Planning under Uncertainty in Dynamic Domains

Jim Blythe

May 25, 1998

CMU-CS-98-147

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:

Jaime Carbonell
Manuela Veloso
Reid Simmons
Judea Pearl, U.C.L.A.

Keywords: artificial intelligence, planning, reasoning under uncertainty, machine learning

Abstract

Planning, the process of finding a course of action which can be executed to achieve some goal, is an important and well-studied area of AI. One of the central assumptions of classical AI-based planning is that after performing an action the resulting state can be predicted completely and with certainty. This assumption has allowed the development of planning algorithms that provably achieve their goals, but it has also hindered the use of planners in many real-world applications because of their inherent uncertainty.

Recently, several planners have been implemented that reason probabilistically about the outcomes of actions and the initial state of a planning problem. However, their representations and algorithms do not scale well enough to handle large problems with many sources of uncertainty. This thesis introduces a probabilistic planning algorithm that can handle such problems by focussing on a smaller set of relevant sources of uncertainty, maintained as the plan is developed. This is achieved by using the candidate plan to constrain the sources of uncertainty that are considered, incrementally considering more sources as they are shown to be relevant. The algorithm is demonstrated in an implemented planner, called Weaver, that can handle uncertainty about actions taken by external agents, in addition to the kinds of uncertainty handled in previous planners. External agents may cause many simultaneous changes to the world that are not relevant to the success of a plan, making the ability to determine and ignore irrelevant events a crucial requirement for an efficient planner. Three additional techniques are presented that improve the planner's efficiency in a number of domains. First, the possible external events are analyzed before planning time to produce factored Markov chains which can greatly speed up the probabilistic evaluation of the plan when structural conditions are met. Second, domain-independent heuristics are introduced for choosing an incremental modification to apply to the current plan. These heuristics are based on the observation that the failure of the candidate plan can be used to condition the probability that the modification will be successful. Third, analogical replay is used to share planning effort across branches of the conditional plan. Empirical evidence shows that Weaver can create high-probability plans in a planning domain for managing the clean-up of oil spills at sea.

Contents

List of Tables

List of Figures

Acknowledgments

Many, many people have helped me with the ideas and presentation of this thesis. Of course my committee have been great at both generating and refining ideas: Jaime Carbonell, Manuela Veloso, Reid Simmons and Judea Pearl. Jaime and Manuela in particular have been very instrumental in the work. The Prodigy project over the years has always been an excellent group of people to work with. Alicia Perez, Mei Wang, Mike Cox, Scott Reilly, Eugene Fink, Rujith de Silva, Karen Haigh, Peter Stone and Will Uther all gave me valuable feedback in many meetings. Eugene and Mike greatly improved the thesis with their detailed comments. Outside the Prodigy group, Lonnie Chrisman, Sven Koenig, Rich Goodwin, John Mount, Bryan Loyall, Gary Pelton, Mike Williamson, Steve Hanks, Martha Pollack, Steve Chien, Louise Pryor and Robert Goldman have been sources of inspiration and of many useful conversations. The AI group at NASA Ames provided a great environment to home in on a thesis topic over the summer, with help from Steve Minton, Mark Drummond and Peter Cheeseman. I owe the greatest debt of thanks, though, to my wife Cathy, whose love and support allowed me to finish this thesis and laugh about it afterwards.

Chapter 1

Introduction

Planning and problem solving are essential characteristics of intelligent behaviour and have held a central place in AI research since the beginning of the field. The emphasis of research in planning has most often been the design of efficient algorithms to find plans given a simplified representation of a planning task, encoding only its essential features. By assuming a certain syntactic representation for planning domains and problems, general-purpose planning algorithms could be designed and in some cases proofs could be given of their correctness and completeness [Fikes & Nilsson 1971; Tate 1977; Chapman 1987; Minton *et al.* 1989; McAllester & Rosenblitt 1991].

While these systems represent significant technical achievements, when we want to apply them to problems that occur in the world around us we find that the assumptions they make can be severely limiting. For example, their representations do not address trade-offs between goals with different value in cases where they cannot all be achieved, or the possibility that the world can be changed independently of the actions taken by the planning agent. Nor do they address the fact that planners often do not know the state of the world or the effects of actions taken in it with certainty.

Uncertainty affects a wide spectrum of planning domains, from Saturn orbit insertion for the Cassini satellite (where the primary rocket may fail requiring the secondary rocket to be primed before the maneuver) to a plan to have a picnic in a park (where one should take into account the chance of rain or that the park may be crowded). Frequently there are many things that can go wrong with each step of a plan and a detailed plan created in advance will almost certainly fail. For example, it is not feasible to create a detailed plan for driving to the office that specifies turns, pauses and levels of acceleration before getting in the car. The planner does not know if the traffic lights will be red or green, or what traffic will be encountered. This led some researchers to abandon programs that choose steps in advance in favour of "reactive" programs that choose each step as it is to be made [Agre & Chapman 1987; Schoppers 1989b].

However it is often necessary to make contingency plans ahead of time even if not every situation can be predicted and planned for. For example, the Cassini spacecraft's secondary rockets take several hours to fire up, but must be ready immediately if the primary rockets should fail during the critical minutes of orbit insertion. The spacecraft cannot wait until this point to think about the problem, and reactive systems that explicitly prime the secondary rocket beg the question of how such behaviour could be designed automatically.

To deal with such problems, a planning system must be able to prioritise the different possible situations that might occur when a candidate plan is executed, and produce a plan that covers the most important ones while leaving others to be completed as more information is available during execution. One way to approach this problem, adopted in this thesis, is to assign probabilities to the various sources of uncertainty in the planning domain and use them to derive probabilities for each of the different possible outcomes when the plan is executed. The outcomes with higher probability are treated as having higher priority. This approach is attractive because a single parameter — a minimum acceptable value for the probability of plan success — can be used to control the planner and values can be combined using the standard axioms of probability. Computing the probability of success of a plan can be expensive, however.

This thesis describes an implemented algorithm to produce plans that meet a threshold probability of success in uncertain domains. The thesis concentrates on plan generation and does not address plan execution, plan monitoring or re-planning although these are important pieces of a complete system for planning in uncertain domains. The central contributions of the thesis are described below after the key concepts of planning under uncertainty are covered in more detail.

1.1 Planning under uncertainty

Planning, in the absence of uncertainty, is the process of finding a sequence of operators to transform an initial state into one that matches a goal description. The initial state is usually specified by giving a list of facts that hold in it, and the goal description is usually a logical sentence using a subset of the facts as terms. Operators are usually described in terms of preconditions and effects, where the precondition of an operator is a logical sentence describing the states in which the operator can legally be applied, and the effects describe the changes that will be brought about in a state if the operator is applied. Following STRIPS [Fikes & Nilsson 1971], the effects are usually represented as a list of facts to be deleted from the current state and a list of facts to be added.

This thesis builds on the Prodigy planner [Minton *et al.* 1989; Carbonell *et al.* 1992; Veloso *et al.* 1995], a domain-independent planner that was designed as a testbed for learning and problem solving. Prodigy provides a rich language for specifying search control knowledge and includes a number of independent machine learning modules that can produce control knowledge from experience in a particular domain or use saved cases.

Uncertainty can enter a planning domain through a number of different types of cause. The initial state may not be completely known, either because it is not completely observable by the agent or because of delays between forming and executing a plan. The outcomes of actions taken in the domain may be non-deterministic and other agents may be changing the world though exogenous events that are not completely known or predictable. Finally the goals of the agent may change over time, perhaps unpredictably. For each of these sources, partial information is still useful, for example a subset of the possible initial states that is guaranteed to contain the actual initial state, or a probability that some new goal will be given over a fixed time scale can be helpful in building a working plan.

This thesis presents a planner that can handle uncertainty from three types of cause: more than one possible initial state, non-deterministic outcomes of actions and non-deterministic exogenous events. For each of these, the planner requires a probabilistic representation. It then attempts to find a plan that exceeds a minimum desired probability of success.

The presence of exogenous events means that each step in a plan that has an appreciable duration is effectively nondeterministic. Explicitly considering each of these alternative outcomes could lead to a combinatorial explosion in the separate cases that a planner would need to consider. The planning algorithm present in the thesis, called Weaver, addresses this problem by lazily including sources of uncertainty in its evaluation of a plan. This is done building creating a plan in two alternating steps, plan creation and plan evaluation. In the plan creation step, Weaver chooses an outcome for a step that is desired in order to achieve a goal and does not explicitly consider the other outcomes, although it does make sure the outcome is reasonably likely. In the plan evaluation step, all sources of uncertainty can potentially be considered, but only those that are shown to have an affect on the plan's chance of success are explicitly represented. After this, the plan creation step is then re-entered, with one or more of the sources of uncertainty that are known to be important being explicitly considered.

1.2 An example planning problem

Consider for example a planning domain describing an oil tanker that has run aground near the coast and is beginning to leak oil into the sea. The goal is to stop the oil from polluting the water or reaching any of several areas of coastline along the shore. The available actions include pumping the oil from the tanker into a secondary vessel, surrounding the tanker with booms to contain the oil, using booms and other equipment to prevent the oil from reaching the shore and cleaning oil up from either the sea or the shore. Cleaning up oil from the water is successful with 70% probability. The amount of oil spilled from the tanker is a stochastic process represented as an exogenous event. Its path of travel once in the water is also treated as an exogenous event. Many of the pieces of equipment used in the recovery process require good weather conditions and change in the weather is also modelled with exogenous events.

Weaver begins by ignoring the sources of uncertainty in the domain. An initial plan might put equipment in place and pump the oil from the tanker into another vessel. When this plan is evaluated, those sources of uncertainty that can affect the plan are considered. For example, in the time taken to move the vessel and the pump, oil may spill from the tanker and the weather around the tanker may change. However nondeterministic changes to the world, such as weather changes in other parts of the sea, are still ignored. The plan critic will select a potential flaw in the plan to have the planner improve, for example that oil can be spilled. This event is now represented to the planner, which may produce a plan to clean oil from the water if it is spilled, or to move booms around the tanker to prevent the oil being spilled.

In either case oil may still be in the water when the new plan is completed, although with lower probability than in the original plan. On the next improvement, the plan critic might direct the planner to pay attention to the situation where oil might reach some shoreline, and the planner might create a conditional branch in the plan to protect the shore or clean it up in that case. Although an alternative plan might always protect the shore without testing the state, in some cases a higher probability of success can be achieved with a conditional plan than otherwise. If there are not enough resources, for example, to protect both potentially threatened pieces of shoreline, a branching plan that tests the oil's position before allocating equipment would be more successful than any non-branching plan.

1.3 Contributions

The thesis describes a novel algorithm for producing conditional plans based on the Prodigy planner. The main contributions of the thesis are techniques that allow the planner both to find plans in situations where there can be many sources of uncertainty and to reason efficiently about these plans. These techniques can be divided into those that affect the process of plan creation and those that affect the process of plan evaluation.

Plan creation

Although previous uncertain planners, such as Buridan and C-Buridan [Kushmerick, Hanks, & Weld 1995], represented probabilistic effects of actions, their techniques did not scale well because of the joint expenses of handling all the contingencies the planner may need to consider and evaluating a plan with a number of sources of uncertainty. The problem is more severe when uncertainty through exogenous events is considered, because there can be a number of different resulting states after applying any action, rather than just those that have explicitly marked uncertain outcomes, and the number of such states climbs exponentially with the number of exogenous events.

A contribution of this thesis is an implemented planner that selectively ignores some of the sources of uncertainty while choosing actions for the plan, and in a separate evaluation stage considers the sources that are known to be relevant to the candidate plan, ignoring others. This process is iterative, with the planner refining its candidate plan and becoming aware of more sources of uncertainty on each iteration. Using this technique, the planner is the first to handle uncertain exogenous events.

Two further contributions of the thesis are aimed at improving the efficiency of plan creation. First, based on a study of the algorithm a family of domain-independent search heuristics is proposed that relax the assumptions of independence typically made in probabilistic planners, using the observation that the failure of a candidate plan can be used to condition the probability that the modification will be successful. Heuristics from this family are demonstrated. Second, a variant of internal analogy is used to reduce the amount of duplicated work in different conditional branches of the plans produced.

Plan evaluation

A plan's probability of success is computed by automatically creating a Bayesian belief net to represent the plan and evaluating it using a standard join tree method [Pearl 1988]. A contribution of this thesis is an algorithm to produce a belief net that can be efficiently evaluated when there are exogenous events. The algorithm exploits two properties that are expected to be common in probabilistic planning domains: a representation for exogenous events that uses stationary probabilities, and a high variability in the length of establishment intervals of domain-level properties in plans. The first property makes it possible to use Markov chains to summarise the action of exogenous events over time and the second property makes it useful to do so. I briefly explain this approach below, and describe it in more detail in Chapter 5.

In the representation for exogenous events developed in this thesis, events have probabilities of occurrence and probability distributions for their possible outcomes. All of these probabilities may depend on the state of the world at the time the event may occur, but do not otherwise depend on the time of occurrence. This stationary property of the probabilities makes it possible to define Markov chains, whose nodes represent abstract world states, that can be used to infer the probability of future abstract world states.

The actions that planners consider can often take widely varying amounts of time to perform. For instance if a planner considers moving a boat or aircraft to a desired location, the action may take from under an hour to several days depending on the current location and state of readiness of the boat or aircraft. Because of these wide variations, the lengths of time between when the value of a domain variable is established in a plan and when it is required can also vary greatly. We call such an interval of time an *establishment interval* for the variable. We see that if time were divided into units of equal length when a belief net is created to evaluate the plan, some establishment intervals would necessarily cross many time units, and if belief net nodes at one time point were connected to parents at the previous time point, long paths of nodes in the belief net would be created. The time to evaluate this belief net would be at least linear in the length of these paths and frequently worse, since belief net evaluation is NP-hard in general [Cooper 1990].

One can use the ability to describe exogenous events using Markov chains to escape this difficulty. The probability distribution of states in a Markov chain at some future time t given the distribution at time 0 can be computed in time logarithmic in t. Methods to define such Markov chains that guarantee correctness are given in Chapter 3. The chains that are computed might have exponential size in the number of exogenous events in the worst case. The improved time performance in evaluating the resulting belief net outweighs the cost of computing the Markov chains when the establishment intervals vary sufficiently and the resulting chains are not too large. I show in Chapter 7 that there are significant benefits in the oil-spill domain and it is likely that this is the case in a large number of interesting planning domains.

Limitations in scope

Creating robust plans under uncertainty is a hard problem that requires a much broader set of algorithms and approaches than can be dealt with properly in one thesis. Handling some of the aspects of the problem that have been ignored here would make a very interesting extension to this work. In particular, the planner described here does not address plan monitoring or re-planning. After the initial plan is created, a complete system would begin to check certain key domain features as the plan is executed, updating its beliefs that the various alternative courses of action will be successful, and revising part or all of the plan if it was deemed necessary. The potential need to revise a plan could also affect the initial plan, since an ideal plan would not make early commitments that would reduce later options for revision. In fact the objective measure of a robust plan might include the potential for re-planning. In this thesis the objective measure views the plan as a static, complete object. While the belief net approach lends itself to plan monitoring and belief updates, this aspect is not addressed.

Also ignored is the important issue of the source of and confidence in the planner's knowledge about its uncertainty. For example single point probabilities are given for each possible action outcome rather than intervals of probability. Again the approach taken to plan evaluation makes some examination possible of the importance of the assumptions about uncertainty, via sensitivity analysis, but this has not been addressed.

1.4 Reader's guide to the thesis

The next chapter surveys some of the related work to this thesis. Chapter 3 describes the model used to describe planning problems that contain sources of uncertainty.

This model is based on the Markov decision process although the planner does not explicitly build one. Chapter 4 describes the planning algorithm used in Weaver and provides a simple example. Chapter 5 presents a method to improve the efficiency of evaluating plans, and chapter 6 presents methods to improve the efficiency of planning when there is uncertainty. Chapter 7 presents an empirical analysis of the planning problem and the methods to improve efficiency in a large planning domain. Finally, chapter 8 summarises the main contributions of the thesis.

Chapter 2

Related work

Work in automatic planning began early in the field of artificial intelligence, with such programs as GPS [Newell & Simon 1963] and STRIPS [Fikes & Nilsson 1971]. As well as introducing new planning algorithms [Chapman 1987; McAllester & Rosenblitt 1991], later work considered more general conditions, for example where external events could take place during plan execution [Vere 1983], where the domain theory is incomplete or incorrect [Gil 1992; Wang 1996; Gervasio 1996] or where the agent must consider the relative quality of alternative plans [Pérez 1995].

There has been a wide variety of work done in the design of systems for planning under uncertainty. In this chapter I provide a survey of some of this work, point out the work most closely related to this thesis and show how the thesis contributes to the body of work in this area.

2.1 Overview of "classical" planning.

Broadly, planning systems aim to construct a pattern of behaviour for a performance system to accomplish some given goal in its environment. Most planning systems are given a representation for possible actions that can be taken in the environment in terms of the conditions required to hold for the action to be taken and the effects on the environment when the action is taken. Systems referred to as "classical" planning systems typically make the following additional assumptions:

- The environment can be represented in a sequence of discrete states, which are represented using a logical language.
- The actions are represented in a language similar to that used by STRIPS [Fikes & Nilsson 1971], in which the action's preconditions form a logical sentence over the language used to define states, and the action's effects are represented by a set of terms to be added to or deleted from the terms used to represent a state.
- The goal is a property of a state, characterised by a logical sentence (rather than a property of a sequence of states).

In addition to these assumption, classical planners return a plan in the form of a sequence of actions, either totally or partially ordered.

This is a much studied family in AI planning because the assumptions are powerful and seem reasonable. An interesting range of behaviours can be captured with STRIPS-style operators and they allow a planner to perform backward chaining since given a goal, one can subgoal on the preconditions required for an action or sequence actions to achieve the goal. In addition the logical representation allows a plan to be proved correct. Prodigy, the planning system extended in this thesis, is a classical planner according to this definition [Veloso *et al.* 1995]. A more detailed description of Prodigy's action representation, which is an example of a STRIPS-like representation, can be found in Section 3.1.

Work in classical planning has typically focussed on improving the efficiency with which a plan is created. For example partial-order planners, introduced in the late eighties [Chapman 1987; McAllester & Rosenblitt 1991; Penberthy & Weld 1992], attempt to reduce the search space using a "least commitment" principle where steps in a plan are only ordered with respect to one another as required to prove that the plan is successful. Planners can be made faster by solving a series of abstractions of the planning problem, each more detailed until the problem is solved in full complexity, at each stage using the solution from the previous stage [Sacerdoti 1974; Yang & Tenenberg 1990; Knoblock 1991]. Work has also been done on controlling search with various heuristics [Blythe & Veloso 1992; Gerevini & Schubert 1996; Pollack, Joslin, & Paolucci 1997] and with explicit control rules [Minton 1988; Minton *et al.* 1989].

The STRIPS action representation does not support non-deterministic action outcomes and classical planners do not have a means to represent sources of change in the domain other than the actions taken by the performance agent. These systems therefore cannot be used for planning problems involving uncertainty. However the ideas and algorithms developed under the assumptions of classical planning form the basis of many approaches to planning under uncertainty including the ones described in this thesis.

2.2 Reactive planning and approaches that mix planning with execution.

When a plan is executed in a stochastic domain in which initial conditions and action outcomes are uncertain, many different scenarios can take place as a result of the execution. A closed-loop plan, one that contains no sensing and always executes the same set of actions, will usually be too brittle to achieve high reliability in such a domain. Instead, planning systems must create branching plans that take into account the intermediate states reached while the plan is being executed and take different actions accordingly.

Some approaches to this problem, motivated in addition by real-time constraints, aim to create strategies for behaviour in which a system repeatedly senses its environment and chooses an action based on this information, similar to a policy on a Markov decision process described in Section 2.4. These systems have no explicit representation of a global plan and some, such as Brook's subsumption architecture [Brooks 1986] and Agre and Chapman's Pengi [Agre & Chapman 1987], have little or no internal state. The subsumption architecture arranges reactive subsytems into hierarchies, so that higher-level behaviour is achieved by one system over-riding, or subsuming, a more basic system. Neither of these systems, however, shows how a reactive plan could be built from a declarative description of the environment. Schoppers [Schoppers 1989a] develops a similar scheme in his "universal plans" and describes how they can be created automatically. Other reactive planning systems such as PRS [Georgeff & Lansky 1987], RAP [Firby 1987; 1989] and HAP [Loyall & Bates 1991] do include internal state — PRS represents the agent's beliefs, desires and intentions explicitly — and also control structures such as iteration.

These systems can produce appropriate behaviour under different execution conditions but for each contingency that arises the appropriate response must be programmed by a human in advance. In many domains, however, it is not feasible to specify the correct action in each possible situation in advance. Instead we would like to combine the speed of a reactive planner in familiar situations with the flexibility of a classical planner in situations not previously anticipated. Systems have been developed for this purpose that combine reactive execution with classical planning, using the latter when no pre-programmed response is available for some contingency.

Both the Theo-agent [Blythe & Mitchell 1989] and the "anytime synthetic projection" technique of Drummond and Bresina [Drummond & Bresina 1990] follow reactive rules if they are present, and otherwise fall back on planning and then compile the resulting plan into new reactive rules to be used in future episodes. The two systems differ on the action and goal representation and on the planning technique. The Theo-agent uses a STRIPS-like action representation and backward chaining. In anytime synthetic projection, forward chaining is used with a richer action and goal language that can specify probabilistic outcomes, exogenous events and goals to maintain predicates over time intervals.

A mixed planning and execution strategy can provide further advantages if it is under explicit control. Some goals can be planned for in advance, while others can be deferred until part way through the execution, when there may be more information about the best course of action. By not always giving priority to reactive rules, the mixed-strategy system can avoid pitfalls in some cases. On the other hand, delaying planning for these goals can drastically reduce the number of alternatives to consider.

Gervasio shows how to build "completable plans" which are designed to be incomplete, and amenable to being further elaborated during execution [Gervasio & DeJong 1994; Gervasio 1996]. Goodwin [Goodwin 1994] considers the question of when to switch from planning to executing in a time-dependent planning problem. Onder and Pollack [Onder & Pollack 1997] describe a probabilistic planner that reasons about which contingencies to plan for before execution and which to defer. Washington makes use an abstraction hierarchy, planning at some abstract level before execution and picking a concrete instance of the abstract plan during execution [Washington 1994].

2.3 Extensions to classical planning systems

Within the broader context of systems for planning and execution in uncertain environments, some recent work has focussed on creating a plan before execution that accounts for a significant subset of the contingencies that might occur during execution, but not necessarily all of them. For this purpose it is useful to have some way to compare the different contingencies to find the most important. A probabilistic representation of the sources of uncertainty is well suited for this, although some have used a Dempster-Shaffer formalism [Mansell 1993], fuzzy reasoning [Bonissone & Dutta 1990] or a minimax approach [Koenig & Simmons 1995].

Work on extensions to planning systems using decision theory began shortly after the initial work on STRIPS [Feldman & Sproull 1977], but this line of work was largely discontinued until the 1990's. Wellman lays the ground-work for developing probabilistic extensions to classical planning systems, pointing out in [Wellman 1990b] that the STRIPS representation of action embodies a Markov assumption — since the success of an operator depends on its preconditions, this is conditionally independent of all other knowledge if the current state is known. This makes it possible to factor the formula that describes the probability of plan success, using for example the standard techniques for Bayesian belief networks [Pearl 1988]. Wellman also shows [Wellman 1990a] how proofs that one partial plan "dominates" another, in that each of its possible completions will have a higher utility than those of the other, can be used to reduce the search space for high-utility plans. Goldszmidt and Darwiche [Goldszmidt & Darwiche 1995] also discuss using belief nets to evaluate plans, but unlike the work described in this thesis they do not construct belief nets automatically. In another important paper, Peot and Smith describe an extension to the SNLP algorithm, called CNLP for building conditional plans when some of the domain actions can have non-deterministic effects [Peot & Smith 1992].

Among the best-known probabilistic planners are Buridan [Kushmerick, Hanks, & Weld 1995] and C-Buridan [Draper, Hanks, & Weld 1994]. Buridan uses a variant of the SNLP algorithm, and a STRIPS-like action representation with probabilistic effects. Each action has a set of *triggers*, each of which is a conjunction of terms that describe the world state, and each is connect to a probability distribution of effect sets. Each effect set in this distribution is a collection of terms to be added to or deleted from the state, as in a STRIPS action. Buridan's input includes a probability threshold value as well as a logical goal, and it tries to find a plan that succeeds with probability at least equal to the threshold value. Weaver's action representation is similar to Buridan's and is described in detail in Section 3.2.

One way that Buridan extends SNLP is by being able to propose more than one action in its plan to achieve any goal or subgoal. This is necessary since each individual action may fail to achieve the goal with some probability. Buridan can provably find a plan that achieves the threshold probability of success, if a non-branching plan exists that does so. C-Buridan is an extension to Buridan that can create branching plans, using a technique similar to that introduced with CNLP [Peot & Smith 1992]. Essentially, one new way that C-Buridan can fix a candidate plan in which two actions might interfere with each other is to add a test and restrict the actions to different conditional branches based on the test. Information about the branch that an action belongs to is then propagated to the action's descendants and ancestors in the goal tree.

Cassandra [Pryor & Collins 1993; 1996] is another planner based on SNLP and uses an explicit representation of decision steps. Bagchi et al. describe a planner that uses a spreading activation technique to choose actions with highest expected utility [Bagchi, Biswas, & Kawamura 1994]. Onder and Pollack also extend SNLP with explicit reasoning about the contingencies to plan for [Onder & Pollack 1997].

Haddawy's thesis work [Haddawy 1991] introduces a logic with extensions to represent probabilities of terms, intended for representing plans. He and his group have developed hierarchical task-network (HTN) planners that use a similar representation [Haddawy & Suwandi 1994; Haddawy, Doan, & Goodwin 1995; Haddawy, Doan, & Kahn 1996].

While this thesis concentrates mainly on extensions to classical planning that support probabilistic planning in uncertain domains, other work develops extensions to enable planners to find high-quality plans according to some specified criteria. Pérez [Pérez & Carbonell 1994; Pérez 1995] describes how to learn control knowledge from interactions with a human expert that allows Prodigy to produce higher-quality plans. Williamson and Hanks [Williamson & Hanks 1994; Williamson 1996] develop a decision-theoretic extension to UCPOP.

The planner described in this thesis is unique in its ability to reason explicitly about exogenous events, and to perform a relevance analysis of those events. Vere [Vere 1983] introduced a planner that could reason about exogenous events that were known to occur at some fixed time, but not about uncertain events.

2.4 Approaches based on Markov decision processes

In the past five years, much of the research activity in AI planning under uncertainty has built on standard techniques for solving Markov decision problems (MDPs) and partially-observable Markov decision problems POMDPs [Dean *et al.* 1995; Littman 1996; Boutilier, Brafman, & Geib 1997]. This formalism has the advantage of a sound mathematical underpinning, and the standard solution techniques aim at an optimal policy while most systems based on classical planning try to meet a lower bound. However these standard techniques are not tractable since they require enumerating all the states of the system. Work to make MDP approaches tractable has attempted to incorporate ideas from classical planning and other areas of AI to exploit any underlying structure in the state space [Dearden & Boutilier 1997] as well as to relax the requirement of optimalty [Parr & Russell 1995].

In this section I give a brief overview of Markov decision processes and survey some of the work in this area. This thesis pursues an approach based on classical planning, but uses an MDP representation to describe planning problems and their solutions.

2.4.1 Overview of Markov Decision Processes

This description of Markov decision processes follows [Littman 1996] and [Boutilier, Dean, & Hanks 1995]. A Markov decision process M is a tuple $M =< S, \mathcal{A}, \Phi, R >$ where

- S is a finite set of states of the system.
- $\mathcal{A}$ is a finite set of actions.
- $\Phi{:}A \times S \rightarrow \Pi(S)$ is the *state transition function*, maping an action and a state to a probability distribution over S for the possible resulting state. The probability of reaching state s' by performing action a in state s is written $\Phi(a, s, s')$.
- $R{:}S \times A \rightarrow \mathcal{R}$ is the *reward function*. $R(s, a)$ is the reward the system receives if it takes action a in state s.

A *policy* for an MDP is a mapping $\pi{:}S \rightarrow A$ that selects an action for each state. Given a policy, we can define its finite-horizon value function $V_n^\pi{:}S \rightarrow \mathcal{R}$, where $V_n^\pi(s)$ is the expected value of applying the policy π for n steps starting in state s. This is defined inductively with $V_0^\pi(s) = R(s, \pi(s))$ and

$$V_m^\pi(s) = R(s, \pi(s)) + \sum_{u \in S} \Phi(\pi(s), s, u) V_{m-1}^\pi(u)$$

Over an infinite horizon, a discounted model is frequently used to ensure policies have a bounded expected value. For some β chosen so that $\beta < 1$, the value of any reward from the transition after the next is discounted by a factor of β and the one after that by a factor of β^2, and so on. Thus if $V^\pi(s)$ is the discounted expected value in state s following policy π forever, we must have

$$V^\pi(s) = R(s, \pi(s)) + \beta \sum_{u \in S} \Phi(\pi(s), s, u) V^\pi(u)$$

This yields a set of linear equations in the values of $V^\pi()$.

A solution to an MDP is a policy that maximises its expected value. For the discounted infinite-horizon case with any given discount factor β, there is a policy

V^* that is optimal regardless of the starting state [Howard 1960], which satisfies the following equation:

$$V^*(s) = \max_a\{R(s,a) + \beta \sum_{u \in S} \Phi(a,s,u)V^*(u)\}$$

Two popular methods for solving this equation and finding an optimal policy for an MDP are *value iteration* and *policy iteration* [Puterman 1994].

In policy iteration, the current policy is repeatedly improved by finding some action in each state that has a higher value than the action chosen by the current policy for that state. The policy is initially chosen at random, and the process terminates when no improvement can be found. Tha algorithm is shown in Table 2.1. This process converges to an optimal policy [Puterman 1994].

Policy-Iteration($S, \mathcal{A}, \Phi, R, \beta$):
1. For each $s \in S$, $\pi(s) = \text{RandomElement}(\mathcal{A})$
2. Compute $V^\pi(.)$
3. For each $s \in S$ {
4. Find some action a such that
$R(s,a) + \beta \sum_{u \in S} \Phi(a,s,u)V^\pi(u) > V^\pi(s)$
5. Set $\pi'(s) = a$ if such an a exists,
6 otherwise set $\pi'(s) = \pi(s)$.
}
7. If $\pi'(s) \neq \pi(s)$ for some $s \in S$ goto 2.
8. Return π

TABLE 2.1: The policy iteration algorithm

In value iteration, optimal policies are produced for successively longer finite horizons, until they converge. It is relatively simple to find an optimal policy over n steps $\pi^*_n(.)$, with value function $V^*_n(.)$, using the recurrence relation:

$$\pi^*_n(s) = \text{argmax}_a\{R(s,a) + \beta \sum_{u \in S} \Phi(a,s,u)V^*_{n-1}(u)\}$$

with starting condition $V^*_0(s) = 0 \forall s \in S$, where V^*_m is derived from the policy π^*_m as described above. Table 2.2 shows the value iteration algorithm, which takes an MDP, a discount value β and a parameter ϵ and produces successive finite-horizon optimal policies, terminating when the maximum change in values between the current and previous value functions is below ϵ. It can also be shown that the algorithm converges to the optimal policy for the discounted infinite case in a number of steps which is polynomial in $|S|$, $|\mathcal{A}|$, $\log \max_{s,a}|R(s,a)|$ and $1/(1-\beta)$.

2.4.2 Planning under uncertainty with MDPs

The algorithms described above can find optimal policies in polynomial time in the size of the state space of the MDP. However, this state space is usually exponentially

Value-Iteration($S, \mathcal{A}, \Phi, R, \beta, \epsilon$):
1. for each $s \in S$, $V_0(s) = 0$
2. $t = 0$
3. $t = t + 1$
4. for each $s \in S$ {
5. for each $a \in A$
6. $Q_t(s,a) = R(s,a) + \beta \sum_{u \in S} \Phi(a,s,u) V_{t-1}(u)$
7. $\pi_t(s) = \mathrm{argmax}_a Q_t(s,a)$
8. $V_t(s) = Q_t(s, \pi_t(s))$
}
9. if ($\max_s |V_t(s) - V_{t-1}(s)| \geq \epsilon$) goto 3
10. return π_t

TABLE 2.2: The value iteration algorithm

large in the inputs to a planning problem, which includes a set of literals whose cross product describes the state space. Attempts to build on these and other techniques for solving MDPs have concentrated on ways to gain leverage from the structure of the planning problem to reduce the computation time require.

Dean et al. used policy iteration in a restricted state space called an *envelope* [Dean *et al.* 1993]. A subset of the states is selected, and each transition in the MDP that leaves the subset is replaced with a new transition to a new state `OUT` with zero reward. No transitions leave the `OUT` state. They developed an algorithm that alternated between solving the restricted-space MDP with policy iteration and expanding the envelope by including the n most likely elements of the state space to be reached by the optimal policy that were not in the envelope. The algorithm converges to an optimal policy considerably more quickly than standard policy iteration on the whole state space, but as the authors point out [Dean *et al.* 1995], it makes some assumptions which limit its applicability, including that of a sparse MDP in which each state has only a small number of outward transitions. Tash and Russell extend the idea of an envelope with an initial estimate of distance-to-goal for each state and a model that takes the time of computation into account [Tash & Russell 1994].

While the envelope extension method ignores portions of the state space, other techniques have considered abstractions of the state space that try to group together sets of states that behave similarly under the chosen actions of the optimal policy. Boutilier and Dearden [Boutilier & Dearden 1994] assume a representation for actions that is similar to that used in Buridan [Kushmerick, Hanks, & Weld 1994] described in Section 2.3 and a state utility function that is described in terms of domain literals. They then pick a subset of the literals that account for the greatest variation in the state utility and use the action representation to find literals which can directly or indirectly affect the chosen set, using a technique similar to the one developed by Knoblock for building abstraction hierarchies for classical planners [Knoblock 1991]. This subset of literals then forms the basis for an abstract MDP by projection of the original states. Since the state space size is exponential in the set of literals, this reduction can lead to considerable time savings over the original MDP. Boutilier and Dearden prove bounds on the difference in value of the abstract policy compared with an optimal policy in the original MDP.

Lin and Dean further refine this idea by splitting the MDP into subsets and allowing a different abstraction of the states to be considered in each one [Dean & Lin 1995]. This approach can have extra power because typically different literals may be relevant in different parts of the state space. However there is an added cost to re-combining the separate pieces unless they happen to decompose very cleanly. Lin and Dean assume the partition of the state space is given by some external oracle.

Boutilier et al. extend *modified policy iteration* to propose a technique called *structured policy iteration* that makes use of a structured action representation in the form of 2-stage Bayesian networks [Boutilier, Dearden, & Goldszmidt 1995]. The representation of the policy and utility functions are also structured in their approach, using decision trees. In standard policy iteration, the value of the candidate policy is computed on each iteration by solving a system of $|S|$ linear equations (step 2 in Table 2.1, which is computationally prohibitive for large real-world planning problems. Modified policy iteration replaces this step with an iterative approximation of the value function V_π by a series of value functions $V^0, V^1, \ldots$ given by

$$V^i(s) = R(s) + \beta \sum_{u \in S} \Phi(\pi(s), s, u) V^{i-1}(u)$$

Stopping criteria are given in [Puterman 1994].

In structured policy iteration, the value function is again built in a series of approximations, but in each one it is represented as a decision tree over the domain literals. Similarly the policy is built up as a decision tree. On each iteration, new literals might be added to these trees as a result of examining the literals mentioned in the action specification Φ and utility function R. In this way the algorithm avoids explicitly enumerating the state space.

Similar work has also been done with *partially-observable Markov decision processes* or POMDPs, in which the assumption of complete observability is relaxed. In a POMDP there is a set of observation labels $\mathcal{O}$ and a set of conditional probabilities $P(o|a,s), o \in \mathcal{O}, a \in \mathcal{A}, s \in S$, such that if the system makes a transition to state s with action a it receives the observation label o with probability $P(o|a,s)$. Cassandra et al. introduce the *witness algorithm* for solving POMDPs [Cassandra, Kaelbling, & Littman 1994]. A standard technique for finding an optimal policy for a POMDP is to construct the MDP whose states are the belief states of the original POMDP, ie each state is a probability distribution over states in the POMDP, with beliefs maintained based on the observation labels using Bayes' rule. A form of value iteration can be performed in this space making use of the fact that each finite-horizon policy will be convex and piecewise-linear. The witness algorithm includes an improved technique for updating the basis of the convex value function on each iteration. Parr and Russell use a smooth approximation of the value function that can be updated with gradient descent [Parr & Russell 1995]. Brafman introduces a grid-based method in [Brafman 1997].

Although work on POMDPs is promising, it is still preliminary, and the largest completely solved POMDPs have about 10 states [Brafman 1997].

Chapter 3

Planning under Uncertainty

In this chapter I provide an overview of the PRODIGY 4.0 planning algorithm and discuss extensions made to allow it to create and evaluate plans under conditions of uncertainty. I present extensions to PRODIGY 4.0's model to represent uncertainty in Section 3.2. I give the meaning of the elements of this language in terms of a Markov decision process in Section 3.3. I illustrate the chapter with an example domain, a plan and the domain's derived MDP in Section 3.4.

3.1 Overview of PRODIGY 4.0

PRODIGY 4.0 takes as input a representation of a planning problem and attempts to produce a solution. The problem consists of the allowable object types, the actions that can be taken, a set of specific objects, an initial state and a goal state description. A solution, if found, will be a sequence of operators that can be applied to the initial state to produce some state satisfying the goal description. PRODIGY 4.0 uses means-ends analysis to search for plans. In this section I give a detailed description of the language PRODIGY 4.0 uses to represent planning problems, how it constructs a plan and how it searches a space of partial plans to find a solution.

3.1.1 Domains, problems and plans

A planning domain in PRODIGY 4.0 consists of a type hierarchy T, a set of literals $\mathcal{L}$ and a set of operators O[1], based on the STRIPS domain definition [Fikes & Nilsson 1971]. Each operator is defined by specifying its preconditions and effects. The preconditions specify conditions about the world that must be true in order for the operator to be applied, and are made of combinations of literal terms, that may include typed variables and may be combined using conjunction, disjunction, negation and existential and universal quantification. The effects specify literals that are either

[1] A domain also contains inference rules, which are very similar to operators but add inferences to the state rather than changing it. Full details can be found in [Carbonell *et al.* 1992].

added to (made true in) the state or deleted from (made false in) the state. The literals in the effects may mention the same variables that were mentioned in the preconditions, or new variables, in which case universal quantification is understood. Conditional effects may also be specified, in which the conditional test has the same form as the preconditions of the operator. An example of an operator is shown in Figure 3.1 and is discussed after a brief description of a planning problem. Full details on the syntax of planning domains and problems in PRODIGY 4.0 can be found in [Carbonell *et al.* 1992].

```
(Operator  Move-Barge
  (preconds
    ((<barge> Barge)
     (<from>  Place)
     (<to>    (and Place ( diff <from> <to>))))
    (at <barge> <from>))
  (effects ()
    ((del (at <barge> <from>))
     (add (at <barge> <to>)))))
```

FIGURE 3.1: An operator that can move an object of type Barge from the location bound to variable <from> to the location bound to variable <to>. Variables have an associated type and can also be constrained with functions.

A planning problem consists of a planning domain, a set of objects each of which belongs to one of the types in the domain, an initial state which is specified by giving all the literals that are true in the state and a goal description. The goal description is a combination of literals formed in the same way as the precondition of an operator, except that there may be no free variables — all the variables in the goal expression must be either existentially or universally quantified. The literals specifying the initial state may contain no variables.

Figure 3.1 shows a simple operator from a domain that will be developed as an example in this chapter as well as later chapters in the thesis. The operator has three variables, <barge>, <from> and <to>[2], and their types are specified in the preconditions slot: <barge> is of type Barge and <from> and <to> are both of type Place. In addition to mentioning a type, the specification of the variable <to> uses a function to further restrict the possible values for the variable. The function call (diff <from> <to>) must also be true — this function is true when the two variables have different values.

The second part of the preconditions slot consists of the precondition statement, which in this case is the single literal (at <barge> <from>). An *instantiation* of this operator consists of the operator along with an object from the planning problem

[2]Angle brackets are used to denote variables, such as <from>, in PRODIGY 4.0.

associated with each variable in the operator's preconditions. The object's type must match the variable's type and the set of objects must agree with all the functions used in the variable specifications. The instantiated operator is applicable in a state if the literal made from its preconditions statement is true in that state.

If an instantiated operator is *applied* to a state, a new state is produced corresponding to the result of applying the operator in the original state. The new state is specified as follows. First, the test corresponding to each conditional effect set is evaluated in the state. If it is true, the corresponding conditional effects are treated as regular effects, otherwise they are discarded. Next the literals mentioned in each add or del statement are examined for variables, and any variables mentioned in the preconditions slot are replaced with their object values from the instantiation of the operator. If there are variables that are not mentioned in the preconditions slot, they are treated as universally quantified. For each such variable, a variable specification must be given in the effects slot, mentioning a type and optionally some bindings functions in the same way as in the preconditions slot. Each literal in the effects that mentions one of these variables is replaced with a set of literals, corresponding to every object in the planning problem that matches the variable specification. After the literals in the effects are all translated into one or more literals that mention no variables, the state transformation is made. First the literals mentioned in del statements are made false in the state. Next the literals mentioned in add statements are made true in the state. Any literal that is not mentioned in the effects of the instantiated operator has the same truth value in the new state as in the old state.

For example, suppose a planning problem uses a domain that includes the Move-Barge operator as shown, and includes the objects barge1 of type Barge and Richmond and Oakland of type Place. If the initial state has the true literal (at barge1 Richmond), then the instantiated operator

```
(Move-Barge <barge>=barge1 <from>=Richmond <to>=Oakland)
```

will be applicable in the initial state. Applying it would lead to a new state, in which (at barge1 Richmond) would be false and (at barge1 Oakland) would be true, with all other literals unchanged.

A *plan* in PRODIGY 4.0 is a totally-ordered sequence of instantiated operators $(O_1, O_2, \ldots O_n)$. The plan can be re-formulated in a partial order after creation if desired [Veloso 1992]. I will refer to an instantiated operator in a plan as a *step* in the plan. A plan is a solution to the planning problem if:

1. O_1 is applicable in the initial state and applying it leads to the state S_1,
2. each step O_i for $2 \leq i \leq n$ is applicable in the state S_{i-1} and applying it leads to the state S_i and
3. the goal description is satisfied in state S_n.

For example, given the initial state above and the goal (at barge1 Oakland), the plan consisting of the single step (Move-Barge <barge>=barge1 <from>=Richmond <to>=Oakland) is a solution for this planning problem.

3.1.2 Constructing plans

Typically, however, a plan consists of a large number of steps and constructing a plan is not a trivial exercise. PRODIGY 4.0 constructs plans using a technique called *means-ends analysis*, in which it uses the difference between the current state and its goals to decide which steps to include in a plan. In order to illustrate how plans are constructed, I introduce two more operators into the barge domain and consider a problem with a slightly harder goal. Figure 3.2 shows the operators `Pump-Oil` and `Unload-Oil`, which can be used respectively to pump oil from a vessel at sea into a barge and to unload the oil from the barge when it is at a dock. I also introduce two new types, which are specializations or *subtypes* of the type `Place` that already exists in the domain.

```
(operator  Pump-Oil
  (preconds ((<barge> Barge)
             (<sector> Sea-Sector))
   (and (at <barge> <sector>)
        (oil-in-tanker <sector>)))
  (effects ()
   ((add (oil-in-barge <barge>))
    (del (oil-in-tanker <sector>)))))
```

```
(operator  Unload-Oil
  (preconds ((<barge> Barge)
             (<dock> Dock))
   (and (at <barge> <dock>)
        (oil-in-barge <barge>)))
  (effects ()
   ((del (oil-in-barge <barge>))
    (add (disposed-oil)))))
```

FIGURE 3.2: Two more operators in the example domain.

The new planning problem includes the objects `barge1` of type `Barge`, `Richmond` of type `Dock` and `west-coast` of type `Sea-Sector`. The type hierarchy and the objects in the planning problem are shown in Figure 3.3. The problem has an initial state in which the literals `(at barge1 Richmond)` and `(oil-in-tanker west-coast)` are the only true literals, and the goal `(disposed-oil)`. This can be achieved by the following four-step plan:

```
(Move-Barge <barge>=barge1 <from>=Richmond <to>=west-coast)
(Pump-Oil <barge>=barge1 <sector>=west-coast)
(Move-Barge <barge>=barge1 <from>=west-coast <to>=Richmond)
(Unload-Oil <barge>=barge1 <dock>=Richmond)
```

PRODIGY 4.0 searches a space of candidate plans, in which each candidate plan is represented by two structures, a *head plan* and a *tail plan* [Fink & Veloso 1995]. The head plan is a totally-ordered sequence of steps that can be applied to the initial state. The tail plan is a partially-ordered set of steps, that is, a directed acyclic graph in which there is an arc from step a to step b if a was added to the tail plan in order to satisfy some precondition of b. If a step was added to satisfy a goal that is part of the planning problem itself, then there is an arc from that step to the special node *Root*

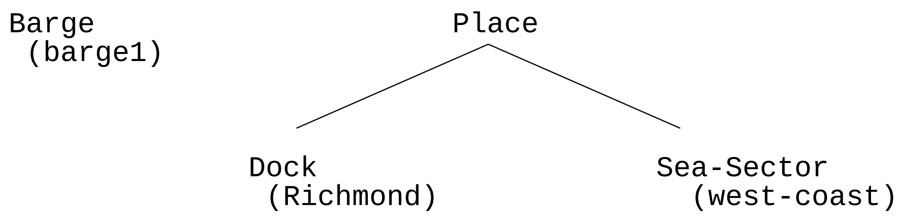

FIGURE 3.3: The type hierarchy for the planning domain. Objects in the example planning problem are shown in brackets below their type.

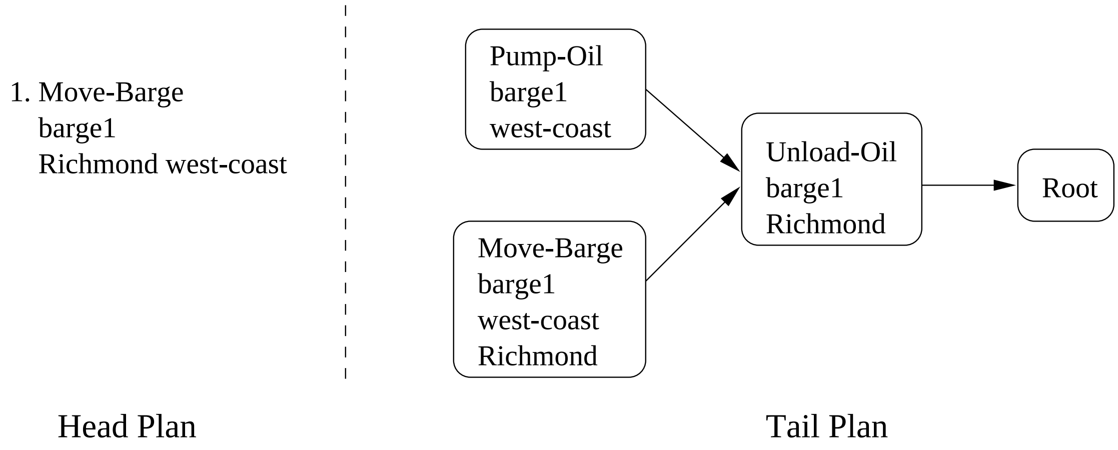

FIGURE 3.4: A candidate plan for the planning problem to dispose of the oil. The variable names have been ommitted from the instantiated operators. The head plan contains one step, to move the barge from Richmond to the west coast.

at the top of the partial order. If the head plan is applied to the initial state, it leads to a new state called the *current state*. This current state is used to guide planning in two ways. First, no steps are added to the tail plan to achieve subgoals that are already achieved in the current state. Second, steps are preferred if they have fewer preconditions that are not true in the current state. Both of these guidance rules are heuristics, and may delay finding a solution. They typically speed the search process, however. Figure 3.4 shows an example candidate plan that might be generated in the search for a solution to the current problem.

PRODIGY 4.0 begins its search with the *null plan*, in which the head plan is empty and the tail plan consists of the single node *Root*. It can expand a candidate plan in one of two ways: it can either add a new step to the tail plan to achieve some open precondition, or it can move a step from the tail plan to the head plan. A step that is moved to the head plan must be applicable in the current state. It is added to the end of the head plan, yielding a new current state. More details on the representation of plans and the search techniques used in PRODIGY 4.0 can be found in [Fink & Veloso 1995; Veloso *et al.* 1995; Veloso & Stone 1995].

For example, in Figure 3.4, the step with operator `Pump-Oil` is applicable and could be moved to the head plan. Moving it would yield a new current state, in which (oil-in-barge barge1) is true. There is only one other way that the plan in Figure 3.4 can be expanded, which is by moving the step (Move-Barge <from>=west-coast <to>=Richmond) to the head plan. However, this candidate plan would include a *state loop*, the situation where the exact same state is visited twice while the head plan is executed. This plan is therefore pruned from the search space, since if it leads to a solution there must be another more efficient solution that avoids the state loop. The plan in Figure 3.4 cannot be expanded by adding a step to the tail plan because there are no open preconditions in the tail plan.

A summary of the PRODIGY 4.0 algorithm is shown in Table 3.1 (taken from al. [Veloso *et al.* 1995]).

PRODIGY 4.0(G,I)
1. Current state C := initial state I,
Head-Plan := *null*,
Tail-Plan := *null*. 2. If the goal statement G is satisfied in the current state C, then return *Head-Plan*.
3. Either
    (A) *Back-Chainer* adds an operator to the *Tail-Plan*, or
    (B) *Operator-Application* moves an operator from *Tail-Plan* to *Head-Plan*.
  Decision point: Decide whether to apply an operator or to add an operator to the tail.
4. Goto 2.

Operator-Application
1. Pick an operator *op* in *Tail-Plan* such that
    (A) there is no operator in *Tail-Plan* ordered before *op*, and
    (B) the preconditions of *op* are satisfied in the current state C.
  Decision point: Choose an operator to apply.
2. Move *op* to the end of *Head-Plan* and update the current state C.

TABLE 3.1: Summary of the PRODIGY 4.0 planning algorithm. Each decision point is a potential backtracking point in the algorithm.

3.1.3 Organization of the search space

Table 3.1 gives a nondeterministic algorithm to construct a plan for a given planning problem. PRODIGY 4.0 maintains an audit trail of the search if performs while constructing the plan, called the *search tree*. A description of the search tree is useful for explaining some extensions made to PRODIGY 4.0 to support conditional planning and planning with external events. It is also used in Chapter 4 to describe how Weaver interacts with PRODIGY 4.0.

The nodes in the search tree represent the choices made at each decision point of the algorithm, and are typically of four types. An *applied operator node* represents the choice to move a particular step from the tail plan to the head plan. As an alternative to moving a step to the head plan, a *goal node* represents the decision to focus on an open condition in the tail plan (also known as a goal). The children of a goal node must all be *operator nodes*, which represent the decision to use a particular operator schema to achieve the goal represented by the parent node. The children of an operator node must all be *bindings nodes*, which represent the decision to use a particular set of bindings to instantiate the operator represented by the parent node. The combination of a goal node, an operator node and a bindings node together represent the act of PRODIGY 4.0 adding a step into the tail plan.

While the tail plan and the head plan together represent one candidate plan, the search tree represents all the candidate plans that have been examined and also all the ways that new candidate plans can be expanded. A path from the root node to any other node in the search tree corresponds to a candidate partial plan, which can be reconstructed by following the decisions encoded in the nodes in the path. For example, the candidate plan in Figure 3.4 is actually generated by PRODIGY 4.0 as the sequence of search tree nodes shown in Table 3.2. In this sequence, each node is the child of the node in the line above[3], so the candidate plan is produced without backtracking over any choices. Other sequences of search nodes could produce the candidate plan just as well: in particular the order in which PRODIGY 4.0 works on the goals `(oil-in-barge barge1)` and `(at barge1 Richmond)` doesn't matter. The choice of this particular sequence was largely due to search heuristics that PRODIGY 4.0 uses, which will not be discussed here (but see [Blythe & Veloso 1992] and [Carbonell *et al.* 1992]).

3.2 A representation for planning under uncertainty

In Section 3.1 I provided a brief description of planning domains, planning problems and plans in PRODIGY 4.0. Here I extend those definitions to the versions used by Weaver, in which planning domains and problems also contain information about uncertainty in the domain, and plans can specify a number of contingencies. In addition to operators, which can have more than one possible outcome, planning domains in Weaver also contain *exogenous events*, which specify ways that the world can be changed independently of the actions in a plan.

In order to give a precise characterisation of the problems that Weaver can represent and solve, I describe Weaver's representation scheme in terms of Markov decision processes. Specifically, I describe how to take a planning problem in Weaver's language and construct a Markov decision process M which is equivalent in the sense that if there is a non-looping policy for M with an expected value greater than 0 then there

[3] except `n5` which is the child of a special node that PRODIGY 4.0 uses to add the top-level goals to its list of goals.

```
n5  Goal     (disposed-oil)  top-level
n6  Operator unload-oil
n7  Bindings <unload-oil barge1 richmond>
n8  Goal     (oil-in-barge barge1)  for n7
n9  Operator pump-oil
n10 Bindings <pump-oil barge1 west-coast>
n11 Goal     (at barge1 west-coast)  for n10
n12 Operator move-barge
n13 Bindings <move-barge barge1 richmond west-coast>
n14 Apply    <MOVE-BARGE BARGE1 RICHMOND WEST-COAST>  apply n13
n16 Goal     (at barge1 richmond)  for n7
n17 Operator move-barge
n18 Bindings <move-barge barge1 west-coast richmond>
```

TABLE 3.2: An annotated trace of PRODIGY 4.0 which shows a sequence of search tree nodes created to produce the candidate plan shown in Figure 3.3. Each node except `n5` is a child of the node in the line above. The text in italics has been added to the trace output for clarity of presentation.

is a plan in the original problem with probability of success equal to that expected value, and conversely if there is a plan then there is such a policy. A description of Markov decision processes was given in Section 2.4.

Recall that in PRODIGY 4.0, a planning domain consists of a type hierarchy T, a set of literals $\mathcal{L}$ and a set of operators O. A planning problem consists of a planning domain, a set of objects belonging to each type in the hierarchy, an initial state and a goal description. Each literal in the domain has a *type signature*, specifying the number of arguments and the type of each argument, which is a member of T. Given the objects in the planning problem, the set of all possible *ground literals* L can be constructed by filling in the arguments of each literal's type signature with all objects that match, which are those objects of the exact type or any subtype of the type in the signature.

Consider the oil-spill planning problem that has been used as an example in the previous sections. The domain includes the literal `at` with signature `(at Barge Place)`. In the planning problem, there are two objects of type `Barge`, `barge1` and `barge2` and two objects belonging to subtypes of the type `Place`, `Richmond` of type `Dock` and `west-coast` of type `Sea-Sector`. Therefore there are four ground literals derived from `at` in the set L: `(at barge1 Richmond)`, `(at barge1 west-coast)`, `(at barge2 Richmond)` and `(at barge2 west-coast)`.

A *state* in a planning problem in PRODIGY 4.0 assigns a value of `true` or `false` to each literal in the ground literals L. Therefore there are $2^{|L|}$ possible states. Some of these assignments may not correspond to valid states in the planning domain being modelled. For example, `(at barge1 Richmond)` and `(at barge1 west-coast)` cannot both be true at the same time. However since any state that is reachable by a sequence of actions in the domain from a valid initial state will also be valid, these invalid states are not an issue in practice. This state property of validity could be partially derived by considering the states reachable by applying actions to any physically possible initial state, or it could be enforced by adding domain axioms such as those used in [Knoblock 1991]. In what follows I will ignore this distinction.

In Weaver the planning domain is generalised as follows.

1. The operators in the domain include a *duration* which is an integer-valued function of the bindings of the operator (and therefore a integer for an instantiated action or step). This integer may represent any time unit, for example seconds or hours, although the unit must be the same for different operators in the same domain.

2. Operators may specify a discrete, conditional probability distribution of possible outcomes rather than the single possible outcome used in PRODIGY 4.0. An example of this will be described in more detail below.

3. A planning domain includes a set of *exogenous events* E as well as the set of operators O. These are syntactically very similar to operators but are used to specify the way that the world can change independently of the actions taken in a plan, as I describe below. For example, they can be used to model the actions of other agents or natural processes.

4. A total precedence order $<$ is given over the actions and events. This is used to resolve conflicts between their effects if more than one action or event produces changes to a state. An example is given below.

Weaver generalises PRODIGY 4.0's definition of a planning problem by specifying a probability distribution of possible initial states rather than a single initial state. The problem also includes a *threshold probability*, τ, a minimum probability of success that a plan must equal or exceed to be considered a solution. The objects and goal statement in the planning problem are unchanged.

In the rest of this section I make the semantics of planning domains and problems in Weaver precise in terms of an underlying Markov decision process M defined by a planning problem. While this definition is needed to prove that Weaver correctly computes probabilities for plans and to discuss its coverage, on a casual reading of the thesis it can be skipped and replaced with the following summary: at each time step, several events may take place simultaneously with one action as a plan is executed. When more than one event or action complete in one time step, their results are applied to the state in parallel. If more than one possible value is specified for some ground literal in the state, the value nominated by the event or action that is highest in the pre-specified precedence order is used. Actions are usually higher than events in the precedence order.

Actions and exogenous events

Weaver operators are very similar to operators in PRODIGY 4.0, but include a duration function and generalise from a single outcome to multiple possible outcomes, one of which will take place when an instantiated operator is executed. Thus each operator has an *effect distribution function*, $edf()$ that takes a state as input and returns a probability distribution of effects, that is, a finite set $edf(s) = \{(p_i, \sigma_i) \mid 1 \leq i \leq n\}$ for some n, where each σ_i is a list of `add` and `delete` statements denoting an effect set as in the operators of PRODIGY 4.0, each p_i is a positive number and $\sum_{i=1}^{n} p_i = 1$.

The effect distribution function is represented as a binary tree whose internal nodes are labelled with literals mentioning variables from the domain and whose leaf nodes are labelled with probability distributions of outcomes. Each internal node has one out-arc labelled `false` and one labelled `true`. To find the appropriate effect distribution for a given state, the tree is traversed from the root node, taking the branch labelled `true` or `false` respectively depending on whether the literal at the node is true or false in the state. Figure 3.5 shows a version of the `Move-Barge` operator from the oil-spill domain used in the previous sections that has a simple branching effect distribution function with one internal node labelled with `(barge-ready <barge>)`. In this case the two possible effect distributions have the same set of outcomes with different probabilities, but this need not be true in general. The probability values are part of the domain definition, assigned externally to Weaver. Deriving them using machine learning or knowledge acquisition techniques would make an feasible extension but is not covered in this thesis.

Informally, when an instantiated operator is applied in some state s in a domain with no exogenous events, the resulting state is determined probabilistically as follows. First the effect distribution function is called to find the effect distribution for s, $edf(s)$. Then an effect set is chosen at random, with the probabilities given in the effect distribution. Finally this effect set is applied to the state in the same way that the single outcome of an instantiated operator in PRODIGY 4.0 is applied to the state. For example, if the instantiated-operator `(Move-Barge <barge>=barge1 <from>=Richmond <to>=west-coast)` is applied in any state in which `(at barge1 Richmond)` and `(operational barge1)` are true but `(barge-ready barge1)` is false, `(operational barge1)` will be true in the resulting state with probability 0.1 and will be false with probability 0.9, while `(at barge1 Richmond)` will be false and `(at barge1 west-coast)` will be true with certainty.

The situation is complicated significantly by the presence of exogenous events in the planning domain. Since events specify ways the world can change independently of an action taken in a plan, the resulting state when an action is taken depends on what events may take effect between the start and end of the action. Since some of these events may have begun before the action is taken, at least some aspects of the history of the world are needed, and in order to model such events it is necessary to introduce a notion of time. A variety of theoretical models in the AI literature can handle exogenous events and are in the same spirit as the situation calculus on

```
(Operator Move-Barge
  (duration (/ (distance <from> <to>) (speed <barge>)))
  (preconds
    ((<barge> Barge)
     (<from>  Place)
     (<to>    (and Place ( diff <from> <to>))))
    (at <barge> <from>))
  (effects
    (branch (barge-ready <barge>)

      ((0.667 ()
        ((add (at <barge> <dest>))
         (del (at <barge> <source>))))
       (0.333 ()
        ((add (at <barge> <dest>))
         (del (at <barge> <source>))
         (del (operational <barge>)))))

      ((0.1 ()
        ((add (at <barge> <dest>))
         (del (at <barge> <source>))))
       (0.9 ()
        ((add (at <barge> <dest>))
         (del (at <barge> <source>))
         (del (operational <barge>))))))))
```

FIGURE 3.5: The `Move-Barge` action from the oil-spill domain used in Weaver has a duration and a probability distribution of possible outcomes.

which the PRODIGY 4.0 representation rests *e.g.* [Allen *et al.* 1991; Kartha 1995; Shanahan 1995; Baral 1995].

In this thesis I have chosen a simple and restricted language for events that is still adequate for planning with some interesting domains. While some approaches, based on circumscription, are independent of an underlying model of the language, my approach is based on the adoption of a Markov decision process as an underlying model. This approach was chosen partly because the existing techniques did not handle probabilistic reasoning well, although the random-worlds approach is promising [Bacchus *et al.* 1997], and partly to make it easier to compare the planning system developed in this thesis with other approaches to planning under uncertainty, many of which are based on Markov decision processes [Boutilier & Dearden 1994; Dean & Lin 1995; Littman 1996].

Exogenous events have a very similar representation to operators. Figure 3.6 shows an example of a simple exogenous event called **Weather-Brightens**. It can only take effect in a state in which (poor-weather) is true, and its application leads to a state in which (poor-weather) is false and (fair-weather) is true. Unlike operators, the agent that executes plans is not free to apply this external event at will. The event will take place randomly in any state that satisfies its preconditions, with probability given in the probability slot of the event. Each event can take place any time that its preconditions are satisfied, independently of its previous occurrences and with the same probability.

As an example, Figure 3.7 shows the set of states that can result from the given initial state when the action (Move-Barge <barge>=barge1 <from>=Richmond <to>=west-coast) is executed in a domain with Weather-Brightens as the sole external event, and the action has a duration of one time unit. The probability that the event takes place in this state is independent of the action outcome in the state and this fact is used to compute the probabilities shown on each arc in the figure. The probability that (poor-weather) is true on completion of the action, for example, is found by summing the probabilities of the two states on the left of the diagram, giving 0.75. However if the action were to have a duration of two time units, the probability of (poor-weather) would be reduced to 0.5625, because the event might take place in either of two states while the action is executed.

```
(event  Weather-Brightens
  (probability 0.25)
  (duration 1)
  (preconds ()
    (poor-weather))
  (effects ()
    ((del (poor-weather))
     (add (fair-weather)))))
```

FIGURE 3.6: An exogenous event.

Suppose that the event and action may be in conflict over the value of some literal. For example, suppose that in some of its possible outcomes, Move-Barge deletes (fair-weather) and adds (poor-weather), perhaps because of pollution. In the cases where both the event and action would affect these literals, the precedence order over events and operators is used to decide their values. If Move-Barge comes before Weather-Brightens in the order, (fair-weather) will be false, otherwise it will be true. More sophisticated schemes for reasoning about simultaneous events are possible, such as the one proposed in [Shanahan 1995].

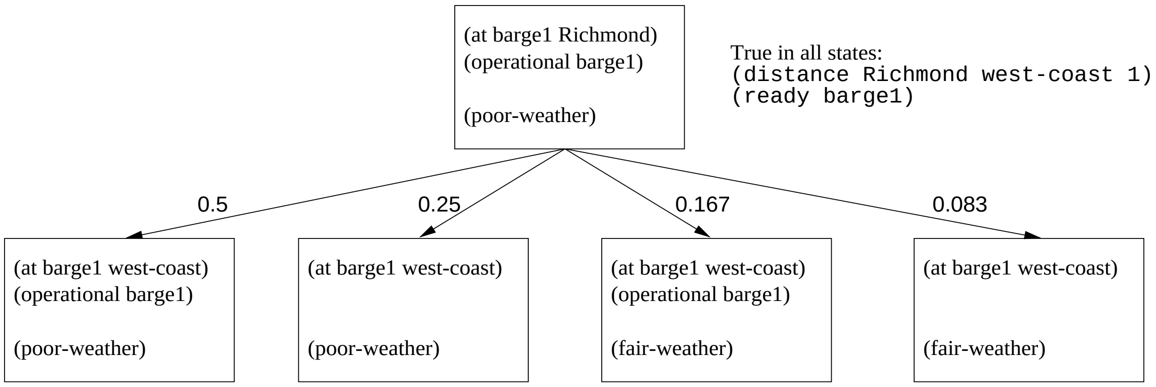

FIGURE 3.7: The set of states reachable when the action `Move-Barge` is executed from the initial state shown, when `Weather-Brightens` is the only exogenous event in the domain and the action has a duration of one time unit. The probability of reaching each state is shown along its arc.

3.3 A Markov decision process model of uncertainty

.

In the previous section I showed the result when an action is applied and one event with the same duration may take place simultaneously. In general, actions and events may have different durations and events may take effect after an action begins but before it ends. Multiple events may take place at the same time, and they may have effects that alter the same literals in different ways. In order to precisely state the probability distributions of state histories that can result from a sequence of actions in a planning problem, I introduce its underlying Markov decision process (MDP).

Recall from Chapter 2, on related work, that an MDP is defined by its state space, its state transition function and its reward function. A state in the state space of the underlying MDP M contains more information than a state in the planning problem, since it contains a limited history of actions and exogenous events that are in progress. This is because the MDP models the interaction of actions and events that have different lengths and whose time intervals may overlap, and it is useful to choose the state space and transition function such that each successor state occurs a fixed unit of time after the predecessor state. For this reason, a state in M contains a history of the events and actions taking place in the world as well as the current state described in terms of literals from the planning problem. Individual state transitions then correspond to the passage of a unit of time rather than an action or event viewed as an atomic unit.

The action and event history contained in a state in M consists of a set of pairs $n{:}\delta$ where n is an integer denoting the time before the action or event will take place, and δ is an effect set, a list of `add` and `delete` statements that specify the changes to be made to the state when the action or event will take place. There is no distinction in the history list between actions and events, since once they are added to the state there is no need for a distinction. These pairs are called *pending effects*.

Figure 3.8 shows the sequence of states in M that is traversed if a deterministic version of `Move-Barge` is executed in the beginning state. In each state, literals from the state space of the planning problem are shown above the dotted line, while below the line is shown a list of pending effects. The initial state and the two final states in the diagram have no pending effects, while each intermediate state has exactly one pending effect. When the action is performed in the left-most state, it gives rise to two possible successor states, with different pending effects corresponding to the different possible outcomes of the action. Since the length of the action is the same in each outcome, the count-down on the pending effects in each state is the same, 3. Whenever the pending effect has a count-down higher than 1, the successor state simply decrements the count-down. When the count-down reaches 1, the pending effect is not present in the successor state, but the effects themselves are applied to the portion of the MDP state corresponding to the literals in the planning problem. Thus in the final states, `(at barge1 west-coast)` is true and `(at barge1 Richmond)` is false. In the lower of the two final states, `(operational barge1)` has become false, while it is still true in the upper final state.

FIGURE 3.8: The two possible sequences of states in the underlying Markov decision process arising from executing the `Move-Barge` operator in the state shown on the left in the absence of exogenous events.

The pending effects in Figure 3.8 were added by choosing the action `Move-Barge` in the initial state. Exogenous events are also modelled by adding pending effects to the state in the MDP. For example, Figure 3.9 shows the states arising from taking the same action in an initial state that also has no pending effects but includes the literal `(poor-weather)` and in a domain that also includes the exogenous event `Weather-Brightens` shown in Figure 3.6, but with a duration of 2 time units. Since the presence of the exogenous event leads to considerably more states, I only show the domain-level state features that differ from those of the parent state. The probability that each state is reached is shown above the state. The probabilities of the state transitions are calculated from the event and action outcome probabilities, which are conditionally independent given the states in which the corresponding pending effects began.

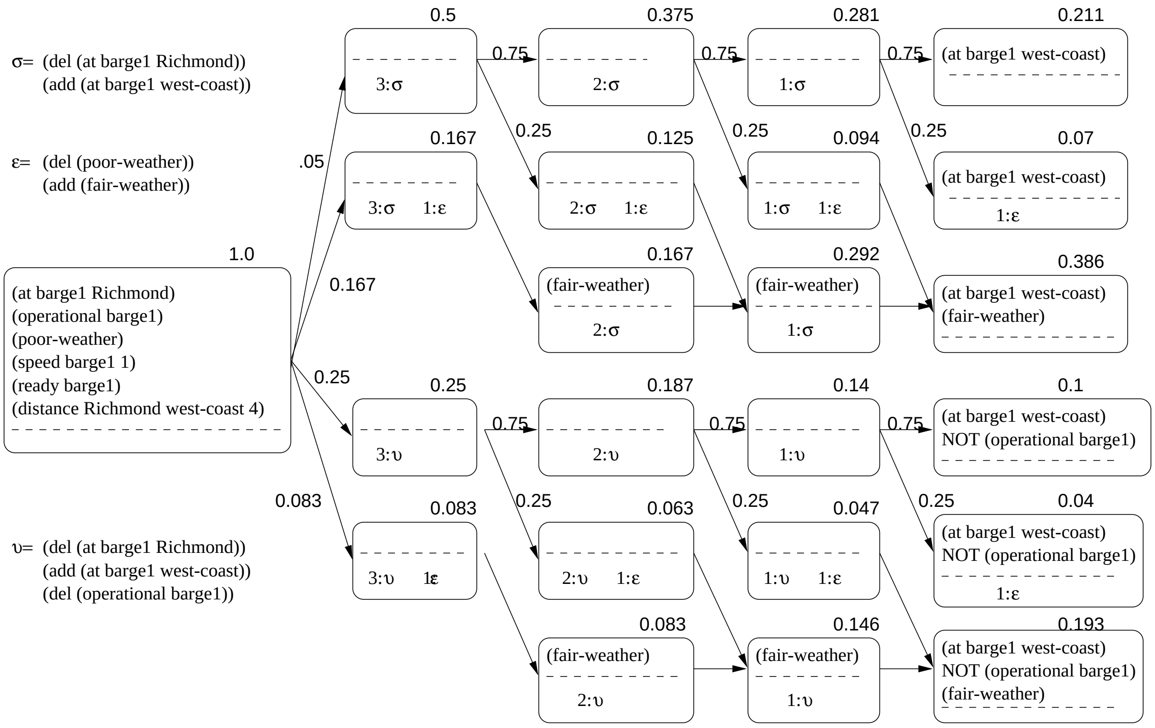

FIGURE 3.9: The possible sequences of states in the underlying Markov decision process arising from executing the `Move-Barge` operator in the state shown on the left, when the exogenous event `Weather-Brightens` can also take place. Only literals that are changed from a parent state are shown.

Formal description of the underlying MDP

Each state s of the underlying MDP M consists of two parts:

1. a truth assignment to the ground literals L, corresponding to a state in the planning domain and referred to as the *literal state* Σ.

2. a set of *pending effects* Δ.

Each element of the pending effects is a triple (t, a, σ) where t is an integer denoting a time interval, a is an operator or exogenous event and σ is a set of "effects", a set of `add` and `delete` statements over ground literals in L. The pending effects represent the events and actions that are currently taking place in the given state of M. Intuitively, t denotes the time left before the effect "completes" or is applied to change the state.

We begin by defining a "successor function" $\mu(.)$ on states of M. This function shows how a state leads to its successor if there are no exogenous events, and is used as the basis for building the transition function for the MDP.

Let s be a state in M and write $s = \Sigma \cup \Delta$ where Σ and Δ are the literal state and the pending effects as described above. To calculate the successor state $\mu(s)$, first find those pending effects whose time-to-complete value is 1:

$$\Delta^I = \{\delta = (1, a, \sigma) \mid \delta \in \Delta\}$$

Δ^I can be thought of as the "imminent" pending effects, and is applied to the literal state Σ to give a new literal state Σ', the literal state in the successor MDP state $\mu(s)$. Σ' is defined by specifying a truth value for each ground literal:

$\Sigma'(l) \;=\;$ `true`, when there is some pending effect $(1, a, \sigma) \in \Delta^I$, σ such that σ contains `(add` l`)` and a is the least operator or event in the total order with a pending effect that mentions l.

$\Sigma'(l) \;=\;$ `false`, when there is some pending effect $(1, a, \sigma) \in \Delta^I$, σ such that σ contains `(del` l`)` and a is the least operator or event in the total order with a pending effect that mentions l.

$\Sigma'(l) \;=\; \Sigma(l)$, if no such pending effect exists.

The remaining pending effects, $\Delta^R = \Delta - \Delta^I$, contribute to the pending effects of the new state $\mu(s)$ with their time count reduced by one:

$$\mu(s) = \Sigma' \cup \{(t-1, a, \sigma) \mid (t, a, \sigma) \in \Delta^R\}$$

This successor function forms the basis of the MDP M describing the actions and external events in the planning domain P. As described below, changes to the world due to actions or external events are expressed by inserting pending effects into the set Δ for the state that is changed. At each time step, the successor function is also performed. Note that the successor function is deterministic: $\mu(s)$ is uniquely defined for each state $s \in M$.

In the special case when there are no external events, the transition function $\Phi(a, s)$ that maps an action a and a state s in M to a probability distribution of new states is defined as follows:

$$\Phi(a, s) = \mu(s \cup \{(d, a, \sigma_i)\}) \text{ with probability } p_i, \text{ with } 1 \leq i \leq n$$

Where d is the duration of the action in the state s, and the effects σ_i and corresponding probabilities p_i are computed from the actions effect distribution function, $edf(s) = \{(p_i, \sigma_i) \text{ such that } 1 \leq i \leq n\}$. Thus if the duration d of action a is 1, the effects take place in the next state after a is performed, otherwise there is a delay corresponding to d before a's effects are realised.

When exogenous events may take place, they modify the possible transitions from the states in which their preconditions are satisfied. For example, the presence of the event `Weather-Brightens` leads to four possible transitions from the initial state in Figure 3.9, while the same state has two transitions when there are no exogenous events in Figure 3.8. Essentially, each combination of occurrences of the applicable exogenous events and each possible outcome of the chosen action leads to a possible transition for the state and the action. The probability of the transition is the product of the probabilities corresponding to the events and the action outcome.

Recall that exogenous events have a precondition, duration and effect distribution similar to actions, and also have a probability of occurrence p. Let E be the set of of exogenous events and let $H(E, s)$ be the set of external events whose preconditions are satisfied in the state s. Intuitively, any combination of the events in $H(E, s)$ might take place when an action is performed in state s. The probability that event h takes place and has effect $ed(s)_i$ is $p \times p_i$.

For each subset $H' \subset H(E, s)$, applying action a in state s can lead to transitions with the following probabilities:

$$\begin{aligned} \Phi(a, s) &= \mu(s \cup \{(d, a, \sigma_i)\} \bigcup_{h \in H'} (d_h, h, edf(s)_{i_h})) \\ &\quad \text{with prob } p_i \prod_{h \in H'} p_h p_{i_h} \prod_{h' \in H - H'} (1 - p_{h'}) \end{aligned}$$

This expression combines the probabilities that (1) action a has its ith possible outcome, that (2) each external event in H' takes place and has its i_hth outcome, and that (3) none of the external events in $H - H'$ takes place. These outcomes are treated as independent. If R is the maximum number of different outcomes that an event can have and $|E|$ is the number of exogenous events, then the worst case complexity of computer $\Phi(a, s)$ is exponential in $|E| \times R$ since each combination of the outcomes must be considered. In the next chapter I show how Weaver evaluates a plan typically without computing $\Phi(a, s)$, using the plan to constrain the events and outcomes considered.

Now that the state space and the state transition function have been defined for the MDP that models a particular planning problem, it remains only to specify the reward function for the MDP. Recall that in an MDP, a reward $R(a, s)$ is given for making a transition to state s using action a. An optimal policy on the MDP is a choice of an action for each state in the MDP that will maximise the expected reward. The original planning problem specifies a goal description, a logical sentence over the literals $\mathcal{L}$ of the domain. In order for the optimal policies on the MDP to have the effect of reaching a goal state and staying there, I specify a reward of 0 for all transitions between goal state, and reward of 1 for each transition from a non-goal to a goal state and a reward of -2 for each transition from a goal to a non-goal state. A new action is also added to the MDP, distinguished from the actions of the planning problem, called `stop`. This action always connects a state to the same state with a reward of 0.

It is easy to see that any optimal policy will choose the `stop` action (or an equivalent) at a goal state, since the negative reward for leaving the state dominates the positive reward for subsequently arriving at a goal state. It is also easy to see that the expected reward of any policy is equal to the probability that the policy will reach some goal state. A plan in the original planning problem can easily be mapped to a policy in the MDP in the manner described in the following example. It is reasonable to then define the probability of success of the plan as the conditional expected value of that policy, conditioned on the initial state distribution of the planning problem.

3.4 Putting it together: a complete example

In the previous sections I have developed operators and events from an oil-spill domain to illustrate elements of the representation for planning problems used in this thesis. In this section I will extend the domain, show the model for a simple planning problem and evaluate a plan as a policy on the model. I will use the planning problem introduced in Section 3.1 with the extensions introduced in Section 4.1. This example is a simplification of the oil-spill clean-up domain described in chapter 7.

Figure 3.10 shows all the operators from the planning domain, and Figure 3.11 shows all the exogenous events. The object types in the domain are the top-level types `Barge` and `Place`, and the two sub-types of `Place`, `Dock` and `Sea-Sector`. The typed literals in the domain are `(at Barge Place)`, `(oil-in-barge Barge)`, `(disposed-oil)`, `(oil-in-tanker Sea-Sector)`, `(operational Barge)`, `(barge-ready Barge)`, `(distance Place Place)`, `(poor-weather)` and `(fair-weather)`.

As in the previous sections, the planning problem considered has one object, `barge1`, of type `Barge`, one object, `Richmond`, of type `Dock` and one object, `west-coast`, of type `Sea-Sector`. Thus the 10 literals in $\mathcal{L}$ yield 11 ground literals in L, for a state space size of $2^{11} = 2048$. In fact only 48 of these states are physically possible, but this is not explicitly modelled as discussed in Section 3.2. The planning initial state and goal are shown in Table 3.3.

In Section 4.1 I showed how a conditional planner based on Prodigy can compute a plan for this problem with one conditional branch. In this section I will describe the underlying MDP for this planning problem, show the expected reward for this plan interpreted as a policy on the MDP and explain the relationship between the expected reward and the plan's probability of success.

Figure 3.12 shows a reachability graph of the states reachable from the initial state of the example planning problem through sequences of operators and exogenous events. These are states in the state space of the planning problem, not the underlying MDP, because they do not contain any information about pending effects. Only 12 states are reachable because `(barge-ready barge1)` is true in the initial state and cannot be made false, and because the weather can only be altered by exogenous events. For brevity, the object `barge1` is not shown in the literals in which it is an

```
(operator  Unload-Oil
  (preconds ((<barge> Barge)
             (<dock> Dock))
   (and (at <barge> <dock>)
        (oil-in-barge <barge>)))
  (effects ()
   ((del (oil-in-barge <barge>))
    (add (disposed-oil)))))
```

```
(operator  Pump-Oil
  (preconds ((<barge> Barge)
             (<sector> Sea-Sector))
   (and (at <barge> <sector>)
        (operational <barge>)
        (fair-weather)
        (oil-in-tanker <sector>)))
  (effects ()
   ((add (oil-in-barge <barge>))
    (del (oil-in-tanker <sector>)))))
```

```
(operator  Make-Ready
  (preconds ((<barge> Barge))
            (true))
  (effects ()
    ((add (barge-ready <barge>))
    )))
```

```
(operator  Move-Barge
  (duration (/ (distance <from> <to>)
               (speed <barge>)))
  (preconds
    ((<barge> Barge)
     (<from>  Place)
     (<to>    (and Place ( diff <from> <to>))))
    (at <barge> <from>))
  (effects
    (branch (barge-ready <barge>)
      ((0.667 ()
        ((add (at <barge> <dest>))
         (del (at <barge> <source>))))
       (0.333 ()
        ((add (at <barge> <dest>))
         (del (at <barge> <source>))
         (del (operational <barge>)))))
      ((0.1 ()
        ((add (at <barge> <dest>))
         (del (at <barge> <source>))))
       (0.9 ()
        ((add (at <barge> <dest>))
         (del (at <barge> <source>))
         (del (operational <barge>))))))))
```

FIGURE 3.10: The operators from the simple oil-spill domain used in this chapter. When the duration of an operator isn't mentioned, it defaults to 1.

argument, only true literals are shown and the literals `(barge-ready barge1)` and `(fair-weather)`, which are always true, are not shown.

If the domain included no exogenous events, then the underlying MDP for the problem would be similar to the reachability graph except that each transition corresponding to the action `move-barge` would lead to a sequence of 3 transitions encoding the count-down of the pending effects. A part of this MDP, restricted only to the four states reachable from the initial state in which `(oil-in-tanker)` is true, is shown in Figure 3.13. In this diagram, literal information is not shown when a state's literals match those of its parents. There are 4 different sets of pending effects that are marked with letters. For example, "a" corresponds to {`(del (at barge1`

```
(event Weather-Brightens        (event Weather-Darkens
  (probability 0.25)              (probability 0.25)
  (duration 1)                    (duration 1)
  (preconds ()                    (preconds ()
    (poor-weather))                 (fair-weather))
  (effects ()                     (effects ()
    ((del (poor-weather))           ((del (fair-weather))
     (add (fair-weather)))))         (add (poor-weather)))))
```

FIGURE 3.11: The exogenous events from the oil-spill domain.

(at barge1 Richmond)	t	Goal: (disposed-oil)
(at barge1 west-coast)	f	
(oil-in-barge barge1)	f	
(disposed-oil)	f	
(oil-in-tanker west-coast)	t	
(operational barge1)	t	
(barge-ready barge1)	t	
(distance Richmond west-coast 4)	t	
(distance west-coast Richmond 4)	t	
(poor-weather)	f	
(fair-weather)	t	

TABLE 3.3: The initial state and goal description. Here t and f stand for "true" and "false".

Richmond)),(add (at barge1 west-coast))} and "c" corresponds to the effects in "a" together with (del (operational barge1)). The transitions that correspond to initiating an action are labelled with their probabilities, but intermediate transitions are not labelled.

The extension of this MDP to one with the full graph including the effects of exogenous events is shown in Figure 3.14. The MDP is composed of two layers. The literal (fair-weather) is true in each state in the top layer and (poor-weather) is true in each state of the lower layer. Within each layer there are three groups of states that are similar to the states in Figure 3.13. In the first, the literal (oil-in-tanker west-coast) is true, in the second, (oil-in-barge barge1) is true and in the third, (disposed-oil) is true. Each layer is therefore similar to the reachability graph shown in Figure 3.12. Including the other actions and the two exogenous events does not increase the number of states because they all have durations of 1 time unit and so do not give rise to intermediate states. Therefore the resulting MDP has 96 states. For each transition within a layer, one of the exogenous events causes another transition, which is 1/3 as likely, with the same starting point but the equivalent state in the other layer as the ending point. A few of these transitions are illustrated in Figure 3.14 by the dotted lines towards the bottom of the figure.

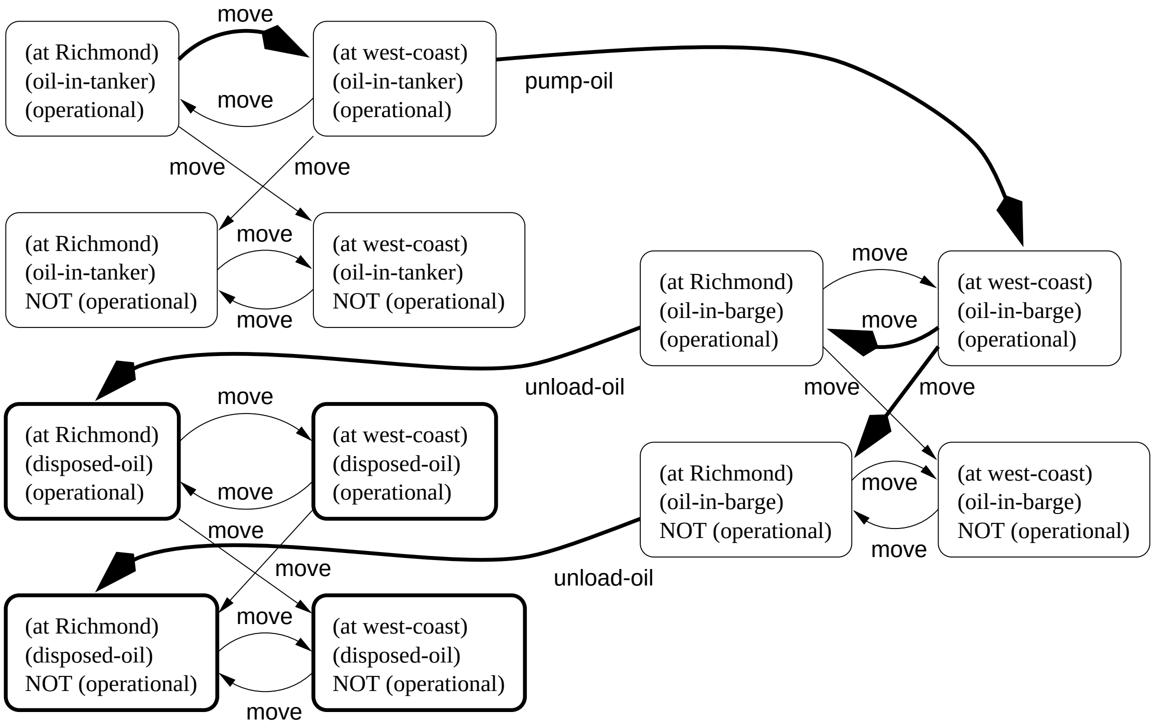

FIGURE 3.12: Reachability graph of literal states in the planning problem from the initial state, considering only operators. The initial state is shown in the top left-hand corner, and the four goal states are shown in bold in the bottom left-hand corner. Actions corresponding to the plan that is produced for this problem are shown in bold.

The plan found by the conditional planner described in Section 4.1 is:

```
(Move-Barge <barge>=barge1 <from>=Richmond <to>=west-coast)
 (Pump-Oil <barge>=barge1 <sector>=west-coast)
 (Move-Barge <barge>=barge1 <from>=west-coast <to>=Richmond)
 (Unload-Oil <barge>=barge1 <dock>=Richmond)
```

This plan is illustrated by the bold lines in Figure 3.14. These bold lines also represent a partial policy on the MDP, a choice of action for some of the states in the MDP. The plan can be extended to a full policy by any arbitrary assignment of actions to the remaining states. I choose the following assignment, which is guaranteed not to increase the expected reward of the policy: A new action called `fail` is added to the set of actions in the MDP. This action does not appear in the planning domain. The action is defined in every state of the MDP to loop back to the same state with probability 1. The reward function assigns a reward of 0 for each such transition. The partial policy is extended by choosing this action in every state where the plan does not choose an action.

The plan used for this problem does not branch, consisting of a linear sequence of actions. The partial policy is made from the plan by choosing the first action in each state that has non-zero probability in the initial state distribution, choosing the

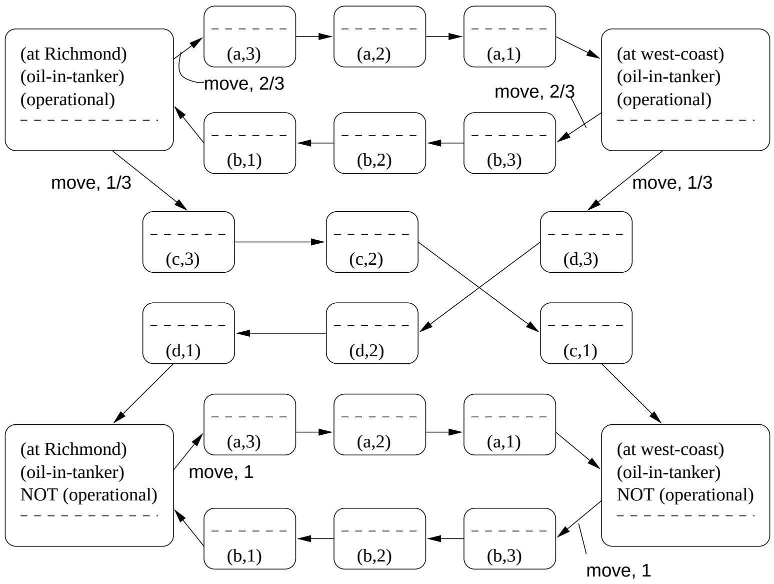

FIGURE 3.13: Part of the MDP that would correspond to the example problem if it had no exogenous events, restricted to the states in which `(oil-in-tanker)` is true. All transitions and intermediate states correspond to a `move-barge` action. The transitions that correspond to initiating an action are labelled with their probabilities. Intermediate transitions are not labelled.

second action in each state that is reached when the first action completes, and so on. If a state that is reached already has an action choice, in other words the plan has visited this state previously with non-zero probability, the action chosen first remains. If every possible sequence of states through the MDP that arose from following the plan had a loop, this algorithm would not be able to assign every action in the plan. However, Prodigy includes a mechanism for avoiding loops and, in the absence of exogenous events, at least one path through its plan will complete. I will discuss the case for planning with exogenous events in the following chapter.

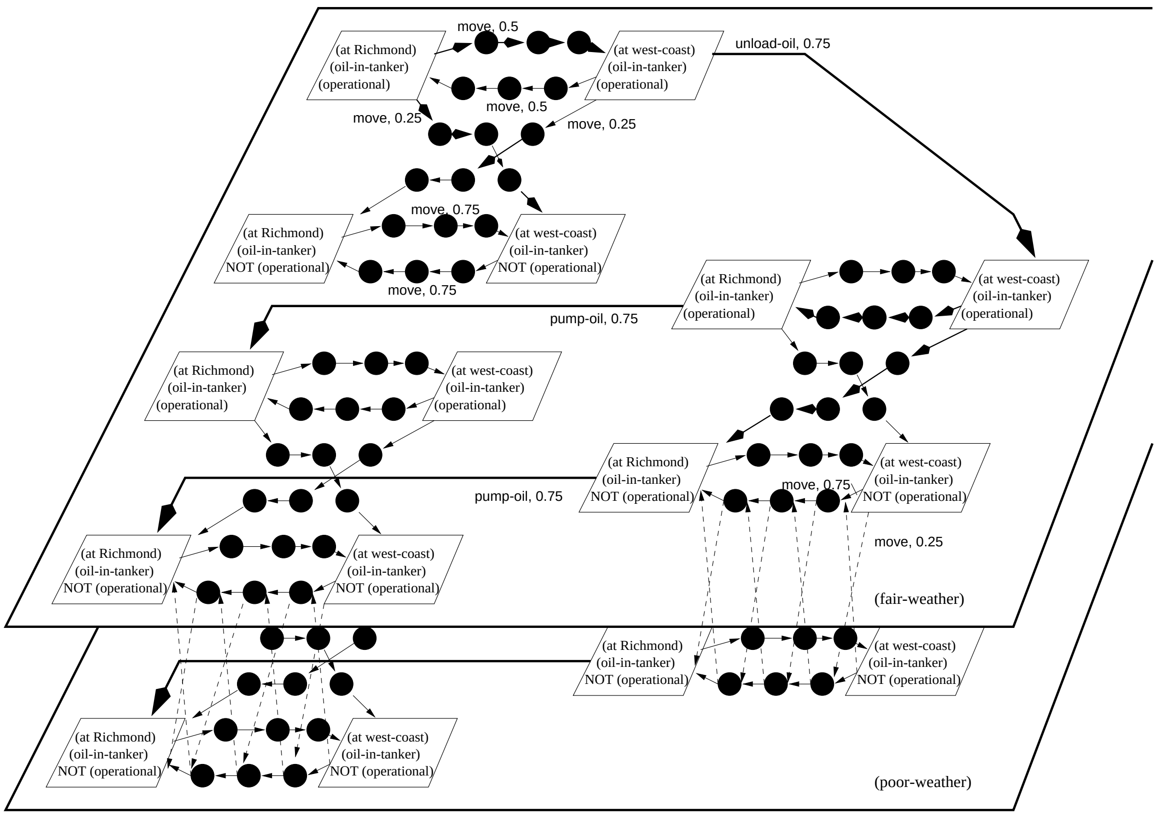

FIGURE 3.14: The full set of states for the MDP for the example problem. The transitions due to exogenous events are shown with dotted lines, connecting the two planes into which the states are divided. In the upper plane the state variable `(fair-weather)` is true and in the lower plane `(poor-weather)` is true.

Chapter 4

The Weaver Algorithm

In the previous chapter I presented a model of planning under uncertainty, describing the representation of uncertain planning problems and the semantics of the representation in terms of a Markov decision process. In this chapter I provide a broad overview of Weaver's algorithm. Weaver operates in a loop, shown in Figure 4.1. On each iteration it creates a candidate plan, evaluates its probability of success and uses the evaluation to suggest ways to improve the candidate plan. Each of these three activities corresponds to a module in the system and is described below.

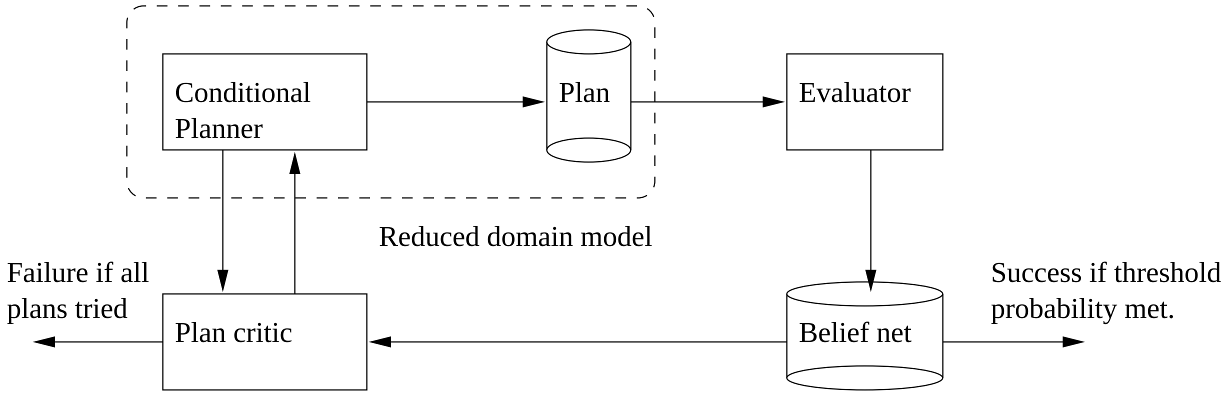

FIGURE 4.1: Weaver's main loop has three modules: a conditional planner, a plan evaluator and a plan critic.

Given a planning problem, one can always produce its underlying Markov decision process as shown in the last chapter and, in principle, solve it using standard techniques such as policy iteration to find a plan. This approach is computationally intractable in practice because the model grows rapidly as objects and actions are added to the domain. The idea behind Weaver is to use goal-directed planning to focus attention on a version of the original problem that ignores irrelevant features, in which a plan can be formulated without explicitly reasoning about the full Markov decision process. After creating such a plan, Weaver creates a parsimonious model of its probability of success, using the "evaluator" module in Figure 4.1. This model is used by the "plan critic" module to increase the subset of features considered to include all the relevant ones, that is, all the features that are used to compute the plan's probability of success. Once the feature set has been increased a new plan is produced if necessary and the process is repeated, allowing the model of the planning problem to grow incrementally.

The smaller version of the planning problem that ignores many features is referred to in Figure 4.1 as the "reduced domain model". As the figure shows, the conditional planner works in this reduced domain model while the plan evaluator and plan critic do not. However, the latter modules still take a constrained view of the planning problem, because they take the plan as their input and only consider domain features that affect it. The precise way in which each module constrains the next is made more clear as the modules are described below. In the next section I describe the conditional planner and in Section 4.2 I describe how the plan is evaluated with a probabilistic model that captures only the relevant features. In Section 4.3 I show how the plan critic updates the plan by increasing the number of sources of uncertainty made visible to the planner.

4.1 Conditional planning

Planners that deal with uncertainty must be able to create and evaluate conditional plans, which contain branches. For example, suppose that the `Move-Barge` operator can have two possible outcomes, in one of which the barge is made inoperable even if it is ready. If the planning problem contains more than one barge, then a plan to transfer the oil with high probability should be conditional on the state of the barge being used. Several planning systems are capable of solving problems like this, such as CNLP [Peot & Smith 1992], Cassandra [Pryor & Collins 1996] and C-Buridan [Draper, Hanks, & Weld 1994]. In this section I first describe some extensions made to PRODIGY 4.0's set of search node types in order to support planning in non-deterministic domains, then describe an algorithm that extends PRODIGY 4.0 to be able to create conditional plans.

4.1.1 Extensions to the PRODIGY 4.0 node set to support probabilistic planning

Weaver interacts with Prodigy by analysing a candidate plan, inserting new nodes into the search tree and re-starting Prodigy's search tree at some node of its choosing. The details of this process are provided in Chapter 4. In order to support this process, two new types of nodes are added to the vocabulary of PRODIGY 4.0 to represent new decisions that can be made about the plan. These are *context nodes* and *protection nodes*. A *context node* represents the decision to treat a particular step in the tail plan as having a set of alternative possible outcomes, rather than just one possible outcome as is the standard. The use of these nodes is described in Section 4.1.2, while here I cover protection nodes.

A *protection node* represents a decision to augment the preconditions of a step that is already in the tail plan with extra preconditions. Weaver uses protection nodes to specify extra preconditions to ensure that either a possible outcome of the step will not occur or that some exogenous event will not occur simultaneously with the step.

Prodigy can make use of protection nodes independently of Weaver to ensure that some chosen conditional effects of a step will not occur. In fact, some mechanism of this kind is required for planners in the Prodigy family to be complete [Blythe & Fink 1997]. As an illustration, suppose that the `Move-Barge` and `Pump-Oil` operators are extended so that `Pump-Oil` requires that the barge be operational, and `Move-Barge` may conditionally delete that the barge is operational if the literal `barge-ready` is false. These altered operators are shown in Figure 4.2.

```
(Operator  Pump-Oil
  (preconds ((<barge> Barge)
             (<sector> Sea-Sector))
   (and (at <barge> <sector>)
        (operational <barge>)
        (oil-in-tanker <sector>)))
  (effects ()
   ((add (oil-in-barge <barge>))
    (del (oil-in-tanker <sector>)))))
```

```
(Operator  Make-Ready
  (preconds ((<barge> Barge))
             (true))
  (effects ()
    ((add (barge-ready <barge>))
    )))
```

```
(Operator  Move-Barge
  (preconds
    ((<barge> Barge)
     (<from>  Place)
     (<to>    (and Place ( diff <from> <to>))))
    (at <barge> <from>))
  (effects ()
   ((del (at <barge> <from>))
    (add (at <barge> <to>))
    (if (~ (barge-ready <barge>))
        ((del (operational <barge>)))))))
```

FIGURE 4.2: The `Pump-Oil` and `Move-Barge` operators are modified respectively to require and conditionally delete `(operational <barge>)`. The new operator `Make-Ready` can add `barge-ready`.

If the initial state is unchanged from that of Section 3.1.2, so that `(barge-ready barge1)` is false, then PRODIGY 4.0 as described is not able to solve the problem with the new operators. This is because it has no way to add the step `(Make-Ready <barge>=barge1)` to the tail plan, since it does not achieve either the top-level goal

```
n5  Goal     (disposed-oil)  top-level
n6  Operator unload-oil
n7  Bindings <unload-oil barge1 richmond>
n8  Goal     (oil-in-barge barge1)  for n7
n9  Operator pump-oil
n10 Bindings <pump-oil barge1 west-coast>
n11 Goal     (at barge1 west-coast)  for n10
n12 Operator move-barge
n13 Bindings <move-barge barge1 richmond west-coast>
n14 Protect  (operational barge1) from n13 with (barge-ready barge1)
n15 Goal     (barge-ready barge1)
n16 Operator make-ready
n17 Bindings <make-ready barge1>
n18 Apply    <MAKE-READY BARGE1>  apply n17
n19 Apply    <MOVE-BARGE BARGE1 RICHMOND WEST-COAST>  apply n13
n16 Goal     (at barge1 richmond)  for n7
n17 Operator move-barge
n18 Bindings <move-barge barge1 west-coast richmond>
```

TABLE 4.1: An annotated trace of Prodigy, showing the use of a protection node (n14) to add a precondition to the step at node n13 which will stop its conditional effect from taking place.

or any open precondition in the problem. However the step is necessary because otherwise the first step of the plan,
`(Move-Barge <barge>=barge1 <from>=Richmond <to>=west-coast)`,
will render the barge inoperable.

The key fact about the original plan is that a literal is made false by a conditional effect that is later required to be true but cannot be made true. This indicates that a plan might be found by stopping the conditional effect from taking place, which may be done by requiring that the conditions of the conditional effect be false when the step is moved to the head plan. I created a version of Prodigy that is able to notice this condition and to add a protection node to the search tree that replaces the preconditions of this step with the conjunction of its preconditions and the negation of the conditions of the conditional effect, similar in spirit to the confrontation step of UCPOP-style planners [McAllester & Rosenblitt 1991]. This is described in [Blythe & Fink 1997]. Table 4.1 shows the sequence of nodes in the search tree that correspond to a solution based to the new problem based on the previous solution sequence in Table 3.2.

4.1.2 B-Prodigy

Weaver's conditional planner, B-PRODIGY, is a modified version of PRODIGY 4.0 described in Section 3.1, that creates branching plans. B-PRODIGY, which stands for Branching Prodigy, allows steps in the tail plan to have more than one possible outcome, and its planning algorithm has two principal modifications to PRODIGY 4.0. First, steps in both the head plan and the tail plan are associated with *contexts*, which represent the branch of the plan under which the step is proposed to be used. A step's context is a first-order logical expression over variable-value pairs for context variables. When a step with more than one possible outcome is introduced into the tail plan, a new context variable is introduced into the plan, with possible values representing the possible outcomes of the step, since each outcome may lead to a separate branch in the plan. The step is termed the *producer* of the context variable. The context associated with a step may involve several context variables. For each variable the context may include a single value, the negation of a value or a disjunction of several values. The different variables are combined as a conjunction.

The second modification to the PRODIGY 4.0 algorithm takes effect when a branching step is moved from the tail plan to the head plan, at which time the B-PRODIGY routine is called recursively for each possible outcome of the step. On each call, the remaining steps in the tail plan that do not match the context corresponding to the chosen outcome of the branching step are ignored. The several calls together produce a branching head plan. These two alterations are sufficient to create branching plans in uncertain domains. Table 4.2 shows the top-level algorithm. The parts printed in bold are only present in B-PRODIGY, the others are from the original PRODIGY 4.0 algorithm.

Figure 4.3 shows a simplified version of the `Move-Barge` operator from the planning domain used in the previous chapter in which moving a barge has a 1/3 probability of rendering it inoperable. The `effects` slot of the operator contains two lists of `add` and `delete` changes labelled with their probability of occurrence instead of the single list used before. When the operator is applied, exactly one of these lists is chosen randomly with the given probabilities and used to specify the effects of the operator. The other operators are unchanged from the domain of Section 3.4.

Figure 4.4 shows a tail plan constructed for a candidate plan for a planning problem in which two barges, `barge1` and `barge2` are at the dock `Richmond`, the oil is in a tanker at `west-coast` as in the previous chapter, and the goal is `(oil-disposed)`.

Step `s3`, `(Move-Barge <barge>=barge1 <from>=Richmond <to>=west-coast)`, is a context-producing step that has two possible sets of effects. Marking a step as context-producing is done externally to B-PRODIGY by Weaver as described in Section 4.3. The choice to suppress the fact that some steps have multiple outcomes is part of the definition of the reduced domain model. Using a reduced model can greatly reduce the complexity of building a plan by directing the planner's effort to the sources of nondeterminism that impact the plan's probability of success. The effects marked with context `s3` $= \alpha$ correspond to the situation where `barge1` remains

B-Prodigy
1. If the goal statement G is satisfied in the current state C, then return *Head-Plan*.
2. Either
   (A) *Back-Chainer* adds an operator **with the same context as the step it is linked to** to the *Tail-Plan*, or
   (B) *Operator-Application* moves an operator from *Tail-Plan* to *Head-Plan*.
   Decision point: Decide whether to apply an operator or to add an operator to the tail.
3. Goto 1.

If an operator with multiple possible consequences was moved to *Head-Plan*, **call** *B-Prodigy* **on each resulting plan, in each case removing from** *Tail-Plan* **all operators that do not match the appropriate context.**

Operator-Application
1. Pick an operator *op* in *Tail-Plan* such that
   (A) there is no operator in *Tail-Plan* ordered before *op*, and
   (B) the preconditions of *op* are satisfied in the current state C.
   Decision point: Choose an operator to apply.
2. Move *op* to the end of *Head-Plan* and update the current state C.

TABLE 4.2: Algorithm for B-PRODIGY, based on PRODIGY 4.0. Steps which only appear in B-PRODIGY are shown in bold font.

operational when it is moved, and the effects marked with context `s3` $= \beta$ correspond to the situation where it is no longer operational.

Step `s2` is marked as belonging to context `s3` $= \alpha$. This means that the planner chose to add a new link for the goal it achieves, oil-pumped, in the context `s3` $\neq \alpha$. Another choice open to the planner would be to use step `s1` in context β as well as α, and then it would have to achieve the precondition that `barge1` be operational in that context. Since no actions are available to make the barge operational, this option was rejected.

In Figure 4.5 the step `s3`, moving `barge1`, has been moved to the head plan. Two recursive calls to B-PRODIGY will now be made, one for each context, in which B-PRODIGY will proceed to create a linear plan for the context. In context `s3` $= \alpha$, steps `s4`, `s5` and `s6` will be removed from the tail plan, and in context `s3` $= \beta$, steps `s1` and `s2` will be removed. Note that no steps are removed from the head plan, whatever their context. Each recursive call produces a valid linear plan, and the result is a valid conditional plan that branches on the context produced by step `s3`.

The final plan produced after the planner solves the α and β contexts is as follows:

```
(Move-Barge <barge>=barge1 <from>=Richmond <to>=west-coast)
If (operational barge1)
     (Pump-Oil <barge>=barge1 <sector>=west-coast)
     (Move-Barge <barge>=barge1 <from>=west-coast <to>=Richmond)
```

```
(Operator Move-Barge
  (preconds
    ((<barge> Barge)
     (<from>  Place)
     (<to>    (and Place ( diff <from> <to>))))
    (at <barge> <from>))
  (effects
    (0.667 ()   ;; context α
      ((add (at <barge> <dest>))
       (del (at <barge> <source>))))
    (0.333 ()   ;; context β
      ((add (at <barge> <dest>))
       (del (at <barge> <source>))
       (del (operational <barge>))))))
```

FIGURE 4.3: A version of the operator `Move-Barge` with two possible outcomes.

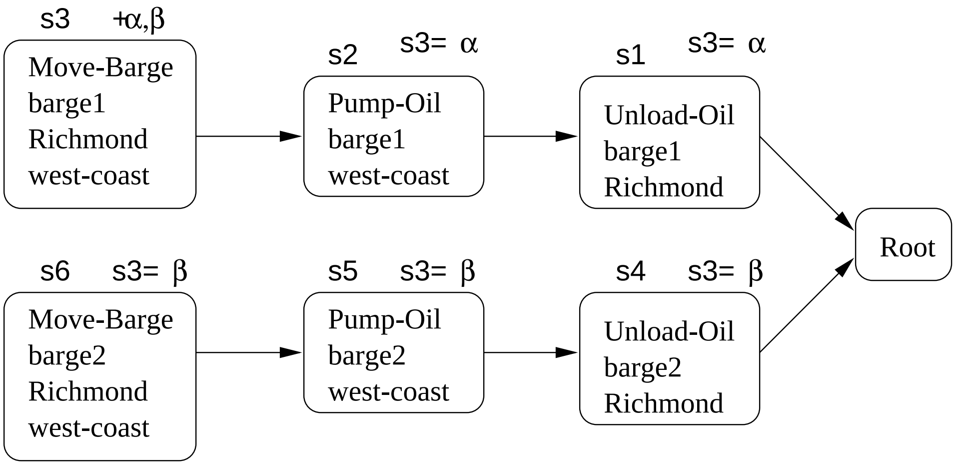

FIGURE 4.4: An initial tail plan to solve the example problem, showing two alternative courses of action. Step `s3` has two different possible outcomes, labelled α and β. The steps along the top row are restricted to outcome α of `s3`, and use `barge1` in the situation when it is operational. The steps along the bottom row are restricted to outcome β of `s3` and use `barge2`.

```
      (Unload-Oil <barge>=barge1 <dock>=Richmond)
otherwise
      (Move-Barge <barge>=barge2 <from>=Richmond <to>=west-coast)
      (Pump-Oil <barge>=barge2 <sector>=west-coast)
      (Move-Barge <barge>=barge2 <from>=west-coast <to>=Richmond)
      (Unload-Oil <barge>=barge2 <dock>=Richmond)
```

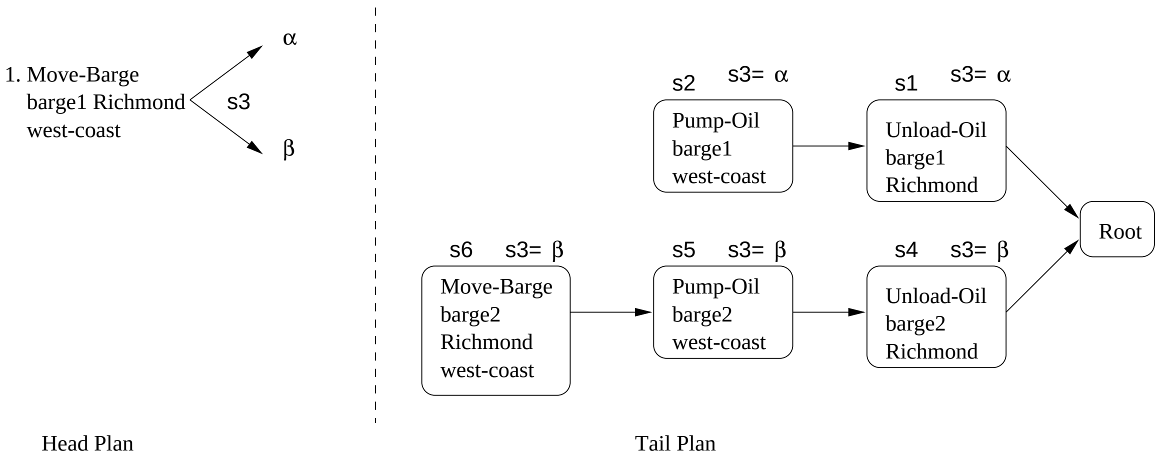

FIGURE 4.5: A more developed version of the candidate plan. One step has been moved to the head plan and now determines two possible current states, given context labels α and β.

When there is only one, universal context, this algorithm is identical to that of PRODIGY 4.0. As with PRODIGY 4.0, it is easy to see that the algorithm is sound, yielding only correct plans. Central to the completeness of *back-chainer* is the use of contexts in the tail plan. Contexts allow steps to appear in the head plan that are required for parts of the plan with a specific context before that context is resolved (for example, taking an umbrella because it might rain).

The completeness of B-PRODIGY rests on the completeness of the underlying classical planner. If this is complete, in the sense that if there is a non-branching plan for a problem in any classical domain then the planner will find a plan, then B-PRODIGY is complete for the nondeterministic problems that can be represented to it. Here I make an informal argument that this is the case. Consider a branching plan that achieves its goal with certainty. It can be viewed as a merge of several linear plans, one for each leaf of the tree-structured plan, with the steps before some branch point possibly establishing preconditions of operators on more than one branch after the branch point. If the classical planner is complete, the tail plan for each individual branch of the plan could in principle be constructed. For each branch point in the original plan, a context-producing action can then be designated allowing the required steps from the component plans to be present in the tail-plan for the branching plan, designated with different contexts. Thus, B-PRODIGY is complete for branching plans if the classical planner on which it is based is complete.

In general it is not possible to guarantee the completeness of Prodigy because it allows arbitrary functions to be used to determine bindings for operators and in the antecedents of control rules. In earlier work, completeness is shown for a version of PRODIGY 4.0 in a limited case when no lisp functions are used in the bindings of operators and no control rules are used [Blythe & Fink 1997]. Extensions to control rules and functions known to conform to certain conditions would be possible.

4.2 Calculating the probability of success

To compute the probability of success of a plan produced by B-PRODIGY in the framework of Section 3.2, we can appeal to the underlying Markov decision process described there. If each barge has an independent probability of 1/3 of becoming unoperational when it is moved, the probability of success of the initial plan drops from 5/8 to 5/12, and the probability of success of the branching plan is approximately 0.535. The improvement from the contingent plan that uses `barge2` is reduced by the continuing chances of deterioration in the weather conditions.

When the plan is implemented as a policy in the MDP, it generates a simple Markov process, part of which is illustrated in Figure 4.6. The probability of success can be found by summing the probabilities of each of the goal states in the Markov process, which are shown in the figure with thick borders. In this figure, the thick dotted line separates the state nodes for the two different branches of the contingent plan. The nodes above the dotted line correspond to the case in which `barge1` is operational after it is moved, and below the line to the case in which `barge1` is not operational. For brevity, Figure 4.6 omits the steps of moving either barge back to the dock and unloading the oil, and the states marked as goals satisfy `(not (oil-in-tanker west-coast))`. Since the omitted steps do not fail unless previous steps have failed, the probability of reaching one of these states is the same as the probability of reaching a goal state in the larger MDP corresponding to the original goal. Thus, the plan evaluated in Figure 4.6 is the following:

```
(Move-Barge <barge>=barge1 <from>=Richmond <to>=west-coast)
If (operational barge1)
      (Pump-Oil <barge>=barge1 <sector>=west-coast)
otherwise
      (Move-Barge <barge>=barge2 <from>=Richmond <to>=west-coast)
      (Pump-Oil <barge>=barge2 <sector>=west-coast)
```

This plan will be used as an example throughout this section.

In order to perform the probability calculations efficiently, a Bayesian belief net is automatically constructed from the plan. This representation has a number of advantages for reasoning probabilistically about plans. It makes efficient use of the dependency structure between domain features when computing probabilities, and has a sound theoretical basis [Pearl 1988]. It can also efficiently maintain beliefs about unobservable world features based on observations during plan execution.

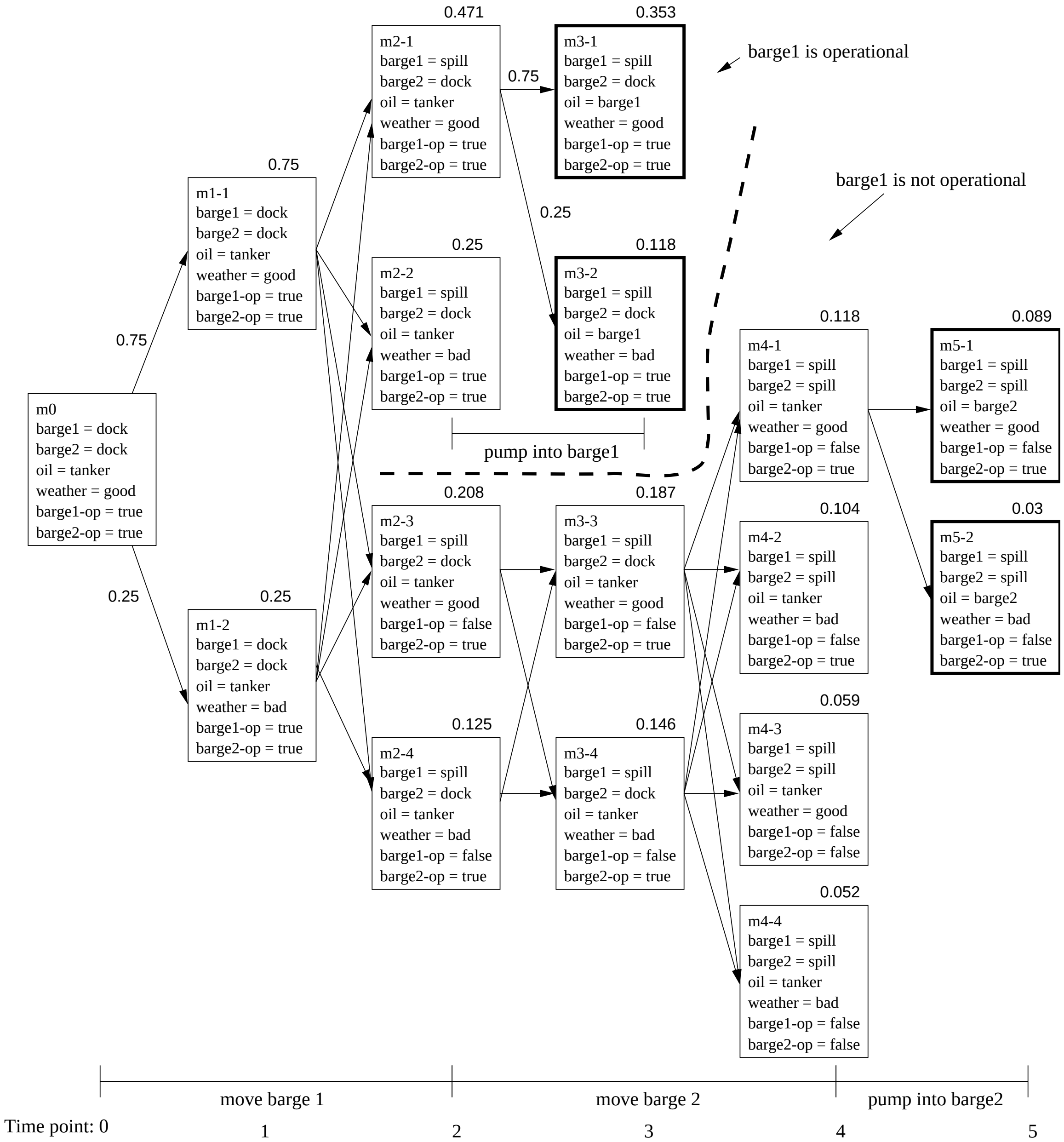

FIGURE 4.6: A reachability graph of states in the planning problem corresponding to the conditional plan to transfer the oil with one of two barges. Goal states are marked with a thick border, and the probability of arriving at each state is shown above it. Time increases from left to right, and the actions taken are shown with their intervals below the nodes.

Overview of Bayesian belief networks

Before describing in detail the algorithm used to construct a belief network that models the plan's probability of success, I give here a brief overview of Bayesian belief networks, based on [Pearl 1988]. In the rest of the thesis I will refer to Bayesian belief networks as "belief networks" or "belief nets".

A Bayesian belief network is a directed acyclic graph (or DAG) whose nodes represent random variables over which it encodes a probability distribution. A simple example of a belief net is shown in Figure 4.7. Essentially, the arcs in a belief net encode information about local dependencies between variables that can be used to construct the full, global probability distribution. Algorithms for computing the probability distribution and for propagating observations on the variables as evidence to update the posterior distributions of the other variables can make use of the dependency information to improve efficiency [Pearl 1988; Lauritzen & Spiegelhalter 1988]. This problem is, however, NP-hard [Cooper 1990].

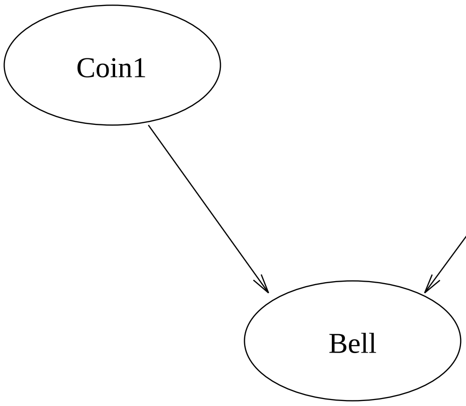

FIGURE 4.7: A small example Bayesian belief network.

Formally, if a belief net models a probability distribution, then dependence relationships among variables in the distribution can be read from the graph using the criterion of d-separation. Let D be a DAG that is a belief net, and let X, Y and Z be disjoint subsets of the variables in D. Then Z is said to d-separate X from Y if along every path from a node in X to a node in Y there is a node w satisfying one of the following conditions: (1) w has converging arrows and none of w or its descendants are in Z or (2) w does not have converging arrows and w is in Z. This is written $< X|Y|Z >_D$. In Figure 4.7, the variable `bell` d-separates `coin1` and `coin2`.

Let D be a DAG and let $I(X,Y,Z)_P$ stand for the statement "Z is conditionally independent of X given knowledge of Y in the probability distribution P", where X, Y and Z are subsets of the random variables in D. Then D is an *I-map* of the probability distribution P if every d-separation displayed in D corresponds to a valid conditional independence relationship in P, *i.e.* for every disjoint sets of vertices X, Y and Z,

$$< X|Y|Z >_D \rightarrow I(X,Z,Y)_P.$$

The DAG D is a Bayesian belief network of probability distribution P if and only if it is a minimal I-map of P. That is, if D is an I-map of P, but removing any arrow from D will make it cease to be an I-map of P.

It would be cumbersome to use this definition to verify that some DAG constructed for a probability distribution P is indeed a belief net for P. The following theorem due to Verma [86] makes this considerably easier:

Theorem *Given a probability distribution $P(x_1, x_2, \ldots, x_n)$ and any ordering d of the variables, the* DAG *created by designating as parents of X_i any minimal set Π_{X_i} of predecessors satisfying*

$$P(x_i|\Pi_{X_i}) = P(x_i|x_1, \ldots, x_{i-1}), \Pi_{X_i} \subseteq \{X_1, X_2, \ldots, X_{i-1}\}$$

is a Bayesian network of P.

Constructing a belief net to evaluate a plan

The belief net constructed to evaluate a plan has time-stamped nodes of two types: those representing the belief that some fact about the world is true at a certain time and those representing the belief that an executed action or exogenous event has a particular outcome at a certain time. The belief net is constructed in two stages. First, nodes are created representing beliefs in the state variables that are relevant to the plan regardless of any external events that may take place. In the second stage nodes are added that represent external events and the state features that are relevant to them. An example of such a net can be found in Figure 4.9.

The algorithm for creating the topology of the belief net is shown in Figure 4.8. Before nodes representing the state of the world are created, a transformation is made to the state representation so that literals that are functionally related are grouped into one random variable in the belief net. For example, since a barge can only be in one place at one time, only one of `(at barge1 Richmond)` and `(at barge1 west-coast)` is true in a state. These literals are grouped together to contribute to one state variable, `location(barge1)` which has the value `Richmond` when `(at barge1 Richmond)` is true and the value `west-coast` when `(at barge1 west-coast)` is true. This grouping of literals into variables is specified through domain axioms added by the user. For an example see Appendix B.

The first stage in the construction of the belief net adds nodes representing state variables that are relevant to the plan based on the actions. This is done by first adding nodes that represent the actions taken in the plan. The assumption is made that the actions are taken sequentially with no pauses in between, although this is not necessary to the belief net construction algorithm. Thus in the example plan, a node is added representing the action `(Move-Barge <barge>=barge1 <from>=Richmond <to>=west-coast)` at time 0, which takes 2 time units, and a node is added representing `(Pump-Oil <barge>=barge1 <sector>=west-coast)` at time 2. Since the planner represents its top-level goals as the preconditions to a final action `finish`, a node is added to represent that action at time 3. The probability of success of the plan is the probability that this final node, has the value `true`. After actions are added, nodes are added to represent their preconditions and effects. For each variable that corresponds to one or more of the literals in the preconditions of the action, a node

$T = 0$, $S = 0$
Stage 1:
For each action A in the plan
    Create a node N_A to represent A at time T.
    For each precondition P of A
        Find or create a node N_P for P at time T.
        Link N_P to N_A.
    $T = T + \text{duration}(A)$
    For each effect E of A
        Find or create a node N_P for P at time T.
        Link N_P to N_A.
Stage 2:
For each node N in the belief net representing a state feature
    If N is not the effect of an action,
        link it to the most recent previous node of the same type.
        Find the set $\mathcal{E}_\mathcal{N}$ of events that can affect N
        For each event ϵ in $\mathcal{E}_\mathcal{N}$
            Add nodes for the event and its preconditions
            in the same way as stage 1.
If new events were added, go back to *Stage 2*.

FIGURE 4.8: The two-stage algorithm to construct a belief net to evaluate a plan. In the first stage, nodes are added to the belief net to represent actions and their preconditions and effects. In the second stage, persistence assumptions are replaced with nodes representing exogenous events and their preconditions and effects when appropriate.

is added representing belief in the variable's value at the time the action is begun, and an arc is added from this node to the action. For each variable that corresponds to one or more literals mentioned in the effect distribution of the action, a node is added representing belief in the variable's value at the time the action completes, and an arc is added from the action to the node. A node representing a variable at some time can be both the effect of some action and the precondition of the next action. In what follows I will refer to a node representing a state variable at a particular time as a *fluent node*.

The first of the two nets shown in Figure 4.9 is the belief net created to evaluate the example plan before any nodes are added for exogenous events. The time stamp of each node is represented by its position along the x-axis. The time value is shown on the x-axis, assuming the first action in the plan begins at time 0. The type of the node is represented by its position along the y-axis. For example, all the nodes in the top row of the diagram represent the location of the oil, which in this plan can be in the tanker or in `barge1`. The action nodes are all in the same row to keep the diagram small, and the particular actions are labelled at the node.

Most of the arcs in this belief net represent the preconditions and effects of actions as described above, but while all such arcs would link a fluent node and an action node, the net also has five arcs that directly link fluent nodes. These links reflect *persistence assumptions* in the belief net. The algorithm initially makes a persistence assumption for every fluent node with a time value greater than 0. After precondition and effect nodes are created, each fluent node is linked to the latest node before it of its type, or if there is no such node a new node of its type is created at time 0 and it is linked to that. The intention is to encode the belief that each state variable at a certain time has the same value as this parent, which is the most recent state variable of the same type, unless some action changes its value. However the conditional probabilities are not added to the net until the possible exogenous events that could challenge these persistence assumptions are examined.

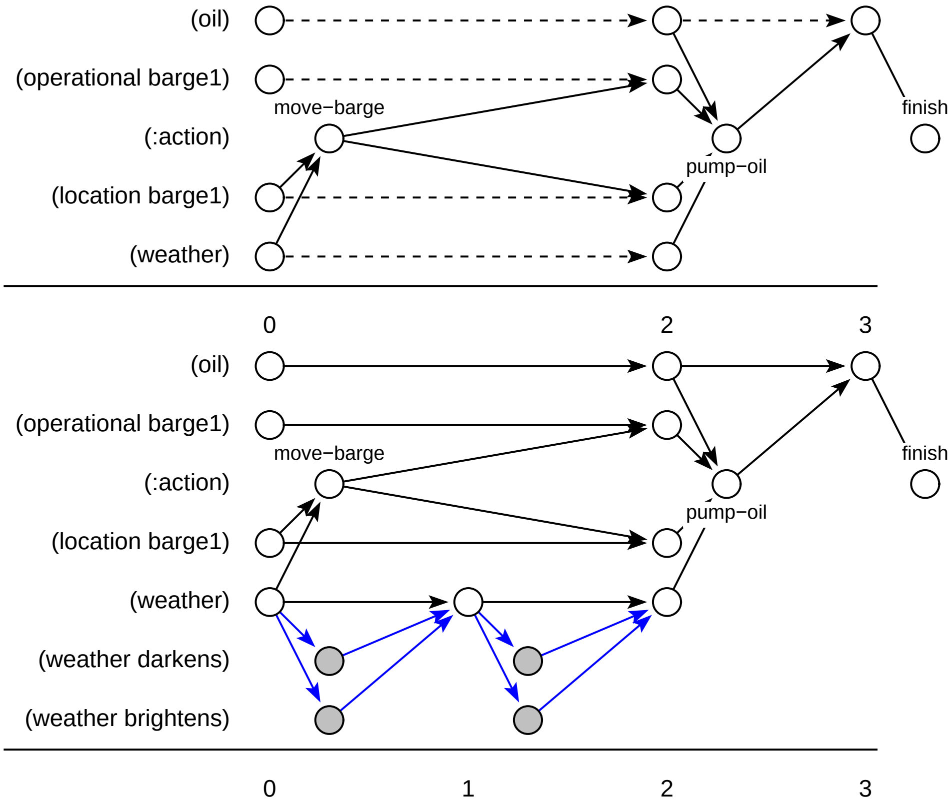

FIGURE 4.9: Belief nets representing the first plan created by the conditional planner for the example problem, before and after event nodes are added to justify the persistence of the `(weather)` variable. In the first graph, the dotted lines show the unexplored persistence intervals. In the second graph, gray nodes represent possible occurrences of external events during one interval. No exogenous events affect the other intervals.

In the second stage of belief net construction nodes are added to represent exogenous events that can affect the plan's probability of success. For each link between two fluent nodes, nodes are added representing the occurrence of any exogenous events that may affect the variable involved. These nodes are added at each time point that the persistence interval spans. The second net in Figure 4.9 shows the example plan after this process. Event nodes, shown in gray, have been added for events of type `Weather-Darkens` and `Weather-Brightens` at time points 0 and 1, affecting the weather before the `Pump-Oil` action.

As with action nodes, fluent nodes are added for the preconditions and effects of event nodes. Persistence intervals are created in turn for these fluent nodes. New event nodes are then recursively sought that can affect these persistence intervals and the process is repeated. This stage terminates because each new interval must be shorter than the one for which the previous event node was introduced and since the plan length was finite the possible intervals over which events can take place will eventually have zero length.

At this point, the conditional probability tables are filled in. Any node with no parents is an "initial state" node, a fluent node with time stamp 0. These nodes have probability distributions determined by the initial state distribution of the problem. Any fluent node with just a fluent node parent is given the identity distribution, since no exogenous events or planned actions affect the value. An example of this is the `(oil)` node at time 2. An action or event node is given a probability distribution that reflects its preconditions. If it is deterministic, so is the distribution: all the probabilities are 1 or 0. If the action has probabilistic outcomes, each possible outcome is labelled and the conditional probabilities in the belief net are assigned from the effect distribution function. In either case the action's value is `false` with probability 1 whenever its preconditions aren't satisfied. Finally, if a fluent has an event or action node parent (or both), then if the action node has a value other than `false`, the fluent node takes the indicated value with probability 1. Otherwise if an event takes place, the fluent takes the value indicated by the event. If neither an action nor an event is successfully executed, the fluent takes the same value as its parent fluent node. Note that Weaver cannot handle two or more parent events occuring simultaneously due to the language restriction that events which affect the same literals must have mutually exclusive preconditions. Figure 4.10 shows the marginal probabilities for the nodes in the belief net for the example plan.

When a conditional plan is evaluated, separate nets are created for each linear set of actions that can be executed in the plan. In principle it is quite possible to represent a branching plan with a single belief net, but in practice it was found that the separate nets are more efficient to evaluate. The test node and value are chosen such that the original plan is known to fail if the value is such that the alternative branch of the plan is chosen. In the belief net for the alternative branch, any fluent node that has a time stamp after the test node but whose parents are time-stamped before the test node is made to have the test node as an extra parent. When the test node takes a value such that the branch of the plan is not taken, the fluent node takes on a new value `branch-not-taken`, which is guaranteed to make any subsequent action fail.

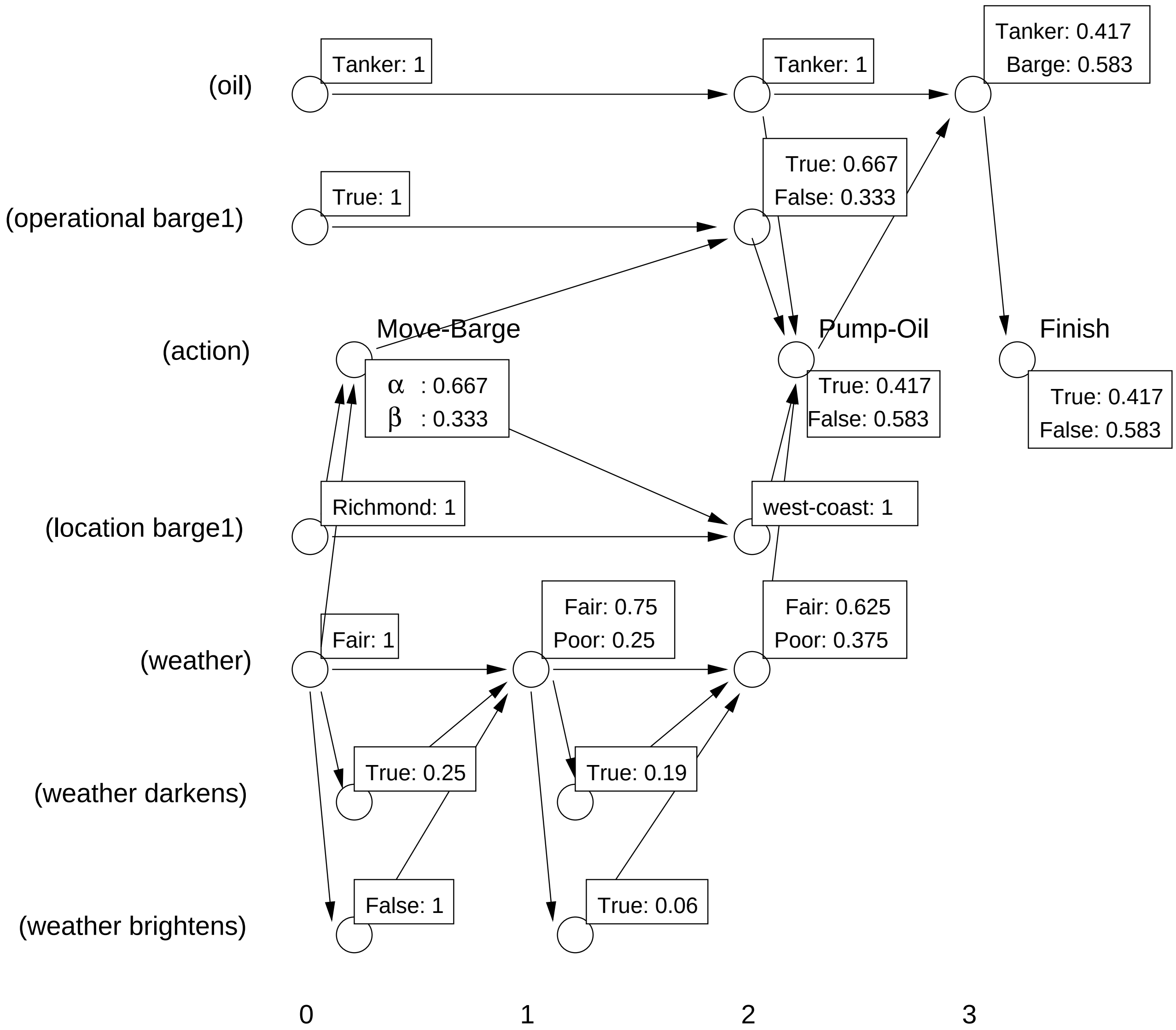

FIGURE 4.10: The belief net for the first conditional branch of this section's example plan, with marginal probabilities shown.

This is done so that the two nets succeed under mutually exclusive conditions and the probabilities of the two `finish` step nodes can be added to get the probability of success of the plan. The nets that are computed for the example plan are shown in Figure 4.11.

This algorithm produces a belief net D which is concise in the sense that only state variables and events known to be relevant to the plan are represented. The benefits of this concise representation are discussed in Section 4.4. The algorithm by which the belief net is constructed will ignore exogenous events that can make variables have desired values, so the probability distribution encoded by the belief net is not the same as the true probability distribution of the set of variables as found

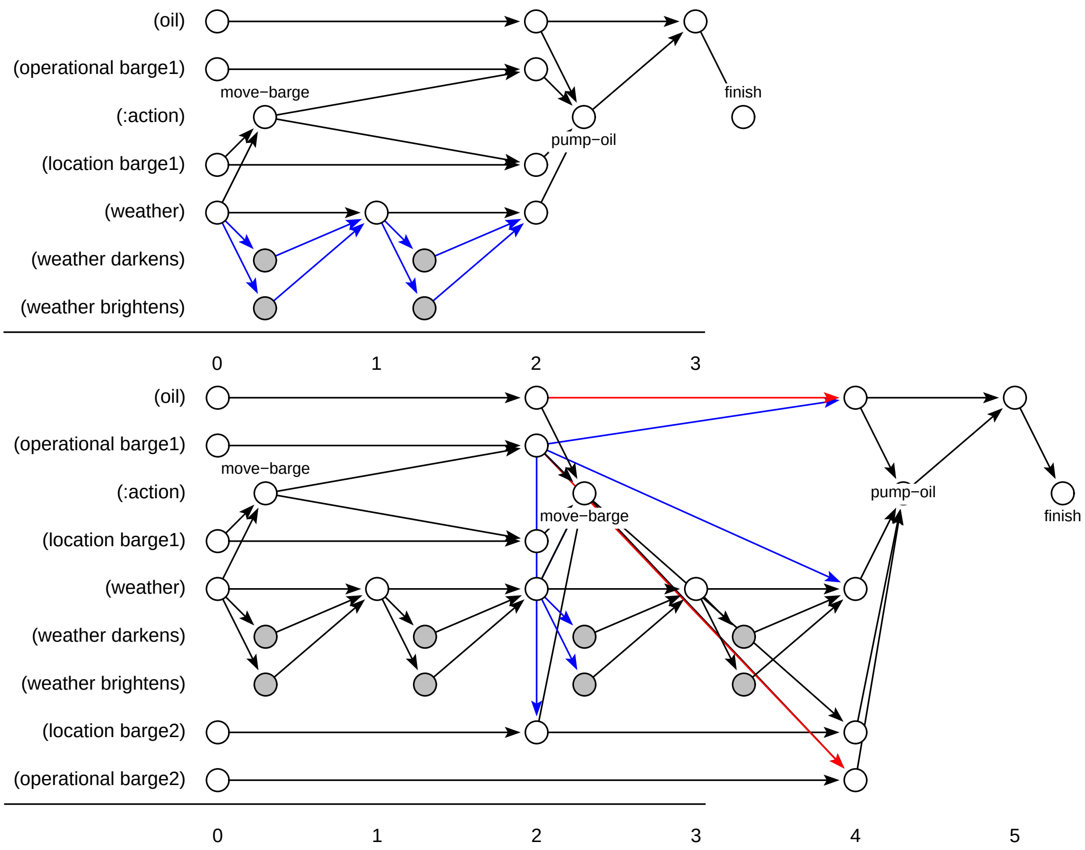

FIGURE 4.11: The two nets constructed for the two conditional branches of this section's example plan.

from the Markov decision process. However, the belief net is guaranteed not to be over-optimistic: the true probability of the plan's success may be greater than the probability predicted by the net but it will not be less. The reason that it may be greater is that some chance events may fortuitously improve the plan's probability of success, and these events are not necessarily represented in the net. A proof that the predicted probability cannot be higher than the true probability is given in detail in Appendix A. Essentially, for each complete assignment to the variables in D for which plan success is true, a set of paths through the underlying Markov decision process is found whose combined probability mass is at least equal to the marginal probability of the assignment in D.

4.3 Incrementally improving a plan with the plan critic

Weaver iterates between creating a plan, evaluating it, finding points where the plan can be improved and creating an improved plan, and so on, as shown in Figure 4.1.

The plan critic, discussed in this section, makes use of the information about the current plan's probability of success, represented in the belief net, to identify points where the current plan can be improved. It then calls the conditional planner to search for a new plan that might address one of these points.

The belief net contains not only the probability of the plan succeeding, but also the probabilities of each possible outcome of each intermediate step in the plan. The plan critic examines each step whose probability of having a desired outcome is less than one. For each one, possible explanations for the undesired outcomes are extracted from the belief net. These may include external events that affect the preconditions of the chosen step as well as previous steps having several outcomes, one or more of which causes the plan to fail. The plan critic chooses one such flaw to work on, picking arbitrarily by default. Once a flaw is chosen, the plan critic may select one of two ways to attempt to remove the flaw. It can either cause the conditional planner to add a new branch which will specifically plan for the goals from a situation corresponding to the flaw, or it can cause the planner to attempt to make the event or undesired step outcome impossible. These actions are described in more detail in this section. Weaver can backtrack over all of these choice points.

4.3.1 Analysing the belief net and choosing a flaw

On each iteration, Weaver examines the belief net for its candidate plan and uses it to find a way to improve the plan. Figure 4.10 shows the belief net that is created to analyse the first plan produced to solve the example problem. This plan has just two steps, to move barge1 to the spill site, and pump oil into it from the tanker. The node representing the first action has three possible values, corresponding to the two possible outcomes α and β as shown in Figure 4.4 and finally the possibility that the action is not successfully executed. These values have the probabilities 2/3, 1/3 and 0 respectively. The second action has two possible values, representing whether the action is successfully or unsuccessfully applied, and their probabilities are respectively 5/12 and 7/12. Since only outcome α for the first action can lead to the plan succeeding, both actions are investigated by the plan critic for possible flaws.

For each action, each fluent node in the belief net that has an arc to the action node is investigated as a possible flaw. The set of *desired values* for the fluent node is computed as the subset of its possible values for which the plan will lead to a goal state, assuming that the other nodes in the belief net take on appropriate values. This set is computed as the union across each conditional branch of the plan of the intersection within each conditional branch of the values for the variable required by each child action node, computed from the action specification. For example, the set of desired values for the fluent node corresponding to weather at time 2 is {fair} since it forms part of the precondition to the pump-oil action. Similarly the desired value for the node corresponding to (operational barge1) is true. Once the set of desired values is computed for a possible flaw, their combined probability mass is computed as the sum of their individual probabilities. If this is less than one, the fluent node is labelled a flaw. Table 4.3 shows the three flaws found for the initial plan in the example problem, as well as the set of desired values for each flaw node and the probability that the node has a desired value.

For each flaw node, the plan critic computes a set of explanations. Each explanation specifies either an external event or an action that is a parent of the flaw node in the belief net, and can take on a value such that the flaw node may have a value outside its set of desired values. All such explanations are produced for each flaw node. For example, the one explanation for the flaw node corresponding to weather at time 2 is the event node `Weather-Darkens` at time 1. Each explanation comprises a flaw node, an event of action outcome that can lead to a set of undesired values, the resulting set of undesired values and their probability mass.

Flaw node	Time	Desired values	P	Explanations
(operational barge1)	2	{true}	0.667	`Move-Barge` = β
(weather)	2	{fair}	0.625	`Weather-Darkens` = true
(oil)	3	{barge1}	0.417	none

TABLE 4.3: The flaws uncovered by the plan critic for the first candidate plan for the example problem. For each flaw a set of desired values is computed from the preconditions of the child action node, and the probability mass of this set is show. The explanations column shows parent node values that can lead to an undesired value.

Once each explanation is produced for each flaw node in the plan, the plan critic selects one explanation and attempts to find an improvement to the current plan that reduces the probability attached to it. If no such improvement can be found, the plan critic backtracks and selects another flaw explanation. For each explanation, the plan critic may choose to add a conditional branch or add preventive steps, as described below.

4.3.2 Adding a conditional branch

If the plan critic chooses to address a flaw explanation by creating a new conditional branch, it achieves this by adding a new outcome to some step that is already present in the candidate plan and making a call to the conditional planner. Since the conditional planner searches for a plan that covers every action outcome that it is aware of, it will add the new branch as it creates the plan. Although the plan critic reasons about flaws with explanations that can consist of action outcomes or exogenous events, it will represent both of these to the conditional planner as a new action outcome. If the explanation names an action outcome, the same action will be chosen by the plan critic for this process. The outcome in the explanation was previously not shown to the conditional planner as a simplification, or otherwise it could not have been a flaw explanation since the candidate plan would have accounted for this value. If the explanation names an exogenous event, the affected step is the earliest that completes after or at the same time as the event. In this case each outcome of the action is made into two outcomes, one representing the case when the event takes place and the other representing the case when it does not.

For example, if the plan critic works on the first flaw in Table 4.3, (operational barge1), there is only one possible explanation, that action `Move-Barge` has outcome β. Both possible outcomes of this action are now shown to the conditional planner, which will produce the branching plan described in Section 4.1. If, on the other hand, the plan critic works on the second flaw, (weather), the only possible explanation is that event `Weather-Darkens` has value `true`. In this case the plan critic also alters the `Move-Barge` action since it completes at the same time as the event. The one outcome that is represented to the conditional planner is now replaced by two, one the same as the previous outcome and one in which the weather changes from fair to poor in addition to the movement of the barge. The conditional planner will fail to solve this problem, however, since there is no solution in the new alternative outcome of `Move-Barge`, so the plan critic will backtrack, try to prevent the flaw as described in the next section and finally try to address another flaw.

Once the outcome to be added to an action in the plan is chosen, the plan critic forces the conditional planner to backtrack to the point where the action is added to the tail plan, and repeat its earlier planning episode with the new action. A context node is added to the search space, as described in Section 4.1.1, that represents the decision to add the extra outcomes to the step. The plan produced by the conditional planner has one or more extra branches at the point where the branching action is added to the head plan. If the extra branch is unsatisfiable, the conditional planner may choose to alter the order in which actions are applied or may choose to add different actions to the tail plan to achieve goals that are outstanding when the altered action was added to the tail plan, but it may not backtrack to any point in its search before the new action was added. This "ceiling" in its search space ensures that the branching step is included in the plan.

4.3.3 Adding preventive steps

In some cases a flaw explanation can be addressed without adding a conditional branch, by adding steps to the plan to reduce the probability of the flaw explanation producing undesired effects. The plan critic causes the planner to add these steps by choosing an action from the current plan and augmenting its preconditions. If the flaw explanation is an action outcome, the plan critic selects a probability distribution for the action that can lead to an undesired value, and adds the negation of the distribution's path preconditions to the preconditions of the operator. It then causes the conditional planner to backtrack to the point where the action was added to the tail plan, and repeat its earlier planner episode with the new action, just as when a new conditional branch is added. The extra preconditions for the action, if satisfiable, will ensure that the effect probability distribution chosen by the critic cannot take place, and this in turn will reduce the probability of the plan flaw, assuming that the plan critic has made a good choice of effect distribution. The plan critic can backtrack and try a new effect distribution for the chosen action and can also add the negations of multiple effect distributions.

If the chosen flaw explanation is an external event, the plan critic selects an effect probability distribution for the event that can lead to an undesired value in the chosen plan flaw. The negation of the path preconditions of the effect distribution are added to the preconditions of the action in the plan during which the event begins. The plan critic then causes the conditional planner to backtrack and find a new plan in exactly the same way as for a flaw explanation that is an action outcome.

The planning domain that has been used as an example for this chapter does not provide any opportunities to add preventive steps. Consider the more complicated version of the `Move-Barge` operator shown in Figure 4.12, however, in which the undesirable outcome that deletes (operational barge1) can only take place if (barge-ready barge1) is false. Suppose that (barge-ready barge1) is false in the initial state. Then after producing the same initial plan and the same flaws, the plan critic could address the flaw that (operational barge1) is false, with the explanation that `Move-Barge` has outcome β by adding preconditions to `Move-Barge` to prevent this outcome from happening, in this case adding `(barge-ready barge1)` to its preconditions.

```
(Operator  Move-Barge
  (preconds
    ((<barge> Barge)
     (<from>  Place)
     (<to>    (and Place ( diff <from> <to>))))
    (at <barge> <from>))
  (effects ()
    (branch (barge-ready <barge>)
      ((add (at <barge> <dest>))
       (del (at <barge> <source>)))
      ((0.667 ()
         ((add (at <barge> <dest>))
          (del (at <barge> <source>))))
       (0.333 ()
         ((add (at <barge> <dest>))
          (del (at <barge> <source>))
          (del (operational <barge>))))))))
```

FIGURE 4.12: A version of the `Move-Barge` operator that depends on the literal (barge-ready <barge>).

4.4 Discussion

In this chapter I have presented the basic algorithms used in Weaver to produce a plan that meets a given threshold probability of success. At a high level, Weaver alternates between stages of plan creation and plan evaluation. A plan critic repeatedly calls a conditional planner with an increasingly faithful representation of the domain although it is initially ignorant of all forms of uncertainty. The plan critic therefore reasons about what sources of uncertainty to pay attention to during plan creation, guided by knowledge of which ones have a significant impact on the current plan's probability of success.

This separation of responsibilities allows Weaver to create plans in domains with exogenous events that essentially add many alternative outcomes to every action in the domain. Without the ability to ignore these alternatives until they are shown to be important the planning task would be intractable. The constraining effect between the modules is important in both directions, both from the critic on the conditional planner and from the planner on the plan evaluator. I illustrate this with two small synthetic domains, in each of which the initial state is empty and the goal is the literal `(goal)`.

In the first domain shown in Figure 4.13, the literal `(sub-0)` is true in the initial state and a sequence of at least $n+1$ steps is required to solve the goal: `Op-1`, `Op-2` ... `Op-`n, `Final`. Each step of the form `Op-`i in the plan has a 0.5 probability of adding the literal `(bad-luck)`, so the success probability of this plan is $1/2^n$. Either of two improvements to the plan can raise the probability to 1. A conditional branch could be added after the step `Op-`n, testing the literal `(bad-luck)` and using the final step `Alt-Final` instead of `Final` if it is true. Alternatively the undesired outcomes of the nondterministic steps could be prevented by adding the step `Protect` to the start of the plan.

When Weaver solves this problem, the conditional planner initially ignores the nondeterminism of the `Op-`i operators and produces the first plan as shown. The critic will then choose one of these operators and choose whether to improve the plan with a conditional branch or a protection step. It is possible for the protection step or conditional branch to be added in the middle of the sequence of `Op-`i operators, in which case several iterations of improvement may be required. By default, however, it will try to put a protection step as early as possible and a conditional branch as late as possible.

If the planner was aware of every source of nondeterminism in its first call, it would attempt to create a branching plan to account for each of the combinations of outcomes for the nondeterministic operators, producing a tree with 2^n leaves. Other conditional planners can represent branching plans whose branches merge later, and would be able to find a conditional plan accounting for all the outcomes in time linear in n. However in the case of causal-link planners such as Buridan or Cassandra, the plan would require n improvements to be made, as each of the alternative outcomes must be linked to the alternative step in the case where a branch is used, or have a link from the protecting step in the other case.

```
(Operator Final
  (preconds ()
    (and (sub-n)
         (not (bad-luck))))
  (effects () ((add (goal)))))

(Operator Alt-Final
  (preconds ()
    (and (sub-n)
         (bad-luck)))
  (effects () ((add (goal)))))
```

```
(Operator Op-i
  (preconds () (sub-i − 1))
  (effects ()
    (branch (protected)
      ((add (sub-i)))
      ((0.5 () ((add (sub-i))))
       (0.5 () ((add (sub-i))
                (add (bad-luck))))))))

(Operator Protect
  (preconds () (true))
  (effects () ((add (protected)))))
```

FIGURE 4.13: Operators in the synthetic domain to illustrate the utility of the constraints on the planner from the critic. The initial plan found by the conditional planner will have a success probability of $1/2^n$. Either one conditional branch or one protection step can increase the probability to 1.

The essential details of this example, a sequence of operators that can with some probability affect a literal that is required after the end of the sequence, would arise frequently if exogenous events were handled by "compiling" them into the effects of actions. This is a simple way to represent exogenous events by representing all the potential effects of the events explicitly as possible outcomes of each operator during which they can take place. For example, if the operators of the form `Op-`i in Figure 4.13 are replaced with deterministic operators and the exogenous event `It-Happens` as shown in Figure 4.14, and the `Op-`i operators all have a duration of 1 unit, the compiled operators are the same as before. In this case Weaver can make an extra savings in time using techniques for compressing the belief net described in the next chapter.

The second synthetic domain, shown in Figure 4.15 requires only two steps for an initial plan to achieve `(goal)`, but there are n independent exogenous events, each of which can take place while the first step is taken. Without an abstraction barrier, this would be represented with 2^n different outcomes for the action. However only $m < n$ of these can affect the plan, by making the literal `(bad-luck)` true. In this case Weaver's abstraction barrier hides the $n - m$ events from the planner and

```
(Operator Op-i
  (preconds () (sub-i − 1))
  (effects ()
    ((add (sub-i)))))
```

```
(Event It-Happens
  (preconds () (not (protected)))
  (effects () ((add (bad-luck))))
  (probability 0.5))
```

FIGURE 4.14: The nondeterminism from the operators in the previous domain is replaced with a single exogenous event. The operators `Final`, `Alt-Final` and `Protect` are unchanged and so are the solutions.

leads to significantly more efficient planning. Weaver does this by using the belief net construction process to test which events are relevant, so this domain provides an example of how the plan can constrain the size of the probabilistic model built for the domain.

```
(Operator  Final
  (preconds ()
    (and (sub)
         (not (bad-luck))))
  (effects () ((add (goal)))))

(Operator  Op
  (preconds () (true))
  (effects ()
    ((add (sub)))))

(Event  Event-i
  (preconds () (true))
  (effects () ((add (e-i))
                (add (bad-luck)))) ;;  If i ≤ m
  (probability 0.5))

(Operator  Alt-Final
  (preconds ()
    (and (sub)
         (bad-luck)))
  (effects () ((add (goal)))))
```

FIGURE 4.15: Operators in the synthetic domain to illustrate the utility of the constraints on the evaluator and critic from the planner.

The conditional planner builds the initial plan `Op`, `Final`. When this is evaluated, the m events `Event-`i, where $1 \leq i \leq m$ are identified as potentially affecting the plan and the belief net includes them. One is chosen and the critic adds an alternative outcome to the plan step `Op`, in which the literal `(bad-luck)` is added (as well as `(e-`j`)` for some j). The planner then produces the same conditional plan as in the last domain, which solves the problem. Only one alternative outcome is examined by the planner.

In contrast, without the plan critic to restrict the outcomes considered by the conditional planner, it would have to examine each of 2^n possible outcomes for the first step in the plan. This is true regardless of the planning method used unless it restricts the outcomes considered in some way. By ignoring the rest of the external events, Weaver forms an abstraction over the outcomes. The two outcomes it considers are aggregates of many indivisible outcomes, that specify the outcome for each exogenous event. Some conditional planners, such as C-Buridan, terminate before planning for each outcome even though they consider each indivisible outcome explicitly. Such planners will not have to consider all 2^n possible outcomes to `Op`, but in order to find a high-probability plan these planners will still have to consider a significant proportion of the indivisible outcomes where the initial plan fails. There are $2^{n-m} \times (2^m - 1)$ of these, each with probability $1/2^n$.

Similarly to the previous domain, although this is an artificial domain the same situation can occur in real domains. It occurs when there is a large number of possible exogenous events but only a small number affect the outcome of the candidate plan. In the oil-spill domain discussed in Chapter 7, there are many exogenous events governing the weather change in different sectors of the domain, and other events could be added governing, for example, equipment failures. These events must be pruned dynamically, using the plan as it is created, because it is not possible to tell which exogenous events are relevant to the plan before a partial plan is developed.

In Chapter 7 I develop measures of the benefit of the mutual constraint between the planner, plan evaluator and critic that consider both the number of steps during which events can take place and the number of exogenous events that are not relevant to the current plan.

In spite of these benefits, the strategy of complete separation of plan evaluation and plan creation does not produce the best behaviour in most planning domains. In particular, faster convergence to a high-probability plan can be obtained if some estimations of the probability of plan completions are used to choose between different candidate actions in a partial plan. In Section 6.1 I present some domain-independent search heuristics that improve the planner by doing local probability estimations.

The basic Weaver algorithm also makes some approximations in the intermediate states in branching plans. This is because it represents the branch to the conditional planner by merging a single new outcome with some existing step (Section 4.3.2). This can produce an impossible state, however, if the action outcome or event being added is only realisable as the end of a chain of events or alternative action outcomes. In theory this will not cause the algorithm to make incorrect probabilistic evaluations since the new plan will be evaluated in a context where all the relevant sources of uncertainty are considered. Nor will it cause a solution to be missed, since if the inconsistency affects the plan's probability of success it will show up as a direct flaw. Still, it can affect the rate of convergence to a plan that passes the threshold probability, which in cases with hard computational resource constraints this can reduce the best probability of success found.

Weaver has three basic mechanisms for improving a plan's probability of success: standard backtracking, adding preventive steps and adding a conditional branch. The question arises whether these are either necessary or adequate for finding plans that pass a probability threshold. Backtracking is obviously needed, and in fact simple examples can show that both conditional branches and preventive steps, or something like them, are needed to be able to find good plans in every domain.

The main example used in this chapter solves a problem in which oil is to be removed from a tanker with a conditional plan, moving `barge1` and using it if it is operational, otherwise using `barge2`. This example does not support the claim that conditional branches are necessary, though, since a possible plan is to move both barges and always attempt to pump oil into each, one after the other. At least one of the `Pump-Oil` steps will always fail, either because a barge is not operational or because the oil has already been pumped. However, the plan has the same probability of success as the original branching plan.

As an example of a problem where a branching plan is needed, suppose that a pump must be deployed in a barge in order to pump oil into it, and that the pump can only be deployed once. If the problem is otherwise identical to the example problem, then a branching plan is required to achieve a probability of success above 0.5, since the second barge could not be used after the pump had been deployed in the first barge. The modified operators for this example are shown in Figure 4.16. Although this simple example is contrived, it follows a pattern that often occurs naturally: a conditional plan is necessary because limited resources must be used sparingly.

```
(operator  Pump-Oil
  (preconds ((<barge> Barge)
             (<sector> Sea-Sector))
   (and (at <barge> <sector>)
        (operational <barge>)
        (fair-weather)
        (pump-deployed <barge>)
        (oil-in-tanker <sector>)))
  (effects ()
   ((add (oil-in-barge <barge>))
    (del (oil-in-tanker <sector>)))))
```

```
(operator  Deploy-Pump
  (preconds ((<barge> Barge))
    (~ (exists ((<b2> Barge))
       (pump-deployed <b2>))))
  (effects ()
    ((add (pump-deployed <barge>)))))
```

FIGURE 4.16: A modified `Pump-Oil` operator and a new `Deploy-Pump` operator that make a conditional plan necessary to achieve a probability of success greater than 0.5 in the example problem.

An example of a planning problem where prevention steps are necessary can be found by replacing the `Move-Barge` operator with one that has different effect probabilities depending on the literal schema (barge-ready <barge>), as in Figure 3.10, and removing (barge-ready barge1) from the initial state. Now the step (Make-Ready <barge>=barge1) must be added to the plan to achieve the required probability, and this can be found by addressing the flaw that the barge is not ready using prevention.

An informal argument can be made that these mechanisms, adding a conditional branch, adding preventive steps and backtracking, are together adequate for finding plans that can be represented as policies on the underlying MDP that do not have loops. A version of Prodigy that can add preventive steps to avoid undesired conditional effects in plans can be shown to be complete when there are no lisp functions used in operator bindings or control rules [Blythe & Fink 1997][1]. Thus, supposing that a planning problem admits a policy π on the underlying MDP which has no loops and reaches the threshold probability of success, a version of Prodigy could find a plan that corresponds to at least one path from an initial state to a goal node in π. If this plan does not meet the threshold probability of success, there must be other paths in π that are not present in the plan. The points where following the plan is less desirable than the action chosen by the policy π will be found as flaws in the plan and a preferable path can be added with a branching plan. While the mechanisms developed here can lead in principle to a complete planner for plans without loops, I do not argue that Weaver is complete. In real-world planning domains, bindings functions and control rules are frequently used to control the search, and the ability to call arbitrary functions makes the planning problem undecidable.

[1]This version of Prodigy also plans for subgoals that are true in the current state, which is not done in the version used in Weaver

Chapter 5

Efficiency improvements in plan evaluation

In the previous chapter I introduced the methods used for probabilistic evaluation of Weaver's candidate plans. These plans are produced by a conditional planner that is ignorant of many of the sources of uncertainty in its planning domain. The task of the plan evaluator is then to identify the relevant sources of uncertainty and assess their impact on the plan. Its output is a belief net structure that encodes dependencies among steps in the plan and external events and which when evaluated computes lower boundaries on the probabilities of success for each step in the plan.

Although there are methods for evaluating belief nets that can make efficient use of the structure that they show for the probability distribution, evaluating belief nets is NP-hard in the general case [Cooper 1990]. Evaluating the belief net can quickly become the bottleneck in the process described in Chapter 4. In this chapter I describe some techniques that can reduce the computation required to evaluate the plan. I focus on representing the plan with an accurate but smaller belief network, rather than seeking algorithms to evaluate belief nets more efficiently, an area which has already received considerable attention [Lauritzen & Spiegelhalter 1988; Pearl 1988; Draper & Hanks 1994; Darwiche 1995].

In particular, I focus on removing the nodes in the belief net that correspond to individual occurrences of external events, replacing them with links that reflect aggregates of several possible events. Consider the belief net shown in Figure 5.1, which captures the probability of success of the plan developed in Chapter 4 to illustrate Weaver. The shaded nodes represent individual occurrences of the external events `Weather-Brightens` and `Weather-Darkens`, accounting for possible changes in the weather between time 0 and time 4. These nodes represent information at a finer level of detail than is required to evaluate the plan, since the weather conditions at times 1 and 3 are not important. Only the weather conditions at times 2 and 4 effect the outcome of any actions in the plan.

The belief net shown in Figure 5.2 does not have the redundant nodes, and instead computes the weather change over the intervals of interest using a Markov chain. Although in this example avoiding the nodes corresponding to external events does not significantly reduce the size of the belief net, notice that the number of these nodes increases linearly with the length of the time interval over which the weather can change. The time to compute the probability of success also increases linearly when event nodes are included, but only logarithmically using a Markov chain representation as shown below.

The rest of this chapter describes how a suitable Markov chain can be constructed, discusses the benefits and pitfalls of the approach and provides further examples.

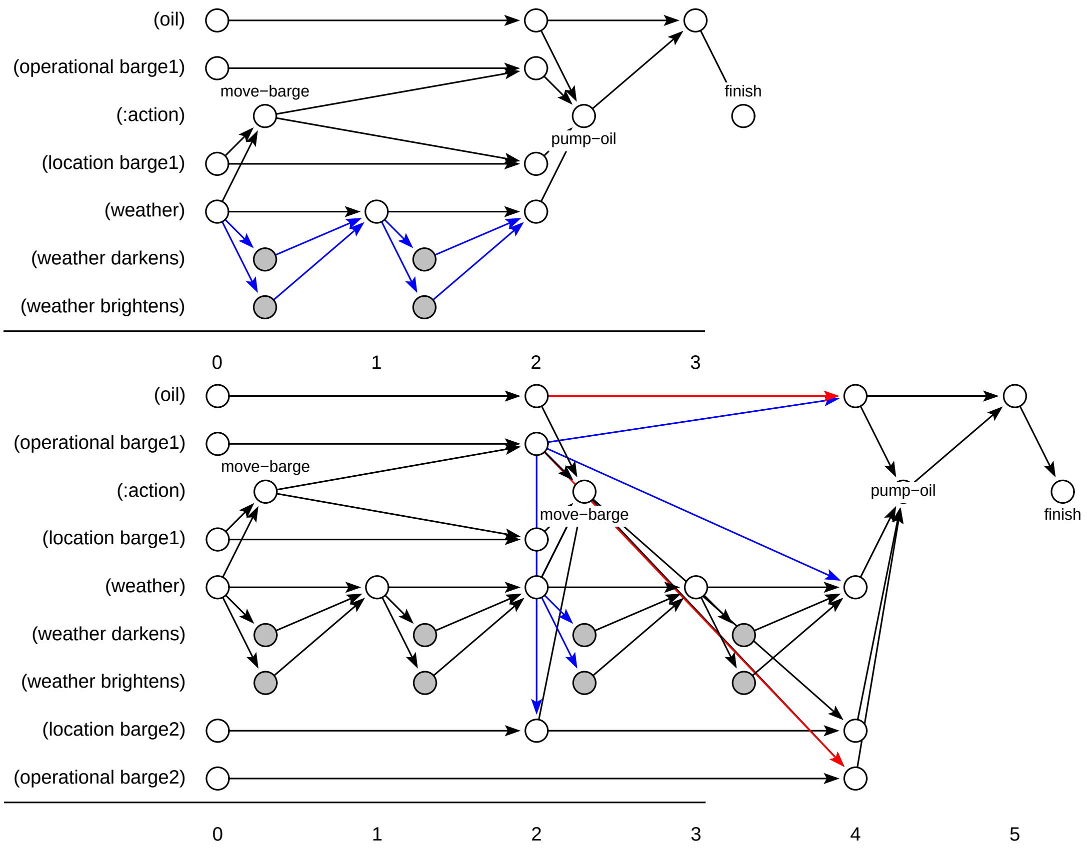

FIGURE 5.1: Belief nets from Chapter 4, in which shaded nodes represent specific occurrences of external events.

5.1 The Markov chain representation for external events

Consider the evolution of the random variable (weather) over time in the graph of Figure 5.1. The conditional probability of the two events affecting the weather is the same at each time step, so the different probability distributions for the variable correspond to four steps of the Markov chain shown to the right in Figure 5.3. The

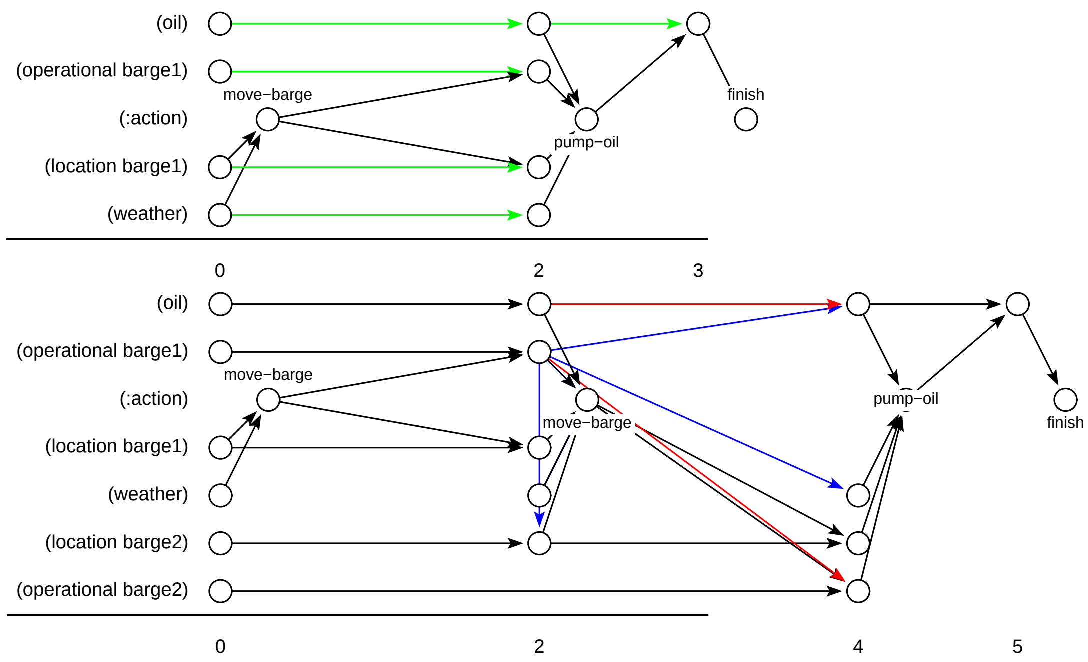

FIGURE 5.2: A belief net representing the same plan but with Markov chains used to compute changes in the weather over the required time intervals rather than a series of event nodes.

Markov chain is described by a transition matrix M whose value in row i and column j is the conditional probability $P(w(1) = j|w(0) = i)$, where $w(n)$ is the value of the weather at time n. This representation can provide a more efficient way to compute the probability distribution for the weather at time N given the distribution at time 0, where N is large, since this is given by applying the transition matrix M^N to the initial distribution, and this transition matrix can be found in time logarithmic in N. The simple technique of adding event nodes directly to the belief net described in Section 4.8 takes time at least linear in N, both for constructing and evaluating the belief net.

In general the evolution of any state variable while an action is being executed can be described by a Markov chain. This is because the planning language itself can be described as a Markov decision process, which reduces to a Markov chain when there are no action choices. However, it would be too expensive to use the underlying MDP as the Markov chain to compute probability distributions of state variables because of its size. Planning problems in the oil spill domain described in Chapter 7 frequently have literal state spaces containing more than 2^{1000} states, for example. So in order to take advantage of the Markov chain representation, a small Markov chain must be found to compute the values of interest.

In the case of the state variable describing the weather, the Markov chain whose states are just the values of this variable is adequate to compute the right probabilities, and the transition matrix M has only 4 entries. In general one must consider the

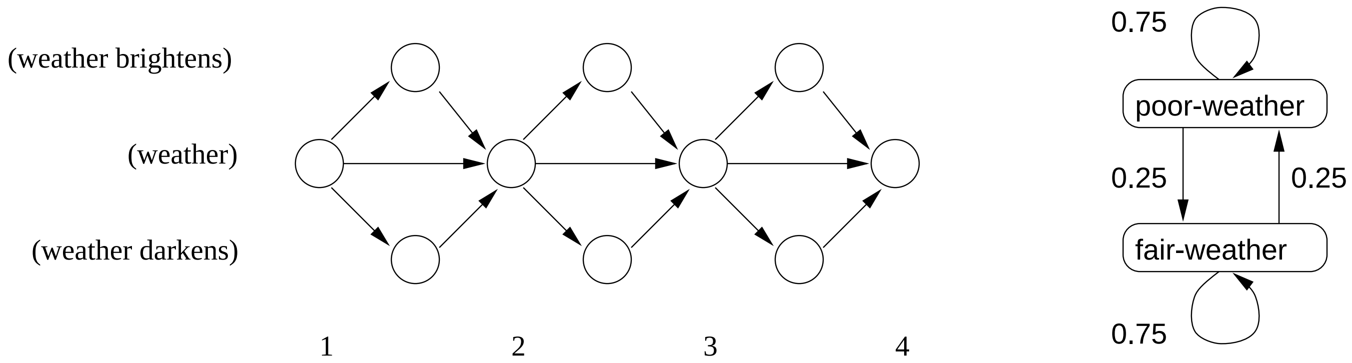

FIGURE 5.3: The nodes governing weather from the plan's belief net follow the Markov chain on the right.

set of state variables whose values need to be known at a particular time in order to see how to build an adequate Markov chain, and this chain may reference state variables other than those of initial interest. To see this, consider adding the event `Oil-Spills` shown in Figure 5.4 to the planning domain and problem described in Chapter 4. This event requires the weather to be poor and the oil to be in the tanker. If this is true, with probability 0.1 the oil will be spilled in the sea in the next state, represented with the new predicate `(oil-in-sea west-coast)`, and will no longer be in the tanker. If this event takes place, it will prevent the `Pump-Oil` action in either branch of the plan from working since the action's preconditions include `(oil-in-tanker west-coast)`.

```
(event oil-spills
  (params <sea-sector>)
  (probability 0.1)
  (preconds
    ((<sea-sector> Sea-sector))
    (and (oil-in-tanker <sea-sector>)
         (poor-weather)))
  (effects ()
    ((del (oil-in-tanker <sea-sector>))
     (add (oil-in-sea <sea-sector>)))))
```

FIGURE 5.4: The event `Oil-Spills` affects the location of the oil and is partly determined by the weather.

Since the event depends on the weather, changes in the weather must be taken into account to compute the probability distribution for the state variable `(oil)`. Otherwise the probability that the oil is spilled into the sea might be wrongly computed as 0, since in the initial state `(fair-weather)` is true with probability 1. In fact, since the `Pump-Oil` action depends on the conjunction `(fair-weather)` $\wedge$ `(oil-in-tanker west-coast)`, the two state variables must be computed together in a single Markov chain. If two separate chains were used, even if the one used to compute the probability distribution for `(oil)` correctly modelled the weather, the probability of the conjunction would be wrong since it would effectively model the `(oil)` and `(weather)` state variables as independently evolving. The next section shows how to take a conjunction of literals whose probability distribution is needed at time N and automatically build a set of small but adequate Markov chains to compute this.

5.2 Using an event graph to guarantee a correct Markov chain

Given a set of literals of interest, we would like to build a submodel that is small enough to answer queries about these literals efficiently, but includes all the literals necessary to ensure the result is correct. Event graphs enable us to do this by capturing the dependencies between the events in the domain.

The *event graph* for a set of event schemas E is a directed graph whose nodes are either event schemas or literal schemas in the domain. Event and literal schemas are distinguished from events and literals in that they have variables rather than domain objects as argument. There is an arc from each event schema $e \in E$ to every literal schema in its effects, and from every literal schema in any of the branches of e to e (so the graph may have cycles) Variables are unified when two different schemas mention the same literal with matching variable types. Figure 5.5 shows the event graph for the example domain. A simple algorithm that adds arcs for each event schema in turn can produce the event graph in time linear in the number of event schemas, the maximum number of branches in an event schema and the maximum number of effects in an effect distribution.

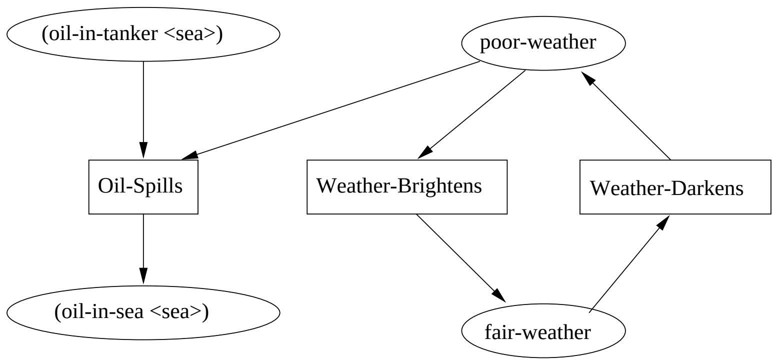

FIGURE 5.5: The event graph for the example domain, with boxes for event schemas and ovals for literal schemas.

Given a query, such as $\mathsf{oil} = \mathsf{tanker} \wedge \mathsf{weather} = \mathsf{fair}$, that contains a set of literals, V, a set of Markov chains is created in two steps. First, create the subgraph of the

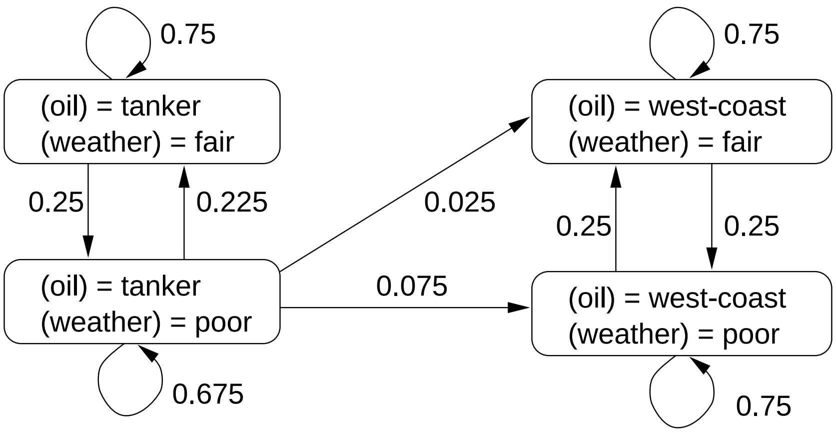

FIGURE 5.6: The Markov model built to model the evolution of the state variable `(oil)`, instantiated by the literal `(oil-in-tanker west-coast)`.

event graph, restricted to the nodes in V and all their ancestors in the graph. Second, create one Markov chain for each component of the resulting graph. The state space for each chain is the cross product of the literals that appear in the corresponding component, and the transitions are formed from the events in the component in the same way as the full-size Markov chain is created from all the events, described above. The probability of the query over literals in V after n stages can then be computed by multiplying together the probability from each chain of the projection of V on that chain after n stages. I refer to this set of Markov chains as the *submodel of the full model induced by V*. For example, the reduced model in Figure 5.6 is the submodel of the full model for the domain induced by {`(oil)`}.

Intuitively, no literal that is not contained in the subgraph of the event graph created for V can affect the way any literal in V changes over time, since if it could affect any event that could affect any of the literals in V, either directly or indirectly, it would be contained in the subgraph by its definition. Similarly the literals in different components of the subgraph change independently of each other, since the value of one can never affect the way the value of the other changes. This intuition underlies the proof of the following theorem. A sketch of the proof is given in Appendix A.

Theorem: *Let Q be a logical expression mentioning a set of literals $V \subset L$. Computing the conditional probability of Q after some fixed number of stages n given an initial state probability distribution over the full state space Ω will yield the same value in the submodel induced by V as in the full model of the domain.*

Using this theorem we can compute the answers to queries about small subsets of the domain variables without consulting the full model of the domain. In fact, the full model is typically never constructed.

5.3 Using the event graph in Weaver

The event graph is used to create smaller belief nets representing plans in Weaver. The plans themselves provide the queries that are used with the event graph to choose the Markov chains that provide input to the belief net. The lack of specific event nodes in the resulting net means that the flaw set found by the plan critic is slightly different. This section describes the changes to the Weaver plan evaluator and plan critic that are necessary to use belief nets built with the event graph.

First, the algorithm to create the belief net to evaluate a plan is altered to make use of the Markov chains that are created from the event graph. Stage 2 of the algorithm from Table 4.8 described in Section 4.2 is replaced by stages 2 and 3 as shown in Table 5.1. A new node is created for each group of nodes that share a Markov chain, that represents the conjunction of the group of nodes. The persistence interval links to this new node and it has an arc to each node in the group specifying its value deterministically.

This new conjunction node could be avoided by clustering the nodes in the group directly, but using it preserves the condition that each precondition node for an action represents a single state variable. Since the Markov chain is built under the assumption that no actions that affect the variables take place during the time interval of the chain, the algorithm splits a persistence interval at each point where an action affects any of the variables involved. The effect of the action is a direct intervention into the natural evolution of the variables represented by the Markov chain, resulting in a model similar to that of Pearl in [Pearl 1994].

Figure 5.7 shows the belief net that is created by this algorithm for the first branch of the example plan along with some of the marginal probabilities. The node shown in gray is added by the algorithm to allow the Markov chain to be expressed as a conditional probability table in the belief net.

Once the belief net has been created, Weaver's plan critic analyses it for possible flaws which are used to find ways to improve the plan. The basic algorithm to find flaws, described in Section 4.3, examines each fluent node that is a parent of an action node and if the fluent node has probability less than 1 of having a desired value it looks for either an action or event node as an explanation for a subset of the undesired values. Since the belief nets created using the event graph do not have event nodes, the set of explanations is modified.

Rather than look for an event node parent, the modified algorithm tests if the fluent node has a conditional distribution that has been created using a Markov chain. If this is the case, the chain is analysed for exogenous events that are responsible for the undesired values. For each member of the set of desired values, the transitions in the Markov chain that move from a state with the desired value to a state with an undesired value are checked and the events responsible for the change are returned as possible explanations for the flaw.

For example, when the fluent node for `(weather)` at time 2 is analysed in the belief

$T = 0$, $S = 0$
Stage 1: For each action A in the plan
  Create a node N_A to represent A at time T.
  For each precondition P of A
    Find or create a node N_P for P at time T.
    Link N_P to N_A.
  $T = T + \text{duration}(A)$
  For each effect E of A
    Find or create a node N_P for P at time T.
    Link N_P to N_A.
Stage 2: For each fluent node N in the belief net
  If N is not the effect of an action,
    link it to the most recent previous node of the same type.
Stage 3: For each time point T with fluent nodes in the plan
  Let F be the set of fluent nodes with this time point
  Let S be the subsets of F grouped into the same Markov chains using the event graph
  For each subset G of S
    Let T' be the time of the most latest fluent node before T that matches a type in G
    Construct nodes for all types in G at time T', making a set G'
    If $|G| > 1$, add a new node N_G at time T with an arc to each node in G
      The values of N_G are the cross product of those of G
      the arcs from N_G implement projection.
    Otherwise set N_G to the single node in G.
    Add an arc from each node in G' to N_G, with probability table given by the Markov chain.
If new nodes were added, go to *Stage 3*.

TABLE 5.1: Algorithm to construct the belief net to evaluate a plan.

net in Figure 5.7, the Markov model in Figure 5.6 is checked for events that lead to transitions from states with (weather) = fair to states with (weather) = poor. The event `Weather-Darkens` is then considered as a possible explanation for the flaw.

Since the explanation for the flaw does not mention a specific event node, the plan critic chooses an action from the entire interval of the Markov chain to adjust, either to prevent the event or to add a conditional branch. This choice is equivalent to choosing the different event nodes that span the interval in the belief net created without using Markov chains. In this example, there is only one action to choose to create a conditional branch, the `Move-Barge` action that was also used in the earlier version. By default, the plan critic adds a conditional branch as late as possible, subject to the condition that backtracking in Prodigy's search space should undo as few other context and protection nodes as possible.

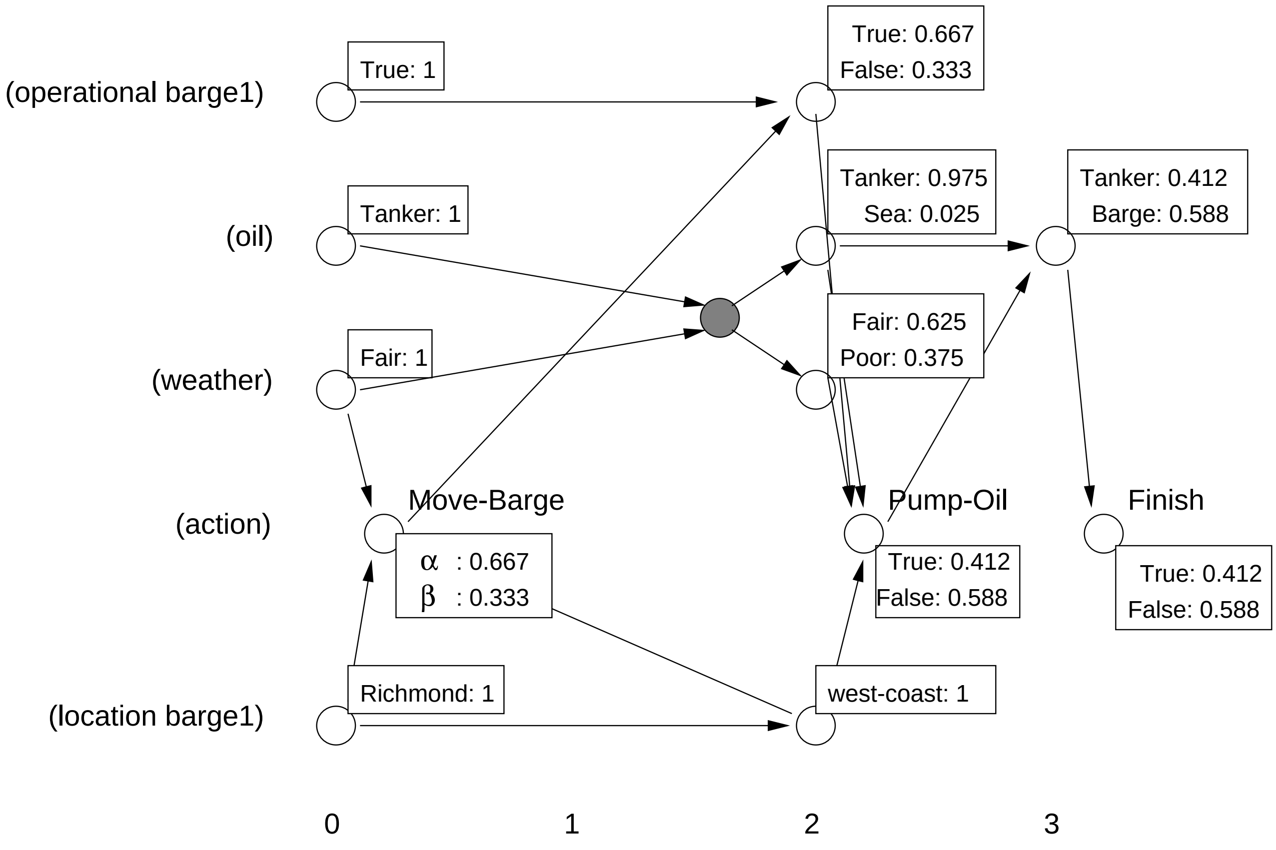

FIGURE 5.7: The belief net created to evaluate the first branch of the example plan with exogenous events affecting both the weather and the oil modelled with a Markov chain.

Chapter 6

Efficiency improvements in planning

6.1 Domain-independent search heuristics

In any search-based planning system, heuristics to control search are extremely important if the planner is to be efficient. Prodigy includes a rich language for specifying search control knowledge in the form of explicit, domain-dependent control rules [Minton *et al.* 1989; Veloso *et al.* 1995]. Domain-independent search heuristics are also vital in generic planning systems where complete domain-specific control knowledge is not always available [Stone, Veloso, & Blythe 1994].

In this chapter I introduce some principles for defining domain-independent search heuristics that are specific to the problem of planning under uncertainty. The "footprint" principle leads to a family of heuristics for probabilistic planners produced by attempting to make subsequent refinements to a plan apply to a set of possible execution traces that is disjoint from the set for which the plan being refined is already successful. Since Weaver's architecture separates the phases of plan creation and plan evaluation, other heuristics can be derived from integrating the two more closely by doing small amounts of probabilistic reasoning during plan search. The rest of this chapter describes these approaches and illustrates them with synthetic domains that help delineate the conditions under which they work well. Experiments in a real-world domain are described in Chapter 7.

6.1.1 Local estimations of probabilities

One source of heuristics for probabilistic planners is to perform local estimations of probabilities, used in greedy maximum-gradient heuristics. For example one can choose the operator for a goal that achieves it with the highest conditional probability, given that its trigger is satisfied. This heuristic is an extremely cheap estimate of the probability that the operator will satisfy its goal as part of the plan, requiring just inspection of the operator definition. A more expensive heuristic that uses gradient ascent to choose between instantiated operators includes an estimate of the probability that the preconditions can be satisfied, for example estimating their probability in the set of possible current states. This can be viewed as a probabilistic version of the *minimum-unsolved-preconditions* heuristic used in Prodigy [Blythe & Veloso 1992] although a more faithful version would minimize the expected number of unsolved preconditions. These heuristics all make estimates using the current state, which arises from applying the operators whose order in the plan prefix is already chosen.

In the oil-spill domain, several control rules can be seen as compiled versions of the heuristic to maximise the probability of the step's preconditions in the current state. These control rules prefer pieces of equipment with higher tolerance to weather conditions. Since the main sources of uncertainty in the oil-spill domain are exogenous events that cause the situation to deteriorate over time, control rules that prefer operators that take less time, for instance using closer equipment or faster transport, also maximise the estimated probability of success, but this is based on predictions about how the world will change rather than on the current state. The effect of these heuristics is explored in Section 7.5.

6.1.2 The footprint principle

Probabilistic planners such as Weaver, Buridan [Kushmerick, Hanks, & Weld 1995] and DRIPS [Haddawy & Suwandi 1994] build plans by incrementally refining some candidate plan to increase its probability of success, stopping when the threshold probability is reached and backtracking if no extensions achieve the desired probability. This incremental refinement might for example be the addition of a new conditional branch of the plan, or some extra steps to improve the probability of some subgoal in the plan. The heuristics based on the footprint principle aim to increase the rate of convergence to a good plan, by choosing improvements to an existing plan that are more likely to significantly increase the probability of success. This is similar to the maximum expected utility approach of Russell and Wefald [91] . While the exact heuristics may differ between different planners, the underlying principles are the same. I begin with two observations about the process of improving the probability of success of plans in uncertain domains.

Coherence

Firstly, while plans in STRIPS-style domains are created for individual initial states, plans in probabilistic domains are created for a set of states, those with sufficiently high probability in the initial distribution to influence the probability of plan success. I define a *trace* of execution of a plan in a stochastic domain to be a complete assignment for each of the choice points during the plan's execution: the initial state is specified and so is each action outcome and potential occurrence of an exogenous event. In Weaver, a trace corresponds to a path through the underlying MDP that can be realised when the plan is executed. The probability of success of a plan is equal to the probability mass of the set of traces of the plan that lead to a goal state. A simple strategy for a planner may be to try to refine a plan to cover an individual trace that has high probability and is not currently covered by the plan. Picking the most likely initial state or the most likely action outcome that lead to plan failure to work on are examples of this strategy.

However it may be that a number of distinct initial states exist that each have low probability but when combined form a set with higher probability that also shares features allowing a single plan or plan refinement to solve them all. I refer to such a set as a *coherent* set of traces with respect to the plan. It can be a valuable strategy for a probabilistic planner to expend computation looking for a coherent set of traces with a high probability mass, and this I refer to as the "coherence" strategy.

As an example, consider the family of domains "Mop(n,ϵ)" parameterised by an integer n and some small number ϵ whose operators are in Figure 6.1. The initial state distribution in an instance of Mop(n, ϵ) consists of n possible states, S_i, $1 \leq i \leq n$, where (S_i) is the sole true fact in each state S_i. State S_i has a probability of $1/2^i$ if $1 \leq i \leq n-1$ and state S_n has a probability of $1/2^{n-1}$, bringing the total to 1. The goal is always the literal (goal). There are n operators S_i which all add (goal) with certainty if their preconditions are met. Each S_i has the single precondition (S_i). In addition the single operator `Mop` achieves (goal) in any state with probability $1-\epsilon$, and with probability ϵ has no effect.

```
(Operator Si                (Operator Mop
  (preconds () (Si ))         (preconds () (true))
  (effects ()                 (effects ()
    ((add (goal)))))            (1-ε () (add (goal)))
                                (ε () nil)))
```

FIGURE 6.1: Operators in the parameterised domain "Mop(n,ϵ)". There are n operators S_i for $1 \leq i \leq n$

Consider finding a plan to achieve (goal) with probability above some threshold τ. If $\tau < 1-\epsilon$ then the one-step plan `Mop` solves the problem. However if the planner concentrates on the most likely state and selects the most promising operator for that state, its initial candidate plan will be S_1. On addressing the flaw that (goal) can be false, it will select the next most likely state S_2 and build the conditional plan:

S_1
If (goal) is false
    S_2

Thus the planner will require approximately $\log_2(1/1-\tau)$ steps to achieve (goal) with probability τ, although a one-step plan exists. If the planner instead estimates the operator most likely to achieve the goal from the probability distribution of possible current states, `Mop` will be chosen as long as $\epsilon < 1/2$. The set of possible initial

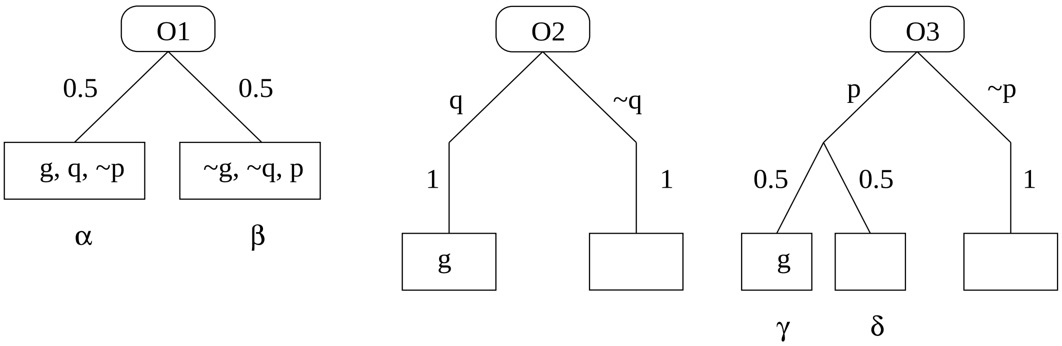

FIGURE 6.2: The operators for an example domain.

states is coherent with respect to the `Mop` action, but not with respect to any of the actions S_i. A practical way to estimate this probability is to make use of the belief net created to evaluate plan success. The marginal probabilities of the action preconditions can be accessed and combined under the heuristic assumption of independence.

`Mop` also leads to shorter plans when $\tau > 1 - \epsilon$, as long as $\epsilon < 1/2$. This is because the probability of the plan $S_1, S_2, ..., S_m$ (with the appropriate conditional branches) is $1 - 1/2^m$ as long as $m < n$. The probability of the plan `Mop`, $S_1, .., S_{m-1}$, which has the same length, is $1 - \epsilon + \epsilon(1 - 1/2^{m-1})$. This can be rewritten as $1 - \epsilon/2^{m-1}$, which is greater than $1 - 1/2^m$ as long as $\epsilon < 1/2$.

The footprint principle

The second observation is that the increase in the probability of success due to a plan refinement is not additive, since the new refinement to the plan typically does not work in a completely different set of traces from the original. Recall that if E and F are expressions, $P(E \vee F) = P(E) + P(F) - P(E \wedge F)$. Thus an estimate of $P(E \vee F)$ found by summing $P(E)$ and $P(F)$ overestimates to the extent that E and F overlap. This can happen when estimates of the probability of success of the refined plan are made from the probability of the original plan and an estimate of the goodness of the refinement.

To illustrate this, consider an example scenario from the domain with the operators shown in Figure 6.2, in which the planner has the goal g and a partial plan consisting of the operator O_1. The task is to choose an operator to add to the plan to improve its probability of success.

A comparison of the operators O_2 and O_3 that assumes independence might lead the planner to prefer O_2, since p and q are both true with probability $1/2$ after O_1 is applied, so O_2 adds g with probability $1/2$ but O_3 adds it with probability only $1/4$. However the plan consisting of O_1 followed by O_2, which I shall write as $[O_1 ; O_2]$, does not improve over $[O_1]$ since p is true only when O_1 achieves g. The plan $[O_1 ; O_3]$ has probability $3/4$ of success because O_3 precisely addresses the case when O_1

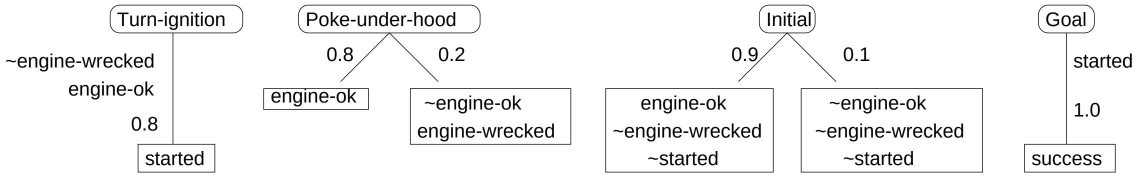

FIGURE 6.3: Operators in the car-starting domain

fails. In the general case, comparisons of operators based on their "average" ability to achieve goals tell us nothing about their usefulness for refining plans.

This simple observation about independence becomes important when we consider heuristics that estimate the increase in probability of success provided by refining a plan. I define the *footprint* of a plan as the set of traces in which it succeeds. Note that when we refine a plan by adding a step, each trace of the new plan is an extension of some trace in the old plan, so the traces in the footprint of the new plan can be classified into two categories: those that extend traces in the footprint of the original plan and those that extend other traces. The new traces that extend traces in the original footprint form the intersection of the refinement's footprint with that of the original plan, and do not improve the probability of success.

Some heuristics estimate the goodness of the refinement without regard to the existing plan, such as noting the maximum or average probability that some step will add a desired subgoal. Such heuristics are likely to over-estimate the efficacy of the refinement when its footprint has a significant intersection with the footprint of the original plan, in terms of its probability mass. This was the case for O_2 — the footprint of $[O_1; O_2]$ is completely contained within the footprint of $[O_1]$. If instead relatively cheap heuristics can be designed that take some of the overlap into account, they can be considerably more powerful than those that assume independence. I describe heuristics built on this principle as belonging to the "footprint" family. Combining the two observations in this section, by discounting the overlap between old and new parts of the plan these heuristics can be viewed as seeking refinements to the plan that have a footprint with high probability mass that is disjoint from the footprint of the original plan. Alternatively, these refinements can be viewed as having a high-probability footprint, conditional on the original plan failing.

Case study: Starting a car

Weaver has several choice points in its search, including which of several potential failure pairs to address, whether to plan to avoid the failure or address it after its occurrence, which extra preconditions to add if avoiding the failure and which state to pick if planning after a failure's occurrence. By default, Weaver deals with failures that occur earlier before those that occur later. It prefers to plan to avoid a failure rather than to branch in case of failure. If it branches, it creates a planning initial state for Prodigy by assuming the undesirable action outcome is the only one to occur.

I illustrate a heuristic based on the footprint principle on the "car-starting domain". The operators for this domain are shown in figure 6.3, and the goal is to achieve started with probability at least 0.95. In addition, the literal engine-ok is not observable, meaning that a branching plan based on this test is not allowed.

There is one plan of length 4 that achieves the goal with the required probability: turn the ignition twice, and if the car has not started, poke under the hood and then turn the ignition once more. If the ignition key is turned only once before the hood is opened, the problem cannot be solved because with too high a probability the car is wrecked[1], so Weaver's default heuristic will not work. If the agent never pokes under the hood the threshold probability cannot be reached.

I focus on the choice point of the planning initial state that Weaver passes to Prodigy when creating a branching plan, and derive a heuristic from the footprint principle . Recall that in this case, Weaver has chosen a literal l that has a low probability of taking a value required for the current plan, and an action that can lead to an undesired value for l. The default heuristic creates the state that would be produced if the operators in the plan gave the outcome nominated by Prodigy up to the suspect action, which then takes the most likely outcome that produces an undesired value for l.

I take a Bayesian view of the footprint principle described earlier in this section and replace the default heuristic by one that attempts to create a likely state given that the original plan fails in a way predicted by l and a. The initial state distribution P_I is taken as a prior distribution, and the posterior, given that a produces an undesired value for l, is computed by propagating the undesired outcome as evidence in the belief net. Then one of the most likely states from the posterior distribution is chosen, and the plan simulated, each time picking one of the most likely outcomes given the new distribution.

When Weaver is run with the default heuristic, it begins solving the start-car problem with the plan turn-ignition, and then attempts to create a branching plan based on the test started after the action is executed. With the default heuristic, Prodigy plans for a state with engine-ok true, producing the branching plan

```
turn-ignition
If not started
     turn-ignition
```

Next Weaver creates a branching plan based on the test started after the second time the ignition is turned. The default heuristic produces exactly the same state, and Weaver appears caught in an endless loop[2], producing a plan whose probability of success asymptotically approaches 0.9.

With the footprint heuristic for choosing a state, Weaver starts in the same way, and tries to improve the plan [turn-ignition] when started is false. Now the probability

[1]This is only a slight exaggeration of my skills as a mechanic.

[2]This is the behaviour with depth-first search, and it can be avoided using a scheme based on depth-first iterative deepening.

of the initial state having engine-ok given that started is false drops from 0.9 to roughly 0.64. Since this is still the most likely state it is chosen and the new plan involves two turns of the ignition key as before. When Weaver creates a state for Prodigy given that the car fails to start twice, however, the probability of the initial state with engine-ok drops from 0.64 to 0.26, and it becomes more likely that the engine is broken. This state is chosen, the current plan is simulated on it, and Prodigy returns the plan poke-under-hood ; turn-ignition. The branching plan that results has probability just over 0.95 of success, and is returned.

In general, propagating evidence through the Bayes net to compute the posterior state distribution could take time exponential in the size of the net. In practice, updates are typically fast since the net can often be constructed to avoid large cycles and nodes with many parents. In other cases, a heuristic based on approximating the posterior distribution may still prove valuable.

6.2 Using derivational analogy in a conditional planner

The work presented in this chapter has two purposes. First, I focus on improving the efficiency of the conditional planner described in Chapter 3. In some cases the planner can duplicate search effort unnecessarily when the same goals must be planned for in different branches of a conditional plan. Analogical replay can be used to share the search effort between the different branches. The approach of derivational analogy, which reconstructs the plan's justification rather than following the plan at a surface level, helps to ensure that the search is shared only when appropriate.

The second purpose of this chapter is to demonstrate that the machine learning techniques developed for Prodigy 4.0, a classical planner that does not handle uncertainty, are still useful and applicable to Weaver, an extension of Prodigy 4.0 as a conditional planner that handles uncertainty. Prodigy's algorithm for derivational analogy was originally developed by Manuela Veloso [Veloso 1994] for No-Limit, a predecessor to Prodigy 4.0, and subsequently ported. Weaver is able to make use of the algorithm with very few changes because it is a conservative extension to Prodigy, itself having as few changes as are needed to perform conditional planning. Given the amount of effort that has been spent on efficient classical planners over the past decade *e.g* [Minton 1988; Etzioni 1990; Knoblock 1991; Gil 1992], it is significant that many of the ideas can be re-used.

I begin this chapter with a discussion of the motivation for using analogical replay in the conditional planner, then discuss the algorithm and illustrate its performance on some synthetic domains. Further experiments are described in Chapter 7, and some comparisons are made with other probabilistic planning systems in Chapter 2. Some of the work described in this section was done jointly with Manuela Veloso [Blythe & Veloso 1997].

6.2.1 The need to share search effort within the conditional planner

Consider a simple planning problem in which a package is to be loaded into a truck. In the initial state, the package is at the depot and the truck is at the warehouse. However, consider also that, in the time it takes to drive the truck to the depot, the package can be misplaced with probability 0.5. When this happens, the package is transferred from the depot to the lost-property department, from where it cannot be lost. The following branching plan solves this problem: drive the truck to the depot and if the package is still there load it into the truck, otherwise drive to lost property and load the package into the truck.

Figure 6.4 shows a tail plan with four steps that may be constructed by Weaver's conditional planner to solve the problem described at the beginning of this section. The truck must be made ready, with step 1 "start-truck" before it can be driven, and the final goal requires that the truck is put away, with step 4 "stop-truck." The arc from "drive to depot" to "load package at depot" indicates that the former step is an establisher of the latter. We have omitted the preconditions from the diagram.

Step 2 "drive to depot," is a context-producing step that has two possible sets of effects. Contexts α and β correspond respectively to the situations where the package is still at the depot and where it is in lost property, when the truck arrives. A step is marked as context-producing externally to B-PRODIGY (in fact by Weaver as we described).

FIGURE 6.4: Initial tail plan to solve example problem. The directed arcs are causal links showing that a step establishes a necessary precondition of the step it points to.

When a branching step is introduced into the tail plan, producing new contexts, the other steps are initially assumed to belong to all contexts. This is true of step 3, "load package at depot." However each step's contexts can be restricted to a subset, and new steps can be added to achieve the same goal in the other contexts. In this example, new operators could be introduced both for the top-level goal and the goal for the truck to be at the depot. The branching step is always introduced with a commitment that one of its outcomes will be used to achieve its goal, and all the ancestors of the step in the tail plan must always apply to that context. In this example step 2 was introduced to establish step 3 using context α, so step 3 may not be restricted to context β.

In Figure 6.5, step 3 has been restricted to context α, new steps have been added to achieve the top-level goal in context β, and steps 1 and 2 been moved to the head plan. Two recursive calls to B-PRODIGY will now be made, one for each context, in each of which B-PRODIGY will proceed to create a totally-ordered (but possibly nonlinear) plan. In context α, the steps labelled with β, driving to and loading at lost-property, will be removed from the tail plan. In context β, the step labelled with α, "load package at depot," will be removed. Step 4, stop-truck, has not been restricted and remains in the tail plan in both contexts. No steps are removed from the head plan, whatever their context. Each recursive call produces a valid totally-ordered plan, and the result is a valid conditional plan that branches on the context produced by the step "drive to depot."

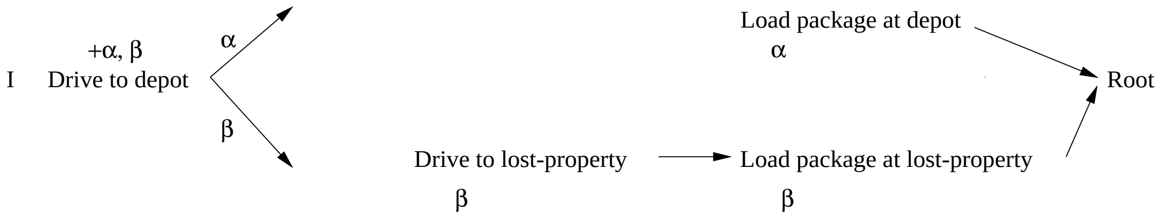

FIGURE 6.5: A second stage of the planner. There are now two possible current states for contexts α and β.

The ability to create conditional plans is vital to planners that deal with uncertainty, however creating them can lead to high computational overheads because of the need for separate planning effort supporting each of the branches of the conditional plan, measured in terms of the computation required to search for an validate the plan. This problem can be alleviated by sharing as much of the planning effort as possible between different branches.

Analogical replay enables sharing the effort to construct plans, while revalidating decisions for each new branch. As the following three examples show, in some cases the planning architecture directly supports sharing this effort and in others it does not.

1. Step 1, start-truck, is shared through the *head plan*. The step is useful for both branches, but is only planned for in one since its effect is shared through the current state. Although in this plan there is only one step, in general an arbitrary amount of planning effort could be shared this way.

2. Step 4, stop-truck, is shared through the *tail plan*. As shown in Figure 6.5, this step does not have a context label and achieves its goal in either context. Thus when the branching step is moved to the head plan, it remains in the tail plan in each recursive call made to B-PRODIGY, making the planning effort available in each call.

3. Sometimes duplicated planning effort is not architecturally shared as in the last two examples. Suppose that extra set-up steps are required for loading a truck. These would be added to the tail-plan in Figure 6.5 in two different places, as establishers of steps 3 and 6, and restricted to different contexts in the two places. Thus, the planning effort cannot be shared by the tail-plan unless it is modified from a tree to a DAG structure. But this approach would lead to

problems if descendant establishing steps were to depend on the context, for instance on the location of the truck. We show below how the use of analogy can transfer planning effort of this kind between contexts. Analogy provides an elegant way to handle other kinds of shared steps as well.

6.2.2 Using derivational analogy in Weaver

Weaver's conditional planner, B-PRODIGY, is integrated with Prodigy/Analogy in such a way that the usual tasks of case retrieval and set-up are made trivial because the case memory is restricted to the current problem. First, a previously visited branch is selected to guide planning for the new branch. By default, the branch that was solved first is used. Next B-PRODIGY is initialized to plan from the branch-point, the point where the new branch diverges from the guiding branch. Then B-PRODIGY plans for the new branch, guided by Prodigy/Analogy, proceeding as usual by analogical-replay: previous decisions are followed if valid, unnecessary steps are not used, and new steps are added when needed. Prodigy/Analogy successfully guides the new planning process based on the high global similarity between the current and past situations. This is particularly well suited for the analogical replay guidance and typically leads to minor interactions and a major sharing of the past planning effort. The smoothness of this integration is made possible by the common underlying framework of the Prodigy planning and learning system.

In this integration of conditional planning and analogy, the analogical replay within the context of different branches of the same problem can be viewed as an instance of internal analogy [Hickman, Shell, & Carbonell 1990]. The accumulation of a library of cases is not required, and there is no need to analyze the similarity between a new problem and a potentially large number of cases. The branches of the problem need only to be cached in memory and most of the domain objects do not need to be mapped into new objects, as the context remains the same. While we currently use this policy in our integration, the full analogical reasoning paradigm leaves us with the freedom to reuse branches across different problems in the same domain. We may also need to merge different branches in a new situation.

Table 6.1 presents the analogical reasoning procedure combined with B-PRODIGY. We follow a single case corresponding to the plan for the last branch visited according to the order selected by Weaver.

The adaptation in the replay procedure involves a validation of the steps proposed by the case. There may be a need to diverge from the proposed case step, because new goals exist in the current branch (step 9). Some steps in the old branch can be skipped, as they may be already true in the new branching situation (step 13). Steps 8 and 12 account for the sharing between different branches and for most of the new planning, since typically the state is only slightly different and most of the goals are the same across branches. This selective use of replay controls the combinatorics of conditional planning.

procedure *b-prodigy-analogical-replay*
1. Let C be the guiding case,
   and C_i be the guiding step in the case.
2. Set the initial case step C_0 based on the branch
   point.
3. Let $i = 0$.
4. Terminate if the goal state is reached.
5. Check which type of decision is C_i:
6. If C_i adds a step O to the head plan,
7.   If O can be added to the current head plan
     and no tail planning is needed before,
8.   then Replay C_i; Link new step to C_i; goto 14.
9.   else Hold the case and call B-PRODIGY,
       if planning for new goals is needed; goto 5.
10. If C_i adds a step O_g to the tail plan, to achieve
    goal g,
11.   If the step O_g is valid and g is needed,
12.   then Replay C_i; Link new step to C_i; goto 15.
13.   else Mark unusable all steps dependent on C_i;
14. Advance the case to the next usable step C_j;
15. $i \leftarrow j$; goto 5.

TABLE 6.1: Overview of the analogical replay procedure combined with B-PRODIGY.

6.2.3 Illustrations of performance using synthetic domains

Each segment of a conditional plan has a corresponding planning cost. If a segment is repeated k times and can be shared but is not, the planner incurs a penalty of $k-1$ times the cost of the segment. Suppose that a plan contains n binary branches in sequence, all of which share steps. Either the first or the last part of the plan may be created 2^n times, but with step sharing it may only need to be created once. This exponential cost increase can quickly become a dominant factor in creating plans for uncertain domains.

Chapter 7 describes experiments to determine the performance of analogical replay in the oil-spill domain. However, to isolate features of the domain or problem that are pertinent to performance, a family of synthetic domains was created in order to comprehensively verify experimentally the effect of analogy in conditional planning. These domains allow precise control over the number of branches in a plan, the amount of planning effort that may be shared between branches and the amount that belongs only to each branch.

Table 6.2 describes the operators. The top-level goal is always g, achieved by the operator Top. There is a single branching operator, Branch, with N different branches corresponding to N contexts. A plan consists of three main segments. The first segment consists of the steps taken before the branch point. These are all in aid of the goal bb, the only goal initially unsatisfied, which is achieved by the step Branch. All of the branches of Branch achieve bb, but delete cx and sh. Each branch also adds a unique "context" fact, c_i. After the branch point, the second segment is a group of steps unique to each individual branch, in aid of the goal cx. We name each of these segments "C_i." Finally, the third "shared" segment contains the steps which are the same in every branch in aid of the goal sh. They must be taken after the branching point, since Branch deletes sh. The planning work done in each segment is controlled by the iterative steps that achieve the predicates $\mathsf{iter\text{-}bb}_0$, $\mathsf{iter\text{-}cx}_{i,0}$ and $\mathsf{iter\text{-}sh}_0$. The plan to achieve $\mathsf{iter\text{-}sh}_0$, for example, has a length determined by some number z for which $\mathsf{iter\text{-}sh}_z$ is the initial state. The planner selects the operators $\mathsf{S\text{-}sh}_k$ which succeed or $\mathsf{F\text{-}sh}_k$ $(k = 0, \ldots, z)$ which fail as their preconditions $\mathsf{u\text{-}sh}_k$ of the operators $\mathsf{A\text{-}sh}_k$ are all unachievable. The domain can have N copies of these operators. So the planning effort to solve $\mathsf{iter\text{-}sh}_0$ is up to $2N \times z$ operators added to the tail plan, all but z of which are removed.

Operator	*Preconds*	*Adds*	*Deletes*
Top	bb, cx, sh	g	–
Branch	$\mathsf{iter\text{-}bb}_0$	bb,c_i	sh, cx
where i refers to each branch i, $i = 1, \ldots, B$			
Contx_i	$\mathsf{iter\text{-}cx}_{i,0}$, c_i	cx	–
Shared	$\mathsf{iter\text{-}sh}_0$	sh	–
$\mathsf{F\text{-}bb}_l$	$\mathsf{a\text{-}bb}_l$	$\mathsf{iter\text{-}bb}_l$	–
$\mathsf{A\text{-}bb}_l$	$\mathsf{u\text{-}bb}_l$	$\mathsf{a\text{-}bb}_l$	–
$\mathsf{S\text{-}bb}_l$	$\mathsf{iter\text{-}bb}_{l+1}$	$\mathsf{iter\text{-}bb}_l$	–
$\mathsf{F\text{-}cx}_{i,m}$	$\mathsf{a\text{-}cx}_{i,m}$	$\mathsf{iter\text{-}cx}_{i,m}$	–
$\mathsf{A\text{-}cx}_{i,m}$	$\mathsf{u\text{-}cx}_{i,m}$	$\mathsf{a\text{-}cx}_{i,m}$	–
$\mathsf{S\text{-}cx}_{i,m}$	$\mathsf{iter\text{-}cx}_{i,m+1}$	$\mathsf{iter\text{-}cx}_{i,m}$	–
$\mathsf{F\text{-}sh}_k$	$\mathsf{a\text{-}sh}_k$	$\mathsf{iter\text{-}sh}_k$	–
$\mathsf{A\text{-}sh}_k$	$\mathsf{u\text{-}sh}_k$	$\mathsf{a\text{-}sh}_k$	–
$\mathsf{S\text{-}sh}_k$	$\mathsf{iter\text{-}sh}_{k+1}$	$\mathsf{iter\text{-}sh}_k$	–
where l, k, m capture the lengths of the segments			

TABLE 6.2: Operator schemas in the test domain.

Figure 6.6 shows an example of a plan generated for for a problem with goal g, and initial state cx, sh, $\mathsf{iter\text{-}bb}_x$, $\mathsf{iter\text{-}cx}_{1,y}$, ..., $\mathsf{iter\text{-}cx}_{B,y}$, and $\mathsf{iter\text{-}sh}_z$.

We have performed extensive experiments with a variety of setups. As an illustrative performance graph, Figure 6.7 shows the planning time in seconds when the

S-bb$_x$...S-bb$_0$ Branch

S-cx$_{1,y}$...S-cx$_{1,0}$ Contx$_1$ S-sh$_z$...S-sh$_0$ Top
...
S-cx$_{B,y}$...S-cx$_{B,0}$ Contx$_B$ S-sh$_z$...S-sh$_0$ Top

FIGURE 6.6: A typical plan in the test domain.

number of branches is increased from 1 to 10. For this graph, the final plan has one step in each of the "C_i" and the "shared" segments. The domain was chosen so that B-PRODIGY examined 4 search nodes to create a "C_i" segment and 96 for the "shared" segment. With less extreme proportions of shared planning time to unique planning time, the shape of the graph is roughly the same and analogical replay still produces significant speedups. With 24 search nodes examined for each unique plan segment, and 72 for the shared segment, B-PRODIGY completes the plan with 10 branches more than twice as quickly with analogical replay as without it.

The improvement in time is similar when the depth and breadth of search are increased for the shared segment and the number of branches is held constant.

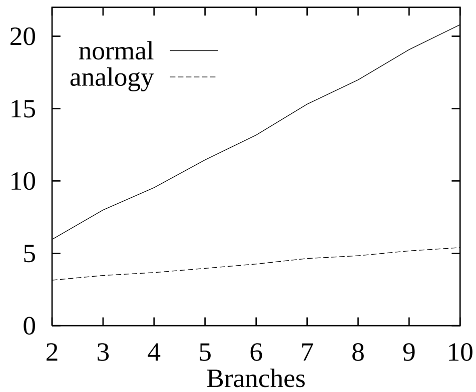

FIGURE 6.7: Time in seconds to solve planning problem with and without analogy plotted against the number of branches in the conditional plan. Each point is an average of five planning episodes.

The use of analogical replay in B-PRODIGY is a heuristic based on the assumption that a significant proportion of planning work can be shared between the branches. We tested the limits of this assumption by experiments holding constant both the number of branches and the effort to create the shared segment, and increasing the effort to create each unique segment. Under these conditions, the time taken by B-PRODIGY grows at the same rate whether or not analogical replay is used, because the overhead of replay is small relative to planning effort, and appears constant. When the planning effort for each C-i was increased from 4 to 100 search nodes while the effort to create a shared branch was held constant at 24 search nodes, B-PRODIGY took about 1 second longer with analogy than without it, an overhead of less than 10 per cent when each C-i took 100 search nodes.

Chapter 7

Experimental results in the Oil-spill domain

Throughout this thesis, most of the techniques introduced have been described using synthetic domains, and these have also been used to demonstrate the potential benefits of the techniques. In this chapter I present an experimental demonstration of these techniques in the domain of oil-spill clean-up. This is a real-world domain, consisting of 62 operator and inference rule schemata, that was originally developed by SRI for the US Coast Guard [Desimone & Agosta 1994].

In the next section I provide an overview of the oil-spill domain. Sections 7.2 and 7.3 discuss how Weaver improves plans in this domain. In section 7.4 I demonstrate the improvement in the time to evaluate the belief net that comes from using the event graph technique describe in Chapter 5. In Section 7.5 I investigate the effect of domain-independent heuristics and in Section 7.6 I investigate the effect of derivational analogy in this domain.

7.1 The oil-spill clean-up domain

The domain models the problem of cleaning up oil-spills from vessels, which comprises 3 kinds of activity: (1) stopping the flow of oil from a tanker, (2) cleaning up oil from the surface of the water and (3) protecting and/or cleaning sensitive areas of coastline. The domain uses 151 object types, of which 52 of the most important are shown in Figure 7.1. The 62 operators and inference rules of the domain can be roughly grouped as follows:

- 8 are used exclusively for stopping the flow of oil from a vessel, by pumping oil from the vessel, towing the vessel to port or using booms to contain the oil around the vessel and skimming it from the surface.
- 18 are used for cleaning oil from the sea, by skimming it into a barge, using chemical dispersants or controlled burns.

- 11 are used for protecting sensitive areas of coastline, either using booms to keep oil from the shore, or using booms or berms and dams to divert oil and vacuum trucks to clean it from the shore.
- 9 are used for shared subgoals of the above three high-level activities. They include placing booms and making inferences about the state of the sea.
- 16 deal with moving equipment, teams and vessels.

The domain is shown in detail in Appendix B.

The main sources of uncertainty in the domain come from the movement of the oil and changes in the weather conditions while the plan is being executed. Equipment is stored at various locations along the coast and may have to be moved large distances to be employed in a clean-up operation. Therefore plans may include long lead times before crucial steps, such as positioning a boom or pumping oil into a barge, can take place. Over these time periods exogenous events model the weather and the amount of oil spilled in the sea as stochastic processes. Certain steps require a level of calmness of the sea in order to be performed, and their probability of success depends on these exogenous events.

The uncertainty in the oil spread over time can also affect the allotment of resources for shore protection. For example in some cases spilled oil may have some probability of reaching one or more of a number of sensitive areas of coastline but resources may not be sufficient to protect all of them. In these cases a conditional plan that moves resources once the direction of the oil is known can have a higher probability of success than any non-conditional plan. An example of this is shown in Section 7.2

The planning problems used for this domain model the geography and equipment available in the San Francisco bay area (Figure 7.2). These problems are modelled with 183 objects belonging to subtypes of `place`, 55 belonging to subtypes of `equipment`, 30 belonging to subtypes of `vessel` and 119 other objects. Details can be found in [Desimone & Agosta 1994].

In the remainder of this chapter I demonstrate Weaver and show experimental results in the oil-spill domain about Weaver's performance and about the impact of the techniques described in this thesis.

7.2 Case studies of improving plan reliability

This section provides demonstrations of how Weaver improves the probability of plan success. The next section provides experimental results and analysis of the improvement obtained for a set of randomly generated examples. Subsequent sections that demonstrate techniques for improving Weaver's speed will focus on the rate of improvement of probability. When Prodigy is used to create plans in the oil-spill domain, it ignores the domain's sources of uncertainty. Consequently Weaver can improve on plans found by Prodigy in three qualitatively different ways:

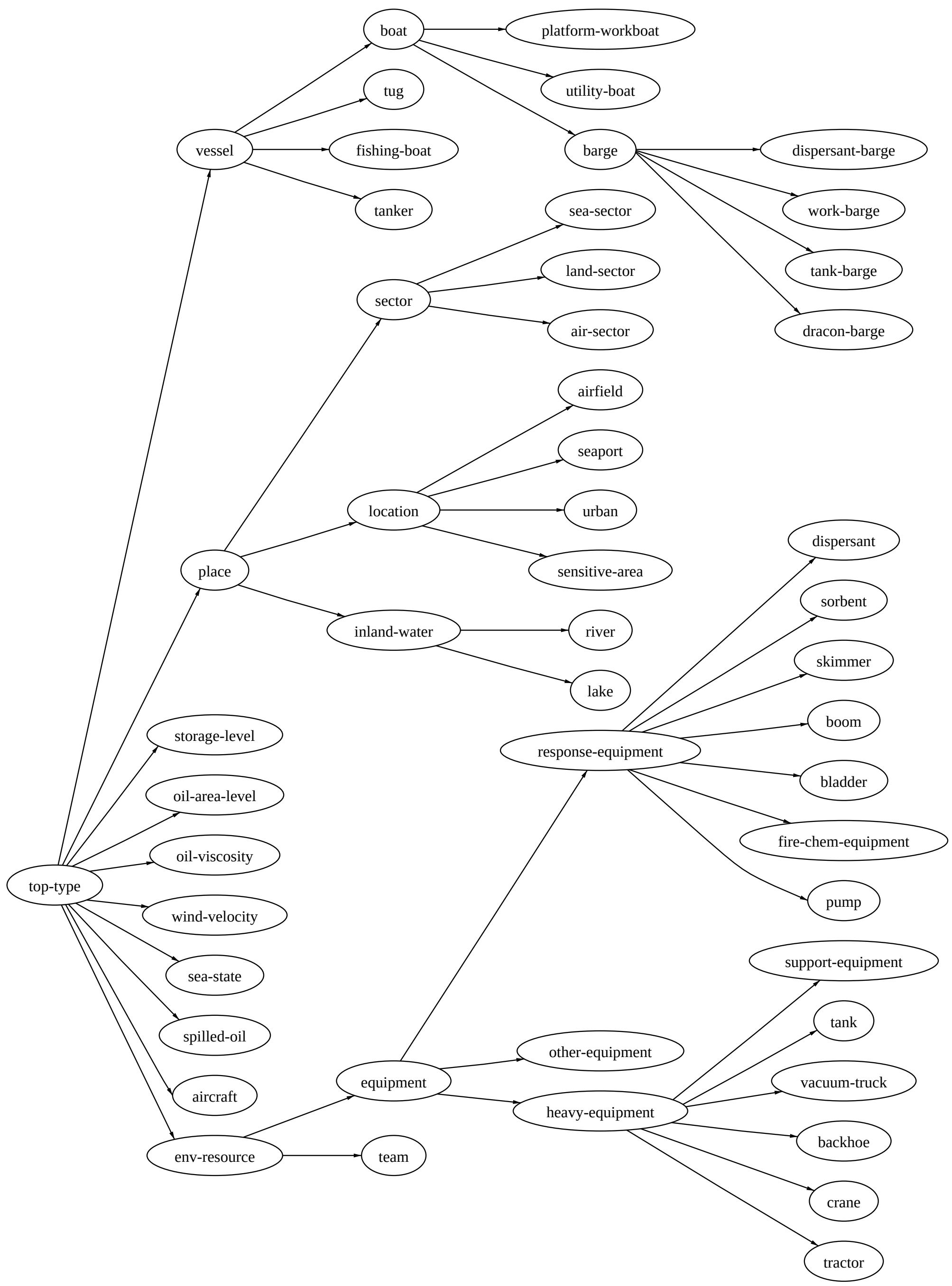

FIGURE 7.1: Part of the type hierarchy, showing the 52 most important of the 151 types in the domain.

1. Better plans can be found by backtracking from one choice of operator or bindings to another that avoids the source of uncertainty.

2. Conditionally branching plans can be created that in some cases have a higher probability of success than is possible without including conditional branches.

FIGURE 7.2: San Francisco Bay area

3. Better plans can be found by including steps whose purpose is to prevent some undesired action outcome or exogenous event.

The mechanisms by which Weaver can use these techniques are described in Chapter 4. Here I provide a brief example of the last two of these effects in the oil-spill domain.

Creating a conditional plan

When oil is spilled, it can move unpredictably on the surface of the water directed by the wind and currents, so that the stretches of coastline that will be hit by oil are uncertain. Consequently the number of sensitive areas potentially under threat from the oil may be much larger than the number that the oil may reach in any single eventuality. Frequently the available resources for cleaning up the shore may be adequate for only a small number of areas and a more reliable plan can be made by delaying the deision on where to deploy the resources as late as possible so that the path of the oil is known with greater certainty. The approach relies on building a conditional plan that tests for the sensitive areas under greatest threat.

In the version of the oil-spill domain under test, the fact that a particular sensitive area of coastline is likely to be hit by oil is modelled with the predicate `(threatened-shoreline <oil> <sensitive-area>)` and the stochastic motion of the oil in the water is modelled by an exogenous event `threatened-shore-changes`, shown in Figure 7.3. This is a primitive model in which the threat from the oil shifts from one area to another without regard to geographic proximity, although the constraint function `potentially-threatened` ensures that the areas are both adjacent to the sea-sector containing the oil.

```
(event threatened-shore-changes
 (params <sensitive-area-1> <sensitive-area-2> <oil>)
 (probability 0.1)
 (preconds
  ((<oil> Spilled-oil)
   (<sea-sector> Sea-Sector)
   (<amount> (and Numerical
                  (gfp (amount-spilled <oil> <sea-sector> <amount>))
                  (> <amount> 0)))
   (<sensitive-area-1> (and Sensitive-Area
                            (potentially-threatened <sensitive-area-1> <oil>)))
   (<sensitive-area-2> (and Sensitive-Area
                            (potentially-threatened <sensitive-area-2> <oil>)
                            (diff <sensitive-area-1> <sensitive-area-2>))))
  (threatened-shoreline <oil> <sensitive-area-1>))
  (effects ()
   ((del (threatened-shoreline <oil> <sensitive-area-1>))
    (add (threatened-shoreline <oil> <sensitive-area-2>)))))
```

FIGURE 7.3: The exogenous event `threatened-shore-changes` models the uncertainty of the oil's path in the water.

Table 7.1 shows the objects and the goal for an example problem, and Table 7.2 shows the initial state for the problem. The tanker `ss-weany` has spilled oil in the `golden-gate` sea sector, whose adjacent shoreline, `west-marin`, contains two sensitive areas, `rodeo-lagoon` and `pt-bonita-cove`. Initially, only `pt-bonita-cove` is threatened. The goal has been simplified by removing the requirement to clean oil up from the surface of the water, leaving the requirements to stop the discharge of oil from the tanker and protect the threatened sensitive area of coastline. There is a tank barge and pump available to pump oil out of the tanker to stop its discharge. There is a tractor and a vacuum truck which can be used to build berms and dams to protect one area of coastline.

The geographic regions and the specific properties of the response equipment in this problem correspond exactly to objects in the scenarios designed by SRI in conjunction with the coast guard [Desimone & Agosta 1994]. The example has been made short by modelling a small oil spill that endangers only a small number of locations. As an uncertain planning problem it has also been simplified by modelling only the movement of the oil, not the change in the sea state or oil spilling.

Prodigy's solution for this problem, ignoring inference rules, is shown in Table 7.3. The first three steps stop the discharge of oil from the tanker and the next three protect `pt-bonita-cove`. Weaver constructs a probabilistic model of this plan and finds it has a probability of success of roughly 1/2, because in the time taken to stop

```
Objects:
    (spill SPILLED-OIL)
    (pt-bonita-cove rodeo-lagoon SENSITIVE-AREA)
    (golden-gate SEA-SECTOR)
    (west-marin LAND-SECTOR)
    (richmond-port SEAPORT)
    (oakland URBAN)

    (ss-weany TANKER)
    (tank-barge2 TANK-BARGE)
    (utility-boat-2 UTILITY-BOAT)
    (cargo-pump2 CARGO-TRANSFER-PUMP)
    (vac-truck2 VACUUM-TRUCK)
    (tractor-2 TRACTOR)

Goal: (and (no-discharge ss-weany spill golden-gate)
           (~ (unprotected-sensitive-area spill)))
```

Table 7.1: The objects and goal for the example problem used to demonstrate a conditional plan formed under scarce resources.

```
(unprotected-sensitive-area spill)
(threatened-shoreline spill pt-bonita-cove)
(amount-spilled spill golden-gate 10)
(discharge-size ss-weany spill 1000)
(discharge-rate ss-weany spill 600)
(sea-state golden-gate 2)
(located-within rodeo-lagoon west-marin)
(located-within pt-bonita-cove west-marin)
(adjacent golden-gate west-marin)
(distance richmond-port golden-gate 25)
(distance oakland rodeo-lagoon 20)
(distance oakland pt-bonita-cove 20)
(distance richmond-port rodeo-lagoon 20)
(distance richmond-port pt-bonita-cove 20)
(shore-personnel-required pt-bonita-cove 10)
(shore-personnel-required rodeo-lagoon 10)

(located ss-weany golden-gate)
(located tank-barge2 richmond-port)
(located cargo-pump2 richmond-port)
(located vac-truck2 richmond-port)
(located tractor-2 oakland)
(max-speed tank-barge2 2)
(barge-capacity-bbl tank-barge2 4000)
(max-sea-state tank-barge2 4)
(max-speed utility-boat-2 5)
(capacity-bbl vac-truck2 1000)
(capacity-bbl-per-hr cargo-pump2 480)
```

Table 7.2: The initial state for the example problem used to demonstrate a conditional plan formed under scarce resources.

the discharge of oil, the sensitive area that is threatened by oil can change a number of times and the probability distribution of the threatened area has approached its steady state value.

Weaver's plan critic forces B-Prodigy to find a branching plan, splitting on the test `(threatened-shoreline spill pt-bonita-cove)`. The exogenous event is modelled in B-Prodigy as taking place due to some action that takes non-zero time. By default the latest possible action is chosen, to push the branch as late as possible. In

```
(move-response-equipment-by-sea1b cargo-pump2 golden-gate richmond-port
                                  utility-boat-2)
(move-to-sea-sector-from-port tank-barge2 richmond-port golden-gate)
(cargo-transfer-oil-to-stabilize ss-weany spill golden-gate
                                 tank-barge2 cargo-pump2 600 120)
(move-heavy-equip-by-ground2 tractor-2 oakland pt-bonita-cove)
(move-heavy-equip-by-ground2 vac-truck2 richmond-port pt-bonita-cove)
(build-berms-and-dams spill pt-bonita-cove)
```

TABLE 7.3: Prodigy's solution to the example problem, ignoring inference rules.

```
(move-response-equipment-by-sea1b cargo-pump2 golden-gate richmond-port
                                  utility-boat-2)
(move-to-sea-sector-from-port tank-barge2 richmond-port golden-gate)
(cargo-transfer-oil-to-stabilize ss-weany spill golden-gate
                                 tank-barge2 cargo-pump2 600 120)
IF (threatened-shoreline (spill)) is in (pt-bonita-cove)
   (move-heavy-equip-by-ground2 tractor-2 oakland pt-bonita-cove)
   (move-heavy-equip-by-ground2 vac-truck2 richmond-port pt-bonita-cove)
   (build-berms-and-dams spill pt-bonita-cove)
ELSE
   (move-heavy-equip-by-ground2 tractor-2 oakland rodeo-lagoon)
   (move-heavy-equip-by-ground2 vac-truck2 richmond-port rodeo-lagoon)
   (build-berms-and-dams spill rodeo-lagoon)
```

TABLE 7.4: Weaver's solution to the example problem, ignoring inference rules.

this case that would be the action of moving `vac-truck2`, but since neither it nor `tractor-2` can be moved more than once, no branching plan is possible at this point and Weaver backtracks to put the branch after the step `cargo-transfer-oil-to-stabilize`. B-Prodigy returns the plan shown in Table 7.4, which Weaver determines has a probability of success of 0.8. This is less than 1 because of the chance that the threatened shoreline will change after the tractor has begun to move to its location, but is higher than any non-branching plan could achieve.

Adding preventive steps

In some cases the probability of success can be improved by inserting steps into the plan that reduce the probability of some undesired event or action outcome from taking place, as described in Chapter 4. An example of this in the oil-spill domain can occur when the actions taken to stabilize the discharge of oil from a tanker take a significant amount of time to set up. During this time, oil may spill from the tanker and spread through the water and methods for cleaning it up may be unreliable and expensive. However, the flow of oil from the tanker can be contained by surrounding it with a length of boom. If this boom can be put in place relatively quickly compared with the time taken to stabilize the discharge, then much of the oil can be prevented from spilling.

Figure 7.4 shows the exogenous event `oil-spills`. The `amount-spilled` predicate models the amount of oil in the water, and is the only predicate changed by the event. The event will not take place if either there is a sufficient amount of boom surrounding the vessel, modelled by the predicate `boom-level>=`, or the tanker has been stabilized to stop the discharge, modelled with the predicate `no-discharge`. Stopping the discharge is usually a top-level goal as well as a way to prevent the `oil-spills` event from happening.

```
(event oil-spills
 (params <spilled-oil> <sea-sector> <rate> <new>)
 (probability 0.2)
 (preconds
  ((<spilled-oil> Spilled-oil)
   (<vessel> Vessel)
   (<vessel-boom-length>
    (and Numerical (gfp (vessel-boom-length <vessel> <vessel-boom-length>))))
   (<sea-sector> Sea-sector)
   (<sea-state> (and Numerical (gfp (sea-state <sea-sector> <sea-state>))))
   (<rate> (and Numerical
             (gfp (discharge-rate <vessel> <spilled-oil> <rate>))))
   (<total> (and Numerical
              (gfp (discharge-size <vessel> <spilled-oil> <total>))))
   (<old> (and Numerical (numberp <old>) (< <old> <total>)))
   (<new> (and Numerical (add-with-ceiling <old> <rate> <total> <new>)
            ;; can't spill more than there is.
            (<= <new> <total>)
            )))
  (and
   (~ (boom-level>= <vessel> <vessel-boom-length> <sea-sector> <sea-state>))
   (~ (no-discharge <vessel> <spilled-oil> <sea-sector>))
   (amount-spilled <spilled-oil> <sea-sector> <old>)
   ))
 (effects ()
  ((del (amount-spilled <spilled-oil> <sea-sector> <old>))
   (add (amount-spilled <spilled-oil> <sea-sector> <new>))
   )))
```

FIGURE 7.4: The exogenous event `oil-spills` models the uncertain flow of oil from the tanker to the sea.

Table 7.5 shows the objects that and goal for an example problem, and Table 7.6 shows the initial state for the problem. The tanker `ss-catastrophe` is discharging oil in the `SF-north-coast` sea sector, although in the initial state no oil has been spilled. A skimmer is located at `Martinez-port` which is 28 miles from the area of the spill. A boom that is long enough to contain the spill around the tanker is located 8 miles away at `Richmond-port`. The goal is to stop the discharge of oil and have no oil spilled in the sea sector.

Prodigy's solution to the problem, ignoring inference rules, is the following two-step plan:

```
Objects:
        (sf-bay-spill SPILLED-OIL)
        (sf-north-coast SEA-SECTOR)
        (martinez-port richmond-port SEAPORT)

        (ss-catastrophe TANKER)
        (weir-skimmer2 PORTABLE-SKIMMER)
        (boom6 BOOM)
        (utility-boat-1 utility-boat-2 UTILITY-BOAT)

Goal: (and (no-discharge ss-catastrophe sf-bay-spill sf-north-coast)
           (amount-spilled sf-bay-spill sf-north-coast 0))
```

TABLE 7.5: The objects and goal for the example problem used to demonstrate adding steps to prevent an event.

```
(amount-spilled sf-bay-spill sf-north-coast 0)
(discharge-size ss-catastrophe sf-bay-spill 12000)
(discharge-rate ss-catastrophe sf-bay-spill 2000)
(sea-state sf-north-coast 2)
(distance martinez-port sf-north-coast 28)
(distance richmond-port sf-north-coast 8)
(located boom6 richmond-port)
(located weir-skimmer2 martinez-port)
(located ss-catastrophe sf-north-coast)
(located utility-boat-1 martinez-port)
(located utility-boat-2 richmond-port)
(vessel-boom-length ss-catastrophe 1000)
(max-sea-state boom6 4)
(length-boom-ft boom6 1600)
(max-sea-state weir-skimmer2 6)
(skim-rate-bbl-per-hr weir-skimmer2 2000)
(max-speed utility-boat-1 5)
(draft utility-boat-1 4)
(max-speed utility-boat-2 5)
(draft utility-boat-2 4)
```

TABLE 7.6: The initial state for the example problem used to demonstrate adding steps to prevent an event.

```
(move-response-equipment-by-sea1b weir-skimmer2 sf-north-coast
                                  martinez-port utility-boat-1)
(get-skimmer-to-skim-near-vessel ss-catastrophe weir-skimmer2
                                 sf-north-coast 2 2000)
```

This achieves the goal of stopping the discharge, but it takes nearly 6 hours to move the skimmer to the spill site and the plan's probability of success is only around 0.26 because in this time oil is likely to spill. One way to improve the probability of success is to negate the event `oil-spills` as quickly as possible, by moving the boom from `Richmond-port` which can be done in under 2 hours. The plan critic tries this approach and adds the appropriate predicate using `boom-level>=` to the preconditions of moving the skimmer and calls B-Prodigy again. The resulting four-step plan has a probability of success of 0.8:

```
(move-response-equipment-by-sea1b boom6 sf-north-coast
                                  richmond-port utility-boat-2)
(use-boom-to-contain-vessel ss-catastrophe boom6 1000 sf-north-coast)
(move-response-equipment-by-sea1b weir-skimmer2 sf-north-coast
                                  martinez-port utility-boat-1)
(get-skimmer-to-skim-near-vessel ss-catastrophe weir-skimmer2
                                 sf-north-coast 2 2000)
```

7.3 Experimental results for improving plan reliability.

In order to experimentally verify the amount of improvement in probability of success that Weaver can produce, Weaver was run with 75 randomly generated problems each solved in 4 different ways, making random choices at the planner and plan critic choice points, yielding a total of 300 trials.

The algorithm to generate the random problems fixes the geography of the area, as defined by the set of objects of types `sea-sector`, `land-sector`, `urban`, `seaport` and `sensitive-area` and by the literals for the predicates `adjacent`, `located-within`, `distance` and `berth-size`. The geographic information models the San Francisco Bay area as defined in the version of the domain written at SRI. Pseudo-code for the algorithm is given at the end of Appendix B. Within this geographic framework, one vessel is placed in a random sea-sector that contains at least one sensitive-area, with a random amount of oil, spill-rate and length, chosen within fixed parameters. A random number of equipment objects, for example, of type `boom`, `tractor` and `tug`, are distributed among the seaports at random. Finally the amount of work required to protect a sensitive area is varied as are the initial weather conditions. This generation algorithm produces a set of problems that, along with random choices made in the problem solver, exercise all possible combinations of the strategies available to the planner while ensuring that the generated problems are physically plausible.

The problems vary in difficulty for the planner in the number of steps required, in the maximum achievable probability of success and in the number of improvement steps needed to achieve a high probability of success. One of the major factors affecting these aspects of the problems is the amount of equipment available in the initial state as compared with the size of the oil spill. With adequate equipment available for a number of clean-up techniques the planner's task is relatively easy, but as equipment becomes more limited the planner may have to produce unusual combinations of equipment or use equipment that is more likely to fail. A second factor is the location of the equipment. If critical equipment is located far from the spill, the plan may take a long time to execute and therefore allow more time for exogenous events that can reduce its probability of success. A third factor is the initial state of the sea. Rough seas can make some clean-up equipment, particularly booms and skimmers, unusable.

Figure 7.5 shows a graph of the average probability of the plans found as the number of iterations through the Weaver algorithm is increased, along with the 10th percentile and 90th percentile probabilities. This graph uses a random sample of 250 examples. The rate of increase in average probability slows as the number of iterations increases. There is only slight improvement after the first two iterations, partly because a significant number of plans succeed with probability 0.9 or greater after two iterations. Figure 7.6 shows how many of the trials reach their maximum value at each iteration.

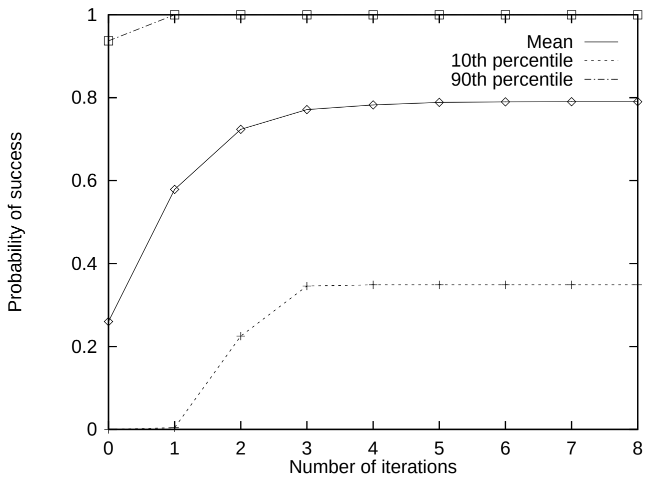

FIGURE 7.5: The average probability of success for the plans for the randomly generated problems, plotted against the number of iterations of improvement. The upper and lower lines show the 90th and 10th percentiles of the probabilities respectively.

Figure 7.7 shows the average improvement per iteration for the first four iterations on the random set of problems

two different types of improvement that Weaver can use: adding a conditional branch and adding preventive steps. The immediate improvement from backtracking is not shown. Its average value in the oil-spill domain is negative, that is, the plan's

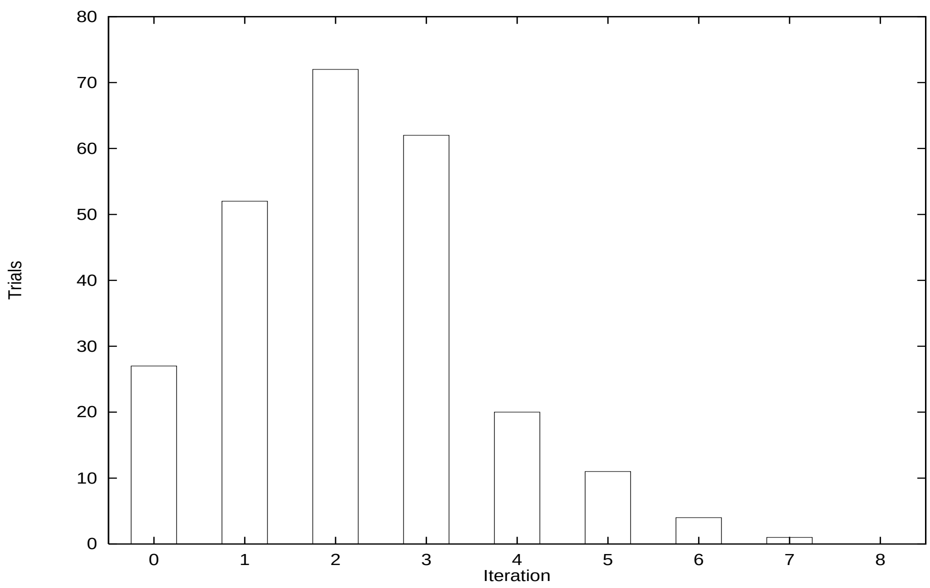

FIGURE 7.6: The number of examples that reach their maximum probability at each iteration.

probability of success typically decreased when the planner backtracked. However, backtracking allows the planner to explore other parts of the space of plans and is frequently used before Weaver finds its best plan. Although the average improvement from adding a conditional branch is approximately one quarter that of adding preventive steps in this domain, the overall increase in probability due to adding conditional branches is higher because this method is used more often. The percentage of the total improvement in probability over the random problem set that is due to each improvement type is shown in the graph on the right of the figure.

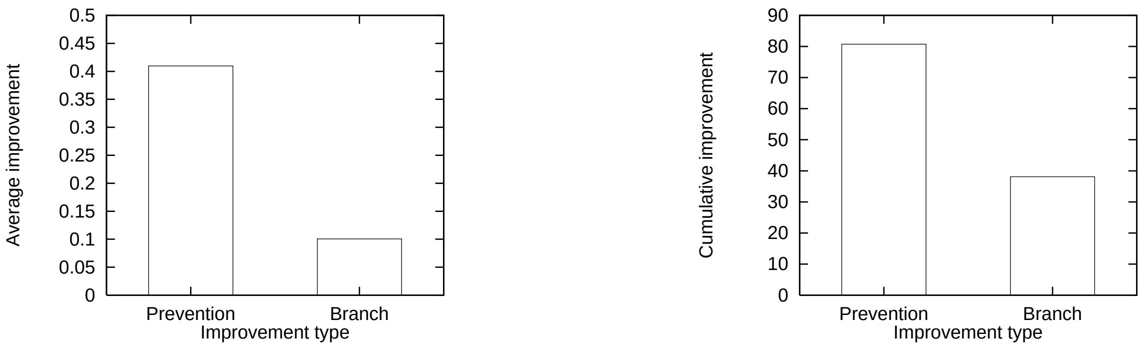

FIGURE 7.7: The graph on the left shows the average improvement gained from each way to improve a plan across the random set of problems. The graph on the right shows their cumulative contribution to success probability. Although in this domain each conditional branch is generally only 1/4 as effective as adding preventing steps, its overall contribution is a higher proportion because it is used more often.

Ablation study

Figure 7.7 shows how much improvement in probability was actually derived during problem solving instances from each of the different ways to improve probability. However, it is frequently the case that while one kind of improvement is used by the system, another could have been used, so a more complete picture of the relative utility of the types of improvement can be gained by selectively disabling each of them and running the same set of examples without it. The results of such a study are shown in Figure 7.8. When Weaver cannot create conditional branches, it reaches an average probability of success of roughly 0.6 after 3 iterations and stays constant on further iterations. When it cannot add steps to prevent an error from taking place, it reaches an average probability just below 0.7. Using both techniques it reaches an average of 0.77 in four iterations and 0.79 in eight iterations.

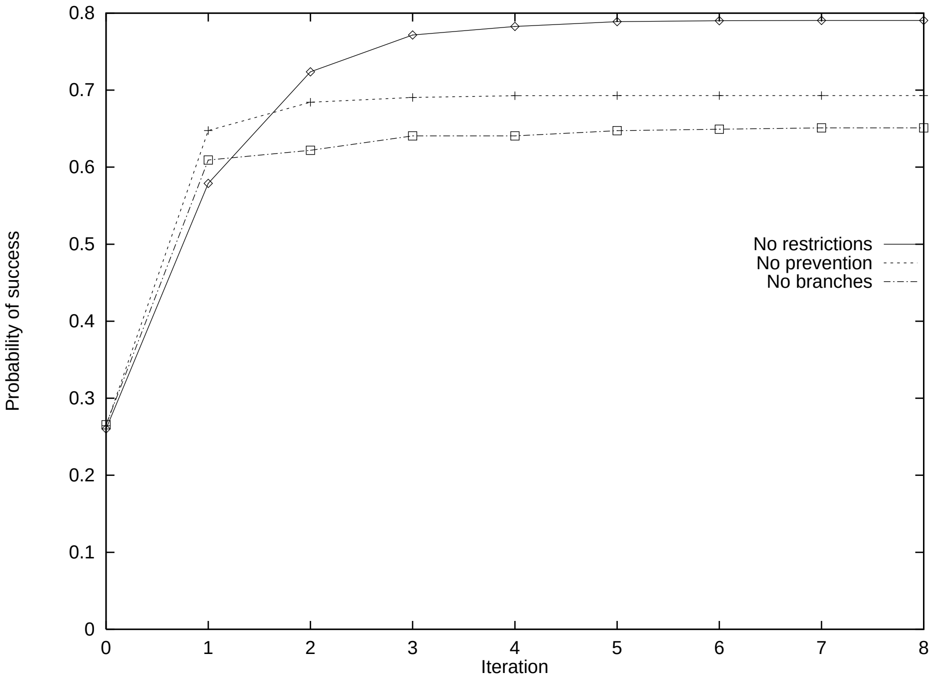

FIGURE 7.8: Average probabilities of plans for the same example set, plotted against the number of iterations, when Weaver's mechanisms for improving plan probability are selectively disabled.

Easy and hard cases for Weaver

Figure 7.9 shows a histogram of the probabilities of the initial plans found by Weaver in the example test sets. In approximately half the examples, Weaver's initial plan has probability below 0.1, and the rest are clustered around a probability of 0.25 with a separate peak at 0.9 - 1.0. To examine whether the cases with low initial probability are rectified quickly or represent examples that have low-probability final plans, Figure 7.3 shows the probabilities plotted against iterations in Weaver for these two cases, with the mean for the whole set plotted as a reference. Although they converge, the initial low probability is not recovered over four iterations of Weaver.

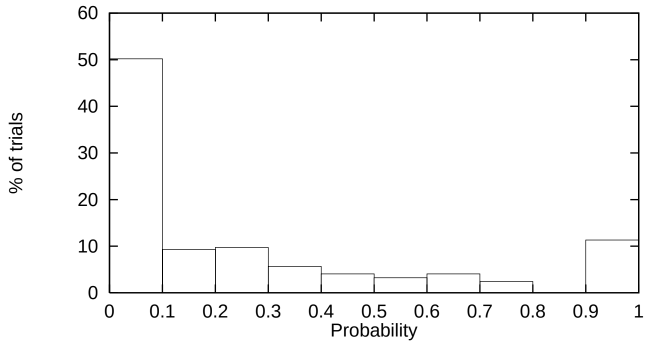

FIGURE 7.9: Histogram of the percentage of trials whose initial probability of success fell in each of the ten probability ranges 0.1 in width. Roughly half the initial cases have probability between 0 and 0.1

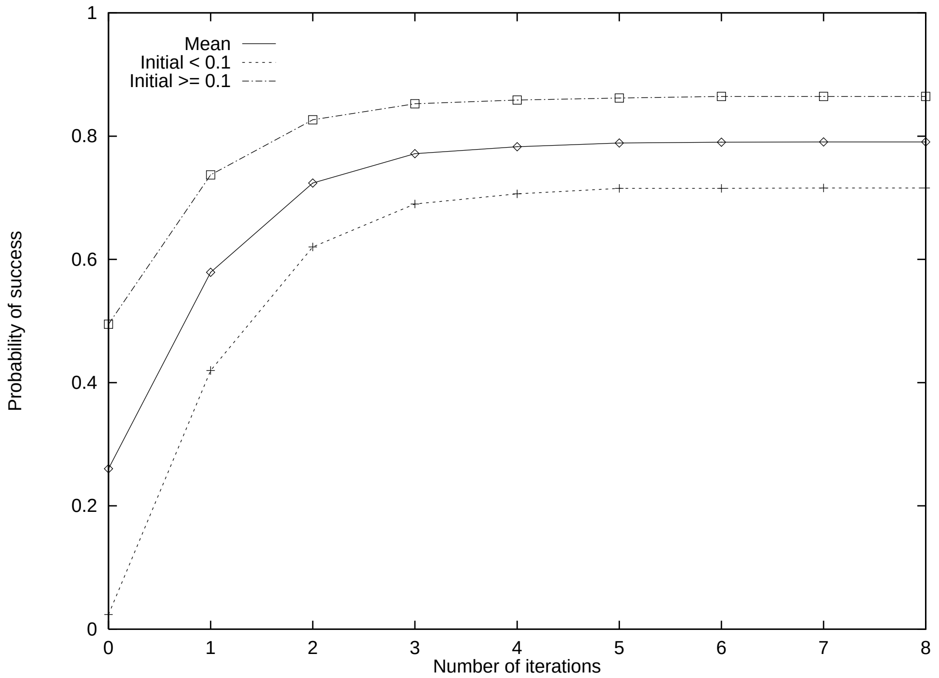

FIGURE 7.10: The lower line shows the mean probability of success plotted against the iteration of Weaver for the cases whose initial probability was between 0 and 0.1. The upper line shows these values for the other cases and the middle line shows the global mean.

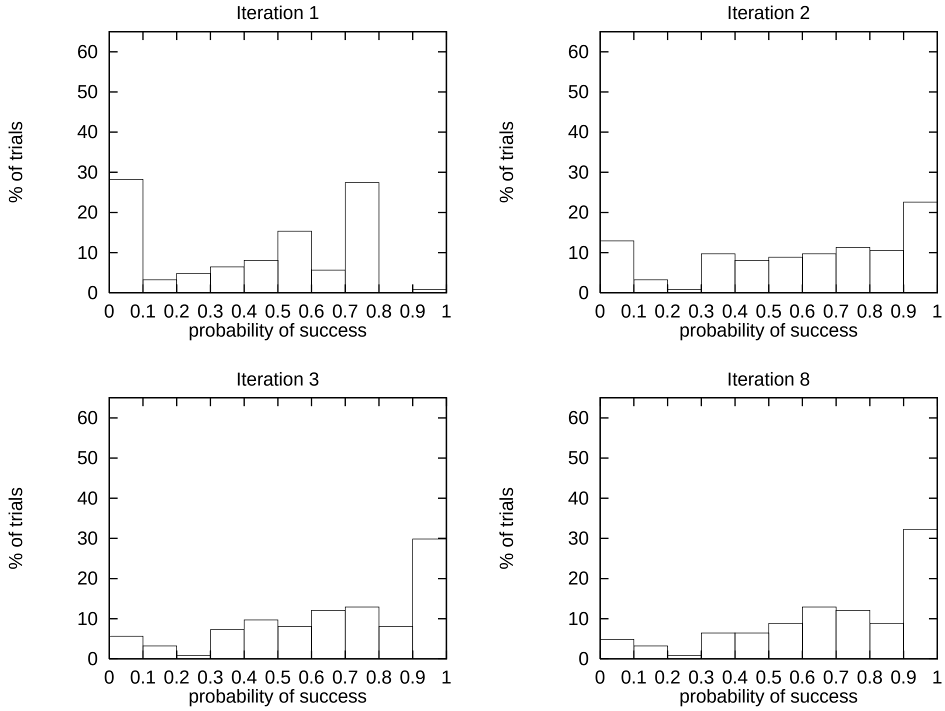

FIGURE 7.11: Evolution of probabilities for the initially low probability cases

7.4 Mutual constraints between the planner, evaluator and critic

In Section 4.4 I mentioned some of the advantages of using the emerging plan to constrain the probabilistic model of the plan and using the model to incrementally increase the amount of domain uncertainty that the planner is aware of. One advantage, demonstrated with an artificial domain, is that the plan can be used to constrain the set of exogenous events that need to be considered to evaluate the plan. This can greatly affect the cost of evaluating the plan, since the size of the joint probability distribution calculated can be exponential in the number of exogenous events. In addition if the number of sources of uncertainty considered by the planner is not constrained, the number of separate cases to be considered by the planner can also grow exponentially. In Section 7.4.1 I show that while it is important to constrain the events considered in the oil-spill domain, it can effectively be done with a domain-dependent algorithm that considers the problem and not the candidate plan.

A second advantage of the mutual constraints between the planner, the plan evaluator and the critic discussed in Section 4.4 is that the effects of events do not need to be modelled as alternative outcomes for every operator during which they can occur, but only where they affect the plan. This observation is used in the event graph technique developed in Chapter 5 to compress the belief net that is built to evaluate the plan. In Section 7.4.2 I show that this technique is useful in the oil-spill domain.

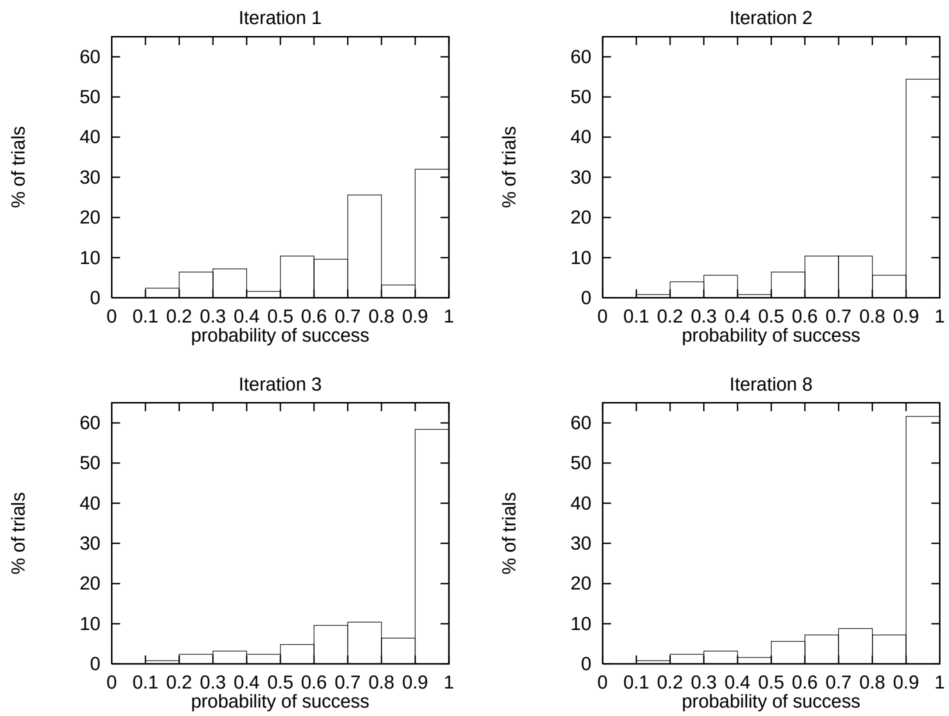

FIGURE 7.12: Evolution of probabilities for the initially high probability cases

7.4.1 Using the plan to constrain events in the evaluator

I focus on the exogenous events of the form `sea-gets-worse` and `sea-gets-better`, which have a numerical sea state and a sea sector as parameters and alter the sea state of the given sector by one. The event `sea-gets-better` is shown in Figure 7.13, and both can be found in the domain specification in Appendix B.

The random problems used in this chapter contain twenty-five sea sectors, each of which takes on one of six sea states independently of the other sea sectors. Since they are independent, they do not automatically lead to an exponential increase in the expense of evaluating the plan. However a planner explicitly considering each outcome of some action due to the full set of events would need to consider $6^{25} \approx 3 \times 10^{19}$ states due to the sea-change events alone.

The number of events considered can be considerably reduced by inspecting a particular planning problem. In the oil-spill domain, only activities concerned with skimming or pumping oil and laying booms depend on the sea state. This means that the only sectors of interest are where the spill takes place and the sensitive areas potentially threatened by the spilled oil. Although there are sixty-nine sensitive areas in the domain, only a maximum of three are reachable by oil from any one spill site, and the three are always in the same sea sector. This means that at most two sea sectors need to be considered to evaluate any plan for any of the test problems. While this method of reducing the sectors considered is very effective, it should be noted that it is domain-dependent, but using the plan to constrain the events is domain-

```
(event sea-gets-worse
 (params <sea-sector> <old-sea-state> <new-sea-state>)
 (duration 1)
 (probability 0.1)
 (preconds
  ((<sea-sector>     Sea-Sector)
   (<old-sea-state>  Sea-State)
   (<new-sea-state>  (and Sea-State
                          (add1 <old-sea-state> <new-sea-state>)
                          ;; stop an infinite markov model..
                          (< <new-sea-state> 7))))
  (sea-state <sea-sector> <old-sea-state>))
 (effects ()
  ((del (sea-state <sea-sector> <old-sea-state>))
   (add (sea-state <sea-sector> <new-sea-state>)))))
```

FIGURE 7.13: The event `sea-gets-worse`.

independent. Even in this domain, the domain-dependent method would be weaker than the plan-based method if more sensitive areas of coastline were reachable, and there is no guarantee that equivalent methods can be found for other domains of interest.

7.4.2 The event graph

To test the improvements in the speed of evaluating a plan that come from using the event graph, the initial plans from the same set of random examples were evaluated both with Markov chains built using the event graph technique described in Chapter 5, and directly representing individual event occurrences. The complexity of evaluating the belief net depends mainly on the number of steps in the plan and the number of precondition nodes required, but it also depends on the number and length of the *persistence intervals* in the plan. These are the time intervals between the achievement of a precondition literal and its use during which exogenous events may take place that affect its value.

The length in time units of the simulated plan is an indication of the length of the persistence intervals within it. For each step in the plan a duration can be calculated from its bindings, and this is used when steps are placed on a timeline to create the belief net as described in Section 4.2. The sum of these is the plan duration, since Weaver does not apply any of the actions in parallel. Plans with high duration in the oil-spill domain are typically caused by a small number of steps that move equipment over long distances. Therefore plans with high duration may not have more steps than shorter plans, but will include many more exogenous events. If the events are modelled explicitly, the corresponding nodes will quickly dominate the time to create and evaluate the belief net. If the events are modelled with a Markov chain, the plan duration must be much higher for the event links to dominate.

Figure 7.14 shows the times to create the Jensen join tree for the 67 plans from the set of problems used in experiments in the previous section that have a plan duration of 12 hours or less. The graph on the left uses explicit nodes for event occurrences while the graph on the right uses Markov chains. There is a clear relationship between the length of the plan and the time required to create the join tree when event nodes are modelled explicitly. There is no clear relationship when Markov chains are used. Note also that the maximum time using Markov chains is 3 seconds, compared with 500 seconds for the original method.

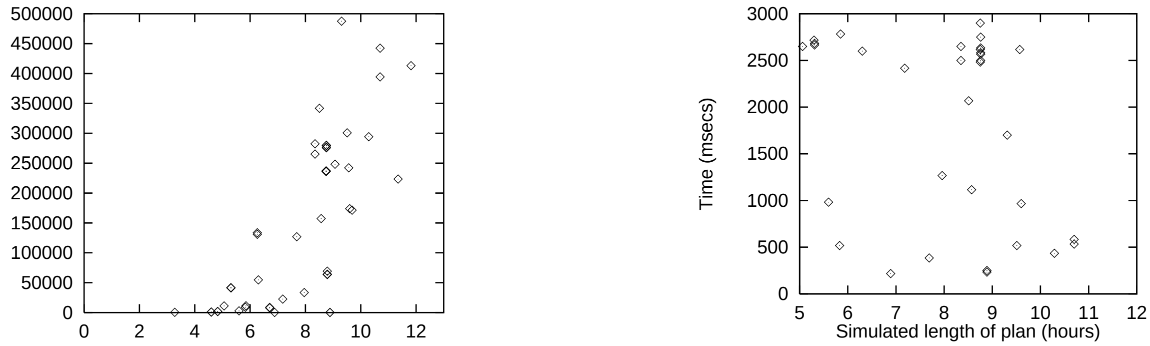

FIGURE 7.14: The times taken to create the Jensen join tree for the initial plans with simulated duration less than 12 hours. On the left, explicit event nodes were used and on the right, Markov chains built using the event graph. Times using the event graph are on average less than one percent of times taken with explicit event nodes.

As Figure 7.15 shows, there is a mild increase in times to create the Jensen join tree for a plan using Markov chains that is apparent when all the plans whose simulated length is below 1,000 hours are considered. Four outliers are removed from the figure, two of which take approximately 300 seconds while the other two have simulated lengths above 1,000 hours but take less than 5 seconds.

7.5 Domain-independent heuristics in the oil-spill domain

In Chapter 6 I mention some heuristics that can be used in the oil-spill domain based on local approximations to maximise the probability of success. They are implemented by means of a number of control rules, which are preference control rules that lead the planner to try preferred alternatives before the others, but do not prune the search space. Therefore the planner can find the same solutions with or without these control rules. They only affect the speed with which the planner reaches a good solution.

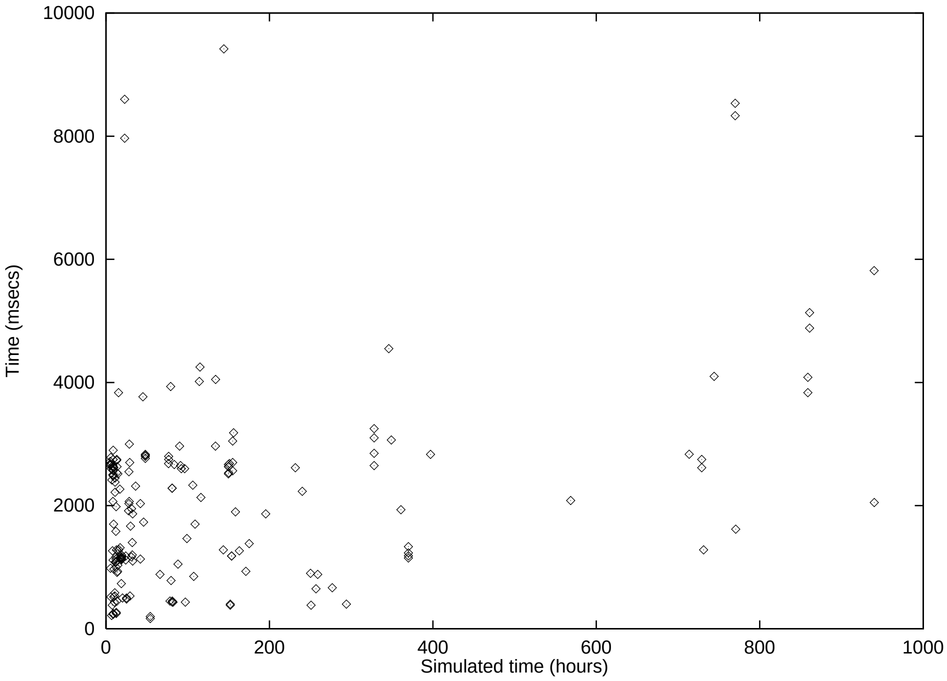

FIGURE 7.15: The times taken to create the Jensen join tree using Markov chains for all initial plans with simulated duration less than 1,000 hours.

The control rules can be grouped into two categories: those that prefer closer things and those that prefer bigger things. Four control rules select operator bindings that choose closer objects — closer equipment, a closer transport boat or a closer port if the leaking vessel is to be towed to safety. These rules typically lead to plans that do not necessarily have fewer steps, but which have a shorter duration in terms of the sum of the durations of their steps. Four control rules choose larger pieces of equipment — a unit of boom which is long enough to surround the vessel or protect the sensitive area and a skimmer or pump that has enough capacity to negate the flow. One control rule is in both categories, maximising the ratio of boom size to its distance from the target area.

As Figure 7.16 shows, the control rules lead to a significant improvement in the average probabilities of plans found in eight iterations of improvement. The figure shows the mean probabilities for 50 random problems, a subset of the problems used in Section 7.3. Using the control rules, the average probability reaches 0.79, and it reaches 0.61 without them.

The fact that both probability curves appear to have become flat indicates the importance of the decisions made early by the control rules. Since the planner has the same search space with and without the control rules, they will ultimately converge to the same values. However the trials without using the control rules show no indication in eight iterations of backtracking over the choices of equipment that lead to the initially poorer plans. Improving Weaver's ability to backtrack over these

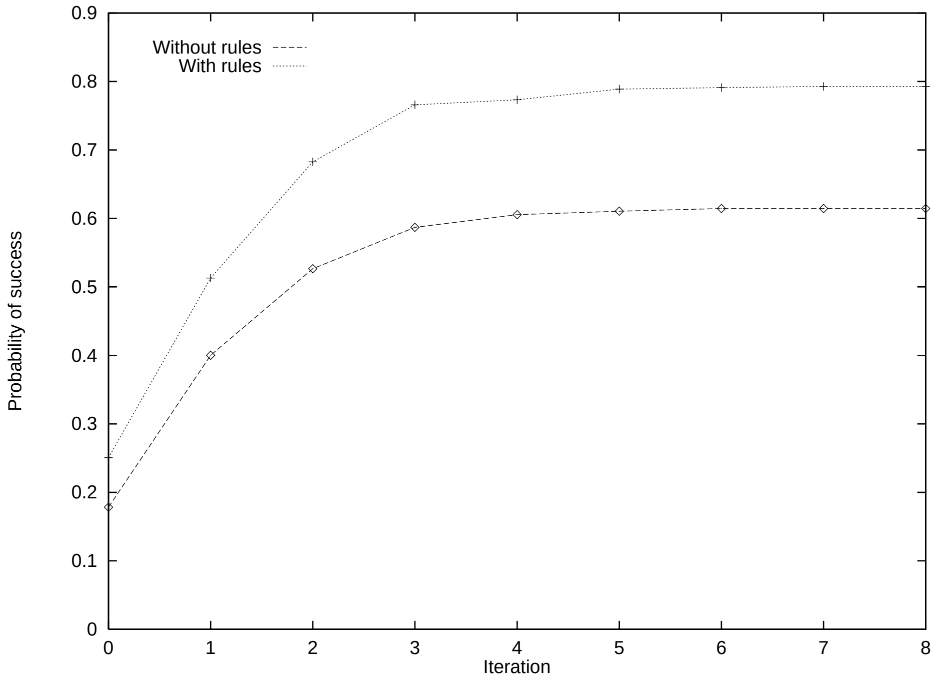

FIGURE 7.16: With and without greedy control rules.

choices is an interesting area for further research.

7.6 Analogical replay

In Chapter 6 I described the use of analogical replay to share planning effort between branches of a conditional plan. This section demonstrates the effectiveness of analogical replay in the oil-spill domain. As in the previous sections, I begin with some case studies that illustrate the technique in this domain, and then describe average performance improvements over a sample population of problems drawn from the domain.

7.6.1 Case studies

Analogical replay allows Weaver to share the computational resources required to create a plan between different branches of a conditional plan. The technique is useful whenever a segment of a larger plan is repeated in different conditional branches but the segment was not created before the branching action was applied, as shown in Chapter 6.

Replay within one clean-up operation.

Repeating segments of a plan after a conditional branch can frequently occur in the clean-up operations for a single spill. The first example used in this chapter can illustrate this if the top-level goals are treated by the planner in the reverse order and a conditional branch is again made on the predicate `threatened-shoreline`. In the original example, shown in Table 7.2, some work is initially done to achieve the goal `(no-discharge ss-weany spill golden-gate)` and after it is completed it is uncertain which of the two shorelines `pt-bonita-cove` or `rodeo-lagoon` is threatened because of the external event `threatened-shore-changes` shown in Figure 7.3. In that case the resulting plan after one conditional branch is added (Table 7.4) is:

```
(move-response-equipment-by-sea1b cargo-pump2 golden-gate richmond-port
                                  utility-boat-2)
(move-to-sea-sector-from-port tank-barge2 richmond-port golden-gate)
(cargo-transfer-oil-to-stabilize ss-weany spill golden-gate
                                 tank-barge2 cargo-pump2 600 120)
IF (threatened-shoreline (spill)) is in (pt-bonita-cove)
   (move-heavy-equip-by-ground2 tractor-2 oakland pt-bonita-cove)
   (move-heavy-equip-by-ground2 vac-truck2 richmond-port pt-bonita-cove)
   (build-berms-and-dams spill pt-bonita-cove)
ELSE
   (move-heavy-equip-by-ground2 tractor-2 oakland rodeo-lagoon)
   (move-heavy-equip-by-ground2 vac-truck2 richmond-port rodeo-lagoon)
   (build-berms-and-dams spill rodeo-lagoon)
```

This plan has no opportunities for analogical replay because each branch contains unique steps. If the planner works on the goals in the opposite order, its first plan looks like this:

```
(move-heavy-equip-by-ground2 tractor-2 oakland pt-bonita-cove)
(move-heavy-equip-by-ground2 vac-truck2 richmond-port pt-bonita-cove)
(build-berms-and-dams spill pt-bonita-cove)
(move-response-equipment-by-sea1b cargo-pump2 golden-gate richmond-port
                                  utility-boat-2)
(move-to-sea-sector-from-port tank-barge2 richmond-port golden-gate)
(cargo-transfer-oil-to-stabilize ss-weany spill golden-gate
                                 tank-barge2 cargo-pump2 600 120)
```

This plan can still be defeated by the event `threatened-shore-changes` which can take place during the two applications of the step `move-heavy-equip-by-ground2`, and after Weaver adds a conditional step to account for this case its plan has the following form:

```
(move-heavy-equip-by-ground2 tractor-2 oakland pt-bonita-cove)
(move-heavy-equip-by-ground2 vac-truck2 richmond-port pt-bonita-cove)
(build-berms-and-dams spill pt-bonita-cove)
IF (threatened-shoreline (spill)) is in (pt-bonita-cove)
   (move-response-equipment-by-sea1b cargo-pump2 golden-gate richmond-port
                                      utility-boat-2)
   (move-to-sea-sector-from-port tank-barge2 richmond-port golden-gate)
   (cargo-transfer-oil-to-stabilize ss-weany spill golden-gate
                                     tank-barge2 cargo-pump2 600 120)
ELSE
   (move-heavy-equip-by-ground2 tractor-2 pt-bonita-cove rodeo-lagoon)
   (move-heavy-equip-by-ground2 vac-truck2-richmond-1000
                                pt-bonita-cove rodeo-lagoon)
   (build-berms-and-dams spill rodeo-lagoon)
   (move-response-equipment-by-sea1b cargo-pump2 golden-gate richmond-port
                                      utility-boat-2)
   (move-to-sea-sector-from-port tank-barge2 richmond-port golden-gate)
   (cargo-transfer-oil-to-stabilize ss-weany spill golden-gate
                                     tank-barge2 cargo-pump2 600 120)
```

In this plan, the steps taken to achieve `(no-discharge ss-weany spill golden-gate)` appear in both branches, and are found independently unless analogical replay is used. In this instance, of 169 search nodes that are present in the final plan, 71 are derived from analogical replay, or about 42%. The total node number is high because a large number of inference rules are used in the plan, which have not been shown here.

7.6.2 Experiments in the oil-spill domain

On 20 randomly generated examples, the average proportion of nodes replayed was 0.53. 7 of the trials formed a conditional branch, these had an average replay proportion of 0.36. 13 of the trials formed a protection, and these had an average replay of 0.64. This set of trials is shown in Table 7.7.

Problem	Nodes	Replayed	Proportion	Type
1	53	35	0.66	protection
2	62	43	0.69	protection
3	48	32	0.67	protection
4	52	35	0.67	protection
5	51	35	0.69	protection
6	40	11	0.27	context
7	40	11	0.27	context
8	35	11	0.31	context
9	23	5	0.22	context
10	75	32	0.43	context
11	44	29	0.66	protection
12	76	32	0.42	context
13	72	29	0.4	context
14	50	33	0.66	protection
15	42	26	0.62	protection
16	48	29	0.6	protection
17	48	29	0.6	protection
18	46	29	0.63	protection
19	42	25	0.59	protection
20	48	29	0.6	protection

TABLE 7.7: Proportion of nodes replayed by derivational analogy in randomly generated problems.

Chapter 8

Conclusions

This thesis presents a novel planner that can build plans to meet a threshold probability of success given a representation of uncertainty in the form of probabilistic action outcomes, a probability distribution over possible initial states and probabilistic information about possible exogenous events that can affect the planning domain. The representation of uncertainty was defined in terms of a Markov decision process model.

The planning algorithm harnesses a novel conditional planner that extends PRODIGY 4.0, and as such is able to make use of control rules, machine learning techniques and a graphical user interface [Veloso *et al.* 1995]. The conditional planner attempts to create a plan that is certain to succeed given a set of non-deterministic actions. A plan critic controls the planner by removing many of the sources of uncertainty from the planner's version of the problem domain, and incrementally adding back sources to improve the probability of success of the final plan. This system has been tested in some synthetic domains and also in a large planning domain for cleaning up oil spilled from a tanker near the coast of California.

8.1 Contributions of the thesis

The main contributions of the thesis are:

- A representation and approach for planning under uncertainty with exogenous events. This is a novel problem area for an AI planner.

- A novel, hybrid representation for the computing the plan's probability using belief nets and Markov chains; a way to automatically decompose the plan into component Markov chains and recombine them using the net.

- Domain-independent heuristics for planning under uncertainty in order to apply techniques for planning under uncertainty to large, realistic planning problems.

- The application of derivational analogy to planning under uncertainty, as a demonstration of how existing machine learning techniques can be used to improve the performance of a probabilistic planner.

- Scaling a system for planning under uncertainty to a real-world domain. The oil-spill domain has more than 50 operators, a large state space and many sources of uncertainty. Typical plans range in length from 30 to 50 steps and may have 3 or more conditional branches.

8.2 Future research directions

The thesis has provided a solid foundation for an exciting and important topic for planning systems. While it gives a promising demonstration of planning under uncertainty in large domains, it raises a number of questions and leaves room for further work in many areas. Some of these are listed below.

- Tighter integration of plan creation and evaluation, and an evaluation of the spectrum of approaches from complete to very loose integration. Some of the power of the system can be claimed to come from judicious use of plan evaluation. However the delayed evaluation sometimes causes the planner to spend precious computational resources in parts of the search space which might quickly be seen to yield only low-probability solutions if an evaluation was made earlier in the process.

- Work on meta-reasoning in the context of planning under uncertainty. An explicit consideration of the tradeoffs between time spent planning and time spent evaluating the plan will allow exploration of ways to reason about how to allot the available computation most effectively between these processes. For example, an upper bound for the contribution of a conditional branch to the total probability of success is the probability that the ranch is reached. If this is determined to be small, the computation of the exact probability could be passed over in favour of more planning work to improve the probability of succes on more likely branches. A decision-theoretic framework for making such decisions such as developed in [Russell & Wefald 1991] and [Zilberstein 1993] could be used.

- More research is needed in machine learning and planning, using derivational analogy and other forms of machine learning such as learning search control rules from experience. Some preliminary work has shown the potential to compile experience gained from the probabilistic evaluation of plans into search control for the planner although it largely avoids probabilistic evaluations [Blythe & Veloso 1996]. This approach may be one way to improve the integration of the plan creation and plan evaluations modules.

- Although this thesis concentrated on a planner based on Prodigy 4.0, it is in principle quite possible to apply the same ideas to other planning algorithms. Despite promising work in planning under uncertainty with partial-order and hierarchical task network planners [Draper, Hanks, & Weld 1994; Haddawy, Doan, & Goodwin 1995], exogenous events have not yet been addressed in these systems. Experience with Weaver suggests that a planner based on iterative repair principles such as [Ambite & Knoblock 1997; Kautz & Selman 1996] could be very appropriate for probabilistic planning.

- Weaver computes a lower bound on the probability of success of a plan, improving its speed by (1) ignoring fortuitous exogenous events that are not explicitly used by the conditional planner and (2) assuming that a plan fails if any component step fails to execute. However it still computes an exact probability for the belief net that it generates. Since the planning task is to pass a threshold probability, and since only comparative values are needed for Weaver to choose between alternative plan improvements, algorithms that compute ranges of probabilities from the belief net can lead to significant performance improvements.

Appendix A

Proofs of theorems

A.1 The plan's belief net provides a lower bound on its probability of success.

Theorem: *Let Π be a solution to a planning problem for which a belief net B is constructed by Weaver. If B predicts a probability of success p, then the true probability of success as given by the interpretation of Π as a policy on the underlying Markov decision process is at least p.*

I show the result for non-branching plans Π, since the predicted probability of success for a branching plan is the sum of such belief nets.

I will show that for each complete assignment to the variables of the belief net B that assigns the value `true` to every action node in the plan, there is a set of paths through the MDP that agree with the assignment to B on the value of all the fluent, action and event nodes and whose combined conditional probability mass, conditioned on the initial state distribution, is at least that of the assignment to B. This is sufficient to prove the result, since the predicted probability of success from B is the sum of such complete assignments.

Let Σ be such an assignment, *i.e.* a mapping from the nodes of B to values such that $\Sigma(a) =$ `true` for each action node a in B. Note that since the `finish` action node has the value `true`, this assignment contributes to the belief in plan success. For the assignment to have probability greater than zero, the preconditions of all actions, and of all events made true by the assignment, must be satisfied in the assignment.

I will show that there is a set of paths in the MDP that match the assignment and have the required conditional probability mass by induction over the length of the plan. Let $F(i)$ denote the set of fluent nodes with time stamp i in B, and let $M(i)$ denote the set of paths through the MDP going through states that are consistent with the values of the fluents in $F(j)$ for all $j \leq i$ as well as with the actions and the events e in B with $\Sigma(e) =$ `true` whose time stamps are less than (but not equal t) i. The inductive strategy is to show that at each time point i, the probability mass of the paths in $M(i)$ is at least that of the restriction of the assignment Σ to nodes with time stamps before or equal to i, which I shall denote $\Sigma|_i$.

Base case: $M(0)$ consists of states in the MDP consistent with the assignment to $F(0)$. By the construction of the belief net B, the probability of the assignment to $F(0)$, $\Sigma|_0$, is equal to the probability of $F(0)$ under the initial state distribution. This is trivially equal to the conditional probability of the matching states in the MDP given the initial state distribution.

Inductive step: Assuming that the conditional probability mass of $M(j)$ is at least the probability that B gives to the assignment $\Sigma|_j$ for all $j \leq i$, we need to show the same for $M(i+1)$ and $\Sigma|_{i+1}$.

Consider any event or action that takes place at time i and according to Σ has some outcome o. Its preconditions appear in $F(i)$ and so are satisfied in the final state of any path in $M(i)$. Therefore the probability of a transition from any such state that adds pending effects corresponding to the correct outcome is the probability of the event taking place multiplied by the probability of the outcome. This is by construction the same as the conditional probability that the event or action node has the outcome o in the belief net B given the values in $F(i)$. Since the events and actions that take place at time i do so with independent probabilities given $F(i)$, the probability mass of all paths that satisfy all these events and actions is above the required value by the inductive hypothesis.

Finally, if a path extends a path from $M(i)$ to match the required events and actions from $\Sigma|_{i+1}$, then it will also match the fluent nodes $F(i+1)$. This is because the fluent node values are determined by the pending effects from these events and actions or earlier ones in $M(i)$. If some other action or event could alter one of these values, it would have corresponding nodes in the belief net.

This argument establishes the required inequality. It does not guarantee equality because the belief net construction algorithm does not search for events that may cause fluent nodes to have their required values for the plan to succeed, only those that could cause other values. Therefore their may be paths in the MDP that reach a goal state even though not all the steps in the plan succeed.

A.2 Event graphs

Theorem: *All queries about the probability of a logical expression over a set of literals $V \subset L$, after some fixed number of transitions n given an initial state probability distribution over the full state space Ω will have the same value in the submodel induced by V as in the full model of the domain.*

It is sufficient to show that probabilities of all conjunctions involving each literal in V or its negation will be the same in each model, since any expression can be expressed as the disjunction of such expressions, whose probabilities can be summed since they are mutually exclusive. I first show that a chain M_r built of the subgraph of the event graph containing the variables in V and their ancestors in the event graph will compute the correct value, and then showing that the same value can be computed by creating independent chains for the separate components of the subgraph.

Each full conjunctive expression corresponds to a unique state of M_r. For each such state i, let $F(i)$ denote the set of states in the full model M_f that agree with i on the literals in V. Let $p^n_{r,(i,j)}$ be the probability that the reduced chain M_r is in state j after n steps given that it began in state i, and let $p^n_{f,(x,y)}$ be the probability that the full model M_f is in state y after n steps given that it began in state x.

A situation in which M_r is in state i may correspond to any probability distribution $P(.)$ over states in $F(i)$ for M_f, where $\sum_{x \in F(i)} P(x) = 1$. Thus given states i and j in M_r, we need to show that, for all n and for all $P(.)$,

$$p^n_{r,(i,j)} = \sum_{x \in F(i)} P(x) \sum_{y \in F(j)} p^n_{f,(x,y)} \tag{A.1}$$

By choosing different probability distributions it can be seen that this is equivalent to

$$p^n_{r,(i,j)} = \sum_{y \in F(j)} p^n_{f,(x,y)} \qquad\qquad \forall x \in F(i) \tag{A.2}$$

since we can choose $P(x) = 1$ for any $x \in F(i)$. Conversely if equation (2) is true, any linear combination given by a probability distribution will also satisfy the equality, so equation (1) is true also.

We use induction on n. The base case to prove is for $n = 1$:

Let E_r be the set of events in the subgraph of the event graph used to build M_r, and let $p(e, i, j)$ be the probability of the effect in $\text{eff}(e, i)$ that changes the literals affected by e to match j, if such an effect exists, and let $p(e, i, j) = 0$ otherwise. By definition, $p^1_{r,(i,j)} = \prod_{e \in E_r} p(e, i, j)$

Similary for any $x \in F(i)$ and for any $y \in M_f$, $p^1_{f,(x,y)} = \prod_{e \in E} p(e, x, y)$. We can arrange this product in terms of events in E_r and the rest:

$$p^1_{f,(x,y)} = \prod_{e \in E_r} p(e, x, y) \times \prod_{e' \in E - E_r} p(e', x, y)$$

Now the events in E_r only effect the literals in M_r and are completely determined by them, by the construction of the event subgraph. So if $y \in F(j)$, $p(e, x, y) = p(e, i, j)$ and the first term is just $p^1_{r,(i,j)}$. So we have

$$\sum_{y \in F(j)} p^1_{f,(x,y)} = p^1_{r,(i,j)} \sum_{y \in F(j)} \prod_{e' \in E - E_r} p(e', x, y) \forall x \in F(i)$$

Also by the construction of the event subgraph, the events in $E - E_r$ do not alter any of the literals in M_r. So for $x \in F(i)$, if we fix the outcomes of events in E_r to lead to j, all transitions with non-zero probability in M_f must lead to a state in $F(j)$. Then under these circumstances $\sum_{y \in F(j)} \prod_{e' \in E - E_r} = 1$, and the induction base case is proved.

Now suppose the result is true for $n \leq m-1$, so that for any $x \in F(i)$ and $z' \in F(k)$,

$$\begin{aligned} p^m_{r,(i,j)} &= \sum_{k \in M_r} p^{m-1}_{r,(i,k)} p^1_{r,(k,j)} \\ &= \sum_{k \in M_r} \Big(\sum_{z \in F(k)} p^{m-1}_{f,(x,z)} \Big) \Big(\sum_{y \in F(j)} p^1_{f,(z',y)} \Big) \end{aligned}$$

We can re-arrange the summands on the right hand side, and choose $z' = z$ separately for each $z \in F(k)$ to get

$$p^m_{r,(i,j)} = \sum_{k \in M_r} \sum_{z \in F(k)} \sum_{y \in F(j)} p^{m-1}_{f,(x,z)} p^1_{f,(z,y)}$$

since each state in M_f is in $F(k)$ for some $k \in M_r$,

$$\begin{aligned} p^m_{r,(i,j)} &= \sum_{z \in M_f} \sum_{y \in F(j)} p^{m-1}_{f,(x,z)} p^1_{f,(z,y)} \\ &= \sum_{y \in F(j)} \sum_{z \in M_f} p^{m-1}_{f,(x,z)} p^1_{f,(z,y)} \\ &= \sum_{y \in F(j)} p^m_{f,(x,y)} \end{aligned}$$

which is the desired result.

For the second stage of the proof we need to show that separate Markov chains derived from the components of the event graph's subgraph and treated independently will yield the same result as a calculation in M_r. The proof uses the same technique as in the first stage, noting that the events in each component effect different literals and are determined by different literals from all the other components, so the transition probabilities can be split in the same way.

Appendix B

The Oil-spill domain

This appendix contains the Weaver encoding of the oil-spill domain and a specification of the algorithm used to generate random examples for Chapter 7. The domain encoding includes operators, inference rules, external events, control rules and bindings functions. The original encoding of this domain in SIPE can be found in [Desimone & Agosta 1994]. In order to keep the specification in this document manageable, several parts have been omitted or contracted. Many simple operators have no preconditions. These have been listed simply by name and effects along with a comment. The specification of the domain problem scenario provided by SRI and used as the basis of the random problem generator, containing over 1500 predicates, has been ommitted. For full details, contact the author by email at *jblythe@cs.cmu.edu*.

B.1 Domain specification

```
(infinite-type numerical #'numberp)

;;; All the names are read in as strings in the object fields
(infinite-type name #'stringp)

;;; Level 1

(inference-rule respond-to-spill-coastal
 (params <spilled-oil> <sea-sector> <vessel>)
 (preconds
  ((<spilled-oil> Spilled-oil)
   (<sea-sector> Sea-sector)
   (<vessel> Vessel)
    )
  (and (no-discharge <vessel> <spilled-oil> <sea-sector>)
       (amount-spilled <spilled-oil> <sea-sector> 0)
       (~ (unprotected-sensitive-area <spilled-oil>))
       ))
 (effects ()
```

```
  ((add (respond-to-spill <spilled-oil> <sea-sector> <vessel>)))))

(inference-rule clean-up-spill
 (params <spilled-oil> <sea-sector> <steps>)
 (preconds
  ((<sea-sector> Sea-Sector)
   (<spilled-oil> Spilled-oil)
   (<steps> (and Numerical (remove-oil-steps <steps>))))
  (remove-oil-steps <spilled-oil> <steps>))
 (effects ()
  ((add (amount-spilled <spilled-oil> <sea-sector> 0)))))

(inference-rule protect-sensitive-areas
 (params <spilled-oil>)
 (preconds
  ((<spilled-oil> Spilled-oil))
  (forall ((<sen> Sensitive-area))
  (or (~ (threatened-shoreline <sen> <spilled-oil>))
      (protect-shore-steps <spilled-oil> <sen> 1))))
 (effects ()
  ((del (unprotected-sensitive-area <spilled-oil>)))))

;;; Level 2

;;; Stop when the discharge-rate is below an acceptable minimum.

(inference-rule end-stabilize
 (params <vessel> <spilled-oil> <rate> <sea-sector>)
 (preconds
  ((<vessel> Vessel)
   (<spilled-oil> Spilled-oil)
   (<sea-sector> sea-sector)
   (<rate> (and Numerical
(gfp (discharge-rate <vessel> <spilled-oil> <rate>)))))
  (stabilize-discharge-rate <vessel> <spilled-oil> <sea-sector> <rate>))
 (effects ()
  ((add (no-discharge <vessel> <spilled-oil> <sea-sector>)))))

(inference-rule begin-stabilize ; no preconditions
  (preconds ((<rate> (and Numerical (acceptable-discharge-rate <rate>)))))
  ((add (stabilize-discharge-rate <vessel> <spilled-oil> <sea-sector>
  <rate>)))))

(operator cargo-transfer-oil-to-stabilize
 (params <vessel> <spilled-oil> <sea-sector>
 <tank-barge> <cargo-transfer-pump> <rate>
 <new-rate>)
 (preconds
  ((<vessel> Vessel)
   (<spilled-oil> Spilled-oil)
   (<sea-sector> Sea-sector)
   (<rate> (and Numerical (> <rate> 0)))
```

```
   (<cargo-transfer-pump> (and Cargo-transfer-pump
       (not (gfp (in-use <cargo-transfer-pump>)))))
   (<tank-barge> Tank-barge)
   (<pump-cap>
    (and Numerical
 (gfp (capacity-bbl-per-hr <cargo-transfer-pump> <pump-cap>))))
   (<barge-cap> (and Numerical
     (gfp (barge-capacity-bbl <tank-barge> <barge-cap>))))
   (<worst-sea> (and Numerical
     (gfp (max-sea-state <tank-barge> <worst-sea>))))
   (<new-rate> (and Numerical (sub <rate> <pump-cap> <new-rate>)))
   ;; Although not used, will stop the operator being considered if
   ;; the literal is not present.
   (<discharge-size>
    (and Numerical
 (gfp (discharge-size <vessel> <spilled-oil> <discharge-size>)))))
  (and (~ (in-use <cargo-transfer-pump>))
       (located <cargo-transfer-pump> <sea-sector>)
       (located <tank-barge> <sea-sector>)
       (sea-state-calmer <sea-sector> <worst-sea>)
       (stabilize-discharge-rate
<vessel> <spilled-oil> <sea-sector> <new-rate>)))
 (effects ()
  ((add (oil-pumped <cargo-transfer-pump> <spilled-oil> <vessel>
    <tank-barge>))
   (add (did-cargo-transfer-oil <vessel> <spilled-oil> <sea-sector>))
   (add (in-use <cargo-transfer-pump>))
   (add (stabilize-discharge-rate <vessel> <spilled-oil> <sea-sector>
  <rate>)))))


(operator stabilize-discharge-by-contain-skim
 (params <vessel> <spilled-oil> <sea-sector> <rate>)
 (preconds
  ((<vessel> Vessel)
   (<spilled-oil> Spilled-oil)
   (<sea-sector> Sea-sector)
   ;; the call to numberp forces <rate> to be bound.
   (<rate> (and Numerical (numberp <rate>) (> <rate> 0)))
   (<sea-state> (and Sea-state (gfp (sea-state <sea-sector> <sea-state>))))
   (<vessel-boom-length> (and Numerical
      (gfp (vessel-boom-length
    <vessel> <vessel-boom-length>)))))
  (and (boom-level>= <vessel> <vessel-boom-length> <sea-sector> <sea-state>)
       (portable-skim-level>= <vessel> <sea-sector> <sea-state> <rate>)
       ))
 (effects ()
  ((add (boom-assembled <vessel> <sea-sector>))
   (add (stabilize-discharge-rate <vessel> <spilled-oil> <sea-sector>
  <rate>)))))
```

```
(operator stabilize-discharge-by-towing-to-port
 (params <vessel> <spilled-oil> <sea-sector1> <sea-port>
 <tug> <rate>)
 (duration
  (let ((distance (or (known-unique '(distance <sea-port> <sea-sector1> <x>))
      100))
(speed (or (known-unique '(tow-speed-knots <tug> <x>)) 2)))
    (/ distance speed)))
 (preconds
  ((<vessel> Vessel)
   (<spilled-oil> Spilled-oil)
   (<sea-sector1> Sea-sector)
   (<rate> (and Numerical (> <rate> 0)))
   (<displacement> (and Numerical
(gfp (displacement <vessel> <displacement>))))
   (<sea-port> (and Seaport (berth-greater <sea-port> <displacement>)))
   (<tug> (and Tug (diff <tug> <vessel>))))
  (and (~ (in-use <tug>))
       (located <tug> <sea-sector1>)))
 (effects ()
  ((add (stabilize-discharge-rate <vessel> <spilled-oil> <sea-sector1>
  <rate>))
   (add (tanker-towed <vessel> <tug> <sea-sector1> <sea-port>)))))


;;; OIL CONTAINMENT, COUNTERMEASURES,  AND RECOVERY OPERATORS

(inference-rule begin-remove-oil ;; no preconditions
 (effects () ((add (remove-oil-steps <spilled-oil> 0)))))

(inference-rule open-water-recovery
 (params <spilled-oil> <sea-sector> )
 (preconds
  ((<spilled-oil> Spilled-oil)
   (<sea-sector> (and Sea-sector
      (get-covered-sector <sea-sector> <spilled-oil>)))
   (<new-steps> (and Numerical (> <new-steps> 0)))
   (<old-steps> (and Numerical (sub1 <new-steps> <old-steps>))))
  (and (remove-oil-steps <spilled-oil> <old-steps>)
       (recovery-open-water <spilled-oil> <sea-sector>)
       ))
 (effects () ((del (remove-oil-steps <spilled-oil> <old-steps>))
      (add (remove-oil-steps <spilled-oil> <new-steps>)))))

(operator perform-open-water-recovery
 (params <spilled-oil> <sea-sector> <discharge-rate>)
 (preconds
  ((<spilled-oil> Spilled-oil)
   (<sea-sector>
    (and Sea-sector (get-covered-sector <sea-sector> <spilled-oil>)))
   (<vessel> Vessel)
   (<discharge-rate>
```

```
    (and Numerical
 (gfp (discharge-rate <vessel> <spilled-oil> <discharge-rate>))))
   (<amount-spilled>
    (and Numerical
 (gfp (amount-spilled <spilled-oil> <sea-sector> <amount-spilled>)))))
  (and (mobile-skim-level>= <sea-sector> <discharge-rate>)
       (store-level>= <sea-sector> <amount-spilled>)))
 (effects ()
  ((add (recovery-open-water <spilled-oil> <sea-sector>))))
 (duration (/ <amount-spilled> <discharge-rate>)))

(inference-rule shallow-water-recovery
 (params <spilled-oil> <sea-sector>)
 (preconds
  ((<spilled-oil> Spilled-oil)
   (<sea-sector>
    (and Sea-sector (get-covered-sector <sea-sector> <spilled-oil>)))
   (<sea-state>
    (and Sea-state (gfp (sea-state <sea-sector> <sea-state>))))
   (<new-steps> (and Numerical (> <new-steps> 0)))
   (<old-steps> (and Numerical (sub1 <new-steps> <old-steps>))))
  (and (remove-oil-steps <spilled-oil> <old-steps>)
       (recovery-shallow-water <spilled-oil> <sea-sector>)))
 (effects ()
  ((del (remove-oil-steps <spilled-oil> <old-steps>))
   (add (remove-oil-steps <spilled-oil> <new-steps>)))))

(operator perform-shallow-water-recovery
 (params <spilled-oil> <sea-sector> )
 (preconds
  ((<spilled-oil> Spilled-oil)
   (<sea-sector>
    (and Sea-sector (get-covered-sector <sea-sector> <spilled-oil>)))
   (<sea-state> (and Sea-state (gfp (sea-state <sea-sector> <sea-state>))))
   (<boom-length>
    (and Numerical
 (gfp (boom-length <spilled-oil> <sea-sector> <boom-length>))))
   (<vessel> vessel)
   (<discharge-rate>
    (and Numerical
 (gfp (discharge-rate <vessel> <spilled-oil> <discharge-rate>))))
   (<amount-spilled>
    (and Numerical
 (gfp (amount-spilled <spilled-oil> <sea-sector> <amount-spilled>)))))
  (and (boom-level>= <sea-sector> <sea-state> <boom-length>)
       (portable-skim-level>= <sea-sector> <discharge-rate> <sea-state>)
       (store-level>= <sea-sector> <amount-spilled>)))
 (effects ()
  ((add (boom-assembled <boom-length> <sea-sector>))
   (add (recovery-shallow-water <spilled-oil> <sea-sector>)))))

(inference-rule apply-chemical-dispersant
```

```
 (params <spilled-oil> <sea-sector>)
 (preconds
  ((<spilled-oil> spilled-oil)
   (<sea-sector>
    (and sea-sector (get-covered-sector <sea-sector> <spilled-oil>)))
   (<new-steps> (and Numerical (> <new-steps> 0)))
   (<old-steps> (and Numerical (sub1 <new-steps> <old-steps>))))
  (and (remove-oil-steps <spilled-oil> <old-steps>)
       (apply-dispersant <spilled-oil> <sea-sector>)))
 (effects
  ()
  ((del (remove-oil-steps <spilled-oil> <old-steps>))
   (add (remove-oil-steps <spilled-oil> <new-steps>)))))

(operator get-chem-dispersant
 (params <dispersant> <spilled-oil> <sea-sector> )
 (duration 1)
 (preconds
  ((<spilled-oil> spilled-oil)
   (<sea-sector> sea-sector)
   (<dispersant> Dispersant))
  (and (~ (use-prohibited <dispersant>))
       (located <dispersant> <sea-sector>)))
 (effects
  ()
  ((add (apply-dispersant <spilled-oil> <sea-sector> )))))

(inference-rule in-situ-oil-burning
 (params <spilled-oil> <sea-sector> )
 (preconds
  ((<spilled-oil> spilled-oil)
   (<sea-sector>
    (and sea-sector (get-covered-sector <sea-sector> <spilled-oil>)))
   (<new-steps> (and Numerical (> <new-steps> 0)))
   (<old-steps> (and Numerical (sub1 <new-steps> <old-steps>))))
  (and (remove-oil-steps <spilled-oil> <old-steps>)
       (burn-oil-in-situ <spilled-oil> <sea-sector> )))
 (effects
  ()
  ((del (remove-oil-steps <spilled-oil> <old-steps>))
   (add (remove-oil-steps <spilled-oil> <new-steps>)))))

(operator perform-in-situ-burning  ;; no preconditions
 (duration 1)
 (effects () ((add (burn-oil-in-situ <spilled-oil> <sea-sector>)))))


;;; SHORE PROTECTION AND CLEAN UP OPERATORS

(inference-rule start-protecting-shore  ;; no preconditions
 (effects ((add (protect-shore-steps <spilled-oil> <sensitive-area> 0)))))
```

```
(operator shore-exclusion-booming
 (params <spilled-oil> <sensitive-area> )
 (preconds
  ((<spilled-oil> spilled-oil)
   (<sensitive-area> sensitive-area)
   (<land-sector>
    (and Land-sector (mygfp 'located-within <sensitive-area> <land-sector>)))
   (<sea-sector>
    (and Sea-sector (adjacent <land-sector> <sea-sector>)))
   (<exc-boom>
    (and Numerical
 (gfp (exclusion-boom-required <sensitive-area> <exc-boom>))))
   (<personnel>
    (and Numerical
 (gfp (boom-personnel-required <sensitive-area> <personnel>))))
   (<new-steps> (and Numerical (> <new-steps> 0)))
   (<old-steps> (and Numerical (sub1 <new-steps> <old-steps>))))
  (and (protect-shore-steps <spilled-oil> <sensitive-area> <old-steps>)
       (boom-level>= <sensitive-area> <exc-boom> <sea-sector>)))
 (effects ()
  ((del (protect-shore-steps <spilled-oil> <sensitive-area> <old-steps>))
   (add (protect-shore-steps <spilled-oil> <sensitive-area> <new-steps>))
   (add (boom-personnel-deployed <personnel> <sensitive-area>))
   (add (boom-assembled <exc-boom> <sensitive-area>))
   (add (barrier-provided <sensitive-area>)))))

(operator shore-diversion-booming
 (params <spilled-oil> <sensitive-area>)
 (preconds
  ((<spilled-oil> spilled-oil)
   (<sensitive-area> sensitive-area)
   (<land-sector>
    (and Land-sector (mygfp 'located-within <sensitive-area> <land-sector>)))
   (<sea-sector>
    (and Sea-sector (adjacent <land-sector> <sea-sector>)))
   (<amount-spilled>
    (and Numerical (gfp (amount-spilled <spilled-oil> <sea-sector>
 <amount-spilled>))))
   (<div-boom>
    (and Numerical
 (gfp (diversion-boom-required <sensitive-area> <div-boom>))))
   (<personnel>
    (and Numerical
 (gfp (boom-personnel-required <sensitive-area> <personnel>))))
   (<new-steps> (and Numerical (> <new-steps> 0)))
   (<old-steps> (and Numerical (sub1 <new-steps> <old-steps>))))
  (and (protect-shore-steps <spilled-oil> <sensitive-area> <old-steps>)
       (boom-level>= <sensitive-area> <div-boom>)
       (vacuum-level>= <sensitive-area> <amount-spilled>)))
 (effects ()
  ((del (protect-shore-steps <spilled-oil> <sensitive-area> <old-steps>))
   (add (protect-shore-steps <spilled-oil> <sensitive-area> <new-steps>))
```

```
   (add (boom-personnel-deployed <personnel> <sensitive-area>))
   (add (boom-assembled <div-boom> <sensitive-area>))
   (add (barrier-provided <sensitive-area>)))))

(operator berms-and-dams
 (params <spilled-oil> <sensitive-area> )
 (preconds
  ((<spilled-oil> spilled-oil)
   (<sensitive-area> sensitive-area)
   (<new-steps> (and Numerical (> <new-steps> 0)))
   (<old-steps> (and Numerical (sub1 <new-steps> <old-steps>)))
   (<personnel>
    (and Numerical (gfp (shore-personnel-required
 <sensitive-area> <personnel>))))
   (<land-sector> (and Land-sector
       (mygfp 'located-within <sensitive-area> <land-sector>)))
   (<sea-sector> (and Sea-sector (adjacent <sea-sector> <land-sector>)))
   (<amount-spilled>
    (and Numerical (gfp (amount-spilled <spilled-oil> <sea-sector>
 <amount-spilled>)))))
  (and (protect-shore-steps <spilled-oil> <sensitive-area> <old-steps>)
       (barrier-provided <sensitive-area>)
       (vacuum-level>= <sensitive-area> <amount-spilled>)))
 (effects ()
  ((del (protect-shore-steps <spilled-oil> <sensitive-area> <old-steps>))
   (add (protect-shore-steps <spilled-oil> <sensitive-area> <new-steps>))
   (add (shore-personnel-deployed <personnel> <sensitive-area>)))))

(operator cleanup-shore
 (params <spilled-oil> <sensitive-area> )
 (duration 1)
 (preconds
  ((<spilled-oil> spilled-oil)
   (<sensitive-area> sensitive-area)
   (<new-steps> (and Numerical (> <new-steps> 0)))
   (<old-steps> (and Numerical (sub1 <new-steps> <old-steps>)))
   (<shore-personnel>
    (and Numerical (gfp (shore-personnel-required
 <sensitive-area> <shore-personnel>))))
   (<land-sector> (and Land-sector
       (mygfp 'located-within <sensitive-area> <land-sector>)))
   (<sea-sector> (and Sea-sector
      (adjacent <sea-sector> <land-sector>)))
   (<amount-spilled>
    (and Numerical (gfp (amount-spilled <spilled-oil> <sea-sector>
 <amount-spilled>))))
   (<boom-personnel>
    (and Numerical (gfp (boom-personnel-required
 <sensitive-area> <boom-personnel>))))
   (<containment-boom-required>
    (and Numerical (gfp (containment-boom-required
 <containment-boom-required>)))))
```

```
  (and (protect-shore-steps <spilled-oil> <sensitive-area> <old-steps>)
       (boom-level>= <sensitive-area> <containment-boom-required>)
       (vacuum-level>= <sensitive-area> <amount-spilled>)))
 (effects ()
  ((del (protect-shore-steps <spilled-oil> <sensitive-area> <old-steps>))
   (add (protect-shore-steps <spilled-oil> <sensitive-area> <new-steps>))
   (add (boom-personnel-deployed <boom-personnel> <sensitive-area>))
   (add (boom-assembled <containment-boom-required> <sensitive-area>))
   (add (barrier-provided <sensitive-area> ))
   (add (shore-personnel-deployed <shore-personnel> <sensitive-area>)))))

;;; RECONNAISSANCE OPERATORS

(operator recon-polluted-sectors-by-sea ;; no preconditions
 (effects () ((add (recon-performed <spilled-oil> <sea-sector>)))))

(operator recon-polluted-sectors-by-air ;; no preconditions
 (effects () ((add (recon-performed <spilled-oil> <sea-sector>)))))

;;; LEVEL 3:

(inference-rule enough-boom1  ;; no preconditions
 (effects () ((add (boom-level>= <vessel> <length> <sea-sector> <sea-state>)))))

(inference-rule enough-boom2  ;; no preconditions
 (effects () ((add (boom-level>= <sea-sector> <sea-state> <length>)))))

;;; get-boom-to-contain-vessel

(operator get-boom-to-contain-vessel
 (params <vessel> <boom> <length-required> <sea-sector> <sea-state>)
 (preconds
  ((<vessel> Vessel)
   (<length-required> (and Numerical (numberp <length-required>)
   (> <length-required> 0)))
   (<sea-sector> sea-sector)
   (<sea-state> Sea-state)
   (<boom> (and Boom (gfp (in-service <boom>))
(sea-state-greater <boom> <sea-state>)))
   (<max-sea-state> (and Numerical
 (gfp (max-sea-state <boom> <max-sea-state>))))
   (<length> (and Numerical (gfp (length-boom-ft <boom> <length>))))
   (<length-left>
    (and Numerical (sub <length-required> <length> <length-left>))))
  (and (~ (boom-deployed <boom>))
       (located <boom> <sea-sector>)
       (sea-state-calmer <sea-sector> <max-sea-state>)
       (boom-level>= <vessel> <length-left> <sea-sector> <sea-state>)))
 (effects ()
  ((del (boom-level>= <vessel> <length-left> <sea-sector> <sea-state>))
   (add (boom-level>= <vessel> <length-required> <sea-sector> <sea-state>))
   (add (boom-assembled <vessel> <sea-sector>))
```

```
   (add (boom-deployed <boom>)))))

;;; get-boom-to-sea-sector

(operator get-boom-to-sea-sector
 (params <sea-sector> <boom> <length-required>)
 (preconds
  ((<sea-sector> sea-sector)
   (<sea-state> sea-state)
   (<length-required> (and numerical (> <length-required> 0)))
   (<boom> (and Boom (gfp (in-service <boom>))
(sea-state-greater <boom> <sea-state>)))
   (<max-sea-state>
    (and Numerical (gfp (max-sea-state <boom> <max-sea-state>))))
   (<length> (and Numerical (gfp (length-boom-ft <boom> <length>))))
   (<length-left>
    (and Numerical (sub <length-required> <length> <length-left>))))
  (and (sea-state-calmer <sea-sector> <max-sea-state>)
       (located <boom> <sea-sector>)
       (boom-level>= <sea-sector> <sea-state> <length-left>)))
 (effects ()
  ((del (boom-level>= <sea-sector> <sea-state> <length-left>))
   (add (boom-level>= <sea-sector> <sea-state> <length-required>))
   (add (boom-deployed <boom>)))))

;;; get-boom-to-sensitive-area

(operator get-boom-to-sensitive-area
 (params <sensitive-area> <boom> <length-required> <sea-sector>)
 (preconds
  ((<sensitive-area> sensitive-area)
   (<length-required> (and Numerical (> <length-required> 0)))
   (<sea-sector> sea-sector)
   (<sea-state> (and Numerical (gfp (sea-state <sea-sector> <sea-state>))))
   (<boom> (and Boom (in-service <boom>)
(sea-state-greater <boom> <sea-state>)))
   (<max-sea-state>
    (and Numerical (gfp (max-sea-state <boom> <max-sea-state>))))
   (<length> (and Numerical (gfp (length-boom-ft <boom> <length>))))
   (<length-left>
    (and Numerical (sub <length-required> <length> <length-left>))))
  (and (sea-state-calmer <sea-sector> <max-sea-state>)
       (located <boom> <sea-sector>)
       (boom-level>= <sensitive-area> <length-left> <sea-sector>)))
 (effects ()
  ((del (boom-level>= <sensitive-area> <length-left> <sea-sector>))
   (add (boom-level>= <sensitive-area> <length-required> <sea-sector>))
   (add (boom-deployed <boom>)))))

(inference-rule begin-skim  ;; no preconditions
 (preconds
  ((<discharge-rate> (and Numerical (>= 0 <discharge-rate>)))))
```

```
 (effects ()
  ((add (portable-skim-level>= <vessel> <sea-sector> <sea-state>
       <discharge-rate>)))))

(inference-rule begin-skim2  ;; no preconditions
 (preconds
  ((<discharge-rate> (and Numerical (>= 0 <discharge-rate>)))))
 (effects ()
  ((add (portable-skim-level>= <sea-sector> <discharge-rate> <sea-state>)))))

(inference-rule begin-skim3  ;; no preconditions
 (preconds
  ((<mskim-rate> (and Numerical (>= 0 <mskim-rate>)))))
 (effects () ((add (mobile-skim-level>= <sea-sector> <mskim-rate>)))))

;;; get-skimmer-to-skim-near-vessel

(operator get-skimmer-to-skim-near-vessel
 (params <vessel> <p-skimmer> <sea-sector> <sea-state> <discharge-rate>)
 (duration 1)
 (preconds
  ((<vessel> vessel)
   (<sea-sector> sea-sector)
   (<sea-state> sea-state)
   (<discharge-rate> (and Numerical (> <discharge-rate> 0)))
   (<p-skimmer> (and Portable-skimmer
     (skimmer-sea-state-greater <p-skimmer> <sea-state>)
     (not (gfp (in-use <p-skimmer>)))))
   (<max-sea-state> (and Numerical
 (gfp (max-sea-state <p-skimmer> <max-sea-state>))))
   (<skim-rate>
    (and Numerical (gfp (skim-rate-bbl-per-hr <p-skimmer> <skim-rate>))))
   (<discharge-left>
    (and Numerical (sub <discharge-rate> <skim-rate> <discharge-left>)))
   (<sweep-product>
    (and Numerical (gfp (sweep-product <p-skimmer> <sweep-product>)))))
  (and
   (~ (in-use <p-skimmer>))
   (~ (skimmer-employed <p-skimmer> <vessel>)) ; sorry about the redundancy
   (located <p-skimmer> <sea-sector>)
   (boom-assembled <vessel> <sea-sector>)
   (sea-state-calmer <sea-sector> <max-sea-state>) ; Jim 4/96
   (portable-skim-level>=
    <vessel> <sea-sector> <sea-state> <discharge-left>)))
 (effects ()
  ((del (portable-skim-level>=
 <vessel> <sea-sector> <sea-state> <discharge-left>))
   (add (portable-skim-level>=
 <vessel> <sea-sector> <sea-state> <discharge-rate>))
   ;;(del (free <p-skimmer>))
   (add (skimmer-employed <p-skimmer> <vessel>)))))
```

```
;;; get-portable-skimmer-to-skim-sea-sector

(operator get-portable-skimmer-to-skim-sea-sector
 (params <sea-sector> <p-skimmer> <pskim-rate-needed>)
 (duration 1)
 (preconds
  ((<sea-sector> sea-sector)
   (<sea-state> sea-state)
   (<pskim-rate-needed> (and Numerical (> <pskim-rate-needed> 0)))
   (<p-skimmer> (and Portable-skimmer
     (skimmer-sea-state-greater <p-skimmer> <sea-state>)))
   (<max-sea-state>
    (and Numerical (gfp (max-sea-state <p-skimmer> <max-sea-state>))))
   (<skim-rate>
    (and Numerical (gfp (skim-rate-bbl-per-hr <p-skimmer> <skim-rate>))))
   (<pskim-rate-left>
    (and Numerical (sub <pskim-rate-needed> <skim-rate> <pskim-rate-left>)))
   (<assembled>
    (and Numerical (gfp (boom-assembled <assembled> <sea-sector>)))))
  (and (~ (in-use <p-skimmer>))
       (located <p-skimmer> <sea-sector>)
       (sea-state-calmer <sea-sector> <max-sea-state>)
       (portable-skim-level>= <sea-sector> <pskim-rate-left> <sea-state>)))
 (effects ()
  ((del (portable-skim-level>= <sea-sector> <pskim-rate-left> <sea-state>))
   (add (portable-skim-level>= <sea-sector> <pskim-rate-needed> <sea-state>))
   (add (in-use <p-skimmer>))
   (add (skimmer-employed <p-skimmer>)))))

(operator get-mobile-skimmer
 (params <sea-sector> <m-skimmer> <mskim-rate-needed> <worst-sea-state>)
 (duration 1)
 (preconds
  ((<sea-sector> sea-sector)
   (<mskim-rate-needed> (and Numerical (> <mskim-rate-needed> 0)))
   ;; use to be generated by skimmer-sea-state-greater
   (<m-skimmer> self-mobile-skimmer)
   (<worst-sea-state> (and sea-state (gfp (max-sea-state <m-skimmer> <worst-sea-state>))))
   (<skim-rate>
    (and Numerical (gfp (skim-rate-bbl-per-hr <m-skimmer> <skim-rate>))))
   (<mskim-rate-left>
    (and Numerical (sub <mskim-rate-needed> <skim-rate>
<mskim-rate-left>)))
   )
  (and (located <m-skimmer> <sea-sector>)
       (sea-state-calmer <sea-sector> <worst-sea-state>)
       (~ (in-use <m-skimmer>))
       (mobile-skim-level>= <sea-sector> <mskim-rate-left>)))
 (effects ()
  ((del (mobile-skim-level>= <sea-sector> <mskim-rate-left>))
   (add (mobile-skim-level>= <sea-sector> <mskim-rate-needed>))
```

```
   (add (skimmer-employed <m-skimmer>)))))

;; (store-level>= <sea-sector> <amount-spilled>)

(inference-rule begin-storing  ;; no preconditions
 (preconds
  ((<store-needed> (and numerical (>= 0 <store-needed>)))))
 (effects () ((add (store-level>= <sea-sector> <store-needed>)))))

(operator get-tank-barge-to-sea-sector
 (params <barge> <sea-sector> <storage-level-needed>)
 (duration 1)
 (preconds
  ((<sea-sector> sea-sector)
   (<storage-level-needed> (and Numerical (> <storage-level-needed> 0)))
   (<barge> tank-barge)
   (<worst-sea> (and sea-state (gfp (max-sea-state <barge> <worst-sea>))))
   (<cap> (and Numerical (gfp (barge-capacity-bbl <barge> <cap>))))
   (<store-left> (and Numerical (sub <storage-level-needed> <cap>
     <store-left>))))
  (and (located <barge> <sea-sector>)
       (sea-state-calmer <sea-sector> <worst-sea>)
       (store-level>= <sea-sector> <store-left>)))
 (effects
  ()
  ((del (store-level>= <sea-sector> <store-left>))
   (add (store-level>= <sea-sector> <storage-level-needed>))
   (add (oil-pumped <sea-sector> <barge>))
   )))

(operator get-dracon-barge-to-sea-sector
 (params <sea-sector> <storage-level-needed>)
 (duration 1)
 (preconds
  ((<sea-sector> sea-sector)
   (<storage-level-needed> (and Numerical (> <storage-level-needed> 0)))
   (<barge> dracon-barge)
   (<worst-sea> (and sea-state
     (mygfp 'max-sea-state <barge> <worst-sea>)))
   (<cap> (and Numerical (gfp (barge-capacity-bbl <barge> <cap>))))
   (<store-left> (and Numerical (sub <storage-level-needed> <cap>
     <store-left>))))
  (and (sea-state-calmer <sea-sector> <worst-sea>)
       (located <barge> <sea-sector>)
       (store-level>= <sea-sector> <store-left>)))
 (effects ()
  ((del (store-level>= <sea-sector> <store-left>))
   (add (store-level>= <sea-sector> <storage-level-needed>))
   (add (oil-pumped <sea-sector> <barge>))
   )))

(operator get-tractor-to-sensitive-area
```

```
 (params <sensitive-area>)
 (duration 1)
 (preconds
  ((<sensitive-area> sensitive-area)
   (<tractor> tractor))
  (and (free <tractor>)
       (located <tractor> <sensitive-area>)))
 (effects ()
  ((del (free <tractor>))
   (add (barrier-provided <sensitive-area>)))))

(inference-rule free-if-available
 (params <resource>)
 (preconds
  ((<resource> Env-resource)
   (<time> (and Numerical (mygfp 'available <resource> <time>))))
  (and (available <resource> <time>)
       (~ (freed-once <resource>))))
 (effects ()
  ((add (free <resource>))
   (add (freed-once <resource>)))))


;;; (vacuum-level>= <sensitive-area> <amount-spilled> <latest>)

(inference-rule begin-vacuum  ;; no preconditions
 (preconds
  ((<amount> (and Numerical (>= 0 <amount>)))))
 (effects () ((add (vacuum-level>= <sensitive-area> <amount>)))))

(operator get-vacuum-truck
 (params <sensitive-area> <amount-needed> <truck>)
 (duration 1)
 (preconds
  ((<sensitive-area> sensitive-area)
   (<amount-needed> (and Numerical (> <amount-needed> 0)))
   (<truck> Vacuum-truck)
   (<cap> (and Numerical (gfp (capacity-bbl <truck> <cap>))))
   (<amount-left> (and Numerical (sub <amount-needed> <cap> <amount-left>))))
  (and (located <truck> <sensitive-area>)
       (barrier-provided <sensitive-area>)
       (vacuum-level>= <sensitive-area> <amount-left>)))
 (effects ()
  ((del (vacuum-level>= <sensitive-area> <amount-left> ))
   (add (vacuum-level>= <sensitive-area> <amount-needed> ))
   (add (oil-removed <truck> <sensitive-area>)))))


;;; DEPLOYMENT OPERATIONS

(operator move-self-mobile-skimmer-by-sea ;; no preconditions
 (preconds
```

```
  ((<seaport> (and seaport (mygfp 'located <skimmer> <seaport>)))))
 (effects ()
  ((del (located <skimmer> <seaport>))
   (add (located <skimmer> <sea-sector>)))))

(operator move-response-equipment-by-sea1a  ;; no preconditions
 (duration
  (let ((dist (known-unique `(distance <seaport> <sea-sector> <d>))))
    (if dist
(/ dist 15) ; assume 15 knots.
      10)))
 (preconds
  ((<boat> (and platform-workboat (mygfp 'located-within <equip> <boat>)))
   (<seaport> (and seaport (mygfp 'located <boat> <seaport>)))))
 (effects ()
  ((del (located <boat> <seaport>))
   (add (located <boat> <sea-sector>))
   (add (located <equip> <sea-sector>)))))

(operator move-response-equipment-by-sea1b  ;; no preconditions
 (duration
  (let ((dist (known-unique `(distance <seaport> <sea-sector> <d>))))
    (if dist
(/ dist 15) ; assume 15 knots.
      10)))
 (preconds
  ((<seaport> (and seaport (mygfp 'located <equip> <seaport>)))))
 (effects ()
  ((del (located <equip> <seaport>))
   (add (located <equip> <sea-sector>))
   )))

(operator move-response-equipment-by-sea2a  ;; no preconditions
 (params <equip> <seaport1>)
 (duration 10)
 (preconds
  ((<boat> (and platform-workboat (mygfp 'located-within <equip> <boat>)))
   (<seaport2> (and seaport (mygfp 'located <boat> <seaport2>)))))
 (effects ()
  ((del (located <boat> <seaport2>))
   (add (located <boat> <seaport1>))
   (add (located <equip> <seaport1>)))))

(operator move-response-equipment-by-sea2b  ;; no preconditions
 (params <equip> <seaport1>)
 (preconds
  ((<seaport2> (and seaport (mygfp 'located <equip> <seaport2>)))))
 (effects ()
  ((del (located <equip> <seaport2>))
   (add (located <equip> <seaport1>)))))

(operator move-response-equipment-by-sea3a  ;; no preconditions
```

```
 (preconds
  ((<pboat> (and platform-workboat (mygfp 'located-within <equip> <pboat>)))
   (<seaport> (and seaport (mygfp 'located <pboat> <seaport>)))))
 (effects ()
  ((del (located <pboat> <seaport>))
   (add (located <pboat> <sensitive-area>))
   (add (located <equip> <sensitive-area>)))))

(operator move-response-equipment-by-sea3b ;; no preconditions
 (preconds
  ((<seaport> (and seaport (mygfp 'located <equip> <seaport>)))))
 (effects () ((add (located <equip> <sensitive-area>)))))

(operator move-response-equipment-by-sea4  ;; no preconditions
 (preconds
  ((<from-sea-sector> (and sea-sector
   (mygfp 'located <equip> <from-sea-sector>)))
   (<to-sea-sector> (and sea-sector (diff <from-sea-sector> <to-sea-sector>)))))
 (effects () ((add (located <equip> <to-sea-sector>)))))

(operator move-response-equipment-by-ground
 (params <equip> <from-location> <to-location>)
 (duration 1)
 (preconds
  ((<equip> response-equipment)
   (<from-location> (and location (mygfp 'located <equip> <from-location>)))
   (<to-location> (and location (diff <from-location> <to-location>)))
   )
  (located <equip> <from-location>))
 (effects ()
  ((add (located <equip> <to-location>)))))

(operator move-response-equipment-by-air
 (params <equip> <from-airfield> <to-airfield>)
 (duration 1)
 (preconds
  ((<equip> response-equipment)
   (<rand-loc> (and location (mygfp 'located <equip> <rand-loc>)))
   (<from-airfield> airfield)
   (<to-airfield> (and airfield (diff <from-airfield> <to-airfield>)))
   )
  (located <equip> <from-airfield>))
 (effects ()
  ((del (located <equip> <from-airfield>))
   (add (located <equip> <to-airfield>)))))

(operator move-heavy-equip-by-ground1  ;; no preconditions
 (params <heavy> <from-loc> <to-land>)
 (duration 1)
 (preconds
  ((<heavy> heavy-equipment)
   (<from-loc> (and location (mygfp 'located <heavy> <from-loc>)))
```

```
   (<to-land> land-sector)))
 (effects ()
  ((del (located <heavy> <from-loc>))
   (add (located <heavy> <to-land>)))))

(operator move-heavy-equip-by-ground2  ;; no preconditions
 (params <heavy> <from-loc> <to-loc>)
 (duration 1)
 (preconds
  ((<heavy> heavy-equipment)
   (<from-loc> (and location (mygfp 'located <heavy> <from-loc>)))))
 (effects ()
  ((del (located <heavy> <from-loc>))
   (add (located <heavy> <to-loc>)))))

(operator move-team-by-ground  ;; no preconditions
 (effects () ((add (located <team> <seaport>)))))

(operator move-team-by-sea   ;; no preconditions
 (preconds
  ((<seaport> (and seaport (mygfp 'located <team> <seaport>)))))
 (effects ()
  ((del (located <team> <seaport>))
   (add (located <team> <sea-sector>)))))

(operator move-team-by-air  ;; no preconditions
 (preconds
  ((<from-airfield> (and airfield (mygfp 'located <team> <from-airfield>)))))
 (effects ()
  ((del (located <team> <from-airfield>))
   (add (located <team> <to-airfield>)))))

(operator move-to-sea-sector-from-port  ;; no preconditions
 (duration
  (let ((distance (or (known-unique `(distance <seaport> <sea-sector> <x>))
      100))
(max-speed (or (known-unique `(max-speed <vessel> <x>)) 10)))
    (/ distance max-speed))) ; a little optimistic..
 (preconds
  ((<seaport> (and seaport (mygfp 'located <vessel> <seaport>)))))
 (effects ()
  ((del (located <vessel> <seaport>))
   (add (located <vessel> <sea-sector>)))))

;;;=============================================================
;;; Weather
;;;=============================================================

(inference-rule calm-enough
 (params <sea-sector> <sea-state> <worst-state>)
 (preconds
  ((<sea-sector> Sea-sector)
```

```
   (<worst-state> sea-state)
   (<sea-state> (and sea-state
     (gfp (sea-state <sea-sector> <sea-state>))
     (<= <sea-state> <worst-state>))))
  (sea-state <sea-sector> <sea-state>))
 (effects ()
  ((add (sea-state-calmer <sea-sector> <worst-state>)))))


;;;============================================================
;;; Eager inference rules
;;;============================================================

;;; sea-state should be primed from sea-state 0. Has to be eager
;;; because the sea states are typically queried with gen-from-pred.

(inference-rule initial-sea-state
 (params <sector> <state>)
 (mode eager)
 (preconds
  ((<sector> Sea-sector)
   (<state> (and Numerical (earliest-sea-state <sector> <state>))))
  (and))
 (effects ()
  ((add (sea-state <sector> <state>)))))

(inference-rule initial-amount-spilled
 (params <oil> <sector> <amount>)
 (mode eager)
 (preconds
  ((<oil> Spilled-oil)
   (<sector> Sea-sector)
   (<amount> (and Numerical
  (earliest-amount-spilled <oil> <sector> <amount>))))
  (and))
 (effects ()
  ((add (amount-spilled <oil> <sector> <amount>)))))


;;;============================================================
;;; Events
;;;============================================================

;;; These declare that the literals are functional on the given argument.
(setf *lit-function-translations*
      '(
((sea-state <sector> <state>)
 ((sea-state <sector>) <state>))
((located <obj> <place>) ((located <obj>) <place>))
((mobile-skim-level>= <sector> <rate>)
 ((mobile-skim-level>= <sector>) <rate>))
((remove-oil-steps <oil> <steps>) ((remove-oil-steps <oil>) <steps>))
```

```
((amount-spilled <oil> <sector> <amount>)
 ((amount-spilled <oil> <sector>) <amount>))
))

(setf *lits-to-ignore* nil)

;;; A range can be handled directly in the Bayes net, which is much
;;; more efficient than the other methods. Note that the link in the
;;; BN is still done through the inference rule (calm-enough),
;;; but the table is filled in using the fact that it is a range.
(setf *lit-ranges*
      '((calm-enough
 (sea-state-calmer <sector> <val>)
 (sea-state <sector>) <= <worst-state>)))

(event sea-gets-worse
 (params <sea-sector> <old-sea-state> <new-sea-state>)
 (probability 0.1)
 (preconds
  ((<sea-sector>     Sea-Sector)
   (<old-sea-state>  Sea-State)
   (<new-sea-state>  (and Sea-State (add1 <old-sea-state> <new-sea-state>)
  (< <new-sea-state> 7)
  )))
  (sea-state <sea-sector> <old-sea-state>))
 (effects ()
  ((del (sea-state <sea-sector> <old-sea-state>))
   (add (sea-state <sea-sector> <new-sea-state>)))))

(event sea-gets-better
 (params <sea-sector> <old-sea-state> <new-sea-state>)
 (probability 0.1)
 (preconds
  ((<sea-sector>     Sea-Sector)
   (<old-sea-state>  Sea-State)
   (<new-sea-state>  (and Sea-State (sub1 <old-sea-state> <new-sea-state>)
  (> <new-sea-state> 0)
  )))
  (sea-state <sea-sector> <old-sea-state>))
 (effects ()
  ((del (sea-state <sea-sector> <old-sea-state>))
   (add (sea-state <sea-sector> <new-sea-state>)))))

(event oil-spills
 (params <spilled-oil> <sea-sector> <rate> <new>)
 (probability 0.2)
 (preconds
  ((<spilled-oil> Spilled-oil)
   (<vessel> Vessel)
   (<vessel-boom-length>
    ;; not gfp because I want this to have a default value (0).
    (and Numerical (vessel-boom-length <vessel> <vessel-boom-length>)))
```

```
   (<sea-sector> ;;(and Sea-sector (gfp (located <vessel> <sea-sector>)))
    Sea-sector
    )
   ;; uh oh. (if left in this will collapse the MC)
   (<sea-state> (and Numerical (gfp (sea-state <sea-sector> <sea-state>))))
   (<rate> (and Numerical
(gfp (discharge-rate <vessel> <spilled-oil> <rate>))))
   (<total> (and Numerical
 (gfp (discharge-size <vessel> <spilled-oil> <total>))))
   (<old> ;; the numberp call forces it to be bound.
  (and Numerical (numberp <old>) (< <old> <total>)))
   (<new> (and Numerical (add-with-ceiling <old> <rate> <total> <new>)
       ;; can't spill more than there is.
       (<= <new> <total>)
       )))
  (and
   (~ (boom-level>= <vessel> <vessel-boom-length> <sea-sector> <sea-state>))
   ;; sorry for the double negative in the next predicate, will fix.
   (~ (no-discharge <vessel> <spilled-oil> <sea-sector>))
   (amount-spilled <spilled-oil> <sea-sector> <old>)
   ;;(~ (covered-sector <spilled-oil> <sea-sector>))
   ))
 ;; taken eveything out so as not to have spurious nodes in the markov
 ;; chain.
 (effects ()
  (;;(add (some-spilled <spilled-oil> <vessel>))
   ;; this is a trigger for the eager inference rule below.
   ;;(add (new-spill <vessel> <spilled-oil> <sea-sector>))
   (del (amount-spilled <spilled-oil> <sea-sector> <old>))
   (add (amount-spilled <spilled-oil> <sea-sector> <new>))
   ;;(add (boom-length <spilled-oil> <sea-sector> 100)) ; entirely arbitrary.
   ;;(add (covered-sector <spilled-oil> <sea-sector>))
   )))

;;; used to change each place from not threatened to threatened
;;; independently. Now "moves" the threat from one shore to another.
;;; This event highlights a problem with the syntax. I wanted to have
;;; only one such event fire, and use multiple outcomes to decide
;;; which new shoreline is chosen. But since there is a variable
;;; number of shorelines I can't represent them. So I will allow
;;; several such events to potentially fire although they are mutually
;;; exclusive because their results conflict.  This means their
;;; probabilities have to be figured as mutually exclusive events
;;; although they're represented here as the independent
;;; probabilities. For example, two of these things at 0.1 probability
;;; means p(event 1) = 0.095 p(event 2) = 0.095 p(nothing) =

(event threatened-shore-changes
 (params <from-area> <to-area> <oil>)
 (probability 0.1)
 (preconds
  ((<oil> Spilled-oil)
```

```
   (<sea-sector> Sea-Sector)
   ;; Strictly, these bindings should make this event depend on oil-spills.
   ;; Syntactically they don't because this requires the predicate to
   ;; be present with any amount, and so does oil-spills.
   (<amount> (and Numerical
  (gfp (amount-spilled <oil> <sea-sector> <amount>))
  (> <amount> 0)))
   (<from-area> (and Sensitive-Area
     (potentially-threatened <from-area> <oil>)))
   (<to-area> (and Sensitive-Area
   (potentially-threatened <to-area> <oil>)
   (diff <from-area> <to-area>))))
  (~ (threatened-shoreline <oil> <sensitive-area-2>)))|#
  ;; the above is better when this is not a functional literal, below
  ;; is better when it is.
  (threatened-shoreline <oil> <from-area>))
  (effects ()
   ((del (threatened-shoreline <oil> <from-area>))
    (add (threatened-shoreline <oil> <to-area>)))))

(setf *events-not-to-negate*
      '(sea-gets-better   ; King Canute might have something
sea-gets-worse      ; to say about this
threatened-shore-changes))


;;;==============================================================
;;; Control rules
;;;==============================================================

;;; This is a general one for conditional planning, that prefers goals
;;; without contexts over goals with them. This might not win in
;;; general, but it does seem to in this domain (so far).
(control-rule prefer-no-context
 (if (and (candidate-goal <g1>)
  (goal-has-context <g2>)
  (~ (goal-has-context <g1>))))
 (then prefer goal <g1> <g2>))

(defun goal-has-context (goal)
  (if (p4::lit-context goal) t nil))

;;; On the other hand if a goal becomes unsolvable in some context, I
;;; want to quickly work on the parent goal.

;;; The specialization makes the planner incomplete, because one
;;; option is to wait for the sea to calm, but I want to quickly shut
;;; planning down on that path because it will have low probability.

;;; Note: this rule must have priority over the rule below that
;;; selects the sea-state-calmer rule only.
```

```
(control-rule promote-parent-goal-in-context
 (if (and (candidate-goal (sea-state-calmer <sector> <state>))
  ;; May as well select it anyway since order won't matter.
  (known (sea-state <sector> <higher-state>))
  (> <higher-state> <state>)
  ;;(sea-state-goal-has-unselected-context-parent
  ;;<sector> <state> ;;<better-goal>)
  ;;)
  (context-affected-goals-are-parents-and-none-is-expanded
   <sector> <state> <goals>)
  ))
 (then select goal <goals>))


;;; If we're in some context that has introduced this unsatisfiable
;;; sea-state goal and if it has parents that are context-dependent
;;; and none of them has already been re-introduced, then re-introduce
;;; one.
;;; This function is called after we know the current state is too
;;; high. So this must be on a branch where the sea-state goal is a
;;; child of the affected goals. Just test if one is expanded and the
;;; sea-state goal is not a child of that expanded goal.
(defun context-affected-goals-are-parents-and-none-is-expanded
  (sector state goal-var)
  (unless (and (p4::get-context *current-node*)
       (some #'(lambda (cnode) (not (member-if #'zerop (cdr cnode))))
     (p4::get-context *current-node*)))
    (return-from context-affected-goals-are-parents-and-none-is-expanded
 nil))
  (let* ((context (p4::get-context *current-node*))
 (affected (p4::context-node-affected (car p4::*context-nodes*))))
    (if (and affected
     (not (some #'(lambda (goal)
    (expanded-below-context
     goal (car p4::*context-nodes*)))
affected)))
(mapcan
 #'(lambda (aff)
     (let ((goal (p4::goal-node-goal
  (p4::nexus-parent
   (p4::nexus-parent aff)))))
       (if (member goal p4::*pending-goals*)
   (list (list (cons goal-var goal))))))
 affected))))

(defun expanded-below-context (goal cnode)
  (do ((node *current-node* (p4::nexus-parent node)))
      ((or (null node) (eq node cnode)
   (and (p4::goal-node-p node)
(eq (p4::goal-node-goal node) goal)))
       (and (p4::goal-node-p node)
    (eq (p4::goal-node-goal node) goal)))))
```

```
;;; Otherwise, select the sea-state goal, either to get it out of the
;;; way or to fail quickly.
(control-rule do-sea-state-early
 (if (and (candidate-goal (sea-state-calmer <sector> <goal-state>))
  (known (sea-state <sector> <actual-state>))
  (or (<= <actual-state> <goal-state>)
      (child-in-this-context <sector> <goal-state>)
      )))
 (then select goal (sea-state-calmer <sector> <goal-state>)))

;;; Find an introducing goal below the lowest context node.
(defun child-in-this-context (sector goal-state)
  (or (null p4::*context-nodes*)
      (let ((lit (p4::instantiate-consed-literal
  `(sea-state-calmer ,(oname sector) ,goal-state))))
(if (some #'(lambda (intro)
      (member (car p4::*context-nodes*)
      (p4::path-from-root
       (p4::instantiated-op-binding-node-back-pointer
intro))))
  (if (p4::literal-state-p lit)
      (p4::literal-neg-goal-p lit)
    (p4::literal-goal-p lit)))
    t))))

;;; First, we want to cut down on the bogus search that this method of
;;; encoding multi-step goals produces, for instance when we are about
;;; to cut off the number of steps of some task. Both force the
;;; inference rule that cuts off to be used, and force the goal to be
;;; picked since it won't interfere with anything else.

(mapc #'(lambda (triple)
  (let ((goal (first triple))
(op (second triple))
(pred (or (third triple) t)))
    (eval `(control-rule ,(intern (format nil "sel-~S" op))
 (if (and (current-goal ,goal) ,pred))
 (then select operator ,op)))
    (eval `(control-rule ,(intern (format nil "sel-~S" op))
 (if (and (candidate-goal ,goal) ,pred))
 (then select goal ,goal)))))
      '(( (stabilize-discharge-rate <v> <o> <s> <r>) begin-stabilize
  (acceptable-discharge-rate <r>))
( (remove-oil-steps <o> 0) begin-remove-oil)
( (protect-shore-steps <o> <a> 0) start-protecting-shore)
( (store-level>= <sector> <level>) begin-storing (< <level> 0))
( (portable-skim-level>= <sector> <level> <state>) begin-skim2
  (< <level> 0))))
```

```
(mapc #'(lambda (pair)
  (let ((goal (first pair))
(op (second pair))
(test (or (third pair) '(> <amount> 0))))
    (eval `(control-rule ,(intern (format nil "rej-~S" op))
     (if (and (current-goal ,goal) ,test))
     (then reject operator ,op)))))
      '(( (remove-oil-steps <o> <amount>) begin-remove-oil)
( (protect-shore-steps <o> <a> <amount>) start-protecting-shore)
( (vacuum-level>= <p> <amount> <t>) begin-vacuum)
( (stabilize-discharge-rate <ship> <spill> <place> <amount>)
  begin-stabilize (~ (acceptable-discharge-rate <amount>)))
))

(control-rule dont-protect-unnecessarily
 (if (and (current-goal (ok-shoreline <area> <oil>))
  (~ (known (threatened-shoreline <oil> <area>)))))
 (then reject operator shoreline-protected))

(control-rule cannot-stop-threat
 (if (and (current-goal (ok-shoreline <area> <oil>))
  (known (threatened-shoreline <oil> <area>))))
 (then reject operator shoreline-not-threatened))

;;; The order in which these inference rules are expanded won't
;;; matter. As soon as unprotected-sensitive-area is expanded, do these.
(control-rule pick-a-shoreline
 (if (lexically-first-candidate-shoreline <area> <oil>))
 (then select goal (ok-shoreline <area> <oil>)))

;;; Look at all the shorelines that could be worked on, force the one
;;; that's first in the yellow pages.
(defun lexically-first-candidate-shoreline (area-var oil-var)
  (let ((shorelines (candidate-goal '(ok-shoreline <area> <oil>))))
    (if shorelines
(sublis (mapcar #'cons '(<area> <oil>) (list area-var oil-var))
(list (car
       (sort shorelines
     #'(lambda (b1 b2)
 (string< (oname (cdr (assoc '<area> b1)))
  (oname (cdr (assoc '<area> b2)))))
     )))))))

;;; Similarly work on barrier-provided first, since it won't mess with
;;; other goals and we just want to fail quickly if there's no free
;;; tractor.
(control-rule provide-barrier-quickly
 (if (candidate-goal (barrier-provided <area>)))
 (then select goal (barrier-provided <area>)))

;;; Implement primary effects
(control-rule provide-barrier-simply1
```

```
 (if (current-goal (barrier-provided <sensitive-area>)))
 (then reject operator shore-exclusion-booming))

(control-rule provide-barrier-simply2
 (if (current-goal (barrier-provided <sensitive-area>)))
 (then reject operator shore-diversion-booming))

(control-rule provide-barrier-simply3
 (if (current-goal (barrier-provided <sensitive-area>)))
 (then reject operator cleanup-shore))

;;; This one forces it to use different equipment when subgoaling to
;;; pick up the slack in stabilizing
(control-rule stabilize-differently
 (if (and (current-ops (cargo-transfer-oil-to-stabilize))
  (expanded-operator
   (cargo-transfer-oil-to-stabilize
    <v> <o> <s> <r> <pump> <b> <pc> <bc>))))
 (then reject bindings ((<cargo-transfer-pump> . <pump>))))

(control-rule different-mobile-skimmer
 (if (and (current-ops (use-mobile-skimmer))
  (expanded-operator
   (use-mobile-skimmer <s> <ss> <srn> <ms> <sr> <srl>))))
 (then reject bindings ((<m-skimmer> . <ms>))))

(control-rule different-portable-skimmer
 (if (and (current-ops (get-skimmer-to-skim-near-vessel
get-portable-skimmer-to-skim-sea-sector))
  (or (expanded-operator
       (get-skimmer-to-skim-near-vessel
<v> <sec> <state> <rate> <skimmer> <srate> <left> <product>))
      (expanded-operator
       (get-portable-skimmer-to-skim-sea-sector
<sec> <state> <rate> <skimmer> <max> <srate> <left>
<assembled>)))))
 (then reject bindings ((<p-skimmer> . <skimmer>))))

(control-rule different-boom
 (if (and (current-ops (use-boom-in-sensitive-area))
  (expanded-operator
   (use-boom-in-sensitive-area <area> <req> <sector> <state>
       <used-boom> <max> <length> <left>))))
 (then reject bindings ((<boom> . <used-boom>))))

;;; These are purely speed-up search control, without which it takes
;;; FOREVER. They remove an operator from the search space if it
;;; requires a resource that has been switched off.
;;; This is a good place to start thinking about iterative
;;; sophistication search.

;;; If a barge has already moved to a sea-sector, it can't be moved to
```

```
;;; another sea-sector - there's no operator to do it.
(control-rule cannot-move-barge-from-sea-sector
 (if (and (current-ops (cargo-transfer-oil-to-stabilize))
  (current-goal (stabilize-discharge-rate
 <barge> <oil> <required-place> <rate>))
  (known (located <barge> <place>))
  (ptype <place> sea-sector)
  (~ (eq <place> <required-place>))))
 (then reject bindings
       ((<tank-barge> . <barge>)
(<sea-sector> . <required-place>))))

(control-rule cannot-move-barge-from-sea-sector
 (if (and (current-ops (use-tank-barge-as-storage))
  (current-goal
   (store-level>= <required-place> <storage-level-needed>))
  (known (located <some-barge> <place>))
  (ptype <place> sea-sector)
  (~ (eq <place> <required-place>))))
 (then reject bindings
       ((<barge> . <some-barge>)
(<sea-sector> . <required-place>))))

(defun ptype (object typename)
  (eq (p4::type-name (p4::prodigy-object-type object)) typename))

(mapc #'(lambda (pair)
  (let ((object (first pair))
(operator (second pair))
(goal (third pair))
(predicate (or (fourth pair) 'in-use)))
    (eval `(control-rule ,(intern (format nil "need-a-free-~S"
  object))
 (if (and (current-goal ,goal)
  (all-objects ,object ,predicate)))
 (then reject operator ,operator)))))
      '( (self-mobile-skimmer open-water-recovery
      (remove-oil-steps <oil> <steps>))
 (portable-skimmer shallow-water-recovery
   (remove-oil-steps <oil> <steps>))
 (dispersant apply-chemical-dispersant
     (remove-oil-steps <oil> <steps>) use-prohibited)
 (cargo-transfer-pump cargo-transfer-oil-to-stabilize
      (stabilize-discharge-rate
       <vessel> <oil> <sector> <rate>))
 (portable-skimmer stabilize-discharge-by-contain-skim
   (stabilize-discharge-rate
    <vessel> <oil> <sector> <rate>))
 ))

;;; known with negated predicates really doesn't seem to work..
(defun all-objects (type pred)
```

```
  (every #'(lambda (object) (known (list pred (oname object))))
 (p4::type-real-instances
  (p4::type-name-to-type type *current-problem-space*))))

;;; Some quick hacks to cut out useless search.
(control-rule no-seaworthy-barges
 (if (and (current-goal (stabilize-discharge-rate <v> <o> <s> <r>))
  (known (sea-state <s> <state>))
  (no-objects-above-sea-state tank-barge <state>)))
 (then reject operator cargo-transfer-oil-to-stabilize))

(control-rule no-seaworthy-portable-skimmers-or-booms
 (if (and (current-goal (stabilize-discharge-rate <v> <o> <s> <r>))
  (known (sea-state <s> <state>))
  (or (no-objects-above-sea-state boom <state>)
      (no-objects-above-sea-state portable-skimmer <state>))))
 (then reject operator stabilize-discharge-by-contain-skim))

(control-rule no-seaworthy-mobile-skimmers-or-booms
 (if (and (current-goal (recovery-open-water <oil> <sector>))
  (known (sea-state <sector> <state>))
  (or (no-objects-above-sea-state self-mobile-skimmer <state>)
      (and (no-objects-above-sea-state tank-barge <state>)
   (no-objects-above-sea-state dracon-barge <state>)))))
 (then reject operator perform-open-water-recovery))

(control-rule no-seaworthy-boom-or-portable-skimmer-or-barge
 (if (and (current-goal (recovery-shallow-water <oil> <sector>))
  (known (sea-state <sector> <state>))
  (or (no-objects-above-sea-state boom <state>)
      (no-objects-above-sea-state portable-skimmer <state>)
      (and (no-objects-above-sea-state tank-barge <state>)
   (no-objects-above-sea-state dracon-barge <state>)))))
 (then reject operator perform-shallow-water-recovery))

;;; Should be generalised to whether there is enough boom for all of them.
;;; But "enough" is governed by a different predicate in each case.
(control-rule no-seaworthy-boom-for-shore-ops
 (if (and (current-goal (protect-shore-steps <oil> <area> <n>))
  (known (located-within <area> <land>))
  (adjacent <sea-sector> <land>)
  (known (sea-state <sea-sector> <state>))
  (no-objects-above-sea-state boom <state>)
  (trying-operator <op> (cleanup-shore shore-diversion-booming
       shore-exclusion-booming))))
 (then reject operator <op>))

(control-rule not-enough-good-boom-for-exclusion-booming
 (if (and (current-goal (protect-shore-steps <oil> <area> <n>))
  (known (located-within <area> <land>))
  (adjacent <sector> <land>)
  (known (sea-state <sector> <state>))
```

```
  (known (exclusion-boom-required <area> <amount>))
  (less-boom-above-state <state> <amount>)))
 (then reject operator shore-exclusion-booming))

(control-rule no-seaworthy-boom-to-clean-up-shore
 (if (and (current-goal (protect-shore-steps <spill> <area> <steps>))
  (known (located-within <area> <land>))
  (adjacent <sea-sector> <land>)
  (known (sea-state <sea-sector> <state>))
  (no-objects-above-sea-state boom <state>)))
 (then reject operator cleanup-shore))

(defun no-objects-above-sea-state (type state)
  (every #'(lambda (object)
     (let ((max-state (known-unique
       (list 'max-sea-state (oname object) '<x>))))
       (< max-state state)))
 (p4::type-real-instances
  (p4::type-name-to-type type *current-problem-space*))))

(defun less-boom-above-state (state amount)
  (let ((ttl 0))
    (dolist (boom (p4::type-instances
   (p4::type-name-to-type 'boom *current-problem-space*)))
      (let ((max (known-unique `(max-sea-state ,(oname boom) <x>))))
(if (> max state)
    (incf ttl (known-unique `(length-boom-ft ,(oname boom) <x>))))))
    (< ttl amount)))

(defun trying-operator (var list)
  ;; If the var is a variable, return every possible binding.
  (if (varp var)
      (mapcar
       #'(lambda (elt)
   (list
    (cons var (p4::rule-name-to-rule elt *current-problem-space*))))
       list)
    ;; Otherwise check if the value is in the list
    (member var list)))

;;; In p1 there are two sources of chemical dispersant, one 6 miles
;;; away and one 345 miles away. Without this rule prodigy chooses the
;;; further one.
(control-rule prefer-closer-dispersant
 (if (and;;(current-goal (apply-dispersant <oil> <place> <time))
      (current-ops (get-chem-dispersant))
      (candidate-bindings ((<spilled-oil> . <sp>)
   (<sea-sector> . <sea>)
   (<earliest-start> . <early>)
   (<dispersant> . <d>)))
      ;; <b2> is bound as the current best bindings
      (inst-val <dispersant> <b2> <other-d>)
```

```
      (closer-obj <sea> <d> <other-d>)))
 (then prefer bindings
       ((<spilled-oil> . <sp>)
(<sea-sector> . <sea>)
(<earliest-start> . <early>)
(<dispersant> . <d>))
       <b2>))

(control-rule prefer-closer-boat
 (if (and (current-ops (move-response-equipment-by-sea1b))
  (candidate-bindings ((<equip> . <e>)
       (<sea-sector> . <s>)
       (<seaport> . <sp>)
       (<earliest> . <time>)
       (<uboat> . <u>)))
  (inst-val <uboat> <b2> <ob>)
  (closer-obj <sp> <u> <ob>)))
 (then prefer bindings ((<equip> . <e>)
(<sea-sector> . <s>)
(<seaport> . <sp>)
(<earliest> . <time>)
(<uboat> . <u>))
       <b2>))

;;; I'd rather not have to write so many similar rules.. on the other
;;; hand this form may be easier to build a tree out of.
(control-rule prefer-closer-skimmer
 (if (and (current-ops (use-mobile-skimmer))
  (candidate-bindings <b>)
  (elt-val 3 <b> <b-sk>)
  (inst-val <m-skimmer> <c> <c-sk>) ; <c> is bound from RHS
  (inst-val <sea-sector> <c> <ss>) ; assumes they are the same.
  (closer-obj <ss> <b-sk> <c-sk>)
  (list-to-bindings <b> use-mobile-skimmer <bb>)))
 (then prefer bindings <bb> <c>))

(control-rule prefer-closer-port
 (if (and (current-ops (stabilize-discharge-by-towing-to-port))
  (candidate-bindings <b>)
  (elt-val 5 <b> <near-port>)
  (inst-val <sea-port> <bad> <far-port>)
  (inst-val <sea-sector> <bad> <sea>)
  (closer-port-to-sea <sea> <near-port> <far-port>)
  (list-to-bindings <b> stabilize-discharge-by-towing-to-port <good>)))
 (then prefer bindings <good> <bad>))

;;; Prefer to use a long enough boom if there is one. I guess two
;;; short ones that can be transported together and are much closer
;;; would be better, but.. Ok, so greedily prefer the boom that
;;; maximises the ratio of proportion of boom added to distance.

(control-rule best-boom-length-per-hour
```

```
 (if (and (current-ops (use-boom-to-contain-vessel))
  (candidate-bindings <b>)
  (elt-val 1 <b> <required>)
  (elt-val 4 <b> <good-boom-obj>)
  (obj-to-name <good-boom-obj> <good-boom>)
  (elt-val 6 <b> <good-length>)
  (inst-val <boom> <other> <other-boom>)
  (inst-val <length> <other> <other-length>)
  (inst-val <sea-sector> <other> <sector>)
  (greater-used-length-per-hr
   <good-boom> <good-length> <other-boom> <other-length>
   <required> <sector>)
  (list-to-bindings <b> use-boom-to-contain-vessel <better>)))
 (then prefer bindings <better> <other>))

;;; Figure out how much of the needed length is delivered "per hour"
;;; assuming the same speed of delivery for each boom.
(defun greater-used-length-per-hr (boom1 length1 boom2 length2
 needed-length sector)
  (let ((delivered1 (min length1 needed-length))
(delivered2 (min length2 needed-length))
(distance1  (obj-distance sector boom1))
(distance2  (obj-distance sector boom2)))
    (cond ((zerop distance2) nil)
  ((zerop distance1) t)
  (t
   (> (/ delivered1 distance1) (/ delivered2 distance2))))))

(defun obj-to-name (object namevar)
  (if (p4::prodigy-object-p object)
      (list (list (cons namevar (p4::prodigy-object-name object))))))

(control-rule prefer-long-enough-boom-in-sensitive-area
 (if (and (current-ops (use-boom-in-sensitive-area))
  (candidate-bindings <b>)
  (elt-val 7 <b> <less-left>)
  (inst-val <length-left> <other> <more-left>)
  (< <less-left> 0)   ; there's enough boom in one bindings set
  (> <more-left> 0)   ; there isn't in the other
  (list-to-bindings <b> use-boom-in-sensitive-area <better>)))
 (then prefer bindings <better> <other>))

(control-rule prefer-big-enough-skimmer
 (if (and (current-ops (get-skimmer-to-skim-near-vessel))
  (candidate-bindings <b>)
  (elt-val 7 <b> <less-left>)
  (inst-val <discharge-left> <other> <more-left>)
  (<= <less-left> 0)
  (> <more-left> 0)
  (list-to-bindings <b> get-skimmer-to-skim-near-vessel <better>)))
 (then prefer bindings <better> <other>))
```

```
;;; prefer to use a bigger pump.

(control-rule prefer-big-enough-pump
 (if (and (current-ops (cargo-transfer-oil-to-stabilize))
  (candidate-bindings <b>)
  ;; bind <b-cap> to the pump capacity in bindings <b>
  (op-val <b> cargo-transfer-oil-to-stabilize <pump-cap> <b-cap>)
  (inst-val <pump-cap> <other> <other-cap>)
  (> <b-cap> <other-cap>)
  (list-to-bindings <b> cargo-transfer-oil-to-stabilize <better>)))
 (then prefer bindings <better> <other>))
```

B.2 Random problem generator

In Chapter 7 I report on experiments with a number of randomly-generated problems from the oil-spill domain. Here, I describe the problem generator in detail.

The algorithm to generate the random problems fixes the geography of the area, as defined by the set of objects of types `sea-sector`, `land-sector`, `urban`, `seaport` and `sensitive-area` and by the literals for the predicates `adjacent`, `located-within`, `distance` and `berth-size`. The geographic information models the San Francisco Bay area as defined in the version of the domain written at SRI [Desimone & Agosta 1994]. This fixed base of information includes 130 objects and 320 predicates. The reader who is interested in this domain and generator is encouraged to send electronic mail to `jblythe@cs.cmu.edu` for a copy.

Within this geographic framework, one vessel is placed in a random sea-sector that contains at least one sensitive-area, of which there are three. The vessel is given a random displacement, distributed uniformly between 30000 and 80000 at 10000 intervals. The spillable oil has size either 12000 or 24000 chosen uniformly at random. The discharge-rate is fixed at 1000 and the boom length required to surround the vessel is also fixed at 1000. With probability 2/3 there is no oil spilled in the initial state and with probability 1/3 some is spilled, the amount chosen according to a uniform distribution between 100 and 4000. With probability 1/3 there is no sensitive area initially threatened by the oil, and with probability 2/3 an area is threatened, chosen uniformly at random from the areas that can be threatened by oil in the chosen sea-sector.

Next, the initial sea-states and some features of the sensitive areas are randomly assigned. The sea-state of a sea-sector can vary from one to six, with one meaning calm and six meaning very rough. Each sea-sector in the domain is assigned a value from one to six randomly according to the uniform distribution. With probability 1/5, each reachable sensitive area is made unthreatenable by the oil. Each reachable sensitive area is given the value 10 for shore-personnel-required, and also assigned three random values according to the uniform distribution: the diversion-boom-required is between 2000 and 10000, the exclusion-boom-required is also between 2000 and 10000 and the boom-personnel-required takes one of the values that occurs naturally in the domain: 4, 15, 18, 42 and 74.

A random number of equipment objects, for example, of types `boom`, `tractor` and `tug`, are distributed among the seaports at random. The number of objects of each type is generated according to a uniform distribution, with the minimum and maximum values show in Table B.1. In addition the properties of these objects are assigned at random, for example the max-speed and the tow-speed of each tug.

Object type	Min	Max
tug	0	4
tractor	0	4
vacuum-truck	0	8
cargo-transfer-pump	0	10
dispersant	0	4
weir-skimmer	0	10
self-mobile-skimmer	0	8
dracon-barge	0	10
dispersant-barge	0	2
tank-barge	0	4
utility-boat	1	3
boom	10	30

TABLE B.1: The number of objects in a problem is chosen randomly according to a uniform distribution. This table shows the minimum and maximum number of objects generated for each type.

This generation algorithm produces a set of problems that, along with random choices made in the problem solver, exercise all possible combinations of the strategies available to the planner while ensuring that the generated problems are physically plausible.

Bibliography

[Agre & Chapman 1987] Agre, P. E., and Chapman, D. 1987. Pengi: An implementation of a theory of activity. In *National Conference on Artificial Intelligence.*

[Allen *et al.* 1991] Allen, J. F.; Kautz, H. A.; Pelavin, R. N.; and Tenenberg, J. D. 1991. *Reasoning About Plans.* 2929 Campus Drive, Suite 260, San Mateo, Ca 94403: Morgan Kaufmann.

[Ambite & Knoblock 1997] Ambite, J. L., and Knoblock, C. A. 1997. Planning by rewriting: Efficiently generating high-quality plans. In *Proc. Ord National Conference on Artificial Intelligence.* AAAI Press.

[Bacchus *et al.* 1997] Bacchus, F.; Grove, A. J.; Halpern, J. Y.; and Koller, D. 1997. From statistical knowledge bases to degrees of belief. *Artificial Intelligence* 87:75–143.

[Bagchi, Biswas, & Kawamura 1994] Bagchi, S.; Biswas, G.; and Kawamura, K. 1994. Generating plans to succeed in uncertain environments. In Hammond, K., ed., *Proc. Second International Conference on Artificial Intelligence Planning Systems*, 1–6. University of Chicago, Illinois: AAAI Press.

[Baral 1995] Baral, C. 1995. Reasoning about actions: Non-deterministic effects, constraints, and qualification. In *Proc. 14th International Joint Conference on Artificial Intelligence*, 2017–2024. Montréal, Quebec: Morgan Kaufmann.

[Blythe & Fink 1997] Blythe, J., and Fink, E. 1997. Rasputin: A complete bidirectional-chaining planner. Technical Report forthcoming, Computer Science Department, Carnegie Mellon University.

[Blythe & Mitchell 1989] Blythe, J., and Mitchell, T. 1989. On becoming reactive. In *International Conference on Machine Learning.*

[Blythe & Veloso 1992] Blythe, J., and Veloso, M. 1992. An analysis of search techniques for a totally-ordered nonlinear planner. In Hendler, J., ed., *Proc. First International Conference on Artificial Intelligence Planning Systems.* Available as http://www.cs.cmu.edu/j̃blythe/papers/search.ps.

[Blythe & Veloso 1996] Blythe, J., and Veloso, M. 1996. Learning to improve uncertainty handling in a hybrid planning system. In Kasif, S., ed., *AAAI Fall Symposium on Learning Complex Behaviors in Intelligent Adaptive Systems*.

[Blythe & Veloso 1997] Blythe, J., and Veloso, M. 1997. Using analogy in conditional planners. In *Proc. Ord National Conference on Artificial Intelligence*. AAAI Press.

[Bonissone & Dutta 1990] Bonissone, P., and Dutta, S. 1990. Merging strategic and tactical planning in dynamic, uncertain environments. In *DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, 379–389.

[Boutilier & Dearden 1994] Boutilier, C., and Dearden, R. 1994. Using abstractions for decision-theoretic planning with time constraints. In *Proc. Ord National Conference on Artificial Intelligence*, 1016–1022. AAAI Press.

[Boutilier, Brafman, & Geib 1997] Boutilier, C.; Brafman, R. I.; and Geib, C. 1997. Prioritized goal decomposition of markov decision processes: Toward a synthesis of classical and decision theoretic planning. In *Proc. 15th International Joint Conference on Artificial Intelligence*. Nagoya, Japan: Morgan Kaufmann.

[Boutilier, Dean, & Hanks 1995] Boutilier, C.; Dean, T.; and Hanks, S. 1995. Planning under uncertainty: structural assumptions and computational leverage. In Ghallab, M., and Milani, A., eds., *New Directions in AI Planning*, 157–172. Assissi, Italy: IOS Press.

[Boutilier, Dearden, & Goldszmidt 1995] Boutilier, C.; Dearden, R.; and Goldszmidt, M. 1995. Exploiting structure in policy construction. In *Proc. 14th International Joint Conference on Artificial Intelligence*, 1104–1111. Montréal, Quebec: Morgan Kaufmann.

[Brafman 1997] Brafman, R. 1997. A heuristic variable grid solution method of pomdps. In *Proc. Ord National Conference on Artificial Intelligence*, 727–733. AAAI Press.

[Brooks 1986] Brooks, R. 1986. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation* RA-2:14–23.

[Carbonell *et al.* 1992] Carbonell, J. G.; Blythe, J.; Etzioni, O.; Gil, Y.; Joseph, R.; Kahn, D.; Knoblock, C.; Minton, S.; Pérez, A.; Reilly, S.; Veloso, M.; and Wang, M. 1992. PRODIGY4.0: The manual and tutorial. Technical Report CMU-CS-92-150, School of Computer Science, Carnegie Mellon University.

[Cassandra, Kaelbling, & Littman 1994] Cassandra, A. R.; Kaelbling, L. P.; and Littman, M. L. 1994. Acting optimally in partially observable stochastic domains. In *Proc. Ord National Conference on Artificial Intelligence*, 1023–1028. AAAI Press.

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

[Cooper 1990] Cooper, G. F. 1990. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence* 42:393–405.

[Darwiche 1995] Darwiche, A. 1995. Conditioning methods for exact and approximate inference in causal networks. In Besnard, P., and Hanks, S., eds., *Proc. Eleventh Conference on Uncertainty in Artificial Intelligence*, 99–107. Montreal, Quebec: Morgan Kaufmann.

[Dean & Lin 1995] Dean, T., and Lin, S.-H. 1995. Decomposition techniques for planning in stochastic domains. In *Proc. 14th International Joint Conference on Artificial Intelligence*, 1121 – 1127. Montréal, Quebec: Morgan Kaufmann.

[Dean & Wellman 1991] Dean, T. L., and Wellman, M. P. 1991. *Planning and Control.* Morgan Kaufmann.

[Dean *et al.* 1993] Dean, T.; Kaelbling, L. P.; Kirman, J.; and Nicholson, A. 1993. Planning with deadlines in stochastic domains. In *National Conference on Artificial Intelligence*, National Conference on Artificial Intelligence.

[Dean *et al.* 1995] Dean, T.; Kaelbling, L.; Kirman, J.; and Nicholson, A. 1995. Planning under time constraints in stochastic domains. *Artificial Intelligence* 76(1-2):35–74.

[Dean 1994] Dean, T. 1994. Decision-theoretic planning and markov decision processes. To be submitted to AI Magazine.

[Dearden & Boutilier 1997] Dearden, R., and Boutilier, C. 1997. Abstraction and approximate decision theoretic planning. *Artificial Intelligence* To appear.

[Desimone & Agosta 1994] Desimone, R. V., and Agosta, J. M. 1994. Spill response system configuration study — final report. Technical Report ITAD-4368-FR-94-236, SRI International.

[Draper & Hanks 1994] Draper, D., and Hanks, S. 1994. Localized partial evaluation of belief networks. In de Mantaras, R. L., and Poole, D., eds., *Proc. Tenth Conference on Uncertainty in Artificial Intelligence*, 170–177. Seattle, WA: Morgan Kaufmann.

[Draper, Hanks, & Weld 1994] Draper, D.; Hanks, S.; and Weld, D. 1994. Probabilistic planning with information gathering and contingent execution. In Hammond, K., ed., *Proc. Second International Conference on Artificial Intelligence Planning Systems*, 31–37. University of Chicago, Illinois: AAAI Press.

[Drummond & Bresina 1990] Drummond, M., and Bresina, J. 1990. Anytime synthetic projection: Maximizing the probability of goal satisfaction. In *Proc. Ord National Conference on Artificial Intelligence*, 138–144. AAAI Press.

[Etzioni 1990] Etzioni, O. 1990. *A Structural Theory of Explanation-Based Learning.* Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University. Available as technical report CMU-CS-90-185.

[Feldman & Sproull 1977] Feldman, J. A., and Sproull, R. F. 1977. Decision theory and artificial intelligence ii: The hungy monkey. *Cognitive Science* 1:158–192.

[Fikes & Nilsson 1971] Fikes, R. E., and Nilsson, N. J. 1971. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

[Fink & Veloso 1995] Fink, E., and Veloso, M. 1995. Formalizing the prodigy planning algorithm. In Ghallab, M., and Milani, A., eds., *New Directions in AI Planning*, 261–272. Assissi, Italy: IOS Press.

[Firby 1987] Firby, J. R. 1987. An investigation into reactive planning in complex domains. In *Proc. Ord National Conference on Artificial Intelligence*, 202–206. AAAI Press.

[Firby 1989] Firby, J. R. 1989. *Adaptive Execution in Complex Dynamic Worlds.* Ph.D. Dissertation, Department of Computer Science, Yale University.

[Georgeff & Lansky 1987] Georgeff, M. P., and Lansky, A. L. 1987. Reactive reasoning and planning. In *Proc. Ord National Conference on Artificial Intelligence*, 677–681. AAAI Press.

[Gerevini & Schubert 1996] Gerevini, A., and Schubert, L. 1996. Accelerating partial-order planners: Some techniques for effective search control and pruning. *Journal or Artificial Intelligence Research* 5:95–137.

[Gervasio & DeJong 1994] Gervasio, M. T., and DeJong, G. F. 1994. An incremental learning approach for completable planning. In *International Conference on Machine Learning.*

[Gervasio 1996] Gervasio, M. T. 1996. *An Incremental Curative Learning Approach for Planning with Incorrect Domain Theories.* Ph.D. Dissertation, University of Illinois at Urbana-Chapmaign.

[Gil 1992] Gil, Y. 1992. *Acquiring Domain Knowledge for Planning by Experimentation.* Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University. Available as technical report CMU-CS-92-175.

[Goldman & Boddy 1994] Goldman, R. P., and Boddy, M. S. 1994. Epsilon-safe planning. In de Mantaras, R. L., and Poole, D., eds., *Proc. Tenth Conference on Uncertainty in Artificial Intelligence*, 253–261. Seattle, WA: Morgan Kaufmann.

[Goldszmidt & Darwiche 1995] Goldszmidt, M., and Darwiche, A. 1995. Plan simulation using bayesian networks. In *IEEE-CAIA*.

[Goodwin 1994] Goodwin, R. 1994. Reasoning about when to start acting. In Hammond, K., ed., *Proc. Second International Conference on Artificial Intelligence Planning Systems*, 86–91. University of Chicago, Illinois: AAAI Press.

[Haddawy & Suwandi 1994] Haddawy, P., and Suwandi, M. 1994. Decision-theoretic refinement planning using inheritance abstraction. In Hammond, K., ed., *Proc. Second International Conference on Artificial Intelligence Planning Systems.* University of Chicago, Illinois: AAAI Press.

[Haddawy, Doan, & Goodwin 1995] Haddawy, P.; Doan, A.; and Goodwin, R. 1995. Efficient decision-theoretic planning: Techniques and empirical analysis. In Besnard, P., and Hanks, S., eds., *Proc. Eleventh Conference on Uncertainty in Artificial Intelligence*, 229–326. Montreal, Quebec: Morgan Kaufmann.

[Haddawy, Doan, & Kahn 1996] Haddawy, P.; Doan, A.; and Kahn, C. E. 1996. Decision-theoretic refinement planning in medical decision making: Management of acute deep venous thrombosis. *Medical Decision Making.*

[Haddawy 1991] Haddawy, P. 1991. *Representing Plans Under Uncertainty: a Logic of Time, Chance and Action.* Ph.D. Dissertation, University of Illinois at Urbana-Champaign.

[Hickman, Shell, & Carbonell 1990] Hickman, A. K.; Shell, P.; and Carbonell, J. G. 1990. Internal analogy: Reducing search during problem solving. In Copetas, C., ed., *The Computer Science Research Review 1990.* The School of Computer Science, Carnegie Mellon University.

[Howard 1960] Howard, R. A. 1960. *Dynamic Programming and Markov Processes.* MIT Press.

[Kartha 1995] Kartha, G. N. 1995. *A Mathematical Investigation of Reasoning About Actions.* Ph.D. Dissertation, University of Texas at Austin.

[Kautz & Selman 1996] Kautz, H. A., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. Ord National Conference on Artificial Intelligence.* AAAI Press.

[Knoblock 1991] Knoblock, C. A. 1991. *Automatically Generating Abstractions for Problem Solving.* Ph.D. Dissertation, Carnegie Mellon University.

[Koenig & Simmons 1995] Koenig, S., and Simmons, R. 1995. Real-time search in nondeterministic domains. In *Proc. 14th International Joint Conference on Artificial Intelligence*, 1660–1667. Montréal, Quebec: Morgan Kaufmann.

[Kushmerick, Hanks, & Weld 1994] Kushmerick, N.; Hanks, S.; and Weld, D. 1994. An algorithm for probabilistic least-commitment planning. In *Proc. Ord National Conference on Artificial Intelligence*, 1073–1078. AAAI Press.

[Kushmerick, Hanks, & Weld 1995] Kushmerick, N.; Hanks, S.; and Weld, D. 1995. An algorithm for probabilistic planning. *Artificial Intelligence* 76:239 – 286.

[Lauritzen & Spiegelhalter 1988] Lauritzen, S., and Spiegelhalter, D. 1988. Local computation with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society* B 50(2):154–227.

[Littman 1996] Littman, M. L. 1996. *Algorithms for Sequential Decision Making.* Ph.D. Dissertation, Department of Computer Science, Brown University.

[Loyall & Bates 1991] Loyall, A. B., and Bates, J. 1991. Hap: A reactive, adaptive architecture for agents. Technical Report CMU-CS-91-147, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[Mansell 1993] Mansell, T. M. 1993. A method for planning given uncertain and incomplete information. In Heckerman, D., and Mamdani, A., eds., *Uncertainty in Artificial Intelligence*, volume 9, 350–358. Washington, D.C.: Morgan Kaufmann.

[McAllester & Rosenblitt 1991] McAllester, D., and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proc. Ord National Conference on Artificial Intelligence*, 634–639. AAAI Press.

[Minton *et al.* 1989] Minton, S.; Carbonell, J. G.; Knoblock, C. A.; Kuokka, D. R.; Etzioni, O.; and Gil, Y. 1989. Explanation-based learning: A problem solving perspective. *Artificial Intelligence* 40:63–118.

[Minton 1988] Minton, S. 1988. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* Boston, MA: Kluwer.

[Newell & Simon 1963] Newell, A., and Simon, H. A. 1963. Gps: a program that simulates human thought. In Feigenbaum, E. A., and Feldman, J., eds., *Computers and Thought*, 279–293. New York: McGraw-Hill.

[Onder & Pollack 1997] Onder, N., and Pollack, M. 1997. Contingency selection in plan generation. In *European Conference on Planning.*

[Parr & Russell 1995] Parr, R., and Russell, S. 1995. Approximating optimal policies for partially observable stochastic domains. In *Proc. 14th International Joint Conference on Artificial Intelligence.* Montréal, Quebec: Morgan Kaufmann.

[Pearl 1988] Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems.* Morgan Kaufmann.

[Pearl 1994] Pearl, J. 1994. A probabilistic calculus of actions. In de Mantaras, R. L., and Poole, D., eds., *Proc. Tenth Conference on Uncertainty in Artificial Intelligence*, 454–462. Seattle, WA: Morgan Kaufmann.

[Penberthy & Weld 1992] Penberthy, J. S., and Weld, D. S. 1992. Ucpop: A sound, complete, partial order planner for adl. In *Third International Conference on Principles of Knowledge Representation and Reasoning*, 103–114.

[Peot & Smith 1992] Peot, M. A., and Smith, D. E. 1992. Conditional nonlinear planning. In Hendler, J., ed., *Proc. First International Conference on Artificial Intelligence Planning Systems*, 189–197. College Park, Maryland: Morgan Kaufmann.

[Pérez & Carbonell 1994] Pérez, M. A., and Carbonell, J. 1994. Control knowledge to improve plan quality. In Hammond, K., ed., *Proc. Second International Conference on Artificial Intelligence Planning Systems*, 323–328. University of Chicago, Illinois: AAAI Press.

[Pérez 1995] Pérez, M. A. 1995. *Learning Search Control Knowledge to Improve Plan Quality.* Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University.

[Pollack, Joslin, & Paolucci 1997] Pollack, M. E.; Joslin, D.; and Paolucci, M. 1997. Flaw selection strategies for partial-order planning. *Journal of Artificial Intelligence Research* 6:223–262.

[Pryor & Collins 1993] Pryor, L., and Collins, G. 1993. Cassandra: Planning for contingencies. Technical Report 41, The Institute for the Learning Sciences.

[Pryor & Collins 1996] Pryor, L., and Collins, G. 1996. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research* 4:287–339.

[Puterman 1994] Puterman, M. 1994. *Markov Decision Processes : Discrete Stochastic Dynamic Programming.* John Wiley & Sons.

[Russell & Wefald 1991] Russell, S., and Wefald, E. 1991. *Do the Right Thing.* MIT Press.

[Sacerdoti 1974] Sacerdoti, E. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5:115–135.

[Schoppers 1989a] Schoppers, M. J. 1989a. In defense of reaction plans as caches. *AI Magazine* 10(4):51–60.

[Schoppers 1989b] Schoppers, M. J. 1989b. *Representation and Automatic Synthesis of Reaction Plans*. Ph.D. Dissertation, University of Illinois at Urbana-Champaign. Available as technical report UIUCDCS-R-89-1546.

[Shanahan 1995] Shanahan, M. 1995. A circumscriptive calculus of events. *Artificial Intelligence* 75(2).

[Stone, Veloso, & Blythe 1994] Stone, P.; Veloso, M.; and Blythe, J. 1994. The need for different domain-independent heuristics. In Hammond, K., ed., *Proc. Second International Conference on Artificial Intelligence Planning Systems*, 164–169. University of Chicago, Illinois: AAAI Press.

[Tash & Russell 1994] Tash, J. K., and Russell, S. 1994. Control strategies for a stochastic planner. In *Proc. Ord National Conference on Artificial Intelligence*, 1079–1085. AAAI Press.

[Tate 1977] Tate, A. 1977. Generating project networks. In *International Joint Conference on Artificial Intelligence*.

[Veloso & Stone 1995] Veloso, M., and Stone, P. 1995. Flecs: Planning with a flexible commitment strategy. *Journal of Artificial Intelligence Research* 3:25–52.

[Veloso *et al.* 1995] Veloso, M.; Carbonell, J.; Pérez, A.; Borrajo, D.; Fink, E.; and Blythe, J. 1995. Integrating planning and learning: The prodigy architecture. *Journal of Experimental and Theoretical AI* 7:81–120.

[Veloso 1992] Veloso, M. M. 1992. *Learning by Analogical Reasoning in General Problem Solving*. Ph.D. Dissertation, Carnegie Mellon University.

[Veloso 1994] Veloso, M. M. 1994. *Planning and Learning by Analogical Reasoning*. Springer Verlag.

[Vere 1983] Vere, S. 1983. Planning in time: Windows and durations for activities and goals. *IEEE Trans. Pattern Analysis and Machine Intelligence* 5(3):246–67.

[Verma 1986] Verma, T. S. 1986. Causal networks: Semantics and expressiveness. Technical Report R-65, UCLA Cognitive Systems Laboratory.

[Wang 1996] Wang, X. 1996. *Learning Planning Operators by Observation and Practice*. Ph.D. Dissertation, Carnegie Mellon University.

[Washington 1994] Washington, R. 1994. *Abstraction planning in real time*. Ph.D. Dissertation, Stanford University.

[Wellman 1990a] Wellman, M. P. 1990a. *Formulation of Tradeoffs in Planning Under Uncertainty*. Pitman.

[Wellman 1990b] Wellman, M. P. 1990b. The strips assumption for planning under uncertainty. In *Proc. Ord National Conference on Artificial Intelligence*, 198–203. AAAI Press.

[Williamson & Hanks 1994] Williamson, M., and Hanks, S. 1994. Optimal planning with a a goal-directed utility model. In Hammond, K., ed., *Proc. Second International Conference on Artificial Intelligence Planning Systems*, 176–181. University of Chicago, Illinois: AAAI Press.

[Williamson 1996] Williamson, M. 1996. *A Value-directed Approach to Planning.* Ph.D. Dissertation, University of Washington.

[Yang & Tenenberg 1990] Yang, Q., and Tenenberg, J. D. 1990. Abtweak, abstracting a nonlinear, least commitment planner. In *Proc. Ord National Conference on Artificial Intelligence*, 204–209. AAAI Press.

[Zilberstein 1993] Zilberstein, S. 1993. *Operational Rationality through Compilation of Anytime Algorithms*. Ph.D. Dissertation, University of California at Berkeley.