ABSTRACTION AND SEARCH FOR DECISION-THEORETIC PLANNING

By

Richard William Dearden

B. Sc. (Computer Science) Victoria University of Wellington, 1990

B. Sc. (Hons.) (Computer Science) Victoria University of Wellington, 1991

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
Department of
COMPUTER SCIENCE

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October 1994

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature)

Department of COMPUTER SCIENCE

The University of British Columbia
Vancouver, Canada

Date OCTOBER 13 1994

Abstract

We investigate the use Markov Decision Processes as a means of representing worlds in which actions have probabilistic effects. Markov Decision Processes provide many representational advantages over traditional planning representations. As well as being able to represent actions with more than one possible result, they also provide a much richer way to represent good and bad states of the world. Conventional approaches for finding optimal plans for Markov Decision Processes are computationally expensive and generally impractical for the large domains and real-time requirements of many planning applications. For this reason, we have concentrated on producing approximately optimal plans using a minimal amount of computation.

We describe two complementary methods for planning. The first is to generate approximately optimal plans using abstraction. By ignoring certain features of a planning problem, we can create a smaller problem for which an optimal plan can be efficiently found by conventional means. The plan for this smaller problem can be directly applied to the original problem, and also provides an estimate of the value of each possible state of the world. Our second technique uses these estimates as a heuristic, and applies game-tree search techniques to try to determine a better action to perform in the current state of the system. By repeatedly choosing an action to perform by searching, and executing the action, we provide a planning algorithm which has a complexity that is independent of the number of possible states of the world.

Table of Contents

List of Tables

List of Figures

Acknowledgement

This thesis has been a true collaboration between my supervisor Craig Boutilier and myself. Craig's enthusiasm and interest have had a profound influence on this work, he first pointed me at the work of Dean et al. which provided much of our motivation to work with Markov Decision Processes. Indeed, some of the work in this thesis is at least as much Craig's research as it is my own. Craig has also been an ideal supervisor, always prepared to talk, and usually full of suggestions on ways to improve things. I can't thank Craig enough for his commitment to his students, and contributions to this research.

I should also thank my reader, David Poole. His knowledge of decision theory and probabilistic reasoning is encyclopaedic, and he has been an invaluable resource throughout this research.

Fiona Humphris also deserves thanks. She helped proofread this thesis, and provided endless encouragement and understanding. Without her, this thesis might never have come into existence. My parents have also showed me nothing but encouragement in the pursuit of this degree.

Finally I would like to thank the faculty, staff and graduate students of the UBC Computer Science department. I have yet to find a better environment for performing research. In particular I would like to thank Michael Horsch, Roger Ford, and Brent Boerlage for comments on various parts of this work, and Paul Lalonde and Graham Denham for providing a place where I could escape from it.

Chapter 1

Introduction

The goal of research in artificial, or computational intelligence is to create an agent which, at least in some limited sense, can be said to behave intelligently. What we mean by the word "behave" varies widely from problem to problem, but in its most general sense, "behaving intelligently" means carrying out some sequence of actions with a rational intent based on information about the agent's environment. The process of choosing the sequence of actions to perform is planning. Informally, planning is the problem of determining, given a set of objectives and a set of actions, a sequence of actions to perform which will achieve some or all of the objectives.

As an example of a planning problem that we might want to solve, imagine a robot that tries to keep its user supplied with coffee. The robot has a goal that it must achieve, that of ensuring the user has coffee, it has a set of actions such as buying coffee, delivering coffee, and moving to the user's office that it can perform, and it has certain information about the current state of the world, for example that the user wants coffee, that the robot is at the coffee shop, and that it is sunny outside. The problem for the robot is to plan a sequence of actions that will make the goal true.

Planning for problems of this type has been investigated in artificial intelligence for many years. One of the earliest planning systems is STRIPS (Fikes and Nilsson 1971). STRIPS uses goal-directed search to plan. The planner begins with the goal state, and searches backwards through the state space until it reaches the initial state. The plan it produces is a linear series of actions which will accomplish the goal. Although not all

classical planners fit precisely into this description — for example, many produce partial orders over actions, rather than total orders, and may find plans that will succeed from a set of starting states, rather than a single one — most make a number of assumptions about the problems they will be used for. In this thesis we are particularly interested in relaxing the following assumptions, and building planning systems that can operate in domains where they do not hold.

Agents achieve goals The definition of a classical planning problem is in terms of 1) the initial state of the world (this need not be unique), 2) a set of actions the agent can perform, and 3) a description of a state or states which constitute the agent's goal. The agent's objective is to produce a sequence of actions which when executed in the initial state (or any initial state if there could be more than one), will leave the world in a goal state. While this representation of states as either goal states or normal states is adequate for some domains, we would typically expect a much richer way of describing how good or bad certain states are. For example, in the coffee robot domain we described above, as well as its main objective of delivering coffee the robot may receive rewards based on how hot the coffee is, whether it disturbed people on its way to get the coffee, etc. Although its goal is to bring the user coffee that is as hot as possible while disturbing as few people as possible, the agent may find that these goals conflict, and hence must reason about which is most important.

Actions always work Classical planning assumes that the results of all actions are deterministic. For example, if an agent carries out the action *Pickup block A*, the resulting state of the world is that in which the agent is holding block A (assuming that it is possible for the agent to pick up the block), and this fact is known with certainty. In many real-world situations, this is a very unreasonable assumption,

the block might slip from the agent's grasp, the agent might make a sensing error and pick up block B instead of A, and so on. Although the most likely outcome may be that the agent is left holding the block, an agent that only considers the most probable result of a real-world action would be courting disaster. By considering the probability and effects of other outcomes, and by making observations of the world to determine the outcome that actually occurred when an action is carried out, the agent can avoid actions with unlikely but catastrophic results, and can also plan to achieve states which only occur rarely.

The whole world is visible Another important assumption of classical planning systems is that the state of the world is completely observable. The agent always knows exactly what is true at any given time. While this assumption may well be true in some domains — some scheduling problems are an example of this — in general it will not be. While planning systems which use this assumption can be useful in a restricted set of problems, a general planner must still be able to operate with partial information about the world.

This thesis examines one approach to planning which addresses at least the first two of these issues, and makes some progress on the third.

We are particularly interested in computationally efficient planning. As we shall see, there are already well-developed planning algorithms for the domain representations which we are interested in, but their computational requirements are often polynomial in the number of states in the system. Many AI planning problems are represented in terms of propositions, and hence have exponentially many possible states (n atoms result in 2^n possible worlds). A real-time planning algorithm for such a problem must perform considerably better than exponentially in the number of atoms.

In summary, the motivation for this work is to build a planning system which operates

in probabilistic worlds with a rich structure of rewards and penalties. The system should have reasonable computational requirements, and should be designed for use in time critical domains.

1.1 Decision Theoretic Planning

Decision-theoretic planning is an attempt to combine the artificial intelligence field of automated planning with decision theory. Although researchers have been interested in this idea for some time, only recently have the problems become well-posed. Decision theory allows us to reason not only about whether a plan satisfies some goal or not, but also about how good a plan is — how quickly or cheaply it satisfies the goal, how probable it is that the goal will have been satisfied by performing the plan, and how valuable the particular goal achieved is, compared with other goals that can be achieved by other plans. The ability to compare two plans and decide which one does best is a great asset that many classical planning algorithms cannot provide.

Much of the research that has gone on in decision-theoretic planning has addressed the class of problems associated with Markov Decision Processes (MDPs) (Howard 1971) (see (Haddawy and Hanks 1992) for an alternative approach). MDPs are a form of stochastic automata where every state has an associated numerical value or *reward*. Most recent decision-theoretic planning algorithms use an MDP model in which there are a finite number of states, a finite number of actions (each of which is a mapping from states to probability distributions over the states) and a reward function (a mapping from states to real numbers). An agent accumulates rewards as it moves through the MDP by performing actions. Although other classes of problems can be addressed using decision-theoretic methods, MDPs provide a representation which is general enough to be used for many interesting problems. They have been extensively studied in Operations Research,

so a number of well understood algorithms for computing plans are available, at least for simple forms of MDP.

Since the problems of interest in MDP planning are generally stochastic dynamical systems, the standard approach to planning is to create a universal plan, referred to as a *policy*, which indicates which action to perform in every situation that could be encountered. For AI planning, computing an optimal policy is generally far too expensive, so most decision-theoretic planning algorithms sacrifice optimality (although some will converge on an optimal policy if enough computation time is available), preferring to produce either a nearly optimal policy for a subset of the state space, or a policy that is applicable in any state, but may be far from optimal.

Artificial intelligence research often concentrates on "tractable islands" — problems which have additional structure that can be used to reduce the computation required to find solutions to reasonable levels. The challenge of applying this type of approach to Markov decision processes is a major motivation for this thesis, and has motivated a number of other researchers in this area (Dean et al. 1993b; Nicholson and Kaelbling 1994). The following have been identified as one set of characteristics that make generating approximately optimal policies easier. Although these are from (Dean et al. 1993b), they also apply to our approach, and a number of other MDP algorithms.

- There are relatively many policies which have high (although not necessarily optimal) value. Since most algorithms do not produce optimal policies, some less-than-optimal policies must be reasonably good.

- From any given state, there are few states to which transitions can be made. In other words, there are relatively few valid actions for each state, and more importantly, actions have few possible outcomes.

- The value of a state can be estimated by considering the values of nearby states.

Nearby states are those for which the expected number of steps to reach one from the other is small.

1.2 Abstraction and Search in Markov Decision Process Planning

As Chapter 2 shows, Markov processes are extremely general, and somewhat laborious to specify. In practice we expect most MDP planning problems not to be specified directly as the MDP itself, but in a more compact representation which makes the structure in the problem explicit. In Section 2.2 we suggest one such representation, other researchers have proposed similar models. The representation we have chosen is in terms of logical propositions. Thus a state in the MDP represents a possible world in the logical model (an assignment of truth values to all the propositions).

A second consideration when developing our approach was the problem of planning in real-time. Since well developed methods already exist for finding optimal policies for MDPs, any contribution from Artificial Intelligence must be directed at taking advantage of structure for computational gains, and at finding close-to-optimal plans efficiently. Designing "anytime" algorithms that always produce a plan, but produce better ones if more time is available has also motivated this thesis.

Although we are very interested in the problem of how to plan if the agent does not know the state of the world, this is an extremely difficult problem, and for the purposes of this thesis, we will concentrate instead on domains where the agent always knows what the current state of the world is, even though it cannot predict exactly the outcome of its actions. In the terminology of decision theory, we say our agents operate in *completely observable* worlds.

Our approach is twofold. Before any actions are performed, we use relatively small amounts of computation to produce a policy which we will treat as a set of *default*

reactions, actions that the agent will do if there is no time to find a better plan. To do this we use an idea from classical planning, namely *abstraction*. The idea of using abstraction is to solve a large problem by constructing smaller but closely related problems, and using the solution to a small problem as the basis for a solution to a larger problem. In the case of our algorithm, we ignore some of the atoms in the propositional representation of the problem to create a smaller MDP which we will refer to as the *abstract MDP*. Because of the way we choose the atoms to ignore, an optimal policy for this small MDP can also be used as a (usually sub-optimal) policy for the original problem.

The second part of our approach is performed at run-time. Since the policy we found using abstraction is usually not optimal, we use search to try to find a better strategy for the current state. By interleaving search with execution — search for the best action to do now, perform it, and then search for the best action in the new state — we can reduce the size of the search considerably, and if there is no time for searching, we still have the default reactions we computed with the abstraction algorithm.

We can compute theoretical bounds on the difference in value between an optimal policy for the original MDP, and a policy produced from any abstract MDP. With these bounds a user can trade off computation time for policy accuracy by choosing an appropriate abstraction. Similarly, the depth of the search tree can be adjusted to fit the available time before the next action must be performed, and actions can be cached to avoid recalculation of the best action when revisiting a state. These properties contribute to making the algorithm ideally suited for time-critical domains.

1.3 Organization of this Thesis

This chapter has introduced the area of decision-theoretic planning, described the motivation behind this work, and briefly outlined the contribution of this thesis. Chapter 2

describes the MDP model in considerably more detail, examines the propositional representation we use for domains, and provides a summary of related work in classical planning, search, and decision-theory as well as in decision-theoretic planning itself.

Chapter 3 is a detailed description of the abstraction algorithm, including theoretical results concerning the value of abstract policies, and experimental results that demonstrate the algorithm in operation. Similarly, Chapter 4 describes the search algorithm. Chapter 5 discusses how the two algorithms combine to form a complete planning system, and describes some of the future research this thesis leads towards.

Chapter 2

Markov Decision Processes for Planning

In this chapter we will describe the basic characteristics of *Markov Decision Processes* (MDPs), compact representations for them, and a standard approach from Operations Research for planning using them. For reasons of simplicity, we will only concern ourselves with completely observable MDPs as this research has almost entirely concentrated on this model. Although partially observable MDPs can accurately represent many more real world problems, and a much larger class of situations, they are computationally much more difficult to deal with. We will leave our discussion of the partially observable case to Chapter 5.

Finite automata provide a general way of representing problems that include an explicit state space, and a set of actions each of which is a mapping from state to state. If the actions are probabilistic — mappings from a state to a probability distribution over all states — such a model is called a stochastic automata. A Markov Decision Process is a stochastic automata in which each state has a numerical reward or cost associated with it (we will use the term reward to describe both rewards and costs, costs can be considered as negative rewards). At each state, a decision is made about what action to perform, with the objective being to maximize the total accumulated reward over a series of such decisions. An MDP represents all possible worlds explicitly as states, represents actions as stochastic transitions from state to state, and represents objectives or goals in terms of rewards associated with states. Although rewards and costs can be associated with actions as well as states, we will not consider that case here.

There are many advantages to using MDPs for decision theoretic planning. They are very general, and allow a wide variety of problems to be represented. The reward structure is much more flexible than traditional goal representations for planning, and MDPs are much better than classical planning at representing processes, so they can be easily used to plan in systems that must keep some condition or conditions true as much as possible, or deal with an infinite sequence of objectives. MDPs have long been used in Operations Research, and well developed algorithms exist for finding optimal universal plans or *policies.*

One difficulty with using MDPs for planning is that the Markov Property (see below) must hold in the domain. Although a model that conforms to the Markov Property can be designed for any domain, in practice this may result in a much larger state space. Another problem is that it is generally not possible to take advantage of the structure of a domain because it is hidden in the MDP model of it. By using more compact representations of domains we can take advantage of certain structure when planning (see Chapter 3).

2.1 The MDP model

For our purposes, an MDP can be defined by the tuple $< \mathcal{S}, \mathcal{A}, T, R >$, where $\mathcal{S}$ is a finite set of world states that the agent can distinguish, $\mathcal{A}$ is a finite set of actions the agent can perform, T is a state transition function that describes the effects of each action, and R is the reward function. Although this need not be the case in general, we will assume that the set of *feasible actions* for any state $s \in \mathcal{S}$ is $\mathcal{A}$ — every action can be attempted in any state. T is a mapping from $\mathcal{S} \times \mathcal{A}$ into discrete probability distributions over $\mathcal{S}$. We write $\Pr(s_2|A, s_1)$ for the probability that state s_2 results if the agent performs action A in state s_1. As we would expect, $0 \leq \Pr(t|A, s) \leq 1$ for all s, t, and for all

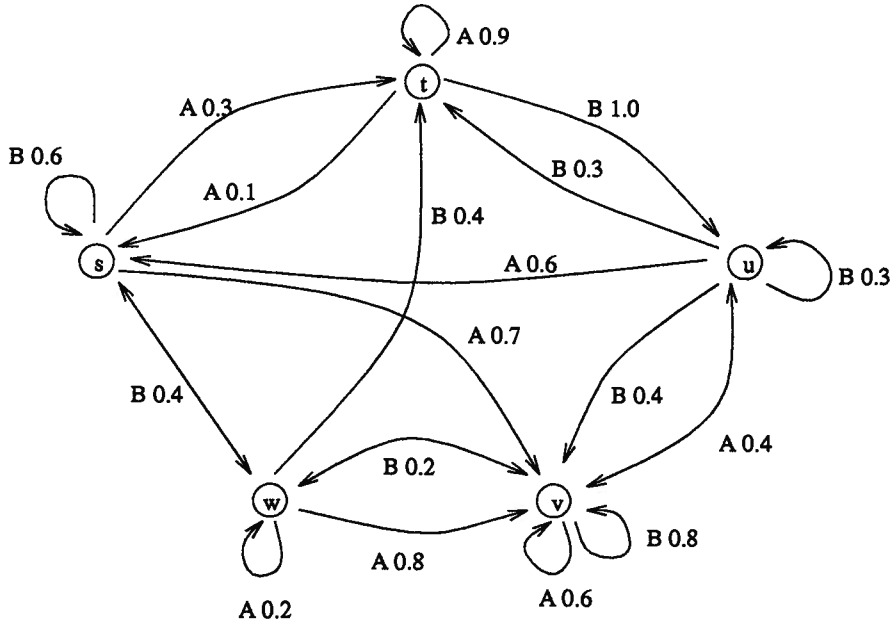

Figure 2.1: An example of an MDP with 5 states and 2 actions.

s, $\sum_{t \in \mathcal{S}} \Pr(t|A,s) = 1$. R is a mapping from $\mathcal{S}$ to $\Re$ which specifies the instantaneous reward the agent gains for entering a state. We will write $R(s)$ to signify the reward for entering state s. By our definition of T, we ensure that the *Markov Property* holds.

Definition 1 [Markov Property] A tuple $< \mathcal{S}, \mathcal{A}, T, R >$ has the *Markov Property* if the result of performing any action $A \in \mathcal{A}$ depends only on the current state $s \in \mathcal{S}$, not on the sequence of actions that left the world in state s.

Figure 2.1 shows an example MDP depicted as a directed graph. Each arrow represents a possible state-to-state transition, and is labeled with the action that could cause the transition to be used, and the probability that it will be used given that action is performed. For example, from state u, if action B is performed, the transition from u to itself will take place with probability 0.3. Here $\mathcal{S}$ is $\{s, t, u, v, w\}$, $\mathcal{R}$ is $\{A, B\}$. T for action A is:

From	*To*	s	t	u	v	w
s		0	0.3	0	0.7	0
t		0.1	0.9	0	0	0
u		0.6	0	0	0.4	0
v		0	0	0.4	0.6	0
w		0	0	0	0.8	0.2

One possible reward function for the MDP is: $R(s) = R(t) = R(u) = R(v) = -1, R(w) = 0$. This produces an MDP where the best actions to take are those that put the world in state w with the minimum number of actions, and then keep it there. The optimal policy for this MDP is to perform action A in states u and w, and action B otherwise.

2.2 Propositional Domains

In many planning problems, we describe the state of the world using a set of propositional atoms. For example, we might say that *Raining* is false in the current world, but that *Cloudy* is true. This section discusses the propositional model of actions that we have chosen to use. In translating such a scheme into a MDP, we make each state in the network correspond to a possible world, so for a world represented by n propositional atoms, the resulting MDP contains 2^n states.

2.2.1 The extended STRIPS model

The representation described here is based on that used in BURIDAN (Kushmerick, Hanks and Weld 1993) with the addition of a method for representing independence between different aspects of the outcome of an action.

In the classical STRIPS (Fikes and Nilsson 1971) representation, actions are represented with a list of preconditions and a list of effects made up of literals which will become true or false if the action is executed. If the preconditions are true, and the action is performed, then the effects list is applied to the current world to determine the new world. In BURIDAN, this is extended by adding multiple mutually exclusive and exhaustive preconditions which we will refer to as *discriminants* (, and by making the effects probabilistic. For each action and discriminant, there is a set of effects lists with associated probabilities. The probabilities sum to one, and represent the probability that the corresponding effects actually occur given that the action is performed with the discriminant true. Table 2.1 shows a problem represented in this way. If we look at the *BuyCoffee* action, we see that it has two discriminants. If *Office* is true, the with probability 1, there will be no changes to the state, while if *Office* is false, then with probability 0.8 *HRC* will be true after executing the action, regardless of the previous value of *HRC*, and with probability 0.2, there will be no effect. If s is the current state (represented as the set of literals true in that state), and $\mathcal{E}$ is the effect list to be applied, then the new state that results is:

$$E(s) = (s \setminus \{p | \neg p \in \mathcal{E}\}) \cup \mathcal{E}$$

We have extended the BURIDAN representation slightly in two ways. Firstly, we have added what we call *action aspects*. These are designed to represent the fact that in some of the effects of an action are dependent on only some of the preconditions. For example, in Table 2.1, the *Move* action has two aspects. The first represents the fact that when the agent performs a *Move*, the resulting location depends on the agent's current location only. It is independent of the values of *Rain* and *Umbrella*. The second aspect deals with whether the agent becomes wet or not. Since this is independent of where the agent is, the precondition only contains *Rain* and *Umbrella*. Actions with multiple action aspects

Table 2.1: An example domain presented as STRIPS-style action descriptions. (a) is the action description for the problem, while (b) is the reward function. Note that HUC and HRC are HasUserCoffee and HasRobotCoffee respectively. UserIsThirsty* is a random event.

Action	*Discriminant*	*Effect*	*Prob.*
Move	Office	¬Office	0.9
			0.1
	¬Office	Office	0.9
			0.1
Move	Rain,¬Umb	Wet	0.9
			0.1
BuyCoffee	¬Office	HRC	0.8
			0.2
	Office		1.0
GetUmbrella	Office	Umbrella	0.9
			0.1
DelCoffee	Office,HRC	HUC,¬HRC	0.8
		¬HRC	0.1
			0.1
	¬Office,HRC	¬HRC	0.8
			0.2
	¬HRC		1.0
UserIsThirsty*		¬HUC	0.01
			0.99

(a)

Sentence	*Value*	*Sentence*	*Value*
HUC, ¬Wet	1.0	¬HUC, ¬Wet	0.2
HUC, Wet	0.8	¬HUC, Wet	0.0

(b)

Table 2.2: Translating action aspects and random events into single actions. (a) is a summary of both aspects of the *Move* action, while (b) is the combination of the random event *UserIsThirsty** with the *GetUmbrella* action.

Action	*Discriminant*	*Effect*	*Prob.*
Move	Office,Rain,¬Umb	¬Office,Wet	0.81
		Wet	0.09
		¬Office	0.09
			0.01
	Office,¬(Rain,¬Umb)	¬Office	0.9
			0.1
	¬Office,Rain,¬Umb	Office,Wet	0.81
		Wet	0.09
		Office	0.09
			0.01
	¬Office,¬(Rain,¬Umb)	Office	0.9
			0.1

(a)

Action	*Discriminant*	*Effect*	*Prob.*
GetUmbrella	Office	Umbrella	0.891
		Umbrella,¬HUC	0.009
		¬HUC	0.001
			0.099
	¬Office	¬HUC	0.01
			0.99

(b)

can be translated into actions with a single aspect by forming the "cross-product" of their effects. Table 2.2 (a) shows the translated form of the *move* action from the example.

The second extension of the BURIDAN model is the addition of *random events.* These are actions over which the agent has no control and which (in our rather naive model) occur randomly at the same time as some action of the agent. If we imagine a situation where an agent must keep some process running smoothly, random events represent things that occur during the process that the agent must deal with. As with action aspects, random events can be combined with normal actions by forming the "cross-product" of

Table 2.3: Parts of two actions to be translated into a single action. A_1 is a normal action, A^* is a random event, and A'_1 is the new action produced by combining them. Discriminants D_1 and D_2 are combined into a single new discriminant. The $\diamond$ operator is defined on two sets of literals E and F such that $E \diamond F \equiv E \cup \{F \setminus \{l : \neg l \in E\}\}$

Action	*Discriminant*	*Effect*	*Prob.*
A_1	D_1	$E_{1,1}$	$p_{1,1}$
		$E_{1,2}$	$p_{1,2}$
		$\vdots$	$\vdots$
		$E_{1,n}$	$p_{1,n}$
	$\vdots$		
A^*	D_2	$E_{2,1}$	$p_{2,1}$
		$E_{2,2}$	$p_{2,2}$
		$\vdots$	$\vdots$
		$E_{2,m}$	$p_{2,m}$
	$\vdots$		
A'_1	$D_1 \wedge D_2$	$E_{1,1} \diamond E_{2,1}$	$p_{1,1} \cdot p_{2,1}$
		$E_{1,1} \diamond E_{2,2}$	$p_{1,1} \cdot p_{2,2}$
		$\vdots$	$\vdots$
		$E_{1,1} \diamond E_{2,m}$	$p_{1,1} \cdot p_{2,m}$
		$E_{1,2} \diamond E_{2,1}$	$p_{1,2} \cdot p_{2,1}$
		$\vdots$	$\vdots$
		$E_{1,n} \diamond E_{2,m}$	$p_{1,n} \cdot p_{2,m}$
	$\vdots$		

their outcomes. If an action of the agent and one or more random events would have a conflicting effect on some atom (for instance if the agent delivered coffee to the user just as the user became thirsty), we arbitrarily assume that the action written first in the representation of the problem takes precedence. Table 2.2 (b) shows the result of adding the random event *UserIsThirsty** to the *GetUmbrella* action. The complete translation of Table 2.1 can be found in the appendix.

Table 2.3 gives a formal definition of the result of combining two discriminants in the

process of translating a regular action and a random event into the compiled form. To compile a sequence of n actions $A_1, A_2, \ldots, A_n$ (one of which is assumed to be a regular action, and the rest random events) into a single action, we proceed by compiling A_1 and A_2, then compile the result with A_3, and so on. To compile two actions A and B, for each discriminant of A and for each discriminant of B we produce a new rule (here we refer to a discriminant and its set of effects and probabilities as a rule) as in Table 2.3. The new rule will have the conjunction of the two discriminants as its discriminant, and will contain a probability and effect for every pair of effects in the original rules. If E and F are effects from A and B with probabilities p and q respectively, then the new rule will contain an effect $E \cup \{F \setminus \{l : \neg l \in E\}\}$ which has probability $p \cdot q$. Obviously, if the actions have discriminants D_1 and D_2 where $D_1 \wedge D_2 \vdash \bot$ then no new discriminant is created for their combination. Similarly, there may be a large number of identical effects in the new rule that can be combined by summing their probabilities, and there may be effects that include literals that appear in the corresponding discriminants. These literals which the action makes true, but which are already known to be true can be deleted from any effects lists they appear in.

We will use a slightly different way of writing the extended STRIPS rules when we are talking about effects. In our alternate representation, a problem consists of a list of rules. Each rule is a tuple consisting of an effect E_i, the probability of it occurring p_i, the discriminant D_i that makes it possible, and the action A_i it relates to. This representation is identical to the standard representation given above — the elements of each tuple are identical — but is a more convenient way to talk about effects lists and their associated actions and discriminants. We will also occasionally want to talk about the set of atoms that appear in the logical sentence that is a discriminant. We will write D_i^S for the set of all atoms that appear either negated or unnegated in discriminant D_i. For example, if D_i is $p \vee \neg q$, then $D_i^S = \{p, q\}$.

We have represented actions in terms of propositions, but we would also like a compact representation of the reward function. There are two simple representations for rewards that we have considered. In the first, each atom is assigned rewards for its truth or falsehood and rewards for states in the MDP are the sum of the rewards for each atom's value in that state. A more general approach (in fact any reward function can be represented by this method) is to assign rewards to some set of mutually exclusive and exhaustive sentences so that the reward for any state in the MDP is the reward for the sentence that is true of that state. Our algorithms work equally well with either of these, so we will generally prefer the second as it can describe a strictly larger class of reward functions.

2.2.2 Using Bayesian networks

Nicholson and Kaelbling (1994) use a representation of actions as time dependent Bayesian networks. They use a two-slice network to represent each action. The first slice represents the values of (possibly multi-valued) variables before the action is performed, while the second slice represents the value after the action. In addition there is a value node which represents the instantaneous reward for the new state of the world. Arcs in the diagram represent probabilistic dependence between variables. As with conventional Bayesian networks, each node contains a table of conditional probabilities over all the nodes with arcs leading to it.

The Bayesian Network representation for actions is more expressive than the extended STRIPS approach in that nodes in the network may have more than two possible values. However, for propositional domains the two are equivalent. Figure 2.2 shows our example domain from Table 2.1 represented with Bayesian Networks.

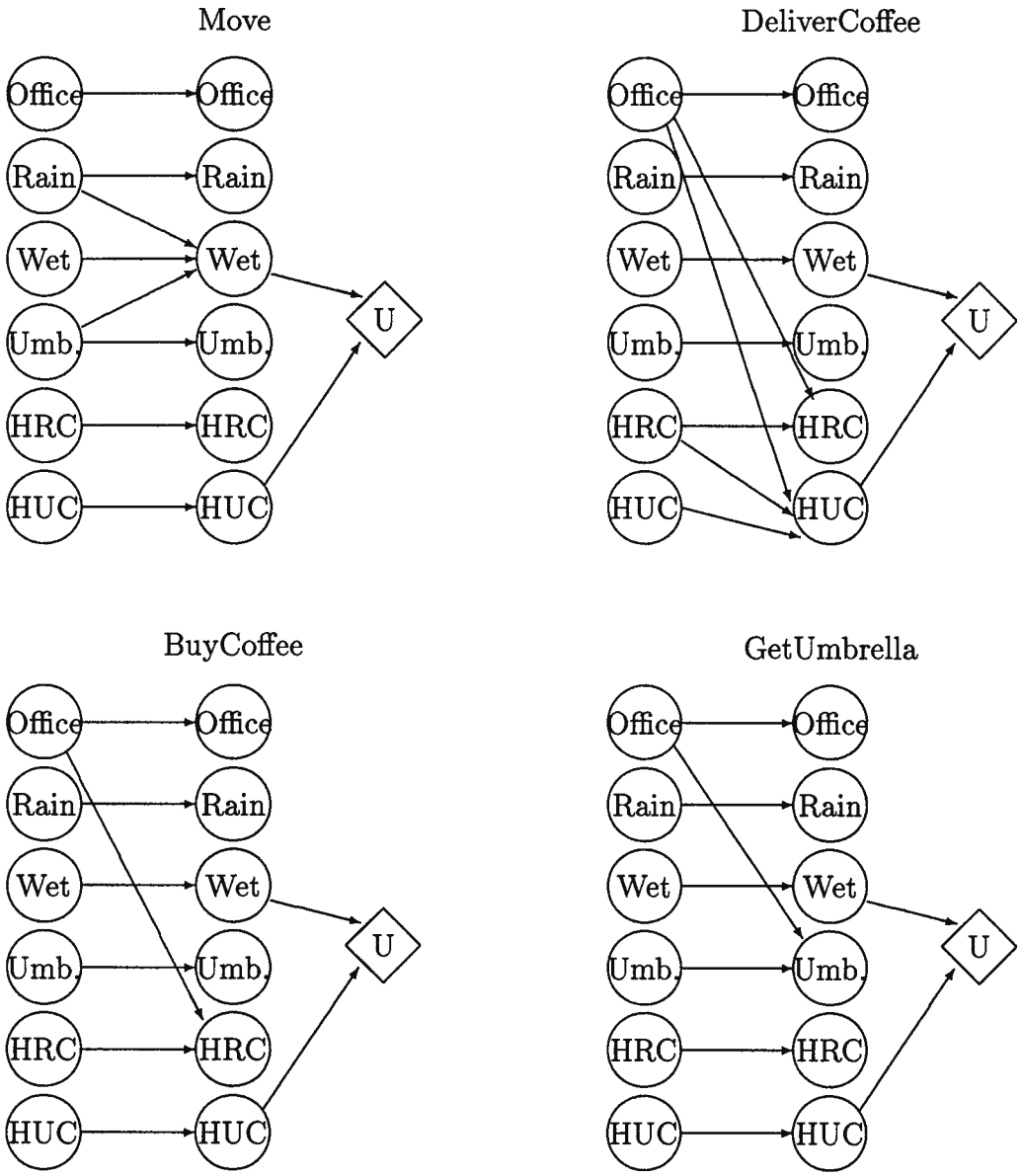

Figure 2.2: The influence diagram representation of the actions for the coffee robot example.

2.3 Goals and Rewards

In classical planning, the objective of the agent is to make some goal true. A planning problem typically consists of a set of actions and a goal, with the agent's task being to find a sequence of actions that make the goal true. Note that for most classical planners, the objective is to find any plan that satisfies the goal, without considering how efficient the plan is. Decision theoretic planning allows a more general description of the aims of the agent. As we have already described, it allows us to think about agents as continuing processes, but we can also use a decision theoretic framework to represent domains where there are multiple goals with different priorities, and goals with more interesting structures. As an example of this, the agent may want to achieve one of two things, but achieving both would be undesirable.

How can we represent classical goals in the framework of MDPs? The standard approach used by (Dean et al. 1993b) is to create a reward function such that $R(s) = 0$ if s is a state in which the goal is satisfied and $R(s) = -1$ otherwise, and to make all states in which the goal is satisfied *absorbing* (an absorbing state is one in which the result of every action is to leave the state unchanged). This reward function will cause the agent to seek out goal states using as few actions as possible. In contrast with a classical planner, it finds a minimal plan, not just any plan that will achieve the goal.

2.3.1 Why use rewards?

As we have already said, rewards provide a more flexible representation of an agents aims, and also allow more complex goals to be represented, but there are other reasons for using the full power of a reward function in decision theoretic planning.

In many situations, producing optimal plans in probabilistic domains is computationally infeasible. The cost of calculating an optimal plan in a time-critical domain may be

greater than the improvement in performance gained. If approximate plans are useful, the reward function can be a very valuable aid in determining the most important tasks for the agent. For example, consider the example domain in Table 2.1. The reward function shows us that the most important goal to achieve is that the user has coffee. In making a plan for this domain, it may well be worth ignoring the small reward for keeping the agent dry, and concentrate on delivering coffee. Classical goals are unable to represent information of this kind.

2.4 Policy Iteration and Planning

A control policy π is a mapping from $\mathcal{S}$ to $\mathcal{A}$. If an agent adopts some policy π, then $\pi(s)$ is the action the agent will perform whenever it finds itself in state s. Thus π is a *universal plan*, an action to perform in every possible circumstance. An agent that adopts policy π can also be thought of as a reactive system. Given an MDP, an agent ought to adopt a policy that maximizes the expected rewards for states visited. Such a policy is called an *optimal policy.* We will concentrate on *discounted infinite horizon* problems where the actions of the agent continue infinitely, and the value of a reward received n actions in the future is discounted by some factor $\beta^n (0 < \beta < 1)$. We do this for a number of reasons. Even though in many situations we can expect the agent to have a finite lifetime, we rarely know in advance how long it will be. We can approximate a long finite horizon process with an infinite horizon one. Discounting is used in Operations Research MDP problems where a dollar earned today is worth more than a dollar earned tomorrow, or where the agent has some probability $1 - \beta$ of "dying" at each time step. While this may not be the case in many planning problems, discounting encourages short plans, and is much easier to compute than alternative methods such as *average reward criteria* (Puterman 1994). As well as an action to perform in each state, we are also interested

in the value of that state. While the total reward received by an agent will diverge over an infinite lifetime, the discounted reward will converge on a finite value (assuming finite rewards). The agent's objective is to maximize the expected accumulated discounted reward for all future states over an infinite time period. One way to look at a problem in decision theoretic planning is as finding a good — or preferably optimal — policy, at least for all states that the agent actually visits.

Given some policy π and a reward function R, we can compute the value of any state $s \in \mathcal{S}$ ($V_\pi(s)$) by the following formula due to Howard (1960):

$$V_\pi(s) = R(s) + \beta \sum_{t \in \mathcal{S}} \Pr(t|\pi(s), s) V_\pi(t)$$

Since the value of each state is dependent on the values of all others, we can find the value of π for all states by solving the set of simultaneous linear equations $V_\pi(s) \forall s \in \mathcal{S}$. A policy π is optimal if:

$$(\forall s \in \mathcal{S})(\forall \pi')(V_\pi(s) \geq V_{\pi'}(s))$$

2.4.1 Calculating optimal policies

There are two commonly used algorithms for calculating optimal policies in discounted infinite horizon problems. *Value iteration* (Bellman 1961) converges on an optimal policy by considering the effects of states further and further from each state. It is guaranteed to converge on an optimal policy, but only approximates the value of states in the policy. The algorithm we have used is *policy iteration*, first presented by Howard (1960). Policy iteration converges on both an optimal policy and accurate values for states under that policy. Although its computation time is exponential in the number of states in the worst case, policy iteration tends to converge quickly in practice.

The policy iteration algorithm begins with some arbitrary policy, and repeatedly improves on it until no more improvement is possible. At that point it is guaranteed to produce an optimal policy. The algorithm in detail is given below:

1. Let π' be any policy on $\mathcal{S}$
2. While $\pi \neq \pi'$ do
   (a) $\pi = \pi'$
   (b) For all $s \in \mathcal{S}$, calculate $V_\pi(s)$ by solving the set of $|\mathcal{S}|$ linear equations given by the equation above.
   (c) For all $s \in \mathcal{S}$, if there is some action $a \in \mathcal{A}$ such that $R(s)+\beta \sum_{t \in \mathcal{S}} \Pr(t|a,s)V_\pi(t) > V_\pi(s)$ then $\pi'(s) = a$;
   otherwise $\pi'(s) = \pi(s)$
3. Return π

The algorithm begins with an arbitrary policy (a good choice is a greedy approach that chooses the action with the greatest immediate reward), and repeatedly improves on the policy until no more improvement is possible. The improvement step is performed by first constructing a vector containing the value of each state by solving the set of $|\mathcal{S}|$ linear equations in $|\mathcal{S}|$ unknowns given above in the equation for the value of a state. Algorithms for solving linear equations of this kind are typically $O(n^3)$ where n is the number of unknowns. Once the value of every state under the current policy is known, the algorithm attempts to find a better action for each state if one exists. To find a better action for a given state, the algorithm holds the rest of the policy and the value vector constant, and checks every action to see if it has a higher value than the current one, using the same equation as before. Although this won't be the true value of that

Table 2.4: The optimal policy for the sample domain.

	HUC, HRC	HUC,HRC	HUC, HRC	HUC,HRC
Office	BuyCoffee	Move	BuyCoffee	Move
Office,Rain, Umbrella, Wet	BuyCoffee	Move	BuyCoffee	BuyCoffee
Office	Move	DeliverCoffee	Move	BuyCoffee
Office,Rain, Umbrella, Wet	GetUmbrella	DeliverCoffee	GetUmbrella	GetUmbrella

action (the value of this state will differ from that in the vector), it is still an indication that the policy is not yet optimal. The algorithm stops performing improvement steps when no better action than the one in the current policy can be found for any state.

For the example problem in Table 2.1, there are 64 states in the MDP. The optimal policy discovered by policy iteration (using $\beta = 0.95$) is shown in Table 2.4. As the table shows, if the robot doesn't have coffee and is in the office, it will get the umbrella if there is rain, the robot is dry, and it doesn't have it already, and will then move to the coffee shop. If the robot is at the coffee shop it will buy coffee if it doesn't have any, and will then move to the office unless the user has coffee and it is likely to get wet, in which case it does nothing (by buying more coffee). If the user wants coffee and the robot has some in the office then it will deliver it, but if the user doesn't want any, the robot will get the umbrella if that might be useful, and then does nothing (by trying to buy coffee in the office). The fact that the robot may repeatedly buy coffee while waiting until the user becomes thirsty is a good example of why an MDP model with rewards (or in this case costs) on actions as well as states would be useful.

The values of states in this policy range from 12 to just under 20, with the lowest values (12–14) for states where the robot is (or will probably become) wet and the user has no coffee. States where the robot is dry, or the user has coffee but not both have

intermediate values (15–18), while the highest values (19–20) are for states where the robot is dry and the user has coffee. Computing this policy required 0.48 seconds.

2.5 Related Work

Decision-theoretic planning is a relatively new field. Consequently, there is little closely related previous work. However, the abstraction procedure is related to other such algorithms from classical planning, and the search algorithm bears similarities to a number of search procedures, especially from game playing, and decision analysis.

2.5.1 MDP planning

This section describes a selection of important papers that are directly related to this thesis in that they are performing decision-theoretic planning using explicit MDP models in domains with many of the same characteristics we are assuming.

Dean, Kaelbling, Kirman and Nicholson — Planning Under Time Constraints in Stochastic Domains

Dean, Kaelbling, Kirman and Nicholson (1993b; 1993a) produced two papers which provided the motivation for a lot of the recent interest in using MDPs for decision-theoretic planning, including this thesis. Although MDPs and algorithms for them were already well known in operations research, these two papers first suggested examining time-critical applications where optimal policies cannot be computed from an AI perspective.

The approach taken in these papers is to use information about the starting state of the system, the reward function, and the transition probabilities for actions in order to restrict the planner's attention to only those states which are likely to be encountered in the process of reaching the goal. By planning in this smaller *envelope* of states, the

agent can find a policy with much less computation than the original domain. The size of the envelope can be tailored to the amount of computation time available, although if the agent leaves the envelope, replanning will be required. These papers also consider the question of when computation should be performed. In their *precursor deliberation* model all computation is performed before any actions are taken, while the *recurrent deliberation* model plans and acts in parallel.

Nicholson and Kaelbling — Toward Approximate Planning in Very Large Stochastic Domains

Nicholson and Kaelbling (1994) are also interested in automatically generating abstractions. As with our approach, Nicholson and Kaelbling assume a compact representation of the domain in terms of *domain attributes*, although they allow theirs to have more than two possible values, and represent actions as two-slice Bayesian networks. They also perform abstraction by ignoring domain attributes, although their abstraction procedure is rather more general as abstractions can always be created. Unfortunately, with this generality comes a great deal of computation, as the effects of actions must all be recomputed by assuming a uniform distribution over the values of the ignored attributes.

To decide which domain attributes to ignore, Nicholson and Kaelbling use sensitivity analysis. Their approach is to generate an abstraction and an envelope, and continually refine these by extending the envelope, building a policy for it (in the abstract MDP), and expanding the abstraction to be more fine-grained if necessary, until the agent must act.

Tash and Russell — Control Strategies for a Stochastic Planner

Tash and Russell (1994) use a rather different form of the envelope approach of Dean et al. Their system begins with an heuristic which estimates the value of each state, and

through a number of iterations, improves this heuristic to more accurately represent the true value of each state. Tash and Russell describe an algorithm whereby an envelope is created around the current state, the states just outside the envelope (the fringe) have their values fixed to the heuristic values, and a local form of policy iteration is performed on the envelope and fringe. The result of the local policy iteration algorithm is an action to perform for every state in the envelope, and new heuristic values for each of those states. As with the search algorithm we present in Chapter 4, the agent in Tash and Russell's planner executes each action as soon as it has computed it, and then rebuilds its envelope to choose the next action. The relationship between our search algorithm, and the approach of Tash and Russell is analogous to that between value iteration and policy iteration except that we make no changes to our heuristic function, preferring to save computation time when revisiting states.

Tash and Russell also present an introspective planner which not only takes into account the problem domain it is working in, but also its knowledge of the domain when planning. The introspective planner is designed to expend extra computational effort in areas where it currently has little information, and to use well-known paths through the state space when little computation time is available. They present results which show the introspective planner finding optimal plans more quickly than their standard algorithm when computation time is relatively cheap, but more slowly if computation time is expensive.

2.5.2 Other decision-theoretic planning algorithms

In this section we will examine a few other decision-theoretic planning algorithms that do not directly use MDPs as their underlying model of the world. Unlike the MDP planners which generally use dynamic programming techniques and find universal plans (at least for a restricted part of the state space), these systems are much closer to classical

planners.

Feldman and Sproull (1977) were among the first researchers to investigate the use of decision theory for AI. They suggested using bounds on utility to guide the planning process, but they left their utility model relatively undeveloped, focusing instead on representational issues for reasoning with uncertainty.

Haddawy and Hanks — Representations for Decision-Theoretic Planning: Utility Functions for Deadline Goals

Haddawy and Hanks (1992) made one of the earliest attempts to couple decision theory with AI planning. In particular, they were interested in using representations of goals to determine plan quality. Although they make no effort to model actions, either probabilistically or otherwise, they have a detailed model of planning goals, and examine ways that the utility of a plan will vary depending on whether all goals are achieved, and on whether there is only partial achievement. Unlike our approach, they also examine temporal aspects of planning. Their goal description language is rich enough to capture requirements such as "Make G true by noon" or "Make sure that H is true between 10am and 1pm," allowing them to reason about goals that are partially completed from a temporal point of view (for example if H is true only between 11am and noon).

Kushmerick, Hanks and Weld — An Algorithm for Probabilistic Planning

Kushmerick, Hanks and Weld (1993) developed the BURIDAN planner, a method for performing probabilistic planning with uncertainty about both state and the effects of actions in a classical planning framework. BURIDAN uses a classical partial-order planner, for example SNLP (McAllester and Rosenblitt 1991) for plan creation, and a probabilistic reasoning system to evaluate the partial plans. The system requires a set of probabilistic actions, a probability distribution over initial states of the world, a goal expression, and

a probability threshold. A solution consists of a partial ordering of actions which will accomplish the goal with probability greater than the threshold given the distribution of initial states. Note that the notion of a goal here is of a set of states that must be reached. BURIDAN is unable to make use of the rich variety of goals that can be represented in MDP planners.

The BURIDAN planner builds a complete plan to reach the goal based on its initial information about the initial state, without executing a single action. One characteristic of the plans it makes is that actions are frequently repeated in order to increase their probability of success. A modification of the BURIDAN algorithm, where the agent can perform actions and gather information to avoid this behaviour, is the C-BURIDAN planner presented in (Draper, Hanks and Weld 1994). Here, actions not only change the world, they also give the agent messages about the state of the world (some actions may only observe the state of the world), by building conditional plans based on these messages, C-BURIDAN can produce plans where an action is repeatedly performed until a message that shows it has succeeded is received.

Haddawy and Doan — Abstracting Probabilistic Actions

Haddawy and Doan (1994) use an action model very similar to that of the BURIDAN planner, although it can handle a much larger class of rewards and costs, including rewards that vary over time. Their DRIPS system attempts to use abstraction to reduce the number of possible plans the agent must generate and evaluate. While we are interested in abstracting states while leaving the actions unchanged, Haddawy and Doan build abstract actions by combining possible outcomes of one or more actions. For example, if the system contains two actions "drive on the mountain road" and "drive on the valley road", the abstraction might abstract them into a single action which combines the outcomes and preconditions of both. The DRIPS system searches for a plan that maximizes

expected utility by first building abstract plans, and then only refining those with the highest expected utility. Although this technique greatly reduces the number of plans that DRIPS evaluates, it is not clear how much computation would be required to find optimal plans in real-world domains by this approach.

2.5.3 Abstraction in classical planning

In this section we examine the use of abstraction in classical planning. Using abstract versions of a problem in order to solve it efficiently is by no means new. The first formal description and application of this idea to planning is in the ABSTRIPS planner described below.

Sacerdoti — Planning in a Hierarchy of Abstraction Spaces

Sacerdoti (1974) described the earliest use of abstraction in his ABSTRIPS planning system. Many of the ideas from this system have influenced our abstraction procedure. Along with a description of the actions and the goal, ABSTRIPS must also be provided with an abstraction hierarchy which is used to automatically assign criticalities (a measure of how difficult a literal is to achieve) to the preconditions of each action. The system uses this to form a hierarchy of problems with the topmost level containing only the literals in the goal that are the hardest to satisfy, since the actions that make them true are the most likely to be clobbered by the effects of other actions, and the lowest level is the original problem. ABSTRIPS builds a plan at the topmost level of the hierarchy, where only the most critical propositions are reasoned about, and then refines the plan level by level by including more preconditions, and operators to satisfy them, until a complete plan has been constructed. Our approach is closely related to this as we form abstract problems by only including the most important propositions. Although we generally don't construct more refined plans from the abstract ones in the same way

ABSTRIPS does, Chapter 3 mentions the possibility of using abstract policies to seed the policy iteration algorithm in the original domain to find optimal policies more efficiently.

Knoblock — Generating Abstraction Hierarchies: An Automated Approach to Reducing Search in Planning

Knoblock (1993) extends the ABSTRIPS approach to abstraction with the ALPINE system, which automatically learns the abstraction hierarchy that ABSTRIPS requires. The approach is to construct a partial ordering of the literals in the system by finding constraints between literals, building a graph of the constraints, and topologically ordering the graph. A constraint from one literal to another represents the fact that achieving the first might interfere with achieving the second, and hence the first should be at least as high in the abstraction hierarchy. This procedure is very similar to — and inspired — the abstraction algorithm we describe in Chapter 3.

Smith and Peot — Postponing Threats in Partial-Order Planning

Partial-order planners, such as SNLP (McAllester and Rosenblitt 1991), operate by constructing plans where two actions are ordered (one must take place before the other) only if one has an effect which threatens the preconditions of the other. Ordering two actions is known as *resolving a threat.* Smith and Peot (1993) describe an algorithm for avoiding having to resolve some threats during the planning process by postponing the threat until the plan is complete. In order to do this, they introduced the concept of *operator graphs*, a generalized form of the graphs constructed by the ALPINE system for total-order planning. Operator graphs represent the fact that certain actions make the preconditions of other actions true: there is a path (sequence of arcs) in the graph between two literals if performing the actions along that path will make the second literal true if the first one is. These graphs show the deterministic equivalent of the relationships

we discover for probabilistic actions when we automatically construct abstractions (see Chapter 3). Although the uses we put them to is rather different, the graphs Smith and Peot construct represent exactly the relationships we are interested in when we construct abstractions.

2.5.4 Markov decision processes and Decision Theory

There is a huge body of work in the area of Markov decision processes, most of it in the field of operations research. There are two well known algorithms for constructing optimal policies for MDPs where rewards are discounted over time. We have already described the most commonly used, *policy iteration*, described by Howard (1960). The other commonly used algorithm is *value iteration*(Bellman 1961). Value iteration begins with a vector of estimated values for each state, and repeatedly updates the value of each state until the vector converges on the optimal value. To perform each update, the value iteration algorithm finds the action which gives the highest expected immediate value (using the values from the previous iteration), and sets the new value of this state to be the immediate reward plus the discounted expected value of performing that action. If V_i is the value vector at the ith iteration, the $(i+1)$th iteration value for state s is:

$$V_{i+1}(s) = R(s) + \beta \max\{\sum_{t \in \mathcal{S}} \Pr(t|a,s)V_i(t) : a \in \mathcal{A}\}$$

This function is guaranteed to converge in the limit to the optimal policy, and in practice, often converges quite quickly. The search procedure we describe in Chapter 4 resembles a local form of policy iteration in that deeper and deeper search will converge on the correct value for a state in the same way that value iteration does.

For a more recent survey of policy construction in MDPs, including discussion of partially observable MDPs, semi-Markov processes, non-discounted MDPs where the average reward earned per action must be maximized, and more efficient algorithms such

as modified policy iteration for computing optimal policies, see (Puterman 1994). This also describes work on finding optimal policies by aggregating states. However, rather than taking advantage of the way the problem is represented as our abstraction algorithm does, these algorithms use the MDP directly when deciding when to aggregate states.

Although we have yet to implement many of its suggestions, we were greatly influenced in this research by (Russell and Wefald 1991). Russell and Wefald discuss decision making in general, and in particular, deciding whether to perform further sensing and/or computation in order to improve a decision, or whether to act now with the current information. Adding the ability to reason introspectively about the value of further computation is an important capability of any planner, and one we would very much like to add to our system.

2.5.5 Search algorithms

The search algorithm described in Chapter 4 is closely related to a number of classic AI search algorithms. Although the algorithm is in essence a decision-tree search, its use of heuristics and pruning techniques is based in game-tree searches.

A decision tree is an explicit representation of all the possible scenarios that could occur following a decision. The root of the tree represents the initial situation, while each path through the tree represents a series of decisions, outcomes of actions, and events that might occur, terminating in a consequence node which contains the utility of the situations that would result from following that path. The search trees which we construct are very similar to this, they contain action nodes, where the agent must decide which action to perform, and chance nodes where all the possible outcomes of some action are investigated. The main difference between our search procedure and decision tree analysis is that we cannot produce a complete tree due to the nature of our problems, so we produce a partial decision tree and use the heuristic to estimate the utility of the

final nodes in each path. However, the algorithm we use to determine which action to perform is essentially identical to the *exp-max* algorithms used in analyzing decision trees. For more detail on decision trees and their uses, see (Pearl 1988) and (Howard, Matheson and Miller 1976).

Ballard — The *-Minimax Search Procedure for Trees Containing Chance Nodes

(Ballard 1983) describes an extension of alpha-beta search for game trees. The *-minimax search algorithm selects game moves in games which have *probability nodes* whose value is defined to be the weighted average of their successors' values. The class of games that can be played by this algorithm are those with random elements, but no hidden information. The treatment of the probability nodes in *-minimax search is essentially identical to that of the AVERAGE nodes of the search procedure we describe in Chapter 4. In fact, the trees we construct during the heuristic search procedure are equivalent to *-minimax trees which contain no MIN nodes (moves by the opposing player). There is a similar relationship between the pruning techniques used in *-minimax search and our *utility pruning* technique. The decision trees we construct during the search process can be seen as game trees for a game played against a randomized opponent, so many of the results shown for *-minimax trees also hold for our problem.

Korf — Real-time Heuristic Search

Korf (1990) presents a single-agent heuristic search procedure adapted from game-tree search. The *Real-Time-A** (RTA*) algorithm combines a limited search horizon and commitment to moves in a constant time to produce a search procedure which is the deterministic equivalent of the one we present in Chapter 4. Using the property that the heuristic evaluation of nodes is monotonic (that is, the heuristic value of any node

is at least as high as that of its parents), Korf provides a form of pruning for these trees which he calls *alpha pruning.* Alpha pruning is similar to the *expectation pruning* that we perform, although the properties of the heuristic that we make use of are different. Paradoxically, Korf finds that as the branching factor of the domains he has examined increases, the search space actually reduces as a result of the effectiveness of alpha-pruning.

Korf also presents a learning form of RTA* which, over a series of runs, will improve its heuristic function, and will eventually converge on the true values of states. Some of the same ideas can be applied to the search procedure we present, but we have not yet had the opportunity to investigate them.

Chapter 3

Creating Approximate Policies Using Abstraction

As we described in Chapter 2, computing the optimal policy for a Markov Decision Process using policy iteration is usually polynomial in the size of the state space. While this might seem to be a reasonable computational cost, in practice the state space may be extremely large. For example, in a propositional domain made up of n atoms, the size of the state space is 2^n. Given that the state space is often too large to compute an optimal policy for, we need a way to produce approximately optimal policies in less than polynomial time.

To produce "good" policies efficiently, we will use a process of abstraction. By abstraction, we mean the creation of a smaller problem which contains the most significant details of the original problem while ignoring some of the less important properties. A solution to the simpler problem can then serve as the outline for a solution to the original problem which will considerably reduce the total computation required. In the case of this work, abstraction provides a mechanism for reducing the size of the problem to be solved to manageable proportions, and allows us to cheaply compute a (probably less than optimal) solution to the original problem directly from an optimal solution to the smaller problem.

The most obvious use for the abstraction process is to supply a close-to-optimal policy for use in the original domain, but this is not the only possible use. We also use the abstract policy to construct a heuristic estimate of the value of each state. By performing a heuristic search from states that are visited in the course of carrying out

actions, we can hope to find better actions than the ones provided directly by abstraction (if better actions exist). See Chapter 4 for the details of this approach. Since the policy iteration algorithm described in Chapter 2 requires an initial policy, we can also use the abstract policy as this "seed" for constructing an optimal policy in the original domain. By first computing a policy in an abstract state space, and then using it to compute a true optimal policy, we can hope for computational savings because fewer iterations of policy iteration should be required before an optimal policy is reached.

3.1 Abstract MDPs and Policies

In overview, our approach to abstraction is to build an exponentially smaller *abstract* MDP which captures the most significant details of the original *concrete* MDP, use policy iteration to compute an optimal policy for the abstract MDP, which we call the *abstract policy*, and then apply the abstract policy to the concrete MDP to produce an approximately optimal policy. The key to this approach is to generate the abstraction automatically. Rather than requiring the user to specify the abstraction, the algorithm will provide a set of possible abstractions each with a bound on the difference between the abstract and optimal policies, and will let the user select an acceptable value. In this way the user can trade off computation time for policy quality.

The abstract MDP must have a number of properties in order to be useful. The first of these is that the structure we produce in the process of abstraction must be a MDP. In particular, the Markov Property (see Section 2.1) must hold. The most important property of the abstract MDP is that the cost of constructing it should be quite small. Producing an abstraction and then computing its optimal policy must require considerably less computation than computing an optimal policy for the original MDP. If this is not the case, the time is better spent computing the optimal policy.

Not only must the construction of the abstract policy be inexpensive, the policy discovered for the abstract domain should be easily applicable in the original MDP as well. Since we want to use the abstract policy to tell the agent what to do in the concrete domain, each abstract action must correspond to some concrete action or sequence of concrete actions. For example, in a navigation domain, the abstract action *GoNorth* might translate to the sequence of concrete actions *TurnRight, Advance* if the agent had been facing west to begin with. If the abstract actions do not correspond directly to concrete actions then they also must be computationally inexpensive to produce or discover.

When constructing the abstract MDP, we want to use as much domain information as possible, whether or not that information is explicitly represented in the concrete MDP. This domain information may take a variety of forms. For example, we may be able to use the fact that the domain is a navigation problem and that certain states are geographically related to each other when producing the abstract state space. As we said in Section 2.2, domains will usually not be represented directly as Markov Networks. Representations such as the extended STRIPS model (see Section 2.2.1) provide additional domain information that can be used to construct more useful abstractions.

The final requirement for the abstraction process is that the abstract policy generated is actually useful in the concrete MDP. Time spent on abstraction is only of value if the resulting policy is fairly close to optimal. As we discuss in Chapter 4, we can perform local search — when sufficient computation time is available — to improve on the policy generated by abstraction so we do not require an optimal policy. However, we do need the abstract policy to be a reasonable approximation to the optimal policy for local search to provide any improvement.

1. Using the STRIPS representation of the domain, decide which atoms are most important for constructing a good policy.
2. Build the abstract state space $\tilde{S}$ by clustering together all the states in S which agree on the values of the important atoms.
3. For each action, build an abstract transition function $\tilde{T}$ by deleting all reference to unimportant atoms from the action description and translating the extended STRIPS representation of the action into an MDP transition function. Note that an explicit transition matrix need not be built for each action as the extended STRIPS rules can be used to generate the linear equations required for policy iteration directly.
4. Construct $\tilde{R}$, the reward function for the abstract problem.
5. Use policy iteration to find the optimal policy $\tilde{\pi}$ for the MDP $< \tilde{S}, \mathcal{A}, \tilde{T}, \tilde{R} >$.
6. Construct the policy π such that for each state $s \in S \in \tilde{S}, \pi(s) = \tilde{\pi}(S)$. π is an approximately optimal policy for the original MDP.

Figure 3.3: Constructing an approximately optimal policy using Abstraction

3.2 Constructing Abstract Policies in Propositional Domains

While abstraction can be used in any domain (see (Nicholson and Kaelbling 1994) for an example of a different approach to abstraction), we have concentrated on domains represented in terms of propositions. In particular, we shall use the extended STRIPS representation described in Section 2.2. For these types of domain, the process of using abstraction to produce an approximately optimal policy is described in Figure 3.3.

3.2.1 Choosing relevant atoms

In order to construct an abstract MDP, we need to select some subset of the atoms which will form the basis of the abstraction. The quality of the policy and the effectiveness of the abstraction process are both closely dependent on the atoms chosen. If too many

atoms are selected, the policy created may be very close to optimal, but the computational savings may not be large enough to justify the loss of optimality. On the other hand, if the set of atoms chosen is small, then the computation required to produce the approximate policy will also be small, but the policy may be quite poor.

As well as choosing an appropriate number of atoms to construct an abstraction, we must consider which atoms should be selected. Obviously, if the reward for each state depends solely on the value of a single atom in that state, it would be foolish to ignore that atom when constructing the abstract state space. However, this is not the only consideration — atoms that have relatively little effect on the reward for a state may be able to be ignored, and an atom which indirectly affects reward should be included in the set of relevant atoms.

In order to construct a set of atoms which meets the criteria described above, we first identify a set $\mathcal{IR}$ of *immediately relevant atoms.* $\mathcal{IR}$ is formed by examining the propositional model of the reward structure, and selecting only those atoms which have the greatest impact on the reward for each state. The larger this set is, the more fine-grained the abstraction will be, so by varying the size of $\mathcal{IR}$, we can find a balance between the quality of the abstraction, and the computation time required.

Although the set $\mathcal{IR}$ contains some of the relevant atoms we would like to use for abstraction, it does not yet include all relevant atoms. For example, in a domain where the reward is large if atom A is true and small otherwise, $\mathcal{IR}$ would be $\{A\}$. But if an action that made A true required B to be true as a precondition, then clearly B is a relevant atom as well. To capture this formally, we define $\mathcal{R}$, the set of relevant atoms as the smallest set such that $\mathcal{R}$ contains every atom in $\mathcal{IR}$, and if some atom $A \in \mathcal{R}$ appears in the effects list of some action aspect, then every atom that appears in the corresponding discriminant is also in $\mathcal{R}$. The set $\mathcal{R}$ is defined as follows:

1. $\mathcal{IR} \subseteq \mathcal{R}$.

2. if $q \in \mathcal{R}$ and for some effects list E_i, either $q \in E_i$ or $\neg q \in E_i$, then $D_i^S \subseteq \mathcal{R}$.

Notice that only the atoms in a discriminant that might probabilistically lead to a relevant effect are deemed relevant; we will call this a *relevant discriminant.* Other conditions associated with the same action aspect are ignored, unless these are also relevant. Furthermore, although $\mathcal{R}$ is defined recursively, it can be easily and automatically computed as a fixed point of the function $\mathcal{R}_i$ given below:

$$\mathcal{R}_0 = \mathcal{IR}$$

$$\mathcal{R}_{i+1} = \mathcal{R}_i \cup \{q | (\exists j)(q \in D_j^S \text{and} E_j \cap \mathcal{R}_i \neq \emptyset)\}$$

The only decision required from the user of the system is that of which atoms should be placed in $\mathcal{IR}$. As we shall see, this fact allows the user to specify the degree of accuracy required of the abstraction, and to have an abstract policy and heuristic function for search calculated automatically.

Having calculated $\mathcal{R}$, we have completed step 1 of the algorithm in Figure 3.3. To achieve step 2, we must use $\mathcal{R}$ to create the abstract state space $\tilde{\mathcal{S}}$. We do this by clustering together all the states in the original MDP which agree on the values of the atoms in $\mathcal{R}$. By treating each cluster as a state in the abstract MDP, we ignore the irrelevant details of atoms that do not appear in $\mathcal{R}$.

Definition 2 The abstract state space generated by $\mathcal{R}$ is $\tilde{\mathcal{S}} = \{\tilde{s}_1, \ldots, \tilde{s}_n\}$, where:

1. $\tilde{s}_i \subseteq \mathcal{S}$.

2. $\bigcup\{\tilde{s}_i\} = \mathcal{S}$.

3. $\tilde{s}_i \cap \tilde{s}_j = \emptyset$ if $i \neq j$.

4. $s, t \in \widetilde{s}_i$ iff $s \models P$ implies $t \models P$ for all $P \in \mathcal{R}$.

For an example of how to construct an abstract MDP by this method, see Section 3.2.3.

3.2.2 Building abstract actions and rewards

If an optimal policy is to be constructed over the abstract state space $\widetilde{\mathcal{S}}$, we require actions and a reward function that are applicable to the new states. In general, computing the transition probabilities for actions associated with an arbitrary clustering of states is computationally prohibitive. It requires that one consider the effect of each action on each state in the cluster. Furthermore, computing the probability of moving from one cluster to another when performing a given action requires some prior distribution over the worlds in the initial cluster. With the information we have, it is impossible to calculate such a distribution. To do so would require a distribution over initial states of the system, and a fixed policy.

The abstraction mechanism we have described is designed to avoid exactly these problems. Our intention in creating $\mathcal{R}$ as described above is to make the problem of building abstract actions and rewards as computationally inexpensive as possible. The following theorems demonstrate the significant characteristics of the abstraction. Theorems 1 and 2 ensure that all the states clustered into one abstract state will behave the same when an abstract action is performed. This justifies using the abstract problem to reason about the concrete problem because it shows that the concrete states in a cluster will behave the same as the corresponding abstract state. If an action is performed on a cluster for which the action has a relevant discriminant, then all the states in the cluster will be mapped to the same probability distribution over clusters, and if the cluster is contained in an irrelevant discriminant, then all the states will be mapped into other states in the

same cluster.

Theorem 1 *If $\tilde{s}$ is an abstract state, and $s, t \in \tilde{s}$, then s satisfies a relevant discriminant for some action aspect iff t does.*

Proof: Assume not, then without loss of generality, there exist $s, t \in \tilde{s}$ and a relevant discriminant D_i such that D_i is true in s but not in t. Hence s and t must disagree on the value of some atom in D_i^S, but since $D_i^S \subseteq \mathcal{R}$ by the definition of $\mathcal{R}$, s and t cannot be in the same cluster $\tilde{s}$, which contradicts our assumption. □

Theorem 2 *If E_i is an effect of some action, and $s, t \in \tilde{s}$ as above, then i) If E_i is associated with an irrelevant discriminant, then $E_i(s) \in \tilde{s}$; and ii) $E_i(s) \in \tilde{u}$ iff $E_i(t) \in \tilde{u}$.*

Proof: i) If E_i is associated with an irrelevant discriminant, then $E_i \cap \mathcal{R} = \emptyset$ since if E_i contained an atom in $\mathcal{R}$, the discriminant would be relevant. Since E_i contains no atoms in $\mathcal{R}$, $E_i(s)$ must agree with s on the value of all atoms in $\mathcal{R}$, and hence $E_i(s) \in \tilde{s}$ as required.

ii) Since s and t agree on the values of all atoms in $\mathcal{R}$, $E_i(s)$ and $E_i(t)$ must also (because E_i changes the same atoms in both), so $E_i(s) \in \tilde{u}$ iff $E_i(t) \in \tilde{u}$. □

The two theorems show that when any action is performed, the effects of that action will be to map all the states in a cluster to the same new cluster. This allows us to construct the abstract actions by simply deleting all reference to irrelevant atoms from the actions in the original problem. This may cause some simplification of the action specification as discriminants and effects which were different in the original problem become the same in the abstract problem. For example, if an action has a discriminant with several effects, none of which contains a relevant atom, they will all collapse into a single empty effects list. The probability of this effect occurring will be the sum of the probabilities of the concrete effects which combined to form it.

To associate a reward with each cluster, we use the midpoint of the range of rewards for the states in that cluster. For example if a cluster contained states with rewards 0, 1 and 10, we would select 5 as the reward for the cluster. Formally, if $\min(\tilde{s})$ and $\max(\tilde{s})$ denote the minimum and maximum values of the set $\{R(s) : s \in \tilde{s}\}$ respectively, then the *abstract reward function* is defined as:

$$\tilde{R}(\tilde{s}) = \frac{\max(\tilde{s}) + \min(\tilde{s})}{2}$$

This choice of $\tilde{R}(\tilde{s})$ minimizes the possible difference between $R(s)$ and $\tilde{R}(\tilde{s})$ for any $s \in \tilde{s}$, and is adopted for reasons explained in Section 3.3. Although using the average of the rewards in the cluster as the abstract reward for the cluster might result in better average-case behavior, it can lead to much worse bounds on the difference between the abstract and optimal policies.

3.2.3 An example domain

To demonstrate the abstraction algorithm, we will use the example from Table 2.1 (Page 14). In this example, there are two atoms that affect the reward for a given state, *HasUserCoffee*, and *Wet*, but since the effect of *HasUserCoffee* is far greater, we will set $\mathcal{IR} = \{HasUserCoffee\}$. Two actions affect the value of *HasUserCoffee*, the relevant effect of *UserIsThirsty** has no preconditions, and can be ignored, while *DeliverCoffee* has discriminant $\{Office, HasRobotCoffee\}$, so these two atoms must be included in $\mathcal{R}$. Similarly, the *Move* action affects the value of *Office*, and the *BuyCoffee* and *DeliverCoffee* actions affect the value of *HasRobotCoffee*, but no new atoms appear in the relevant discriminants of these actions, so $\mathcal{R} = \{HasUserCoffee, HasRobotCoffee, Office\}$.

Note that if we had chosen $\mathcal{IR}$ to include *Wet*, then $\mathcal{R}$ would also have included *Rain* and *Umbrella* from the second aspect of the *Move* action, so all the atoms from the original problem appear in $\mathcal{IR}$, and the abstract state space is identical to the concrete

Table 3.5: The STRIPS representation of the abstract MDP.

Action	*Discriminant*	*Effect*	*Prob.*
Move	Office	¬Office	0.9
			0.1
	¬Office	Office	0.9
			0.1
BuyCoffee	¬Office	HRC	0.8
			0.2
	Office		1.0
GetUmbrella			1.0
DelCoffee	Office,HRC	HUC,¬HRC	0.8
		¬HRC	0.1
			0.1
	¬Office,HRC	¬HRC	0.8
			0.2
	¬HRC		1.0
UserIsThirsty*		¬HUC	0.01
			0.99

Sentence	*Value*
HUC	0.9
¬HUC	0.1

one.

Having constructed $\mathcal{R}$, we now have an abstract MDP with a STRIPS representation as given in Table 3.5. We can then treat the abstract MDP as any other MDP and compute an optimal policy for it using policy iteration. This gives us the policy shown in Table 3.6. When this policy is translated into a policy in the original domain, it differs from the optimal policy for only three of the 64 states (although in states where *Office*, *HasUserCoffee*, and *HasRobotCoffee* are all true, it substitutes *GetUmbrella* for *BuyCoffee*, neither of which has any effect in those states) and — as we would expect given the method of its construction — is optimal whenever *Rain* is false or *Umbrella* is true. The values for states in this abstract policy are around the midpoints of the ranges

Table 3.6: The policy computed using the abstract MDP.

	HUC	Val.	$\neg$*HUC*	Val.
HRC, Office	GetUmbrella	17.8	DelCoffee	16.5
HRC,$\neg$*Office*	Move	17.8	Move	15.7
$\neg$*HRC, Office*	Move	17.7	Move	14.1
$\neg$*HRC,*$\neg$*Office*	BuyCoffee	17.7	BuyCoffee	14.8

of values for the states in each cluster according to the optimal policy. Moreover, the time required to compute the abstract policy was 0.01 seconds, only 2.1 percent of the optimal policy (both using policy iteration).

3.2.4 Properties of the abstract MDP

The most important property that the domain formed by the abstraction process must have is the Markov property. The Markov property states that the probabilistic outcome of every action depends only on the current state, not on any previously visited states, or previous actions.

Theorem 3 *The Markov property holds in the abstract domain (and hence it is an MDP).*

Proof: Since the Markov property holds in the original MDP, it is sufficient to show that for a given abstract action, the result of performing that action on two states in the same cluster will be the same probability distribution over clusters. Formally, if S and T are clusters and $\widetilde{S}$ is the abstract state-space, we must show that:

$$(\forall S, T \in \widetilde{S})(\forall s, s' \in S)(\sum_{t \in T} \Pr(t|A, s) = \sum_{t \in T} \Pr(t|A, s') = \Pr(T|A, S))$$

Consider two such states s and s', both in cluster S. Since they are in the same cluster, s and s' must differ only on the values of atoms not in $\mathcal{R}$ (the set of relevant

atoms).

Now consider some action A such that $\sum_{t \in T} \Pr(t|A, s) \neq \sum_{t \in T} \Pr(t|A, s')$. For this to be the case there must be (at least) one atom $p \in \mathcal{R}$ which appears negated and unnegated respectively in the effects lists of the discriminants containing s and s' for A. Since $p \in \mathcal{R}$, the discriminants of S and s' for A must also be in $\mathcal{R}$ (by the definition of $\mathcal{R}$), so s and s' must differ on the values of atoms in $\mathcal{R}$ and this contradicts the statement made in the previous paragraph, so no such action can exist. □

Since the Markov property holds, we know that the abstract domain is an MDP, so policy iteration will produce an optimal policy (for the abstract domain). If the Markov property does not hold, a policy produced using policy iteration is no longer guaranteed to be optimal. The proof shows that for any action and any two states in a single cluster, the sum of the transition probabilities from the two states to all the states in any other cluster is the same.

3.3 Properties of the Abstract Policy

We are interested in two properties of the abstract domain. The time required to compute a policy using abstraction compared with the time required to find an optimal policy, and how close to optimal the policy is.

As to the first question, since the time required for policy iteration is a function of the size of the state space, and the size of the state space is exponential in the number of underlying atoms, any reduction in the cardinality of $\mathcal{R}$ will result in an exponential reduction in the size of the state space and hence in computation time. Even reducing the domain by a single atom will halve the size of the state space, and produce a large computational saving when performing policy iteration.

This speed-up comes at the cost of generating possibly less-than-optimal policies.

However, we can at least bound the difference in value between an optimal policy $\pi*$ and the policy for the concrete domain π derived from the optimal abstract policy $\tilde{\pi}$ as in step 6 of Figure 3.3. Along with $\tilde{\pi}$, policy iteration will produce an abstract value function $V_{\tilde{\pi}}$. We can treat $V_{\tilde{\pi}}$ as an estimate of the true value of policy π; that is, $V_{\tilde{\pi}}(\tilde{s})$ approximates the value of $V_\pi(s)$ where $s \in \tilde{s}$ is any state. The difference between $V_{\tilde{\pi}}(\tilde{s})$ and $V_\pi(s)$ is a measure of the accuracy of policy iteration over the abstract state space in estimating the value of the induced policy in the concrete state space. To find this bound, we will need to use the discounting factor β, and the *reward span* of each cluster $\tilde{s}$.

Definition 3 The reward span of a cluster $\tilde{s}$, $span(\tilde{s})$ is the maximum range of possible rewards for that cluster. That is; $span(\tilde{s}) = \max(\tilde{s}) - \min(\tilde{s})$.

The reward span for a cluster is the maximum degree to which the estimate $\tilde{R}(\tilde{s})$ of the immediate reward associated with a state $s \in \tilde{s}$ differs from the true reward $R(s)$ for that state. Let δ denote the maximum reward span over all the clusters in $\tilde{\mathcal{S}}$.

Theorem 4 *For any* $s \in \tilde{s} \subseteq \mathcal{S}$,

$$|V_{\tilde{\pi}}(\tilde{s}) - V_\pi(s)| \leq \frac{\delta}{2(1-\beta)}$$

Proof: Let ρ_i be the expected (undiscounted) reward the agent will receive for performing its ith action from policy π starting in state s (ρ_i is just the reward received at that single step). Hence

$$V_\pi(s) = R(s) + \beta\rho_1 + \beta^2\rho_2 + \ldots$$

Let $s \in \tilde{s}$, then $R(s) - \frac{\delta}{2} \leq \tilde{R}(\tilde{s}) \leq R(s) + \frac{\delta}{2}$, so

$$V_{\tilde{\pi}}(\tilde{s}) \leq R(s) + \frac{\delta}{2} + \beta(\rho_1 + \frac{\delta}{2}) + \beta^2(\rho_2 + \frac{\delta}{2}) + \ldots$$

and

$$V_{\widetilde{\pi}}(\widetilde{s}) \geq R(s) - \frac{\delta}{2} + \beta(\rho_1 - \frac{\delta}{2}) + \beta^2(\rho_2 - \frac{\delta}{2}) + \ldots$$

$$\begin{aligned} |V_\pi(s) - V_{\widetilde{\pi}}(\widetilde{s})| &\leq \frac{\delta}{2} + \beta\frac{\delta}{2} + \beta^2\frac{\delta}{2} + \ldots \\ &= \frac{\delta}{2}(1 + \beta + \beta^2 + \ldots) \\ &= \frac{\delta}{2(1-\beta)} \end{aligned}$$

□

A more useful bound is on the difference between the generated policy π and an optimal policy $\pi*$. As above, we will use the value function $V_{\pi*}$ associated with the optimal policy. The true measure of the optimality of the abstract policy is the difference between $V_\pi(s)$ and $V_{\pi*}(s)$ for any state s.

Theorem 5 *For any $s \in S$,*

$$|V_{\pi*}(s) - V_\pi(s)| \leq \frac{\beta\delta}{1-\beta}$$

Intuitively, this bound results in the same way as the previous one. The difference in value is simply the sum of all the bounds at each step. In this case, the first state bound is zero since both values are in the concrete domain. The value of the state one step ahead could be in error by at most δ since the true reward can be at most $\delta/2$ greater than the abstract reward for the optimal action, and at most $\delta/2$ less for the action selected by policy π. This value must be discounted by β. A similar argument shows that at each future step the bound on the reward received at that step is at most δ, so the sum is

$$\beta\delta + \beta^2\delta + \ldots = \frac{\beta\delta}{1-\beta}$$

as required. To formally prove the theorem, we make use of the following lemmata:

Lemma 6 *Let $\tilde{V}$ denote the (optimal) value function for the abstract MDP (Hence $\tilde{V}(\tilde{s})$ is the value given to $\tilde{s}$ by any optimal policy for the abstract MDP). Let V denote the value function for the concrete MDP. Let $s \in \mathcal{S}$, $\tilde{s} \in \tilde{S}$, and $s \in \tilde{s}$, then*

$$\left|V(s) - \tilde{V}(\tilde{s})\right| \leq \frac{\delta}{2(1-\beta)}$$

Proof: Define $V_0(s) = R(s)$,

$$V_n(s) = R(s) + \beta \max_{A \in \mathcal{A}} \sum_{t \in \mathcal{S}} \Pr(t|A, s) V_{n-1}(t)$$

This is the discounted finite horizon formulation of V. Similarly, define $\tilde{V}_0(\tilde{s}) = \tilde{R}(\tilde{s})$,

$$\tilde{V}_n(\tilde{s}) = \tilde{R}(\tilde{s}) + \beta \max_{A \in \mathcal{A}} \sum_{\tilde{t} \in \tilde{\mathcal{S}}} \Pr(\tilde{t}|A, \tilde{s}) \tilde{V}_{n-1}(\tilde{t})$$

Then

$$V(s) = \lim_{n \to \infty} V_n(s)$$

$$\tilde{V}(\tilde{s}) = \lim_{n \to \infty} \tilde{V}_n(\tilde{s})$$

And since

$$\frac{\delta}{2(1-\beta)} = \frac{\delta}{2} + \beta\frac{\delta}{2} + \beta^2\frac{\delta}{2} + \ldots$$

We must show that:

$$\left|V_n(s) - \tilde{V}_n(\tilde{s})\right| \leq \sum_{i=0}^{n} \beta^i \frac{\delta}{2}$$

Base Case: By the construction of $\tilde{R}$ it follows that

$$\left|V_0(s) - \tilde{V}_0(\tilde{s})\right| = \left|R(s) - \tilde{R}(\tilde{s})\right| \leq \frac{\delta}{2}$$

Inductive Hypothesis: Assume for all s that

$$\left|V_{n-1}(s) - \tilde{V}_{n-1}(\tilde{s})\right| \leq \sum_{i=0}^{n-1} \beta^i \frac{\delta}{2}$$

Lemma 7 *Assuming the inductive hypothesis, we have for any action A,*

$$\left|\sum_{t\in\mathcal{S}} \Pr(t|A,s)V_{n-1}(t) - \sum_{\tilde{t}\in\tilde{\mathcal{S}}} \Pr(\tilde{t}|A,\tilde{s})\tilde{V}_{n-1}(\tilde{t})\right| \leq \sum_{i=0}^{n-1}\beta^i\frac{\delta}{2}$$

Proof: By Theorem 3, if $s \in \tilde{s}$, then for all A we have

$$\Pr(\tilde{t}|A,\tilde{s}) = \sum_{t\in\tilde{t}} \Pr(t|A,s)$$

So

$$\begin{aligned}\left|\sum_{t\in\mathcal{S}} \Pr(t|A,s)V_{n-1}(t) - \sum_{\tilde{t}\in\tilde{\mathcal{S}}} \Pr(\tilde{t}|A,\tilde{s})\tilde{V}_{n-1}(\tilde{t})\right| &= \sum_{tin\mathcal{S}} \Pr(t|A,s)\left|V_{n-1}(t) - \tilde{V}_{n-1}(\tilde{t})\right| \\ &\leq \sum_{t\in\mathcal{S}} \Pr(t|A,s)\sum_{i=0}^{n-1}\beta^i\frac{\delta}{2} \\ &\leq \sum_{i=0}^{n-1}\beta^i\frac{\delta}{2}\text{as required.}\end{aligned}$$

□

Let A be the action that maximizes $\tilde{V}_n(\tilde{s})$, let B maximize $V_n(s)$. Then

$$V_n(s) = R(s) + \beta\sum_{t\in\mathcal{S}} \Pr(t|B,s)V_{n-1}(t)$$

$$\tilde{V}_n(\tilde{s}) = \tilde{R}(\tilde{s}) + \beta\sum_{\tilde{t}\in\tilde{\mathcal{S}}} \Pr(\tilde{t}|A,\tilde{s})\tilde{V}_{n-1}(\tilde{t})$$

Let $f = \sum_{i=0}^{n-1}\beta^i\delta/2$, let

$$\begin{aligned}\mathbf{A} &= \sum_{t\in\mathcal{S}} \Pr(t|B,s)V_{n-1}(t) \\ \mathbf{B} &= \sum_{t\in\mathcal{S}} \Pr(t|A,s)V_{n-1}(t) \\ \mathbf{C} &= \sum_{\tilde{t}\in\tilde{\mathcal{S}}} \Pr(\tilde{t}|A,\tilde{s})\tilde{V}_{n-1}(\tilde{t}) \\ \mathbf{D} &= \sum_{\tilde{t}\in\tilde{\mathcal{S}}} \Pr(\tilde{t}|B,\tilde{s})\tilde{V}_{n-1}(\tilde{t})\end{aligned}$$

Then by the definitions of A and B, we have $\mathbf{A} \geq \mathbf{B}$, and $\mathbf{C} \geq \mathbf{D}$, and by Lemma 7, $|\mathbf{B} - \mathbf{C}| \leq f$, and $|\mathbf{A} - \mathbf{D}| \leq f$.

If $\mathbf{C} \geq \mathbf{B}$, then $\mathbf{C}-\mathbf{A} \leq f$. From $\mathbf{C} \geq \mathbf{D}$ and $|\mathbf{A}-\mathbf{D}| \leq f$ it follows that $\mathbf{A}-\mathbf{C} \leq f$,so $|\mathbf{C}-\mathbf{A}| \leq f$. Similarly, if $\mathbf{C} < \mathbf{B}$, then $\mathbf{A} > \mathbf{C}$, and since $\mathbf{C} \geq \mathbf{D}$ and $\mathbf{A}-\mathbf{D} \leq f$, we have that $|\mathbf{A}-\mathbf{C}| \leq f$. In full,we have

$$\left|\sum_{t\in\mathcal{S}} \Pr(t|B,s)V_{n-1}(t) - \sum_{\widetilde{t}\in\widetilde{\mathcal{S}}} \Pr(\widetilde{t}|A,\widetilde{s})\widetilde{V}_{n-1}(\widetilde{t})\right| \leq \sum_{i=0}^{n-1}\beta^i\frac{\delta}{2}$$

From the definitions given above,

$$\begin{aligned}
\left|V_n(s) - \widetilde{V}_n(\widetilde{s})\right| &= \left|R(s) + \beta\sum_{t\in\mathcal{S}} \Pr(t|B,s)V_{n-1}(t) - \widetilde{R}\widetilde{s} - \beta\sum_{\widetilde{t}\in\widetilde{\mathcal{S}}} \Pr(\widetilde{t}|A,\widetilde{s})\widetilde{V}_{n-1}(\widetilde{t})\right| \\
&\leq \left|R(s) - \widetilde{R}(\widetilde{s})\right| + \beta\left|\sum_{t\in\mathcal{S}} \Pr(t|B,s)V_{n-1}(t) - \sum_{\widetilde{t}\in\widetilde{\mathcal{S}}} \Pr(\widetilde{t}|A,\widetilde{s})\widetilde{V}_{n-1}(\widetilde{t})\right| \\
&\leq \frac{\delta}{2} + \beta\sum_{i=0}^{n-1}\beta^i\frac{\delta}{2} \\
&\leq \sum_{i=0}^{n}\beta^i\frac{\delta}{2} \text{ as required.}
\end{aligned}$$

□

Corollary 8 *The preceding lemma states that*

$$\left|V(s) - \widetilde{V}(\widetilde{s})\right| \leq \frac{\delta}{2(1-\beta)}$$

Since $V(s)$ is the value of state s for any optimal policy in the concrete MDP, and $\widetilde{V}(\widetilde{s})$ is the value of $\widetilde{s}$ for any optimal policy in the abstract MDP, the same relationship applies to the values of specific optimal policies, namely for all s,

$$|V_{\pi*}(s) - V_{\widetilde{\pi}}(s)| \leq \frac{\delta}{2(1-\beta)}$$

Proof: (of Theorem 5). From Corollary 8, we have that for all s

$$|V_{\pi*}(s) - V_{\widetilde{\pi}}(s)| \leq \frac{\delta}{2(1-\beta)}$$

Now, from Theorem 4, we have that for all state s,

$$|V_{\tilde{\pi}}(s) - V_{\pi}(s)| \leq \frac{\delta}{2(1-\beta)}$$

So

$$|V_{\pi*}(s) - V_{\pi}(s)| \leq \frac{\delta}{1-\beta}$$

The first term in each series is $\delta/2$. This is the term for the difference between $R(s)$ and $\widetilde{R}(\tilde{s})$, but since $V_{\pi*}(s)$ and $V_{\pi}(s)$ both have $R(s)$ as their first term, they must agree on this value, so

$$\begin{aligned} |V_{\pi*}(s) - V_{\pi}(s)| &\leq \frac{\delta}{1-\beta} - \delta \\ &\leq \frac{\beta\delta}{1-\beta} \text{ as required.} \end{aligned}$$

□

3.4 Results

We examined two domains in our investigation of the abstraction algorithms. Since we have yet to apply these algorithms in any real-world domains, both have been created by hand, and the first was designed specifically with this abstraction scheme in mind. This should result in the first domain performing particularly well as an example of abstraction. The second domain was adapted from (Smith and Peot 1993), and was specifically chosen as a typical planning problem for other planners. It is not particularly well suited for abstraction as there is only a single possible set $\mathcal{IR}$ that results in a state space smaller than the concrete problem (a second possible $\mathcal{IR}$ exists, but has a larger state space, and worse reward span than the one we use here). See Appendix A for the details of both these domains and their abstractions.

The COFFEE domain, is an extension of the example in Chapter 1. It contains nine propositions and nine actions, producing a 512 state MDP. There are three possible

Table 3.7: Results of abstraction for the COFFEE domain.

Abstract value vs. true policy value	5 Proposition domain	6 Proposition domain	8 Proposition domain
No. of errors in value of state	512	512	512
Average error	5.00	2.59	1.00
Standard Deviation	2.71	0.89	0.00
Maximum Error	8.5	3.5	1.0
Predicted Bound	8.5	3.5	1.0
Time required to compute policy	0.14s	0.89s	35.55s
true abstract policy value vs. optimal policy			
No. of errors in action to choose	187 36.5%	85 16.6%	39 7.6%
No. of errors in value of state	348 68.0%	256 50.0%	192 37.5%
Average error	4.12	0.91	0.48
Standard Deviation	3.80	1.39	0.76
Maximum Error	14.17	5.93	1.89
Predicted Bound	16.15	6.65	1.90

abstractions for this domain, with five, six and eight propositions respectively. Policy iteration on the complete problem required 254.33 seconds on a Sun 4/60. State values ranged from -7.0 to 30.0. The results of the abstractions are summarized in Table 3.7.

As the table shows, the more fine-grained the abstraction, the better the resulting policy is. The number of errors in the action to select drops until over 90 percent of states have their correct action in the eight proposition domain, and the errors in the estimates of state values drop as well. Requiring very little computation time, the five proposition abstraction still produces quite a reasonable policy. Although state values can be anywhere from zero to twenty, the policy produced from five propositions will differ from the optimal policy by an average value of just 4.12, only an eleven percent

Table 3.8: Results of using optimal abstract policies as initial policies when computing an optimal policy for the concrete MDP.

(a) Time to find optimal policy

MDP to solve	Initial policy	Time	Iterations
32 state	None	0.89s	6
64 state	None	4.93s	7
	32 state	5.59s	8
256 state	None	214.71s	8
	32 state	214.68s	8
	64 state	84.71s	3
512 state	None	1531.67s	8
	32 state	1564.14s	8
	64 state	643.58s	3
	256 state	472.53s	2

(b) Total time to find optimal policy for 512 state MDP

Use 256 state	Use 64 state	Use 32 state	Total time
Yes	Yes	Yes	563.72
Yes	Yes	No	562.17
Yes	No	Yes	688.10
Yes	No	No	687.24
No	Yes	Yes	650.06
No	Yes	No	648.51
No	No	Yes	1565.03
No	No	No	1531.67

error for a 99 percent reduction in computation time. The eight proposition abstraction performs even better, producing a 1.3 percent average error in the value of a state for an 86 percent reduction in computation time.

We can also examine the possibility of computing an optimal policy by using the result of an abstract policy as the initial policy for applying policy iteration to the concrete (or a less abstract) policy. The results for this experiment, again in the COFFEE domain are given in Table 3.8. As the table shows, there can be considerable savings by using a series of abstractions to calculate the optimal value for the concrete domain. The fastest

way to compute the optimal value for the concrete domain is to compute the optimal value for the 64 state domain, use that to compute the 256 state optimal value, and then use that to find the optimal policy. This requires only 37 percent of the computation time of computing the optimal policy directly. Perhaps surprisingly, computing the 32 state optimal policy is o help at all for computing more fine-grained policies, and in the case of the 64 state domain, actually slows the process of policy iteration. At present, we have no way of predicting when this will occur.

In order to use these results for efficient computation of the optimal policy, the results suggest that using as many levels of abstraction as possible is a good strategy. Although it isn't the best possible series of abstractions in this case, it is very close, and informal arguments suggest that using all available abstractions should be the best strategy in most cases. If multiple processors are available, all the possible permutations of abstractions could be run in parallel, and computation halted as soon as any processor computed an optimal policy. This method allows us to guarantee that computing the optimal policy using abstractions will be no worse than computing it directly.

The second domain, the BUILDER domain involves an agent that must join two objects together. For maximum reward, the objects must be machined to the correct shape, clean, painted, and joined together. The reward for any given state is simply the sum of the individual rewards for all of these attributes. The state contains nine propositions (512 states) and ten actions. Policy iteration on the entire state space required 201.86 seconds. State values range from 0.0 to 20.0. The results are summarized in Table 3.9.

As the table shows, the abstraction was again good, especially considering the small size of the abstraction (only 32 states). The average error in the value of a state is 5.99, which is quite large at 30 percent of the possible range of values, but the abstract policy required less than one percent of the computation time of the optimal policy. For this domain, since the abstract state space is so much smaller than the concrete one, some

Table 3.9: Results of abstraction for the BUILDER domain.

Abstract value vs true policy value	
No. of errors in value of state	$478 = 93.4\%$
Average error	2.69
Standard Deviation	1.86
Maximum Error	6.0
Predicted Bound	6.0
Time required to compute policy	$0.11s = 0.05\%$
true abstract policy value vs. optimal policy	
No. of errors in action	$309 = 60.4\%$
No. of errors in value	512
Average error	5.99
Standard Deviation	2.16
Maximum Error	10.00
Predicted Bound	11.40

local way of improving the policy, such as the search procedure described in the next chapter, may be very valuable. Since there is no abstraction that is more fine-grained than this, we cannot choose another abstraction if the bound on the difference between the abstract and optimal policies of 11.4 is unacceptable.

Chapter 4

Planning by Heuristic Search

The abstraction scheme described in the previous chapter produces an approximately optimal policy for a relatively small investment of computation time. This computation must be performed before any actions are carried out, but if time is available as the plan is executed, extra computations could be used to make local improvements to the policy. This chapter introduces the idea of treating the abstract policy as a set of default reactions, and using *heuristic search* to try to improve on these at run time if computation time is available. A major advantage to be gained by using search is that its computation time generally does not depend on the size of the state space, only on its connectivity (the number of states that can be reached by performing a single action from some state).

Although the abstraction process of Chapter 3 is an obvious place to obtain a heuristic from, the algorithm described in this chapter can be (and has been) used with heuristics obtained from other sources. In many domains, a heuristic may already be provided by the user. In others, such as high-level robot navigation, a domain specific heuristic — for example, one based on geographic relationships — may be easy to construct by hand. We will assume throughout this chapter that a heuristic function $\mathcal{V}$ is available, without concerning ourselves with its source.

If a heuristic exists which is a fairly reliable estimate of the value of each state, we can use it to select a good action to perform in each state in a way analogous to minimax search. A simple search algorithm will tend to be greedy, selecting actions which lead to the greatest immediate reward, but by using a heuristic we can hope to increase the

importance of long-term gains, and hence produce a better planner.

In general, the more computation time is spent generating the heuristic, the more accurate it will be, and similarly, if deeper search is performed, we can typically expect a better decision to be made — this may not always be the case (see (Pearl 1984) for a proof of this for minimax search). In practice, searching deeper generally leads to better performance. There is an obvious tradeoff between initial computation time while constructing the heuristic and computation time spent searching. At one extreme, we can compute a perfect valuation function for states initially, and then perform no search, while at the other, we have a greedy search with no heuristic information to guide it.

Classical planning makes use of search as well, but there are important differences between the classical approach, and our search algorithm. Classical planning is typically goal directed, which is impossible with the complex reward structures we are interested in; but even when the search isn't goal directed, classical planning algorithms search until a goal state is found. Even ignoring the fact that, like game playing problems, the real-time nature of our approach prohibits exhaustive searches, the fact that we have no explicit goal states, and are planning for the infinite horizon case, means that there is no final state where search stops. Our use of partial decision tree search and game-like heuristic techniques is dictated by this inability to perform complete searches.

4.1 Interleaving Search and Execution

Time-critical domains provide a minimum of computation time in which to plan, hence it is important to restrict the space to be searched as much as possible. To this end, we propose an algorithm where actions are executed as soon as they have been selected. Because of the stochastic nature of the domains we are interested in, this results in a considerable saving in computation. Performing an action in a certain state can leave the

system in a number of different states, so a planning algorithm that constructs a sequence of actions would need to find an action to perform for each of the possible outcomes of the action it selects first. By executing actions as soon as they are selected, we know (since the MDP is completely observable) which of the possible outcomes actually occurred, and only need search for the next action to perform from a single state, rather than many.

As we would expect, the saving in computation gained by interleaving execution with planning is considerable. Let b be the maximum number of outcomes each action could have. In a search for a sequence of a actions, search without execution will require an action to be selected for $\sum_{i=0}^{a-1} b^i$ states compared with only a states for search with execution.

4.2 Planning in MDPs by Searching

As we described in the previous section, there is a great deal to be gained in terms of computational savings by interleaving planning with execution. In view of this, the planning algorithm can be viewed at the highest level as follows:

1. Calculate the best action to perform in the current state, using the heuristic function if necessary.
2. Execute the selected action.
3. Observe the new state of the system and return to step 1.

Steps 2 and 3 require little explanation. Although the algorithm as it stands never terminates, this is consistent with the process-like domains that the MDP representation is ideally suited for. If the domain contains goal states in which the agent has no more to do or other self-absorbing states, the algorithm might terminate when such a state is reached. In general, however, the agent will continue planning and acting indefinitely.

4.3 The Search Algorithm

This section deals with Step 1 of the algorithm described above. The first point to make is that the algorithm we will present makes no changes to the heuristic function — it doesn't learn more accurate estimates of the values of states — so for any given state, the action chosen to execute for that state will be the same every time. To take advantage of this, the algorithm caches the action chosen for each state it computes so that should the state be visited again, no extra computation need be performed.

The search algorithm constructs a partial decision tree rooted at the current state to determine the best action to perform. The decision tree is built to a fixed depth, and the heuristic function is used to estimate the value of the leaf nodes. Although fixed-depth search is not necessary for the algorithm to function, it allows the use of depth-first search, which is computationally efficient. Using breadth-first search (or one of its variations) would make some of the pruning algorithms more efficient, but these techniques often require considerable extra book-keeping costs. If certain information about the heuristic function is available, then pruning of the search tree is possible, and as the results below show, can be quite effective. The cached values of previously selected actions provide an even more effective form of pruning.

4.3.1 Action selection

Let s and t be states, let β be the discounting factor as before, and let $\mathcal{V}(t)$ be the value of the heuristic function at state t. Then the *estimated* expected utility of action A_i in state s is:

$$U(A_i|s) = \sum_{t \in \mathcal{S}} \Pr(t|A_i, s)V(t)$$

Where $V(s)$, the value (estimated by the search process) of a state is:

$$V(s) = \begin{cases} \mathcal{V}(s) & \text{if } s \text{ is a leaf node} \\ R(s) + \gamma(max\{U(A_j|s) : A_j \in \mathcal{A}\}) & \text{otherwise} \end{cases}$$

As we would expect, the utility of an action is simply the weighted average of the value of every state reachable by performing the action. The value $V(s)$ of state s is computed as follows: if s is at the bottom of the search tree (s is a leaf node), then we use the heuristic to approximate its value. Otherwise we add the immediate reward for state s to the discounted value of the best action to perform in that state. Figure 4.4 gives the algorithmic description of the basic search algorithm.

Figure 4.5 illustrates the search process with a partial tree of actions two levels deep, and a discounting factor β of 0.9. From the initial state s, if the agent performs action A, state t will result with probability 0.8 and state u with probability 0.2. When the tree is expanded again to include states reachable from these, the set of second states result. At this point, the heuristic is used to estimate the values of the second states as they are leaf nodes in the tree. Given that $\mathcal{V}(x) = 2$ and $\mathcal{V}(y) = 3$, the utility of action A if the agent were in state t, is $0.9 \times 2 + 0.1 \times 3 = 2.1$, and since $U(B|t) = 0.3$, A is the better action from state t, and $V(t) = R(t) + \beta \times U(A|t) = 0.5 + 0.9 \times 2.1 = 2.39$. In a similar fashion, we can compute the value of state u, which is 1.58. Given this, we determine the estimated utility of action A if the agent is in state s. $U(A|s) = 0.8 \times 2.39 + 0.2 \times 1.58 = 2.23$, and similarly, $U(B|s) = 2.94$. Since the estimated utility of action B is higher than action A, we select B as the best action to perform in state s, record that fact in the cache of previously calculated best actions, and execute action B. By observing the outcome of the action, we now know which of states v and w actually resulted, and can hence avoid searching for a best action for the one that did not.

In Figure 4.5, the tree is expanded to depth two, but this does not need to be the case. As available computation time varies, the depth of the search tree can also vary.

search-state$(s, depth)$ — **Returns the best action and its value for state** s

1. If $depth > 0$ (this isn't a leaf node)
   (a) Set $MaxValue$ to be lower than any possible value of an action
   (b) Set $BestAction$ to be null
   (c) for each action $a \in \mathcal{A}$
      i. Let $value$ be **search-action**$(a, s, depth)$, the (estimated) value of action a in state s.
      ii. If $value > MaxValue$ (this is the best action so far), let $MaxValue$ be $value$, let $BestAction$ be a
   (d) Return $BestAction, R(s) + \gamma \times MaxValue$
2. otherwise (s is a leaf node of the search tree) return $\mathcal{V}(s)$, the heuristic value of state s

search-action$(a, s, depth)$ — **Returns the value of action** a **in state** s

1. Let $value$ be 0
2. For each state t that could result from performing a in s do
   (a) Let $value$ be $value + \Pr(t|a, s) \times$ **search-state**$(t, depth - 1)$, the probability of reaching state t times the computed value of t.
3. Return $value$

Figure 4.4: The search algorithm. To begin search from state s to depth d, search-state(s, d) is called. The action that is returned is the estimate of the best action to perform.

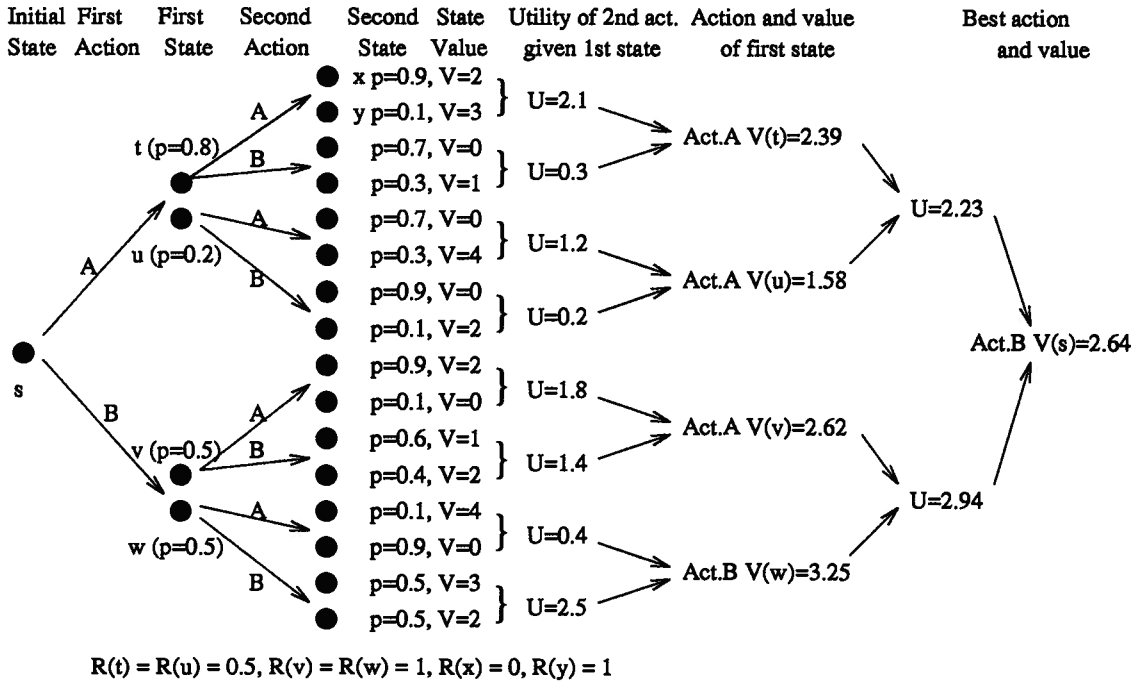

Figure 4.5: An example of a two-level search for the best action from state s.

In practice, an interruptible search using an iterative deepening technique may well be used, so that at any time, the algorithm can be interrupted, and the current best action can be performed. We cannot guarantee that deeper search will produce better results, in general however, the deeper the tree is expanded, the more accurate the estimates of action utility will tend to be, and the more confidence we should have that the action selected approaches optimality. One can view this search process as a form of *directed value iteration* (Bellman 1961). A search to depth n will result in exactly the same action for each state as performing n iterations of the value iteration algorithm with the heuristic as the initial vector of state values. Hence we can use the result that value iteration converges on an optimal policy to show that deep enough search will also find optimal actions, but such a search is too expensive to be performed in practice.

There is also a close relationship between the search algorithm and Ballard's $\star$-minimax search (Ballard 1983). Determining the value of a state is analogous to the MAX step for minimax search, while calculating the utility of an action is an AVERAGE step and is analogous to a move made by a randomized opponent in Ballard's game-playing search.

4.3.2 Pruning algorithms

As we said above, there is a close relationship between our search algorithm and minimax search. Just as alpha-beta search is a version of minimax with pruning of irrelevant subtrees, we can perform two different forms of pruning on our search trees. To make our description of these pruning techniques clearer, we will treat a single ply of search as consisting of two steps, *MAX* in which all the possible actions are considered, and the best is chosen, and *AVERAGE* where the outcomes of a particular action are combined to determine the expected value of that action.

Two type of pruning are possible. The first, *utility pruning*, is very similar to α- and

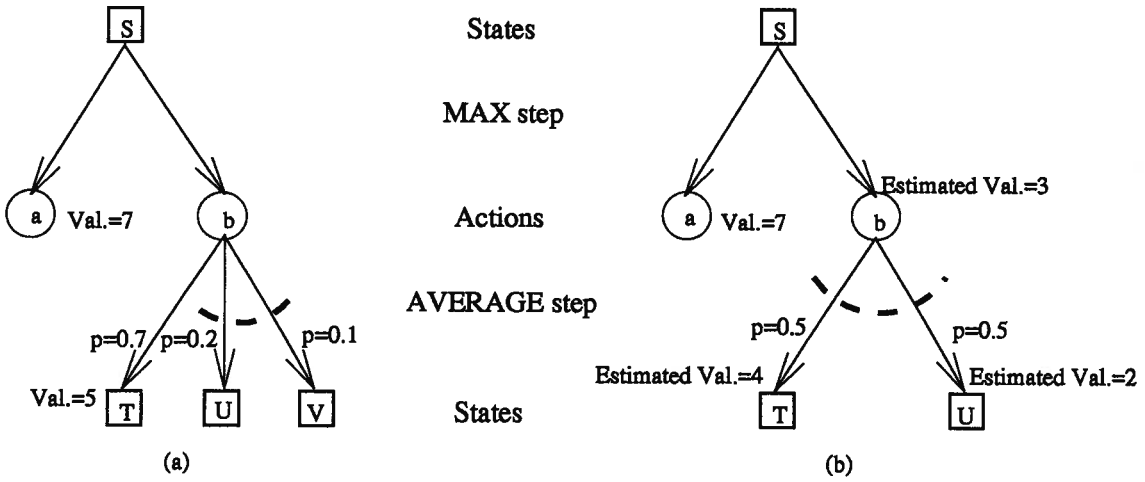

Figure 4.6: Two kinds of pruning where $\mathcal{V}(s) \leq 10$ and is accurate to ± 1. In (a), utility pruning, the trees at U and V need not be searched, while in (b), expectation pruning, the trees below T and U are ignored, although the states themselves are evaluated.

β-cuts in minimax search, and requires knowledge of the maximum and minimum values of the heuristic function. For heuristics produced by the abstraction algorithm described in Chapter 3, if R^+ and R^- are the maximum and minimum rewards for any state in the MDP, then the maximum and minimum expected values for any state are bounded by $R^+/(1-\beta)$ and $R^-/(1-\beta)$ respectively. For the second type of pruning, *expectation pruning*, we require bounds on the error associated with the heuristic function. Again, for heuristics based on our abstraction algorithm, these bounds can be computed using the result of Corollary 8 (see Section 3.3).

Utility Pruning We can prune the search at an AVERAGE step if we know that no matter what the value of the remaining outcomes of this action, we can never exceed the utility of some other action at the preceding MAX step. For example, consider the search tree in Figure 4.6 (a). We assume that the maximum value that the heuristic function can take is 10. When evaluating action b, since we know that

the value of the subtree rooted at T is 5, and the best that the subtrees below U and V could be is $0.1 \times 10 + 0.2 \times 10 = 3$, the total cannot be larger that $3.5 + 3 = 6.5$, so neither of nodes U and V need be expanded. This type of pruning requires knowledge of the maximum value of the heuristic. Although we haven't implemented it, we can use the minimum value in a more restricted fashion. If, for example, the value of action a was 3, and the minimum value of the heuristic was 0, then the value of b must be at least $0.7 \times 5 + 0.3 \times 0 = 3.5$, so we can tell that b is the best action without searching nodes U and V. This process will only work if b is the last action to be evaluated, and we are at the topmost level of the search tree, so the value of b is unimportant. For the maximum amount of pruning to be performed, the possible outcomes of each action should be searched in order of their probability of occurring, with outcomes of high probability searched first.

To implement Utility pruning, we need to modify the algorithm in Figure 4.4 as follows:

We need a new constant, $MaximumStateValue$, which is the maximum expected value of any state.

search-state$(s, depth)$

The only change needed is in the call to **search-action** in 1(a)i. The function call becomes **search-action**$(a, s, depth, MaxValue)$.

search-action$(a, s, depth, \alpha)$ — **Find the value of action a in state s by searching to depth *depth*. The value is irrelevant if it is less that α.**

1. Let *value* be 0
2. Let *RemainingProb* be 1
3. For each state t that could result from performing a in s do

(a) Let *value* be *value*+$\Pr(t|a,s)\times$ **search-state**$(t, depth-1)$, the probability of reaching state t times the computed value of t.

(b) Let *RemainingProb* be *RemainingProb* $-\Pr(t|a,s)$

(c) If $value+RemainingProb\times MaximumStateValue < \alpha$ then return $\alpha-1$. We can stop searching, this cannot be the best action.

4. Return *value*

Expectation Pruning For this type of pruning we need to know the maximum error associated with the heuristic function. If the search is at a maximizing step, and, even taking into account the error in the heuristic function, the action we are investigating cannot have as high a utility as some other action for which a utility is already known, then we do not need to expand this action further. For example, consider Figure 4.6 (b), where we assume that for all states s, $\mathcal{V}(s)$ is within ± 1 of the true (optimal) value of s. Having determined that $U(a|S) = 7$, we know that any potentially better action must have a heuristic value of at least 6. Since $\Pr(T|b,S)\mathcal{V}(T) + \Pr(U|b,S)\mathcal{V}(U) \leq 4$, even if b is as good as possible, it cannot be better than a, so there is no need to search the subtrees below states T and U.

To implement Expectation pruning, we make the following further modifications to Figure 4.4:

Again, we need a new global variable *HeuristicError*, which is the maximum possible difference between the heuristic value of a state and its value in an optimal policy.

search-action$(a, s, depth, \alpha)$

The following additional steps must be included at the beginning of the algorithm (before step 1. in Figure 4.4).

1. Let $ValueEstimate$ be 0
2. For each state t that could result from performing a in s do
   (a) Let $ValueEstimate$ be $ValueEstimate + \Pr(t|a,s) \times \mathcal{V}(t)$, the probability of state t times its heuristic value.
3. If $ValueEstimate + HeuristicError < \alpha$ then return $\alpha - 1$, we need not look any deeper into this action.

As we have implemented it above, utility pruning is very simple to perform. It requires very little extra computation, and as our results below indicate, can result in quite a respectable improvement in computation time. Expectation pruning requires a more significant modification of the search algorithm to check all the outcomes of an action to see if they justify searching the tree below that action, but in domains where the heuristic function is quite accurate, it can still offer a substantial performance improvement, as the results in the next section will show. Expectation pruning is quite closely related to what Korf (1990) calls alpha-pruning. The difference is that while Korf relies on a property of the heuristic that it is always increasing, we rely on an estimate of the actual error in the heuristic.

The comments in the previous paragraph are specific to the implementation of the search algorithm as a depth-first search. Using an iterative deepening search seriously limits the applicability of utility pruning since the final value of an action is only known when the last round of deepening is performed. On the other hand, iterative deepening removes the additional computation requirements that make expectation pruning more expensive to perform.

4.4 Results

As before, let $n = |\mathcal{A}|$ be the number of actions. Let b be the maximum number of possible outcomes for any action and state. We will assume that we are constructing a search tree without pruning to a fixed depth d. The number of nodes (states) expanded while calculating the best action for a single state is $1 + bn + (bn)^2 + \ldots + (bn)^d = ((bn)^{d+1} - 1)/(bn - 1)$. Over a series of such calculations, the cost is slightly less than this because we can reuse previous calculations, but the complexity is $O((bn)^d)$. The actual size of the state space has no effect on the algorithm; rather it is the number of states visited that determines the cost. In most domains this number should be considerably less than the total number of states. More importantly, the complexity of the algorithm is constant (with regard to the number of states), and execution time per action can be bounded for a fixed search depth and branching factor.

We have performed experiments to test the effectiveness of the searching algorithm in a number of different domains. Figure 4.7 shows how both the time required for searching and the value of the induced policy (the policy that results if searching is conducted on every state) tends to increase with the search depth. The experiments were performed on a Sun 4/60, with no pruning of the search tree.

We also examined in detail the way the induced policy changes as search depth increases for three different problems. The first two are the COFFEE domain, with a heuristic computed using the 32 state abstract MDP, and the 256 state abstract MDP respectively. The third problem we examined was the BUILDER domain, using the 32 state abstract MDP to compute the heuristic. The results are presented in Table 4.10.

As the table shows, the effects of searching vary widely from domain to domain. In the first problem, the heuristic is quite poor, and the policy produced by searching converges quite slowly to the optimal one. Even though the number of states for which

Table 4.10: Comparison of the induced policy as search depth increases for three different search problems.

COFFEE **domain, coarse-grained heuristic**

	Optimal policy	1 step search	2 step search	3 step search	4 step search
Average state value	22.607	18.686	19.961	20.363	20.509
Percentage of optimal	100.0	82.7	88.3	90.1	90.7
Maximum error	0	14.169	10.607	10.607	10.607
Average error	0	3.921	2.646	2.245	2.098
No. of non-zero errors	0	320	288	288	288
Average non-zero error	0	6.274	4.704	3.991	3.730

COFFEE **domain, fine-grained heuristic**

	Optimal policy	1 step search	2 step search	3 step search	4 step search
Average state value	22.607	21.928	22.607	22.607	22.607
Percentage of optimal	100.0	97.0	100.0	100.0	100.0
Maximum error	0	1.890	0	0	0
Average error	0	0.679	0	0	0
No. of non-zero errors	0	224	0	0	0
Average non-zero error	0	1.553	0	0	0

BUILDER **domain**

	Optimal policy	1 step search	2 step search	3 step search	4 step search
Average state value	18.173	12.227	18.112	18.008	18.021
Percentage of optimal	100.0	67.3	99.7	99.1	99.2
Maximum error	0	10.003	0.702	5.050	5.050
Average error	0	5.947	0.062	0.166	0.152
No. of non-zero errors	0	512	207	141	91
Average non-zero error	0	5.947	0.153	0.602	0.857

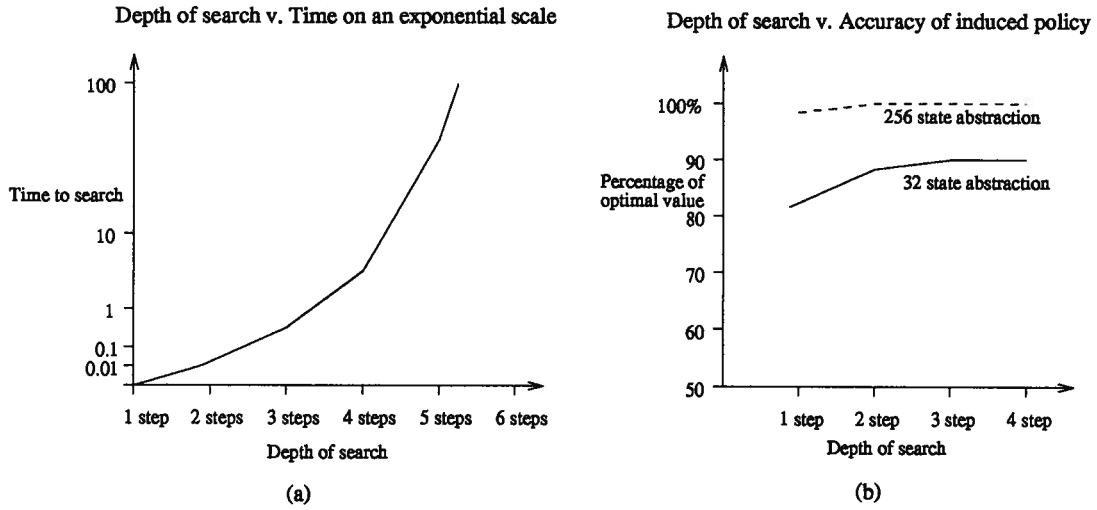

Figure 4.7: Graphs of (a) search time, and (b) policy quality against search depth for the COFFEE domain.

the optimal policy has not been found[1] remains relatively constant, the expected value of states continues to rise, which shows that the optimal action is being selected for a greater and greater number of states, although a few pathological ones with high errors remain, even after searching four steps deep.

In the second problem, the heuristic is already very accurate. Searching to one step is essentially equivalent to selecting the same actions as the abstract policy, and even one step search performs very well (better than four step search with the coarse-grained abstraction). This illustrates the trade-off between abstraction complexity and search depth. By spending 35 seconds initially rather than 0.14 seconds, we can spend only 0.003 seconds computing each action, rather than five seconds, and still perform better. In fact, since the search procedure converges on the optimal policy when searching to depth two in this domain, we can guarantee an optimal policy, and still only spend 0.03

[1] A non-zero error means that some state that could be visited by following the policy has a less-than-optimal action.

seconds per action in the search process.

The third search problem illustrates an important point; the search procedure doesn't always perform better as search depth increases. In this domain, we once again have a coarse-grained abstraction, which makes searching to depth one perform quite poorly. However, a two-step search increases performance dramatically, and in fact is better than searching to three or four steps, at least in terms of the average value of a state. More detailed analysis of the policies produced by each depth of search reveals that for almost all states the value of the policy continues to improve as search depth increases, but there are a small number of pathological states for which the search algorithm performs very badly.

4.4.1 Execution and caching of values

We have also performed experiments to investigate the value of caching previously computed best actions for state, and the value of interleaving execution with search. Table 4.11 summarizes our results. The columns where execution is interleaved with search show the standard algorithm as we presented it. For search without execution, the agent performs the standard search, determines the best action, and then rather than executing it, searches again to find the best action to perform for all possible outcomes of the action.

Unsurprisingly, cached search interleaved with execution is the most efficient method. The size of the domain will have a considerable effect on the value of caching. In this case, the domain contains 256 states but we are only performing ten actions, so caching has relatively little effect. However, since in most cases more searches will be performed, if the space is available to cache values, it is a worthwhile tradeoff against the time required to recompute previously computed values. One surprise from this table is how well the cached search without execution performs. This is due to the small number

Table 4.11: A comparison of time to search for ten actions both with and without caching of previous best actions, and execution.

Search depth	Execution Interleaved		No Execution	
	Caching	No cache	Caching	No cache
1	0.01	0.02	5.19	26.7
2	0.04	0.06	5.41	281
3	0.42	0.51	7.14	2780
4	4.48	5.68	15.4	-
5	55.9	56.9	102	-
6	219	230	272	-

of states. Because the algorithm caches the action whenever it computes for a state, the interleaved execution algorithm will only cache values for at most ten states. In comparison, if each action has n possible outcomes, the no-execution algorithm can cache up to $1 + n + n^2 + \ldots + n^9$. In practice this means that for a domain of this size, the algorithm quickly finds itself looking for actions for states it has already evaluated, and hence not performing any computation. The column for search without caching or execution gives an idea of how badly search without execution but with caching would perform if the state space was big enough that it more rarely got to reuse its calculations.

4.4.2 Effectiveness of pruning

Figure 4.8 shows the performance of the pruning algorithm in both the COFFEE and BUILDER domains. The two graphs illustrate the large differences in the performance of the pruning algorithms on different domains. For the COFFEE domain, utility pruning is ineffective — it requires more time than no pruning at all — while expectation pruning performs remarkably well, saving over 60 percent of the computation time for deep search trees. For the BUILDER domain, utility pruning is extremely effective, pruning more than 40 percent of all states for depth five search, and required only half the computation

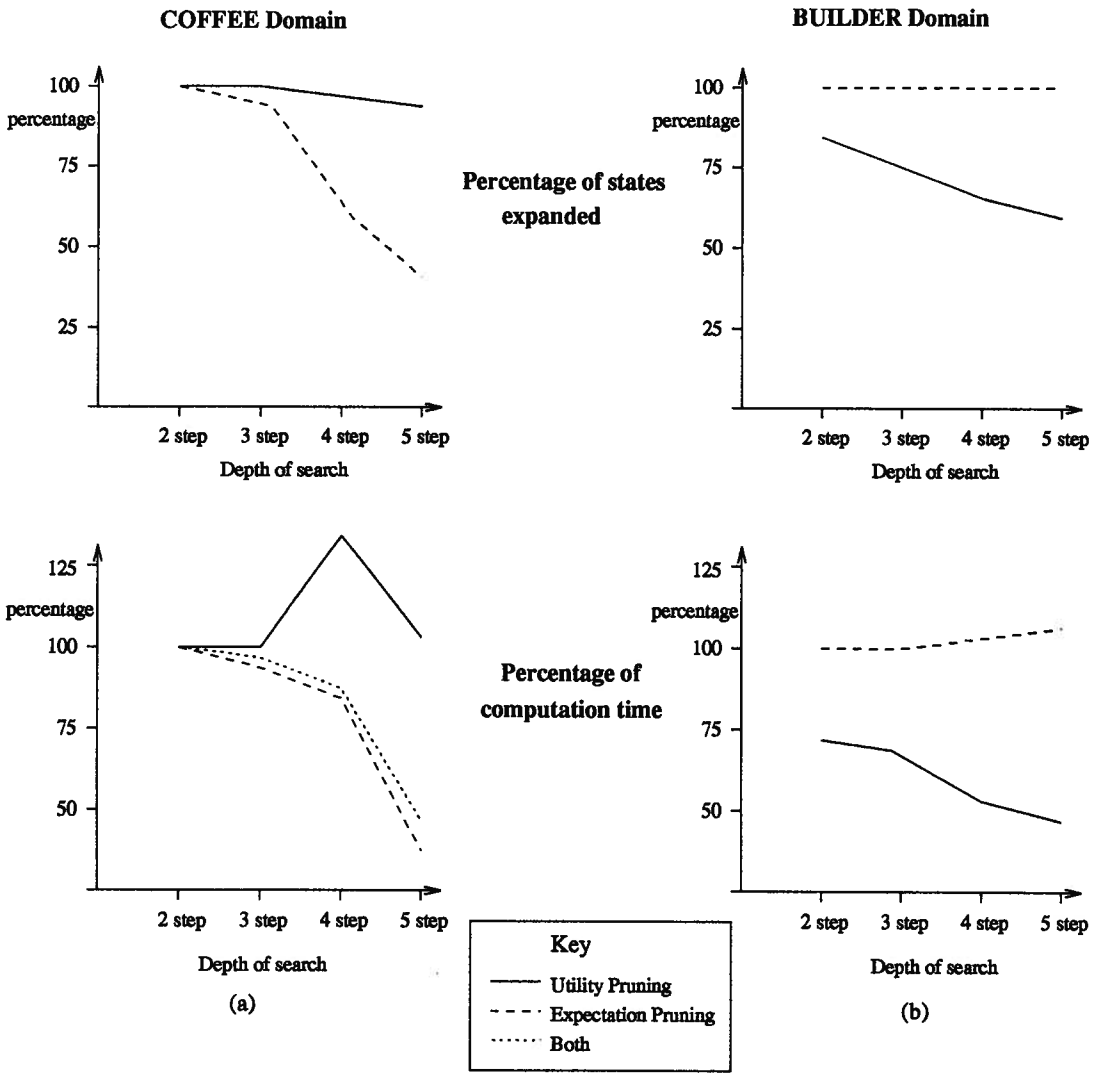

Figure 4.8: Pruning effectiveness for (a) the COFFEE, and (b) the BUILDER domains.

Table 4.12: Expectation pruning when only the top of the search tree is pruned. Values are percentage of value with no pruning.

Search depth	Prune depth	Percentage of states pruned	Percentage of search time
3	3	89.3	80.6
	2	92.6	89.0
4	4	85.7	97.3
	3	88.3	88.9
	2	92.4	94.5
5	5	82.3	89.8
	4	84.3	84.8
	3	87.4	89.5
	2	91.2	95.6

time of search without pruning. The coarse-grained abstraction scheme of the BUILDER domain results in quite large errors being possible in the heuristic function, so expectation pruning is ineffective in this domain. We used the heuristic produced from the 256 state abstract MDP for the COFFEE domain, and the error bound of ± 1 is low enough that expectation pruning works particularly well. This suggests that the pruning algorithms applied in various different problems should be tailored to the quality of the heuristic function. The two pruning algorithms will often prune the same nodes, so using utility pruning when the heuristic is poor, and expectation pruning when it is good may prove to be a good strategy.

In many domains, the gains due to expectation pruning in term of states expanded are not matched in terms of time to search because of the additional cost of pruning. We have run some experiments in a smaller (256 state) domain which show that for maximal savings of computation time, expectation pruning should not be performed all the way to the bottom of the search tree. These results are summarized in Table 4.12.

In this domain, expectation pruning is much less effective than in the COFFEE domain

given above. This emphasizes the computational requirements. As we would expect, as the depth to which pruning is performed decreases, the number of states which must be examined in the search increases, but the time required to perform the search tends to decrease at first as the cost of attempting to prune the large numbers of states at bottom of the tree is reduced, and then increases again as the lack of pruning means that more nodes must be searched. At least for this search problem, the optimal pruning depth seems to be one level earlier than the bottom of the tree, but we have no way at present of estimating this value in advance for a given domain and heuristic.

Chapter 5

Conclusions

This thesis has presented two components of a planning system for domains represented as Markov decision processes. Chapter 3 describes a way of creating close to optimal policies using abstraction, while Chapter 4 demonstrates the use of heuristic search to plan and execute actions at run-time. Putting the two together to build a complete planning system produces the following high-level algorithm:

1. *Prior to run-time:* Use abstraction to produce 1) an abstract policy that can be used as a set of default reactions, and 2) an estimate of the value of each state, to be used as a heuristic function.

2. *At run-time:* If there is sufficient computation time, use the search algorithm to compute the best action for the current state, and execute the action. Otherwise, execute the default reaction provided by the abstraction process.

3. If a goal state has been reached (if such states exist), end. Otherwise, observe the new state, and return to step 2.

This algorithm is designed for use in domains where time is critical. The accuracy of the heuristic can be tailored to the amount of start-up computation time available, and the depth of the search can be varied from action to action in order to take advantage of the time available before the next action must be performed, or an anytime search algorithm such as iterative deepening can be used. However, because the plans produced

are only approximately optimal, the domain should have the following characteristics if the algorithm is to work effectively.

- *Structure:* The abstraction algorithm relies on a great deal of structure existing in the domain. We expect other methods for creating heuristics to require similar structure.

- *Time criticality:* As we have said before, the approach is designed for use in domains where reaction time is significant. If there is no restriction on the computation time available, then a better approach would be to use policy iteration (Howard 1971) or a similar algorithm to find an optimal policy, and execute actions according to the policy.

- *Recoverability:* By recoverability, we mean that performing a poor action is generally not fatal. For example, if the connectivity of some parts of the state space is particularly low (there are very few transitions between certain classes of states), the fact that actions are executed as planning proceeds may become a liability. The agent may select an action and execute it, only to discover later that its choice has made reaching certain desirable states very difficult. An algorithm that plans more thoroughly may well be much more effective in such a domain.

- *Complete Observability:* As we have observed before, we have only applied these techniques in completely observable domains. Unfortunately, many interesting real-world domains do not have this characteristic.

- *Stable goals:* Over a series of planning tasks, the agent will work especially well if the reward function remains constant. If the start state changes, only the search need be redone, while making any significant change to rewards requires creating a new abstraction.

- *Few actions:* The abstraction and search algorithms are both very sensitive to the number of actions in the domain. Even a small reduction in the number of actions to be considered can result in a significant reduction in the performance of the search algorithm. Action elimination techniques (see Section 6.7 of (Puterman 1994)) may be a useful addition to the approach, as may the work of Dean and Greenwald (1994) on applying MDP techniques to scheduling problems.

The tradeoff between the two parts of the algorithm is also important. If a large amount of initial computation time is available, or actions must be executed very quickly once planning has begun, a more accurate heuristic can be constructed, and the search can be correspondingly less deep. There are a number of other factors that affect this balance.

- *Continuity:* if actions have similar effects in large classes of states and most of the goal states are fairly similar, we can use a less detailed heuristic function (a more abstract policy).

- *Fan-out:* if there are relatively few actions, and each action has a small number of outcomes, we can afford to increase the depth of the search tree.

- *Extreme probabilities:* with extreme probabilities it may be worth only expanding the tree for the most probable outcomes of each action. This seems to bear some relationship to the envelope reconstruction phase of the recurrent deliberation model of (Dean et al. 1993a).

5.1 Real World Domains

As we said above, there are relatively few real world domains which are completely observable. This considerably restricts the real world applicability of the work as it now

stands. Although we have not investigated it as yet, one promising class of real problems, scheduling, has been proposed in which our techniques can be applied. We describe two such domains.

Airline Gate Scheduling The problem is, given a set of gates at an airport, and a set of incoming and outgoing flights, to allocate gates to each flight in order to minimize deviation from the published schedule. The domain is stochastic since flights can be delayed, or canceled altogether, and gates may be unavailable for various reasons. Actions in this domain are assignments of gates to flights, and rewards (and penalties) are received depending on whether flights leave on time, and are delayed on the ground because no gate is available. Rewards may also be received for allocating nearby gates to flights with a large number of connecting passengers.

One difficulty with this domain is the very large number of actions — each assignment of a flight to a gate is a distinct action. The techniques discussed above for reducing the action space will have a critical part to play in applying our algorithms in many scheduling domains, as will schematic descriptions of actions, and their exploitation.

Oil Pipeline Management Here the problem is to supply a number of customers with various petroleum products. Actions are opening and closing valves, while rewards are received for delivering correct shipments on time, and minimizing oil spoilage due to leaks and contamination of one product with another. The system is stochastic since pumps may fail, pipes may leak, etc.

One open research question is whether the abstraction procedure described in Chapter 3 will be applicable in real world domains. Although we have not investigated any real

domains to test this, we have reason to be optimistic, as we would expect a number of irrelevant or only slightly relevant propositions to be present in any such domain. The fact that Bayesian networks can be applied in real-world domains suggests that there is a great deal of independence that can be exploited.

5.2 Future Work

There are a great many possible lines of research that present themselves. A major one, which we have already discussed, is extending the approach to partially observable MDPs. The major difficulty in the partially observable case is that the current state of the system is a probability distribution over all states, rather than a single state. Although partially observable MDPs can be translated into completely observable ones (Sondik 1978), the resulting MDP has an uncountably infinite state space. One naive approach to this problem is to perform heuristic search as described in Chapter 4 from each state with a non-zero probability (assuming that there are relatively few such states) or with probability above some threshold, and then calculate the weighted average of the utility of each action over all such states. For complex probability distributions, such a procedure may be very expensive to perform, but there are currently no other computationally efficient ways of computing even close-to-optimal plans in partially observable MDPs. From an AI perspective, it is important to consider sensor models to determine what kinds of probability distribution are possible when considering planning in partially observable domains.

A fairly easy extension to this work would be to add rewards and costs associated with actions as well as states. This would involve a relatively small change to the algorithms, and would provide an increase in representational power.

In both the abstraction and planning algorithms as we have presented them, the agent

is aiming to maximize its discounted total reward over an infinite horizon. However, this is not the only value the agent might be interested in. Without considering the finite horizon case, the agent might also be interested in the undiscounted reward. In the MDP literature, this is referred to as *average reward criteria*, where the agent aims to maximize its average reward per action (or per unit time). In many of the domains we are interested in, it is difficult to justify discounting, so an agent using average reward criteria may well perform better. Average reward criteria considerably complicates policy iteration, and other methods of finding optimal policies, and we have not yet investigated the difficulties involved in applying it with our algorithms.

Extending our approach to operate in infinite domains is another area we have yet to investigate. Nor have we looked at the possibility of planning with semi-Markov processes (Howard 1971).

The areas we have mentioned so far are all general problems that we would like to see investigated. There are also a series of extensions that could be made to both the abstraction and planning algorithms individually.

5.2.1 Abstraction

As it stands, the abstraction algorithm is rather restricted in some ways. We have tended to be very conservative when producing the abstract state space — every atom that could possibly affect the eventual reward is deemed relevant. There are a number of possible ways to relax this requirement. The first is to ignore atoms which make only a very small difference to the outcome of some action. For instance, in the example presented in Table 2.1, if carrying the umbrella made a very slight difference to the probability of delivering the coffee successfully, then the abstraction described in Section 3.2.3 would have included *Umbrella* as a relevant atom. If the difference in the probability of *DeliverCoffee* was sufficiently small, we would prefer to ignore *Umbrella*, and perform

the same abstraction as in the example. There are two difficulties with this. The first is that the Markov property no longer holds if these atoms are ignored, and the second is determining which atoms should be ignored, and which need to be included.

A second relaxation of our requirements is to use logical sentences in the place of atoms when constructing the abstract state space. This makes the construction rather more difficult, but could result in equally good abstract policies produced from even smaller abstract state spaces.

One area that would greatly benefit by further investigation is ways of expanding the abstraction process to operate with variables that can take more than two values. For instance, a robot's position in some navigation problem might be represented by two variables, one giving the room that the robot is currently in, and the other the robot's heading (north, south, etc.). An abstraction process that could decide which values of a given variable could be grouped together would be very valuable. Michael Horsch (Horsch 1994) has been looking at this problem as a way of using abstraction to evaluate influence diagrams.

Another interesting way to improve the abstraction, although only in certain domains, is to take advantage of knowledge of the initial state when constructing the abstract state space. For example, if we knew that in our example domain from Table 2.1 that *Rain* was false in the initial state, and since the value of *Rain* can never change, we would like the abstraction procedure to treat *Rain* as an irrelevant atom even if it has a large effect on reward.

An important research area that we have yet to explore is the question of how the abstraction algorithm we describe can be combined with the envelope method of finding policies described in (Dean et al. 1993b). To some extent, this is related to the idea of using the start state to eliminate atoms from the abstraction, but Dean's approach goes further, ignoring states that are possible but very unlikely to be reached from the

initial state. Nicholson and Kaelbling (1994) have already investigated one form of abstraction using sensitivity analysis. Combining their approach with ours might provide a very powerful approach which might well go some way towards some of the other open problems described in this section.

5.2.2 Search

We have tried to describe the search algorithm in a fairly general way, making little mention of implementation details such as whether to use depth first search or iterative deepening. As it stands, the algorithm searches to a fixed depth. However, this need not be the case. Searching to variable depth depending on the utility of the current action, or some other criteria, would be an interesting area to investigate.

If there is little time to perform search, an agent may want to plan to take advantage of previous computation. For example, an agent might want to reject an apparently superior action in favor of a slightly worse one which leads to an area of the state space where the agent has already cached many actions, thus saving the agent computation time.

Another fruitful area for research is the idea of learning a better heuristic while searching to select actions. We have deliberately shied away from an investigation of this area because of our interest in producing as efficient a planning procedure as possible. The idea is that as the agent computes values for states in the search process, it updates the heuristic with these new values. Over time, the heuristic should approach the true value of each visited state. However, if the value of the heuristic changes during the planning procedure, then the agent needs to replan every time it reaches a certain state, rather than using the previously computed action. As state spaces become larger, the probability of reaching a particular state many times will in general decrease, so this caching of values will be of less value. In these situations, an agent that constantly

attempts to improve its heuristic may well perform better.

References

Ballard, B. W. 1983. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21:327–350.

Bellman, R. 1961. *Adaptive Control Processes.* Princeton University Press, Princeton, New Jersey.

Dean, T., Kaelbling, L. P., Kirman, J., and Nicholson, A. 1993a. Deliberation scheduling for time-critical decision making. In *Proc. of UAI-93*, pages 309–316, Washington, D.C.

Dean, T., Kaelbling, L. P., Kirman, J., and Nicholson, A. 1993b. Planning with deadlines in stochastic domains. In *Proc. of AAAI-93*, pages 574–579, Washington, D.C.

Draper, D., Hanks, S., and Weld, D. 1994. A probabilistic model of action for least commitment planning with information gathering. In de Mantaras, R. L. and Poole, D., editors, *Proceeding of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 178–186. Morgan Kaufmann.

Feldman, J. and Sproull, R. 1977. Decision theory and artificial intelligence ii: The hungry monkey. *Cognitive Science*, 1:158–192.

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.

Greenwald, L. and Dean, T. 1994. Monte carlo simulation and bottleneck-centered heuristics for time-critical scheduling in stochastic domains. Unpublished Manuscript.

Haddawy, P. and Doan, A. 1994. Abstracting probabilistic actions. In de Mantaras, R. L. and Poole, D., editors, *Proceeding of the Tenth Conference on Uncertainty in*

Artificial Intelligence, pages 270–277. Morgan Kaufmann.

Haddawy, P. and Hanks, S. 1992. Representations for decision-theoretic planning: Utility functions for deadline goals. In *Proc. of KR-92*, pages 71–82, Cambridge.

Horsch, M. 1994. Personal communication.

Howard, R. A. 1960. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, MA.

Howard, R. A. 1971. *Dynamic Probabilistic Systems.* Wiley, New York.

Howard, R. A., Matheson, J. E., and Miller, K. L. 1976. *Readings in decision analysis.* Stanford Research Institute, Menlo Park, California.

Knoblock, C. A. 1993. *Generating Abstraction Hierarchies: An Automated Approach to Reducing Search in Planning.* Kluwer, Boston.

Korf, R. E. 1990. Real-time heuristic search. *Artificial Intelligence*, 42:189–211.

Kushmerick, N., Hanks, S., and Weld, D. 1993. An algorithm for probabilistic planning. Technical Report 93-06-04, University of Washington, Seattle.

McAllester, D. and Rosenblitt, D. 1991. Systematic non-linear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 634–639.

Nicholson, A. E. and Kaelbling, L. P. 1994. Toward approximate planning in very large stochastic domains. In *AAAI Spring Symposium on Decision Theoretic Planning*, pages 190–196, Stanford.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Reading, Massachusetts.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Mateo.

Puterman, M. L. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley and Sons, New York.

Russell, S. J. and Wefald, E. 1991. *Do the Right Thing: Studies in Limited Rationality.* MIT Press, Cambridge.

Sacerdoti, E. D. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135.

Smith, D. E. and Peot, M. A. 1993. Postponing threats in partial-order planning. In *Proc. of AAAI-93*, pages 500–506, Washington, D.C.

Sondik, E. J. 1978. The optimal control of partially observable markov processes over the infinite horizon: Discounted cost. *Operations Research*, 26(2):282–304.

Tash, J. and Russell, S. 1994. Control strategies for a stochastic planner. In *Proceeding of the Twelfth National Conference on Artificial Intelligence*, pages 1079–1085. AAAI Press.

Appendix A

Experimental Domains

The fully compiled form of the coffee-making domain from Chapter 2

Action	Discriminant	Effect	Prob.
Move	¬Umbrella,Rain,Office	Wet,¬Office,¬HUC	0.0081
		Wet,¬Office	0.8019
		Wet,¬HUC	0.0009
		Wet	0.0891
		¬Office,¬HUC	0.0009
		¬Office	0.0891
		¬HUC	0.0001
			0.0099
	Umbrella,Rain,Office	¬Office,¬HUC	0.0090
		¬Office	0.8910
		¬HUC	0.0010
			0.0990
	¬Rain,Office	¬Office,¬HUC	0.0090
		¬Office	0.8910
		¬HUC	0.0010
			0.0990
	¬Umbrella,Rain,¬Office	Wet,Office,¬HUC	0.0081
		Wet,Office	0.8019
		Wet,¬HUC	0.0009
		Wet	0.0891
		Office,¬HUC	0.0009
		Office	0.0891
		¬HUC	0.0001
			0.0099

Action	Discriminant	Effect	Prob.
Move	Umbrella,Rain,¬Office	Office,¬HUC	0.0090
		Office	0.8910
		¬HUC	0.0010
			0.0990
	¬Rain,¬Office	Office,¬HUC	0.0090
		Office	0.8910
		¬HUC	0.0010
			0.0990
BuyCoffee	¬Office	HRC,¬HUC	0.0080
		HRC	0.7920
		¬HUC	0.0020
			0.1980
	Office	¬HUC	0.0100
			0.9900
GetUmbrella	Office	Umbrella	0.8910
		Umbrella,¬HUC	0.0090
		¬HUC	0.0010
			0.0990
	¬Office	¬HUC	0.0100
			0.9900
DeliverCoffee	Office,HRC	¬HRC,HUC	0.8000
		¬HRC,¬HUC	0.0010
		¬HRC	0.0990
		¬HUC	0.0010
			0.0990
	¬Office,HRC	¬HRC,¬HUC	0.0080
		¬HRC	0.7920
		¬HUC	0.0020
			0.1980
	¬HRC	¬HUC	0.0100
			0.9900

The value function for this domain is exactly the one presented on Page 14.

The COFFEE domain from the results sections of Chapters 3 and 4

Action	Discriminant	Effect	Prob.
GoBarn	umb,¬la	la,¬lb	0.9000
			0.1000
	¬umb,¬la	wet,la,¬lb	0.8100
		wet	0.0900
		la,¬lb	0.0900
			0.0100
	la		1.0000
GoOffice	¬la	la,lb	0.9000
			0.1000
	la		1.0000
GoAILab	¬umb,la,¬lb	wet,dist,¬la,lb	0.8100
		dist,¬la,lb	0.0900
		wet	0.0900
			0.0100
	umb,la,¬lb	dist,¬la,lb	0.9000
			0.1000
	la,lb	dist,¬la,lb	0.9000
			0.1000
	¬la		1.0000
GoGraphicsLab	¬umb,la,¬lb	wet,¬la,¬lb	0.8100
		¬la,¬lb	0.0900
		wet	0.0900
			0.0100
	umb,la,¬lb	¬la,¬lb	0.9000
			0.1000
	la,lb	¬la,¬lb	0.9000
			0.1000
	¬la		1.0000
BuyCoffee	hrs,la,¬lb	¬hrs,hrc	0.5000
			0.5000
	¬hrs,la,¬lb	hrc	0.8000
			0.2000
	la,lb		1.0000
	¬la		1.0000
DeliverCoffee	hrc,la,lb	huc,¬hrc	0.8000
		¬hrc	0.1000
			0.1000
	¬hrc,la,lb		1.0000
	la,¬lb		1.0000
	¬la		1.0000

Action	Discriminant	Effect	Prob.
BuySnack	hrc,la,¬lb	hrs,¬hrc	0.5000
			0.5000
	¬hrc,la,¬lb	hrs	0.8000
			0.2000
	la,lb		1.0000
	¬la		1.0000
DelSnack	hrs,la,lb	hus,¬hrs	0.8000
		¬hrs	0.1000
			0.1000
	¬hrs,la,lb		1.0000
	la,¬lb		1.0000
	¬la		1.0000
GetUmbrella	la,lb	umb	0.9000
			0.1000
	la,¬lb		1.0000
	¬la		1.0000

Sentence	Value
huc,hus,wet,dist	1.15
huc,hus,wet,¬dist	1.25
huc,hus,¬wet,dist	1.4
huc,hus,¬wet,¬dist	1.5
huc,¬hus,wet,dist	0.65
huc,¬hus,wet,¬dist	0.75
huc,¬hus,¬wet,dist	0.9
huc,¬hus,¬wet,¬dist	1.0
¬huc,hus,wet,dist	0.15
¬huc,hus,wet,¬dist	0.25
¬huc,hus,¬wet,dist	0.4
¬huc,hus,¬wet,¬dist	0.5
¬huc,¬hus,wet,dist	-0.35
¬huc,¬hus,wet,¬dist	-0.25
¬huc,¬hus,¬wet,dist	-0.1
¬huc,¬hus,¬wet,¬dist	0.0

The BUILDER domain from Chapters 3 and 4

Action	Discriminant	Effect	Prob.
PaintA	AClean	APainted	0.75
		¬AClean	0.20
			0.05
	¬AClean		1.00
PaintB	BClean	BPainted	0.75
		¬BClean	0.20
			0.05
	¬BClean		1.00
ShapeA	¬Joined	¬APainted,AShaped	0.80
		¬APainted,¬AClean,¬AShaped,¬ADrilled	0.10
		¬APainted	0.10
	Joined	¬BPainted,¬APainted	1.00
ShapeB	¬Joined	¬BPainted,BShaped	0.80
		¬BPainted,¬BClean,¬BShaped,¬BDrilled	0.10
		¬BPainted	0.10
	Joined	¬BPainted,¬APainted	1.00
DrillA	¬Joined	ADrilled	0.90
			0.10
	Joined		1.00
DrillB	¬Joined	BDrilled	0.90
			0.10
	Joined		1.00
WashA		AClean	0.90
			0.10
WashB		BClean	0.90
			0.10
Bolt	BShaped,AShaped, BDrilled,ADrilled	Joined	0.80
			0.20
	¬ADrilled		1.00
	¬BDrilled,ADrilled		1.00
	¬AShaped,BDrilled,ADrilled		1.00
	¬BShaped,AShaped, BDrilled,ADrilled		1.00

Action	Discriminant	Effect	Prob.
Glue	BShaped,AShaped	¬BClean,¬AClean,Joined	0.35
		Joined	0.35
		¬BClean,¬AClean	0.15
			0.15
	¬AShaped	¬BClean,¬AClean	0.50
			0.50
	¬BShaped,AShaped	¬BClean,¬AClean	0.50
			0.50

Sentence	Value
AClean,BClean,APainted,BPainted,Joined	1
¬AClean,BClean,APainted,BPainted,Joined	.9
AClean,¬BClean,APainted,BPainted,Joined	.9
¬AClean,¬BClean,APainted,BPainted,Joined	.8
AClean,BClean,¬APainted,BPainted,Joined	.8
¬AClean,BClean,¬APainted,BPainted,Joined	.7
AClean,¬BClean,¬APainted,BPainted,Joined	.7
¬AClean,¬BClean,¬APainted,BPainted,Joined	.6
AClean,BClean,APainted,¬BPainted,Joined	.8
¬AClean,BClean,APainted,¬BPainted,Joined	.7
AClean,¬BClean,APainted,¬BPainted,Joined	.7
¬AClean,¬BClean,APainted,¬BPainted,Joined	.6
AClean,BClean,¬APainted,¬BPainted,Joined	.6
¬AClean,BClean,¬APainted,¬BPainted,Joined	.5
AClean,¬BClean,¬APainted,¬BPainted,Joined	.5
¬AClean,¬BClean,¬APainted,¬BPainted,Joined	.4
AClean,BClean,APainted,BPainted,¬Joined	.6
¬AClean,BClean,APainted,BPainted,¬Joined	.5
AClean,¬BClean,APainted,BPainted,¬Joined	.5
¬AClean,¬BClean,APainted,BPainted,¬Joined	.4
AClean,BClean,¬APainted,BPainted,¬Joined	.4
¬AClean,BClean,¬APainted,BPainted,¬Joined	.3
AClean,¬BClean,¬APainted,BPainted,¬Joined	.3
¬AClean,¬BClean,¬APainted,BPainted,¬Joined	.2
AClean,BClean,APainted,¬BPainted,¬Joined	.4
¬AClean,BClean,APainted,¬BPainted,¬Joined	.3
AClean,¬BClean,APainted,¬BPainted,¬Joined	.3
¬AClean,¬BClean,APainted,¬BPainted,¬Joined	.2
AClean,BClean,¬APainted,¬BPainted,¬Joined	.2
¬AClean,BClean,¬APainted,¬BPainted,¬Joined	.1
AClean,¬BClean,¬APainted,¬BPainted,¬Joined	.1
¬AClean,¬BClean,¬APainted,¬BPainted,¬Joined	0