 bos-po)
```

The items marked with * are the ones the expert picked, and that appear in the rule after generalization.

Obviously the state can be very large with many irrelevant features. Remember that we are asking the expert why he preferred a given operator, or a given ordering between two operators. In order to focus on the relevant state features, we could make use of these operators.

Note that the rule TEST2 has more conditions that HAND-CODED, the one I wrote initially by hand. Note also that HAND-CODED is more general (it applies to more situations). We will have to analyze if there is a way to get rid automatically of some of the preconditions of TEST2.

The goal mentioned in (expanded-goal (at-obj <obj> <dest-po>)) is an ancestror in the subgoaling structure of the operator mentioned in the second precondition <load-airplane <obj> <airplane> <orig-airport>>. But if we remove it the rule may be overgeneral; it would fire also if (at-obj <obj> <dest-po>) is not an expanded goal, for example when the top level goal is (inside-airplane <obj> <airplane>). The generalization may be good, and in addition we could ask the user to guide it:

```
You mentioned that a relevant goal was:
 (at-obj package1 bos-po)

Is it ok if the expanded goal is instead one of these:
 (inside-truck package1 bos-truck)
 (at-obj package1 bos-airport)
 (inside-airplane package1 airplane1)
```

where the goals were obtained by going up the search tree. In this case the answer is yes, because the only thing that matters is that load-airplane, already expanded, should be applied before working on fly-airplane.

5.3 Assumptions and Limitations

The approach presented here makes some assumptions about the way control knowledge is available and can be expressed to guide the problem solver. First, we assume that global strategies to obtain good plans can be encoded as control rules that guide local decisions. Also we are assuming that an optimal decision is not required at each point, but rather that a qualitative model of the utility of the actions can be acquired and expressed as a set of control rules. A third assumption is that the factual knowledge, or domain knowledge, is correct and complete. We only worry about acquiring search control knowledge. Other work on the PRODIGY architecture [Carbonell and Gil, 1990, Joseph, 1992] has focused on the acquisition and refinement of domain knowledge. Lastly, this model assumes that experts can give reasons for specific strategic decisions, rather than simply intuit correct sequences of decisions.

6 Learning Quality-Enhancing Control Rules

This section presents the work currently in progress for the automatic acquisition of quality-enhancing control knowledge. Our goal is to have a system that improves over experience the quality of the plans it generates by learning control knowledge to guide the search. Figure 8 shows the architecture of the current system. The system is given a domain theory (operators and inference rules) and a domain-dependent objective function that describes the quality of the plans. It is also given problems to solve in that domain. The system learns control knowledge by analyzing the problem-solving episodes, in particular comparing the search trace for the planner solution given the current control knowledge, and another search trace corresponding to a better solution (*better* according to the evaluation function). The latter search trace is obtained by letting the problem solver search further until a better solution is found, or by asking a human expert for a better solution and then producing a search trace that leads to that solution. The control knowledge learned in this way leads future problem solving towards better quality plans. In some sense there is a change of representation from the knowledge about quality encoded on the objective function into knowledge that the planner may use at problem solving time. We do not claim that this control knowledge will necessarily guide the planner to find optimal solutions, but that the quality of the plans will incrementally improve with experience, as the planner sees new interesting problems in the domain.

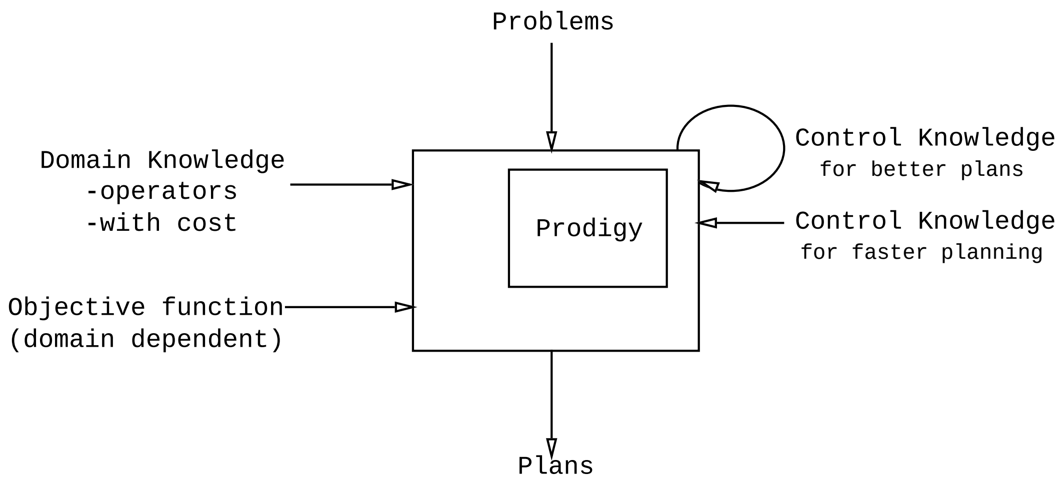

Figure 8: Architecture of a system to learn control knowledge to improve the quality of plans.

Section 2 presented a taxonomy of quality metrics. A goal of our work is to validate and expand that taxonomy. We are exploring in more depth some parts of the taxonomy, giving precise definitions of quality metrics for them. In particular we focus on reducing the execution cost of the plans generated, taking cost as sum of the execution cost of plan operators. We also expect to explore the possibility of using cases as control knowledge and of derivational analogy to extract and store those cases. Currently the derivational analogy module of PRODIGY [Veloso, 1992] stores past problem-solving experiences as cases, and reuses them to solve similar problems, obtaining a considerable improvement in problem solving efficiency. The use of one or more past cases to solve the current problem may lead to shorter plans, as reported in [Veloso, 1992]. This was a surprising result but not the focus of the work. Note in fact that if the solution stored to solve a problem was not a good one, it may be reused to solve subsequent problems without trying to find a better solution. The use of this approach to learn quality-enhancing control rule may affect the similarity metric to find relevant cases, and the language to express the justifications, in order to capture guidance about why the proposed solution is good.

We plan to evaluate this work at each phase by testing empirically how plan quality improves over experience, as the number of problems seen by the system increases. The evaluation will be done for two complex domains, namely those presented in Section 4.1 with sets of randomly generated problems in each domain. We also want to study if the addition of quality-enhancing control knowledge degrades problem solving efficiency, enhances it, or is orthogonal. If the first alternative is true, we need to establish trade offs between planning efficiency and plan execution efficiency.

7 Related Work

Although there have been a number of systems that learn control knowledge for planning systems, most of them are oriented towards improving search efficiency. There has not been much research done on either improving the quality of plans or acquiring control knowledge from human experts. In this section we present some work on planning systems that worry about solution quality, and also on systems that acquire control knowledge.

7.1 Work on Planning Systems and Plan Quality

Several domain-dependent planners for the process planning domain deal with the quality of plans. Hayes' MACHINIST program [Hayes, 1990] generates plans in a machining process planning domain. In this work the measure of plan quality is solution length. Human machinists often spend a large amount of time in the early planning stages looking over the part specification for feature interactions and exploring the limitations that those features impose on the plan. Machinists have specialized knowledge which helps them to quickly focus on the situations in which interactions are likely. This knowledge is acquired through experience. Hayes analyzed the way features interact and encoded this specialized knowledge in form of rules in the MACHINIST program. This program first constructs a plan that deals with feature interactions, and retrieves from memory a plan to square the part (squaring is getting the raw material into a square and accurate shape with the minimum waste of material). Then these two plans are merged to produce a final plan as short as possible.

SIPP [Nau and Chang, 1985] is a process planning system that produces plans for the creation of metal parts. It utilizes a frame hierarchy to represent problem solving knowledge. In particular, actions have cost slots that contain relative costs derived from actual process costs and shop preferences. The problem solving strategy utilizes a least-cost-first branch and bound algorithm to find the least-cost sequence of processes for making each of the part's machinable surfaces. SIPP selects the least cost manufacturing method for an individual feature, in isolation from considerations about other features, and therefore it does not care to find an overall low-cost plan. SIPP is domain-dependent although they anticipate it will be useful in other domains as well.

Descotte and Latombe's planner [Descotte and Latombe, 1985] is a process planner for metal cutting that uses a constraint satisfaction algorithm. Knowledge is represented by manufacturing rules: the left-hand side consists of conditions about the desired part, the available machines, and/or the machining plan. The right-hand side contains pieces of advice representing technological and economical preferences, and so it encodes knowledge about the quality of the plans. In contrast with PRODIGY, there is no separation in the representation between domain knowledge and search control knowledge. Experts have to assign a weight to each piece of advice according to the importance of its satisfaction. These weights are an extremely condensed representation of a large body of knowledge, and human experts have difficulty in expliciting them. The initial state of a problem contains global pieces of information such as the quality desired for the part. As far as we know this system has not been applied to other domains.

ISIS [Fox and Smith, 1984] is a constraint-directed factory scheduling system. The conflicting nature of constraints in this domain prompted the introduction of constraint relaxation techniques. The problem solving strategy finds a solution that best satisfies the constraints. Some of the constraints are physical and must be satisfied, while others can be seen as approximations of a simple profit constraint. Different constraints have different importance or priority, that may change from order to order. ISIS provides tools with which the user can construct and alter schedules interactively, at different levels of abstraction (ISIS automatically fills the other levels). The tools are in charge of maintaining the consistency of the schedule and identify decisions that result in poorly satisfied constraints. However they do not facilitate the acquisition of new knowledge.

SIPE [Wilkins, 1988] is a domain-independent *classical* planner that reasons with partially ordered plans. It incorporates reasoning about resources, and interleaving of planning and execution. SIPE has been applied to the scheduling of packaging lines at a brewery [Wilkins, 1989]. The problem is to generate plans that meet as many orders as possible, while meeting all the physical constraints, and to minimize the waste from flushing lines. Domain-specific knowledge about search control and the utility of the plans, such as why flushing is/is not needed, is encoded in the operators. To take advantage of existing connections to avoid flushing SIPE relies on the preference order given to the operators. Therefore there is no separation between domain knowledge and control knowledge. According to Wilkins it would be straightforward to implement a best-first search using SIPE's context mechanism to find optimal plans, if a good measure of plan utility is provided by someone, but that is true also for PRODIGY. SIPE offers a domain-independent graphical interface to view the partial plans produced as graphs. This interface allows interactive control of the search by letting the user watch and, when desired, guide and/or control the planning or replanning process. This is useful for debugging and to guide the solution of larger problems that would not be solved in a reasonable amount of time. However, as in the case of ISIS, this interaction mechanism does not facilitate explicitly the acquisition of new knowledge.

SteppingStone [Ruby and Kibler, 1990] heuristically decomposes a problem into simpler subproblems, and then learns to deal with the interactions that arise between the subproblems. The system allows hard-constraints, that must be met and usually outline key aspects of the problem, and soft constraints that measure the quality of a solution, and are usually real-valued. The system learns to optimize soft constraints as well as solve hard constraints. Problem solving is viewed as moving to states where the goal is successively closer to completion. EASe [Ruby and Kibler, 1992] is a generalization of SteppingStone, in which additional problem solving knowledge is learned and stored in the form of episodes (or cases). These episodes encode exceptions to the core knowledge. EASe has been applied to design problems in which an initial solution obtained by an application system is improved by using the episodes.

In Section 3 we analyzed the relationship between goal interactions and plan quality. Wilensky's planner [Wilensky, 1983] takes advantage of this relationship. He makes an analysis of the different types of goal interactions and develops meta-planning mechanisms that deal with them. When a goal overlap, or positive goal interaction between a planner's goals, occurs, his planner is able to carry out an action that is in the service of a number of goals at once. This might involve executing a single plan that simultaneously fulfills several goals, achieving a goal that serves more than one purpose, or employing a plan that is worthwhile only when a sufficient number of similar goals is involved. External positive goal interactions may also occur in which two or more planners have similar goals (they are termed *goal concord*). In Wilensky's system then a goal overlap situation provides an opportunity to achieve goals more economically than they could be achieved otherwise, and the planner prefers efficient plans over inefficient ones. This principle would appear to be the underlying justification for a number of processes incorporated in other planning systems. For example, several of NOAH's critics [Sacerdoti, 1977] including "use existing objects," "eliminate redundant preconditions," and "optimize disjuncts" are motivated by this idea and correspond to particular kinds of goal overlap situations.

Some existing domain-independent planning systems solve multiple-goal problems by developing separate plans for the individual goals, combining these plans to form a naive plan for the conjoined goal, and then performing optimizations to yield a better combined plan [Nau *et al.*, 1990]. However they restrict the types of goal interactions that may happen. In this context, the quality of a plan is only considered as far as dealing with and taking as much advantage as possible of goal interactions. A similar mechanism is also used by some domain-dependent planners [Hayes, 1990, Nau, 1987].

Several systems perform plan debugging as their problem solving strategy [Sussman, 1975, Hammond, 1987, Simmons, 1988]. They employ heuristic rules to generate an initial hypothesis and then debugging if the hypothesis is incorrect. Therefore they fix planning failures (not execution failures). An example is the Generate, Test and Debug paradigm [Simmons, 1988] in which the debugger analyzes causal explanations for why a bug arises and fixes it by replacing those assumptions. The debugger is only used if the heuristic generator produces an incorrect hypothesis. In contrast our planner generates correct plans and our goal is not to fix them but to improve their quality. Our approach does not perform post-facto modification of the plans, but analyzes the problem solving process to extract knowledge that will guide the problem solver towards better solutions.

The problem of finding optimal plans has been attacked by decision theorists. However this problem is computationally very expensive. Simon introduced the idea of "satisficing" [Simon, 1981] arguing that a rational agent does not always have the resources to determine what the optimal action is, and instead should attempt only to make good enough, to satisfice. Some current work on planning (for example [Pollack, 1992]) is about the tradeoff between getting around to acting, and spending enough time thinking. Such resource-bounded reasoning leads to suboptimal behavior. In our work we do not consider the tradeoff between acting and planning time. We acknowledge the computational cost of finding the optimal behavior and do not claim that the acquired control knowledge will necessarily guide the planner to optimal plans, but that plan quality will improve incrementally over experience as the planner sees new interesting problems and interacts with the human expert. The framework presented in [Feldman and Sproull, 1977] incorporates decision theory into planning. The utility function of decision theory is used to decide which strategy to use to achieve a goal, taking into account such factors as reliability, the complexity of the strategy, and the value of the goal. It allows also to improve an existing plan prior to its execution to increment its utility. The costs and utilities of each plan step has to be expressed as numbers, and these numbers have to be obtained in some way. We believe that this limits the range of quality measures that can be expressed. In addition, it seems hard to encode the expert knowledge into these values. On the other hand in this framework it is easier to deal with conflicts among measures. There is also a body of recent work on methods to choose and/or learn optimal policies of action. Some examples are reinforcement learning (e.g. [Lin, 1992]), dynamic programming, and real-time A^* [Korf, 1988]. However these methods have been applied to more reactive models and not so much to solve complex planning problems.

The complexity of the problem of finding optimal solutions in the blocks world is analyzed in [Gupta and Nau, 1991] and [Chenoweth, 1991]. In both cases optimal solutions are shortest-length plans. [Korf, 1985] analyzes how the use of macro-operators in problem solving affects solution quality, but only from the point of view of solution length. In particular in the Eight Puzzle and the 3x3x3 Rubik's Cube domains the solution lengths are approximately equal to or less than those of human strategies.

7.2 Work on Acquisition of Control Knowledge

ASK [Gruber, 1989] is probably the piece of work on knowledge acquisition most related to ours. ASK is an interactive knowledge acquisition tool that elicits strategic knowledge from people in the form of justifications for action choices, and generates strategy rules that operationalize and generalize the expert's advice. Gruber distinguishes between control knowledge and strategic knowledge. Control knowledge refers to knowledge used to decide what to do next. Strategic knowledge is a subset of control knowledge, and it is used by an agent to decide what action to perform next, when actions have consequences external to the agent. Search-control knowledge is used to choose internal actions that increase the likelihood of reaching a solution state and improve the speed of computation.

ASK is integrated with the MU architecture, used typically for heuristic classification. MU organizes the factual knowledge as a symbolic inference network, where inferences are propagated from evidence to hypotheses by local combination functions. ASK acquires strategy rules, inspired by the meta-rules in NEOMYCIN, that map strategic situations to sets of recommended actions. There are three categories of strategy rules: focus rules (propose a set of possible actions), filter rules (prune actions that violate constraints), and selection rules (prefer a subset of remaining actions). These rules are similar to PRODIGY's select, reject and prefer control rules. ASK's knowledge acquisition process has five steps:

1. eliciting the user's critique by presenting a list of chosen actions and ask what should have been done differently.

2. credit assignment analysis: check how existing rules matched and determine the requirements for a new rule.

3. eliciting justifications, by displaying features of the knowledge base and allowing the user to choose some. New features can be acquired if needed if they are analog to existing features.

4. generating and generalizing a strategy rule, possibly with user's guidance in the generalization process.

5. verifying a rule, by presenting the user with a paraphrased description of the rule and its consequences.

ASK's success relies on the relevant control features being defined in advance or are analogous to existing ones, and on the user understanding of the opportunistic control model that underlies the strategy-rule representation. These limitations are somehow shared by our approach. However due to ASK's control model it is awkward to acquire goal directed strategies: ASK's planning method is purely reactive with no projection (lookahead) and no possibility to undo actions. In addition ASK does not focus on plan quality.

The system presented in [Golding *et al.*, 1987] acquires general search-control knowledge for Soar from a human advisor. In general when Soar has to decide among several courses of action it reaches an impasse and generates automatically a subgoal and searches to solve it. Instead this system requests advice from a human to solve an impasse. This request can take one of two forms. In the first mode, direct advice, the advisor tells Soar which alternative to select. There is no operationalization problem as the system forces the expert to name a particular operator. In the second mode the advisor supplies a problem within Soar's grasp that illustrates what to do. In both cases, after the problem is solved, Soar's learning mechanism, chunking, transforms the advice in search-control knowledge. Chunking takes care of the generalization. The system may learn from failure if the advice is incorrect. There is no clear distinction between the knowledge acquired to reduce search (by avoiding subgoaling to solve an impasse) and the knowledge that guides search to a particular solution. Chunking captures the reasons why the advice is correct but it is not clear how it may justify why one path is better than other in terms of the quality of the solution.

7.3 Other Work on Knowledge Acquisition for PRODIGY

Section 2.6.1 presented briefly the work done in the context of the PRODIGY architecture to speed up the problem solving process. Our work does not focus on speed-up learning, but on improving PRODIGY's solution quality. In addition these systems are fully automated and acquire control knowledge by introspection, while we interact with a external source, the human expert.

Other work in our project has focused on the acquisition of factual, or domain, knowledge. APPRENTICE [Joseph, 1992] performs knowledge acquisition of domain knowledge through a graphical interface. It provides also tools to view plan execution. Work on learning by experimentation [Carbonell and Gil, 1990, Gil, 1992] focus on the automatic refinement of incomplete operators by performing directed experimentation on the environment. However none of this work deals with the problem of acquiring control knowledge that improves the quality of PRODIGY's solutions with experience.

8 Conclusion

In this section we present the expected contributions of this work. This is the first piece of work that explores and taxonomizes quality metrics for planning systems. The analysis is taken into practice in the form of control knowledge that guides the planner's search towards greater quality solutions according to a given plan evaluation function. This knowledge can be learned from the system's problem solving experience. We believe that both knowledge about plan quality and its automated acquisition from problem solving experience are key factors for planning systems that move towards more complex and realistic domains.

9 Acknowledgements

We thank the members of the first author's thesis committee (Tom Mitchell, Reid Simmons, Manuela Veloso, and Martha Pollack) for many fruitful discussions and feedback on previous versions of this document. Xuemei Wang and Yolanda Gil gave provided comments on previous drafts, and Manuela Veloso suggested many ideas on the relationship between plan quality and goal interactions. Finally many thanks to all the members of the PRODIGY project for their technical and personal support.

References

[Carbonell and Gil, 1990] J. G. Carbonell and Y. Gil. Learning by experimentation: The operator refinement method. In *Machine Learning: An Artificial Intelligence Approach, Volume III*. Morgan Kaufmann, San Mateo, CA, 1990.

[Carbonell *et al.*, 1992] Jaime G. Carbonell, and The PRODIGY Research Group. PRODIGY4.0: The manual and tutorial. Technical Report CMU-CS-92-150, School of Computer Science, Carnegie Mellon University, June 1992. Editor: M. Alicia Pérez.

[Chapman, 1987] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378, 1987.

[Chenoweth, 1991] Stephen V. Chenoweth. On the NP-hardness of blocks world. In *Proceedings of Ninth National Conference on Artificial Intelligence*, pages 623–628, Anaheim, CA, July 1991.

[Converse and Hammond, 1992] Timothy M. Converse and Kristian J. Hammond. Learning to satisfy conjunctive goals. In D. Sleeman and P. Edwards, editors, *Machine Learning: Proceedings of the Ninth International Conference (ML92)*, pages 117–122. Morgan Kaufmann, San Mateo, CA., 1992.

[Currie and Tate, 1991] Ken Currie and Austin Tate. O-Plan: the open planning architecture. *Artificial Intelligence*, 52:49–86, 1991.

[Descotte and Latombe, 1985] Yannick Descotte and Jean-Claude Latombe. Making compromises among antagonist constraints in a planner. *Artificial Intelligence*, 27:183–217, 1985.

[Doyle, 1969] Lawrence E. Doyle. *Manufacturing Processes and Materials for Engineers*. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1969.

[Drummond and Currie, 1987] Mark Drummond and Ken Currie. Goal ordering in partially ordered plans. In *Proceedings of the Eleventh International Conference on Artificial Intelligence*, pages 960–965, Detroit, MI, 1987.

[Etzioni, 1990] O. Etzioni. *A Structural Theory of Explanation-Based Learning*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1990. Also appeared as Technical Report CMU-CS-90-185.

[Feldman and Sproull, 1977] Jerome A. Feldman and Robert F. Sproull. Decision theory and artificial intelligence II: The hungry monkey. *Cognitive Science*, 1:158–192, 1977.

[Fox and Smith, 1984] Mark S. Fox and Stephen F. Smith. ISIS – a knowledge-based system for factory scheduling. *Expert Systems*, 1(1), July 1984.

[Gil, 1991] Yolanda Gil. A specification of process planning for PRODIGY. Technical Report CMU-CS-91-179, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, August 1991.

[Gil, 1992] Yolanda Gil. *Acquiring Domain Knowledge for Planning by Experimentation*. PhD thesis, Carnegie Mellon University, School of Computer Science, August 1992. Available as technical report CMU-CS-92-175.

[Golding *et al.*, 1987] Andrew Golding, Paul S. Rosenbloom, and John E. Laird. Learning general search control from outside guidance. In *Proceedings of the Tenth International Conference on Artificial Intelligence*, pages 334–337, Milan, Italy, 1987.

[Gratch and DeJong, 1992] Jonathan Gratch and Gerald DeJong. An analysis of learning to plan as a search problem. In D. Sleeman and P. Edwards, editors, *Machine Learning: Proceedings of the Ninth International Conference (ML92)*, pages 179–188. Morgan Kaufmann, San Mateo, CA., 1992.

[Gruber, 1989] Thomas R. Gruber. Automated knowledge acquisition for strategic knowledge. *Machine Learning*, 4:293–336, 1989.

[Gupta and Nau, 1991] Naresh Gupta and Dana S. Nau. Complexity results for blocks-world planning. In *Proceedings of Ninth National Conference on Artificial Intelligence*, pages 629–633, Anaheim, CA, July 1991.

[Hammond, 1986] Kristian J. Hammond. CHEF: A model of case-based planning. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 267–271, 1986.

[Hammond, 1987] K. Hammond. Explaining and repairing plans that fail. In *Proceedings of the Tenth International Conference on Artificial Intelligence*, Milan, Italy, 1987.

[Hayes, 1990] Caroline Hayes. *Machining Planning: A Model; of an Expert Level Planning Process*. PhD thesis, The Robotics Institute, Carnegie Mellon University, December 1990.

[Howe and Cohen, 1991] Adele E. Howe and Paul R. Cohen. Failure recovery: A model and experiments. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 801–808, Anaheim, CA, 1991. AAAI Press/The MIT Press.

[Joseph, 1992] Robert L. Joseph. *Graphical Knowledge Acquisition for Visually-Oriented Planning Domains*. PhD thesis, Carnegie Mellon University, School of Computer Science, August 1992.

[Kambhampati, 1990] Subbarao Kambhampati. Mapping and retrieval during plan reuse: A validation structure based approaches. In *Proceedings of Eighth National Conference on Artificial Intelligence*, pages 170–175, Boston, MA, July 1990.

[Knoblock, 1991a] C.A. Knoblock. *Automatically Generating Abstractions for Problem Solving*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1991. Also appeared as Technical Report CMU-CS-91-120.

[Knoblock, 1991b] Craig A. Knoblock. Search reduction in hierarchical problem solving. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 686–691, Anaheim, CA, 1991. AAAI Press/The MIT Press.

[Korf, 1985] R. E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.

[Korf, 1988] R. E. Korf. Real-time heuristic search: New results. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, St Paul, MN, 1988. Morgan Kaufmann.

[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.

[Langley and Drummond, 1990] Pat Langley and Mark Drummond. Toward an experimental science of planning. In *Proceedings of the Darpa Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 109–114, San Diego, CA, November 1990.

[Lin, 1992] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8, 1992.

[Minton *et al.*, 1989] Steven Minton, Craig A. Knoblock, Daniel 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, Pittsburgh, PA, 1989.

[Minton, 1988] S. Minton. *Learning Effective Search Control Knowledge: An Explanation-based Approach*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1988. Also appeared as Technical Report CMU-CS-88-133.

[Mitchell *et al.*, 1983] Tom M. Mitchell, Paul E. Utgoff, and Ranan Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristics. In R. S. Michalsky, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning, An Artificial Intelligence Approach*. Tioga Press, Palo Alto, CA., 1983.

[Muscettola and Smith, 1990] Nicola Muscettola and Stephen F. Smith. Integrating planning and scheduling to solve space mission scheduling problems. In *Proceedings of the Darpa Workshop on Innovative Approaches to Planning, Scheduling and Control*, San Diego, CA, November 1990.

[Nau and Chang, 1985] Dana S. Nau and Tien-Chien Chang. Hierarchical representation of problem-solving knowledge in a frame-based process planning system. Technical Report TR-1592, Computer Science Department, University of Maryland, November 1985.

[Nau *et al.*, 1990] Dana S. Nau, Qiang Yang, and James Hendler. Optimization of multiple-goal plans with limited interaction. In *Proceedings of the Darpa Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 160–165, San Diego, CA, November 1990.

[Nau, 1987] Dana S. Nau. Automated process planning using hierarchical abstraction. *Texas Instruments Technical Journal*, Winter:39–46, 1987.

[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, *Machine Learning: Proceedings of the Ninth International Conference (ML92)*. Morgan Kaufmann, San Mateo, CA., 1992. Also available in the Working Notes of the 1992 Spring Symposium Series, Symposium on Knowledge Assimilation.

[Pérez and Veloso, 1993] M. Alicia Pérez and Manuela M. Veloso. Goal interactions and plan quality. In *Preprints of the AAAI 1993 Spring Symposium Series, Workshop on Foundations of Automatic Planning: The Classical Approach and Beyond*, Stanford University, CA, March 1993.

[Pérez, 1991] M. Alicia Pérez. Multiagent planning in Prodigy. Technical Report CMU-CS-91-139, School of Computer Science, Carnegie Mellon University, May 1991.

[Perry, 1990] E. L. Perry. Solution of time constrained scheduling problems with parallel tabu search. In *Proceedings of the Darpa Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 231–239, San Diego, CA, November 1990.

[Pollack, 1992] Martha E. Pollack. The uses of plans. *Artificial Intelligence*, 57:43–68, 1992.

[Rich and Knight, 1991] E. Rich and K. Knight. *Artificial Intelligence*. Mc Graw Hill, New York, second edition, 1991.

[Ruby and Kibler, 1990] David Ruby and Dennis Kibler. Learning steppinstones for problem solving. In *Proceedings of the Darpa Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 366–373, San Diego, CA, November 1990.

[Ruby and Kibler, 1992] David Ruby and Dennis Kibler. Learning episodes for optimization. In D. Sleeman and P. Edwards, editors, *Machine Learning: Proceedings of the Ninth International Conference (ML92)*, pages 379–384. Morgan Kaufmann, San Mateo, CA., 1992.

[Ryu and Irani, 1992] Kwang R. Ryu and Keki B. Irani. Learning from goal interactions in planning: Goal stack analysis and generalization. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 401–407, San Jose, CA, July 1992. AAAI Press/The MIT Press.

[Sacerdoti, 1977] Earl Sacerdoti. *A Structure for Plans and Behavior*. Elsevier, North Holland, New York, 1977.

[Simmons, 1988] Reid G. Simmons. A theory of debugging plans and interpretations. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 94–99, St Paul, MN, 1988. Morgan Kaufmann.

[Simon, 1981] Herbert A. Simon. *The Sciences of the Artificial*. The MIT Press, Cambridge, MA, second edition, 1981.

[Sussman, 1975] Gerald J. Sussman. *A Computer Model of Skill Acquisition*. American Elsevier, New York, 1975. Also available as technical report AI-TR-297, Artificial Intelligence Laboratory, MIT, 1975.

[Tate, 1977] Austin Tate. Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 888–900, Cambridge, MA, 1977.

[Veloso *et al.*, 1990] Manuela M. Veloso, M. Alicia Pérez, and Jaime G. Carbonell. Nonlinear planning with parallel resource allocation. In *Proceedings of the Darpa Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 207–212, San Diego, CA, November 1990.

[Veloso, 1989] M. Veloso. Nonlinear problem solving using intelligent casual-commitment. Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1989.

[Veloso, 1992] Manuela M. Veloso. *Learning by Analogical Reasoning in General Problem Solving*. PhD thesis, Carnegie Mellon University, School of Computer Science, August 1992. Available as technical report CMU-CS-92-174.

[Wilensky, 1983] Robert Wilensky. *Planning and Understanding*. Addison-Wesley, Reading, MA, 1983.

[Wilkins, 1988] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, San Mateo, CA, 1988.

[Wilkins, 1989] David E. Wilkins. Can AI planners solve practical problems? Technical Report Techical Note 468R, SRI International, November 1989.

[Zelle and Mooney, 1992] John M. Zelle and Raymond J. Mooney. Combining FOIL and EBG to speed-up logic programs. In *Submitted to IJCAI 93*, 1992.