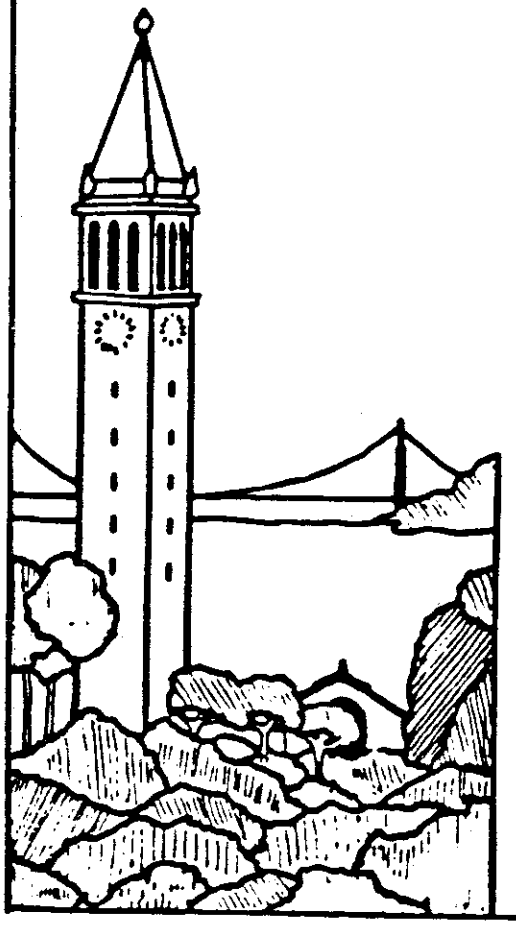
Optimal Probabilistic and Decision-Theoretic Planning using Markovian Decision Theory

Sven Koenig



Report No. UCB/CSD 92/685

May 1992

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Abstract

In this report, we describe a simple probabilistic and decision-theoretic planning problem. We show how algorithms developed in the field of Markovian decision theory, a subfield of stochastic dynamic programming (operations research), can be used to construct optimal plans for this planning problem, and we present some of the complexity results known. Although their computational complexity allows only small problems to be solved optimally, the methods presented here are helpful as a theoretical framework. They allow one to make statements about the structure of an optimal plan, to guide the development of heuristic planning methods, and to evaluate their performance. We show the connection between this normative theory and universal planning, reinforcement learning, and anytime algorithms.

One can easily construct a one-step planner by using a Markovian decision algorithm and a random assignment of actions to states as the initial plan. In many planning domains, it is easy for human problem solvers to construct a working plan, although it is difficult for them to find the optimal (or a close-to-optimal) plan. Therefore, we propose a two-step planning method: During the first planning phase, a working (i.e. ergodic), but not necessarily optimal plan is constructed. Sometimes, a domain specific planning method might be available for this task. We show that such a planning method can be obtained even in probabilistic domains by taking advantage of deterministic or quasi-deterministic actions. Thus, traditional (deterministic) planners can be useful in probabilistic domains. We also state a general greedy algorithm that accomplishes this task if no domain specific method is available. During the second planning phase, a Markovian decision algorithm is used to incrementally refine the initial plan and derive increasingly better plans, until the optimal plan is finally found. Since this algorithm is an anytime algorithm, we can trade off planning time and the execution reward of the plan.

Finally, we briefly present a software package that implements our ideas. The probabilistic domain can be modeled using an augmented STRIPS notation. It is automatically translated into a Markovian decision problem, that is then solved.

Contents

List of Figures

1. Introduction

In the introduction, we will give an overview of the problem of optimal probabilistic and decision-theoretic planning and our approach to the problem. First, we explain what we mean by probabilistic and decision-theoretic domains, how such domains can be modeled, and why they are interesting. Then, we describe the planning task and what constitutes a solution of the planning problem. We show why traditional planning approaches, i.e. planning methods for deterministic domains, cannot directly be used to solve probabilistic planning problems. To give an overview of our approach and to introduce our terminology, we will use a deterministic (instead of a probabilistic) domain and proceed along the lines of the body of this report. Finally, we will give a short summary of the report and describe its organization.

In this report, we investigate optimal probabilistic and decision theoretic planning methods. We adapt and augment the state space planning model from GPS. There, the planning problem consists of a number of states, the start state, a set of goal states, and the actions, each mapping one state into its successor state. This mapping is deterministic and independent from the way the source state was reached. (STRIPS, a successor of GPS, represents actions as operators that have precondition, add, and delete lists.) A plan is a linear sequence of action applications that leads from the start state to an arbitrary goal state. The common measure for the goodness of a plan is its length.

The blocks-world is a classical example for a STRIPS domain: There are some toy blocks on a table, and the planner has to devise instructions for an agent that specify which move actions to execute in order to achieve one of the pre-specified goal configurations of the blocks. This problem is a very old and well-studied planning problem.

The GPS approach to planning imposes two restrictions on the planning problems that we will relax: The deterministic effects of actions, and the length of a plan as its goodness measure. We will augment GPS-type planning problems in these two points and investigate probabilistic and decision-theoretic planning methods to solve them.

- Probabilistic Planning

  Planning has mainly been studied in deterministic domains, although almost every non-artificial problem domain is not completely deterministic. Thus, it is an important problem how to deal with uncertain action outcomes. There can be several reasons why

the action outcomes cannot be predicted with certainty. Uncertainty might inherently be part of the domain, because the action outcomes are truly probabilistic. But usually it is caused by limitations of the planner or the agent that executes the plan: The agent might not be able to execute its (physical or perceptual) actions with the required precision, or the planner might have an insufficient model of the domain. For example, instead of using correct algorithmic solutions, the planner might work with heuristics, statistics, exact methods on models that abstract from relevant details, or a combination thereof, such as using statistics to relate heuristic values to unknown correct values. There can be many reasons for working with approximate models rather than exact ones: First, the planner might not know the underlying cause-effect relations. Second, the planner or the agent might have resource constraints that render it impossible to work with sufficiently detailed models.[1] Third, since an on-line planner has to take into account not only the goodness (optimality) of the plan, but also the resources used for planning and executing the plan, it might be optimal under decision-theoretic aspects not to model all relevant cause-effect relations. In general, restrictive time or other resource constraints, or a rapidly changing environment will encourage to use the approximation approach.

To deal with uncertain domains as if they were deterministic means to lose information. The traditional way of dealing with uncertainty is to regard the most probable outcome of an action as the result of the action. The other outcomes are either ignored (and lead to unsuccessful plan executions) or they are treated as exceptions that lead to deviations from the plan and are detected by monitoring the execution. More recent approaches for dealing with uncertain action outcomes are reactive planning and universal planning. A universal plan consists of state-action rules. During execution, a loop is executed that repeatedly determines the state of the world, looks up the action that is assigned to the state, and executes it. Determining the state of the world in each iteration provides a form of execution monitoring. Also, truly probabilistic planning methods have been suggested. These approaches usually assume that the planner knows the probability distribution over the successor states given an action and the state it is executed in. Our approach belongs to the last two classes: Given the probability distributions, we

[1]For example, the exact trajectory of a stone rolling down a hill could be determined using traditional physics. Since collecting data about the rough surface and calculating the exact trajectory takes a long time, the planner might be forced to use a coarser model to manage the complexity of the domain.

find a universal plan.

- Decision-Theoretic Planning

  A plan cannot always be evaluated by its length. The execution of different actions can require different efforts, and not all goal states might be equally desirable. In this case, there is a trade-off between the quality of the result, the resources needed for the execution of the plan, and the resources needed for planning. Agents that take their resource constraints into account are called limited rational agents. Our approach is an off-line planning approach: planning is done before execution. Once a plan is found, it can be used as often as needed. Thus, we do not need to take into account the resources needed for planning. But we still have to trade-off between the quality of the goal state reached and the effort that is required to reach it.

We will augment GPS-type planning problems in the following way:

- Probabilistic Augmentation

  The start state is determined according to some known probability distribution over all states, and actions can have probabilistic outcomes. The planner knows the probability distribution over the action outcomes (i.e. the successor states) given an action and the state it is executed in. Like GPS, we require that the probability distributions are independent from the way the source state has been reached. Since actions are no longer deterministic, we have to characterize the goodness of a plan by an *expected* (i.e. average) value.

  In the blocks-world, for example, the move action might succeed only 90 percent of the times. In the other cases, the block that the agent wanted to move might accidentally drop onto the table.

- Decision-Theoretic Augmentation

  Actions have costs, and goal states rewards associated with them. The cost of an action can depend on the source state, the action executed in the source state, and the successor state reached. For every action executed, the agent has to pay the associated cost.

  The costs of the actions are negative. The agent has a special stop action available that has no costs and that can be executed in every goal state to stop the execution of the

plan. When stopping in a goal state, it receives the reward associated with the goal state. A goal reward can be any value. The total reward of the execution of a plan is the sum of the costs of the actions plus the reward of the goal state in which the execution was stopped. (If the execution continues forever, its reward is minus infinity.) A good measure for the goodness of a plan is its expected total execution reward.[2]

In the blocks-world, for example, stacking a block onto a 2-block stack might require less effort (i.e. might be less expensive) than stacking it onto a 10-block stack, if the stacking succeeds. The goal might be to assemble at least a 2-block stack, but the more blocks are stacked, the larger is the reward.

The planning problem investigated in this report is a task-driven planning problem: Despite the non-deterministic effects of the actions, the plan has to guarantee that its execution solves a given task. The task is achieved in any of the goal states. Thus, the plan has to guarantee that a stop action is executed eventually, since this is the only way to achieve a goal. The reward of a goal state specifies how well the task is solved in this state. If the agent leaves a goal state (instead of stopping in that state), it does not receive a reward for the state. Therefore, the agent can receive a reward only once, namely at the end of the execution phase.

Some behavioral approaches to planning do not impose the restriction on a plan that its execution eventually has to achieve a goal. They allow actions to have positive or negative rewards attached, and are content with maximizing the average execution reward per action. This is appropriate for some planning problems, but not for task-driven ones: If one sent a robot to Mars to collect rock samples, one would like to make sure that the task is completed at some point in time. Without the restriction, it would be possible that the planning program finds other activities (e.g. playing hide-and-seek with a fellow robot) that yield a larger average execution reward per action for the robot than rock sampling.

A plan that satisfies the restriction is called a solution of the planning problem. Usually, there will be more than one solution. We prefer plans with larger expected total execution rewards over plans with lower ones. Although the plan has to guarantee that its execution

[2]Using this measure implies that there are no deadlines, and that a goal is to be achieved only once and not repeatedly. Another measure that we will investigate in this report is the expected execution reward per action, or (more generally) the expected execution reward per unit time needed to complete the execution.

eventually stops in a goal state, it can trade off between the execution costs of the actions and the reward of the goal state in which the execution is stopped. Since the goal states can have different rewards associated with them, it can happen that the planner is no longer indifferent about which goal state to reach. If achieving a goal state with a large reward is very costly, it might be better to strive for a goal state with a lower reward that can be reached with smaller costs. If the start state is also a goal state, it might even be optimal to stop immediately.[3] Also, it can be rewarding not to stop in the first goal state reached, if a different goal state can be reached and the difference in the rewards of the goal states is larger than the costs of the additional actions needed to reach the other goal state. (The problem of whether to stop in a goal state or continue the execution is known as "optimal stopping problem" in operations research.)

Planning is a type of search with a particular structure. Determining a good node to expand in a search tree is guided by macro operators, abstraction, goal reduction, and subgoal ordering techniques in order to be able to cope with the size of the state space (see [63] and [49]). Planners usually do not guarantee the optimality of their plans. This is often not a problem, since one is satisfied if the planner finds a plan at all. However, we are interested in a normative theory for solving the probabilistic and decision-theoretic planning problem. We will investigate how to find the *best* plan, i.e. the plan with the largest expected total execution reward. Such a plan is called an optimal solution of the planning problem. (Thus, the task of the planner is to find the optimal solution, not just an arbitrary solution. We chose the term "solution" for plans that guarantee their execution to stop, for the following reason: If planning time is limited and the planner has to deliver a plan without having found an optimal one, it may return a suboptimal plan as long as this plan can be used to solve the given task. Thus, these plans are admissible solutions of the planning problem. However, given enough time, the planner has to be able to determine an optimal solution.)

A normative theory can be used as a basis to explore and prove properties of the planning problem, e.g. to make statements about the structure of an optimal plan, to guide the selection or the development of heuristic planning methods (e.g. by relaxing the algorithmic method), and to evaluate their performance. The knowledge how to solve planning problems

[3]Imagine you feel like eating ice cream, but the next ice cream parlor is 30 miles away. Probably you will then decide that it is not worth the effort to go and buy some ice cream. In this case, you have decided to stay in the start state and refrain from executing any actions.

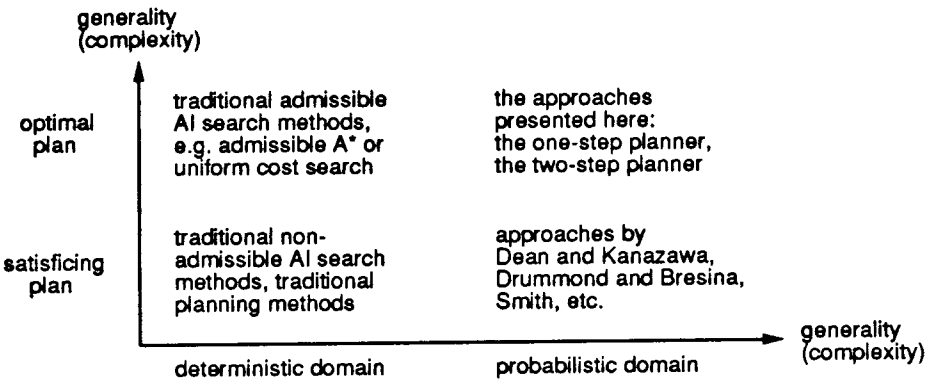


Figure 1: Two Dimensions of a Planning Problem

optimally is also helpful to evaluate how well a planner will perform in domains with given domain characteristics, and to determine whether and where changes have to be performed on the planner.

Our approach can be characterized as *optimal* planning in *probabilistic* domains. (Adding costs and rewards turns out not to be the hard part of the problem.) The two main dimensions are shown in figure 1. The probabilistic and optimal case is the most general one: it subsumes the other cases.

An obvious question is whether we can generalize a method that works for one of the other cases to the probabilistic and optimal case. It turns out that the optimal AI search methods, such as the informed A* search method with an admissible heuristic or the uniformed uniform cost search method[4], cannot easily be converted to search methods that work for probabilistic planning domains.

To explain why one cannot simply use deterministic search methods in probabilistic domains, we will use the (deterministic) GPS domain. A GPS plan (usually) is defined to be a sequence of actions (or a procedural representation of an action sequence) that leads from the given start state to one of the given goal states. Every action has a cost of -1, and every goal state has a reward of 0. In other words, the goodness of a plan (its total execution reward) is determined by its length: the more actions are needed to achieve a goal, the worse the plan.

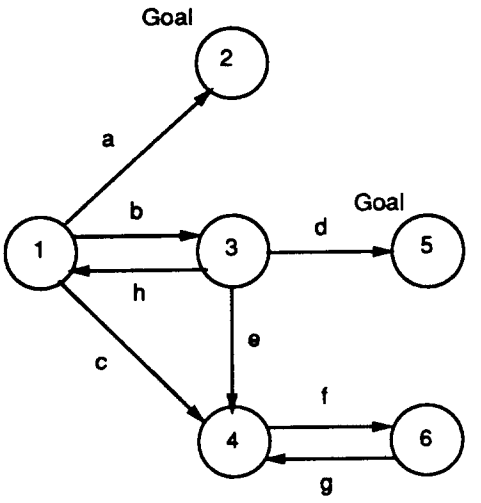
[4]The uniform cost method works like the breadth-first method, but instead of progressing in layers of equals depth, the search space is unraveled in layers of equal cost. It is a special case of the A* method with $h = 0$, see [79].

For a GPS planning problem it is always best to stop in the first goal state reached.

Instead of using an action sequence to denote a plan, we can use a partial mapping from actions to states: Every state that is encountered during the execution of the plan gets assigned the action that the plan requires to execute in that state. (If the action sequence requires to execute different actions in the same state at different points in time, we can always find a better plan for which the actions are uniquely determined by the states (by "patching" the plan, see chapter 12). An example is given in figure 2. The execution of the plan remains unaltered, when we make the mapping from states to actions total by assigning an arbitrary action to every state that is not encountered during the plan execution. We call such a mapping a universal plan (or, equivalently, a stationary plan). A universal plan assigns to every state the action to be executed when this state is reached during the execution of the plan. (We assume that the plans are stored as a state-action tables. This differs from Schoppers [93], who used decision trees to represent them.) Unless stated otherwise, we will use "plan" and "universal plan" synonymously.

For every state s, we define its goal distance under a given plan (which we call its v value and which we denote by $v(s)$) as the negative of the number of actions needed to reach the first goal state when starting the plan execution in s and following the instructions of the plan. (If no goal state can be reached, the goal distance of s is minus infinity.) The smaller the absolute value of the goal distance of a state, the better is the plan for that state as start state. Goal distances are always non-positive. For simplicity, when we state "the goal distance decreases", "... is large", or "... is to be minimized", we refer to the absolute value of the goal distance.

Given a plan, one can determine the goal distances of all states. The inverse is also true: Given values for all states, one can construct a plan for which these values are the goal distances of the states under that plan (if such a plan exists and one knows for every state the actions applicable in that state). Note that different plans do not necessarily differ in the goal distances that they assign to the states. In conclusion, there are two different representations of a universal plan, that are easily transformable: One can describe it as an assignment of actions to states, or, alternatively, as an assignment of goal distances to states. In the following, we will often use the goal distances as the plan representation, because they do not only describe a plan, but also evaluate the goodness of the plan. In figure 3, we show the two different representations for the state-action space that is depicted in figure 2.



GPS plan (for state 1 as start state): b, h, a
(actions a and b are executed in state 1)
alternative, better GPS plan a
(every state is assigned only one action)

Figure 2: Converting GPS Plans to Universal Plans

(a)

state	action
1	a
2	(stop)
3	d
4	f
5	(stop)
6	g

(b)

state	goal distance (v-value)
1	-1
2	0
3	-1
4	-infinity
5	0
6	-infinity

Figure 3: (a) a (State, Action) Table, and (b) a (State, Goal Distance) Table

The reason why one cannot simply use deterministic search methods to solve probabilistic planning problems is the following: In a deterministic domain, universal plans that allow the agent to visit the same state two or more times during plan execution cannot be solution plans. Once caught in a cycle, the agent can no longer reach a goal state. If we use a planning method that avoids such cycles and change the action of a state x in a way that the new action leads to a state y which currently has a goal distance of $v(y)$, then we can safely conclude that the goal distance $v(x)$ equals $v(y) - 1$. In a probabilistic domain, however, universal plans that allow the agent to visit the same state more than once cause no problems, if the cycle can be left with a positive probability, see figure 4. In this case, changing the action assigned to state x can change not only $v(x)$ but also $v(y)$. Thus, one can no longer conclude that the goal distance of x under the new plan is minus one plus the average goal distance of its successor nodes weighted with the probabilities to reach them. For example, in figure 5 the new goal distance of x is not $-1 + 0.5 \times (-1) + 0.5 \times (-2) = -2.5$, but -3. There are two ways to calculate the correct goal distance of x:

- We can repeatedly calculate $v(x) := -1 + 0.5 \times v(x) + 0.5 \times v(y)$. This leads to the assignments $v(x) := -2.5$, $v(x) := -2.75$, $v(x) := -2.875$, $v(x) := -2.9375$, $v(x) := -2.96875$, etc. The advantage of this method is that the computations are local (as in the deterministic case): To approximate the goal distance of a state, we only need to access the goal distances of its successor states. Its disadvantage is that we are usually only able to approximate the correct goal distances asymptotically. This method leads to the value-iteration method, a roll-back algorithm, that is well known in operations research and that can be utilized to solve probabilistic planning problems. (Most of the satisficing probabilistic planners, e.g. Dean and Kanazawa's temporal belief networks [24], Drummond and Bresina's probabilistic planner [15], or Smith's decision theoretic planner [98] use methods similar to the value-iteration method, but make some decisions based on heuristics or perform the calculation only a small, constant number of times.)

- We can solve the system of equations: $v(x) = -1 + 0.5 \times v(x) + 0.5 \times v(y)$, $v(y) = -1$, which simplifies to $v(x) = -3$, $v(y) = -1$. The advantage of this method is that we can easily determine the correct goal distances. Its disadvantage is the non-locality of the computations: We have to determine the goal distances for all states simultaneously, and thus to access all goal distances. This method is part of the policy-iteration methods, that are (like the value-iteration method) well known in operations research and that

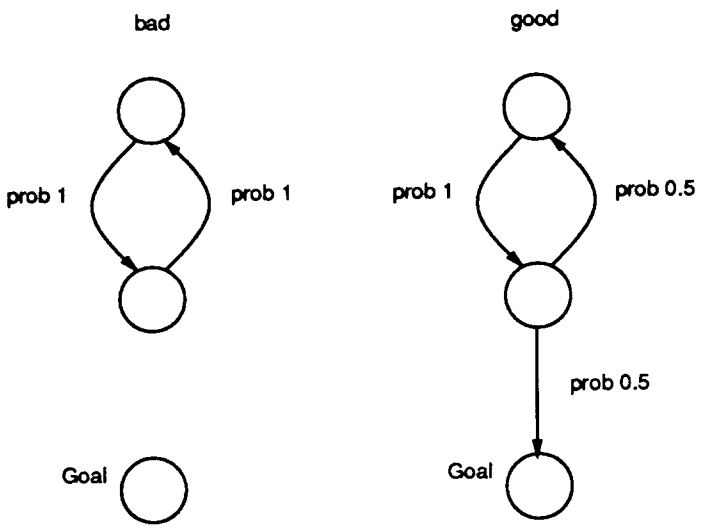


Figure 4: Cyclic Plans in Deterministic and Probabilistic Planning Domains

can also be utilized to solve probabilistic planning problems.

The common framework of the value-iteration method and the policy-iteration methods is sketched below.

In the following, we illustrate our approach and introduce the terminology using the GPS domain as an example. For a deterministic domain, all of the following statements can easily be seen to be true. In the rest of the report, we will show how to generalize these definitions and problem solving methods to probabilistic and decision-theoretic domains using the terms introduced here and proceeding along the lines outlined here for the deterministic domain:

A state is called solvable, if there exists a plan such that the goal distance of the state under the plan is finite. If the start state is solvable, then a solution plan (also called an admissible plan) exists. A plan that solves every state is called ergodic (a term from Markovian decision theory). Ergodic plans are admissible. They are interesting, because we will show how to improve a suboptimal plan that is ergodic. A planning domain for which every plan is ergodic is called completely ergodic.

If a state is solvable, then we want to minimize the absolute value of its goal distance. There is always one plan that simultaneously solves every solvable state such that every solvable state is assigned its smallest possible goal distance. Such a plan is optimal for every solvable state as start state. (If the start state is unsolvable, no solution exists.)

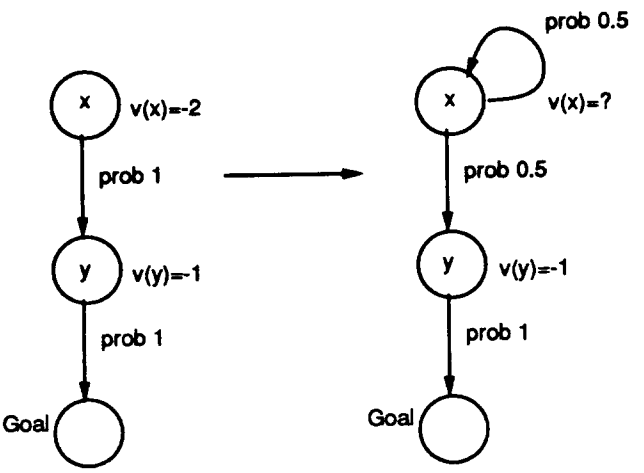


Figure 5: Determining the Goal Distances for Cyclic Plans

One method for finding the optimal solution of a planning problem is the deterministic multiple-chain policy-iteration method:

1. For every state, randomly pick an applicable action and assign it to the state.
2. Determine the goal distance for every state under the current plan.
3. For every state, choose the action that leads to the successor state with the smallest goal distance. If the old action assigned to a state satisfies this requirement, do not change the action assignment of that state.
4. If the action assignment was changed for at least one state, go to step 2. Otherwise stop and output the state-action assignment as the optimal plan.

This search algorithm incrementally refines the initial plan and derives increasingly better plans (in the sense that during every iteration the absolute value of the goal distance of every state cannot increase and at least one absolute value strictly decreases), until a plan is finally found that assigns every state its smallest possible goal distance. (Such a plan is shown in figure 3 for the state-action space that is depicted in figure 2.) Then, the algorithm terminates. Note that an intermediate plan might not solve the start state, and thus might not be a solution of the planning problem. A state is solvable, iff its goal distance under the final plan is finite. If the start state is solvable under the final plan, this plan is an optimal solution, otherwise no solution exists.

Assume now that the algorithm has to obey the restriction that at every point in time the goal distance of every state is finite. (This restriction allows us to solve a system of linear equations easily to determine the goal distances in the probabilistic case.) The search algorithm above (in this case called simple-policy iteration algorithm, because the corresponding algorithm differs from the multiple-chain policy-iteration algorithm for the probabilistic case) still solves the problem for completely ergodic planning domains. (Since the planning domain is completely ergodic, every goal distance will be finite no matter what the plan is.) Since every state is solvable, the final plan is an optimal solution, no matter what the start state is. (Thus, the start state does not need to be specified.)

This algorithm is an anytime algorithm. At every point in time it can deliver an ergodic plan, i.e. a solution plan. The plan gets better, the more time the planner is allowed to run. When we stop the algorithm before its termination, we can trade off planning time and the goodness of the plan. Since every intermediate plan is ergodic, we can also interleave planning with execution without losing the guarantee that the execution will eventually stop. In the probabilistic and decision-theoretic case, we can also learn the transition probabilities, costs and rewards when interleaving planning with execution, thus having developed a reinforcement learning system.

The algorithm above also works in domains that are not completely ergodic, if we choose in step 1 an ergodic plan (instead of a random plan). All of the following plans produced will then be ergodic. Eventually, an optimal solution is found. The initial ergodic plan can only be found if every state is solvable. Thus, we have to delete the unsolvable states first, then we can determine an ergodic plan for the remaining states, that we finally optimize using the simple policy-iteration algorithm. This is the two-step planning method:

1. (first planning phase) Find all unsolvable states, delete them together with all actions applicable in them or leading to them, and determine an ergodic plan for the remaining states.
2. If the start state was deleted, then stop. The planning problem is unsolvable.
3. (second planning phase) Run the simple policy-iteration algorithm on the modified planning problem using the plan determined in step 1 as initial plan.
4. Stop. The plan found is the optimal solution of the original planning problem. (A state

that was deleted in step 2 can get an arbitrary action assigned that is applicable in that state.)

In the probabilistic case, it still holds that one of the optimal plans is always a universal plan. Such a plan determines a Markov chain. A Markov chain is a triple (S, p, c): S is a finite number of states, p specifies the transition probabilities between the states, and c specifies the transition costs. (We will model the rewards of the goal states as transition costs.) States are usually depicted as circles. Transition links are drawn from a state to every potential successor state and labeled with the transition probabilities and the transition costs. The Markov chain is always in exactly one state. It changes its state after a unit time interval according to the rule: If it is at time t in state s, then it will be in state s' at time $t+1$ with probability $p(s'|s)$. During this transition, the cost $c(s, s')$ occurs. It must hold that $\sum_{s' \in S} p(s'|s) = 1$ for all $s \in S$. The assumption that the transition probability only depends on the current state, but not on how the current state was reached (i.e. the history) is called Markov property. It is common to assume that the Markov property holds. For example, GPS domains satisfy that the successor state depends only on the source state and the action executed in that state. Once one has chosen a plan (i.e. assigned an action to every state), the successor state is determined solely by the source state. Thus, GPS domains satisfy the Markov property. (For a longer explanation see [109].)

Probabilistic analogues of the deterministic simple policy-iteration algorithm and the deterministic multiple-chain policy-iteration algorithm exist. (Recently, this connection between work in operations research, namely methods used to solve Markovian decision problems, and algorithms for probabilistic planning was also pointed out by other researchers in probabilistic planning [60] and reinforcement learning [3].) We could transform our probabilistic planning problem into a Markovian decision problem, and then use the multiple-chain policy-iteration algorithm or the simple policy-iteration algorithm, which were developed by Howard [54]. Howard uses the simple policy-iteration algorithm for completely-ergodic problems only. The more general and more complicated multiple-chain policy-iteration algorithm can be applied to all problems.

In this report, we formalize the connection between Markovian decision algorithms and probabilistic and decision-theoretic planning by showing how the GPS framework can be augmented to incorporate probabilistic and decision-theoretic domains. We will also introduce an augmented STRIPS notation for these domains. Then, we will explain the transforma-

tion of a planning problem into a Markovian decision problem. We will show the usefulness and limitations of the results transferred from operations research to AI. These results lead immediately to a one-step planner using the *multiple-chain* policy-iteration algorithm. The main innovation of this work is then to show how the *simple* policy-iteration algorithm can be used to solve the transformed planning problem, even if the problem is not completely ergodic. As outlined above for a deterministic domain, we will introduce a two-step planning approach. Before applying the simple policy-iteration algorithm, one has to delete every unsolvable state and determine an ergodic plan for the remaining states. (In other words, first one finds a working, but not necessarily optimal plan, then one optimizes it.) Sometimes, a domain specific planning method might be available for the first planning phase. We will investigate helpful domain characteristics and, for example, show that such a planning method can be obtained for some probabilistic domains by taking advantage of either leaking actions or deterministic (and quasi-deterministic) actions. (A leaking action is one that leads with a positive probability to a goal state. A quasi-deterministic action is an action whose effect under the optimal plan can be modeled using a deterministic action.) We will also state a general greedy algorithm that always solves this task, and that can be used if no faster domain specific method is available. This greedy method resembles an AND-OR search, and does not need to know the transition probabilities (except for the knowledge which transition probabilities are zero), costs, or rewards.

The following chapters are organized as follows: First we will give a historical overview, in which we will introduce the general terms, planning concepts, and specific planners from the literature that we will need later in the text. Then, we will state the planning problem informally, and show how it generalizes the STRIPS-type planning problem by explaining its probabilistic and decision-theoretic augmentation. As an example, we will introduce the blocks-world domain, and generalize it to the augmented blocks-world domain, in which we have costs, rewards, and probabilistic action outcomes. After having formally stated the planning problem, we will introduce Markovian decision algorithms. First, we motivate the algorithms with utility/decision trees, then we state the results from the operations research literature. This allows us to explain how to transform the planning problem into a Markovian decision problem, and how to solve it with a one-step planning approach. Then, we motivate the two-step planner and discuss both of its phases in detail. We demonstrate its flexibility by showing how to incorporate execution times of the actions into the two-step planner and how it can be used as an on-line planning system or a reinforcement learning system. Finally,

we remark on the complexity of the planning methods, describe our implementation of the planning methods, and discuss related work. The assumptions about the probabilistic and decision-theoretic planning domains made and the proofs of the theorems are given in the appendix.

The whole report is written as free of notational ballast and mathematical prerequisites as possible to allow AI researchers without an extensive background in dynamic programming easy access to the results.

2. Historical Overview

In this chapter, we will give a brief historical overview of planning. This overview is not meant to be complete, since we will only introduce the general terms, planning concepts, and specific planners from the literature that are needed to describe our work. Thus, we will stress work that is relevant to probabilistic and decision-theoretic planning, but we will also mention work in universal planning, reinforcement learning, and anytime algorithms, since these areas are important for our work as well. (The reason is that our planner uses Markovian decision algorithms, that can also be used for reinforcement learning. They are anytime algorithms that find an optimal universal plan.) For a more general overview of planning (that includes topics such as planning under time and resource constraints) see for example [32].

The General Problem Solver (GPS) [73, 74] was one of the first AI planners. As described in chapter 1, it uses a deterministic state space model of the planning domain. Plans are evaluated according to their lengths.

STRIPS [38], a successor of GPS, uses GPS's domain model, but represents states as unique, finite sets of predicates (lists containing only constants). The set of goal states is given as a partially described state (an intersection of states). A state is a goal state iff it is a superset of this partially described state. STRIPS represents the actions as templates, called operators. An operator is described by three lists: a precondition list, a delete list, and an add list. All lists are finite sets of predicates possibly containing variables that match constants or lists. Every variable that appears in the delete list or add list must also appear in the precondition list. An operator is applicable in a state, iff a binding for the variables can be found such that every predicate in the precondition list of the operator is matched by a predicate of

the state. If an operator is applicable in a state under different variable bindings and the effects of the operator applications differ, then the operator represents different actions. (In some STRIPS-like planners, the precondition list is split into two different lists in order to be able to better support means-end analysis: One list contains predicates that will be used as subgoals for means-end analysis, the other list contains predicates that will render the operator inapplicable in states for which the predicates do not hold, and abort further planning for these states.) Universal quantification is not provided, neither is negation. In finite domains, universal quantification can be expressed as conjunction. Likewise, instead of negating a predicate, one can introduce a new predicate representing the negated predicate, and always modify the predicate and its negated counterpart in opposite ways. For each binding of the variables, the application of an operator to a state deterministically leads to a successor state. The new state is determined by deleting every predicate in the delete list from the old state, and then adding every predicate in the add list.

Later AI planning systems continued to study planning in deterministic domains. Unexpected outcomes of actions were either not dealt with at all, or they were detected by execution monitoring and resulted in either one of the following: A re-invocation of the planner to find a new plan that leads from the current state to a goal state, an invocation of an exception handling procedure that was supplied as part of the plan, or the invocation of a plan repair procedure that modifies the plan to fit the new state [95]. (A special case of the last approach is to find and execute a small plan that coerces the unexpected state to a state in which the old plan is again applicable.) A different approach was to check for non-deterministic effects of actions by putting conditionals into the plan [106] (and thus abandoning linear plans).

A newer approach for dealing with uncertain action outcomes that overcomes some deficiencies of classical planners is reactive (reflexive) planning and universal planning. Such a plan is executed by repeatedly determining the current state and then executing the action that is associated with that state. Thus, classical planners assume that the world is predictable, whereas reactive and universal planners do not make this assumption and use sensing operations during execution to determine the state. Schoppers' universal plans [93], situated activity [1], situated automata [59, 83], and action networks [75, 76, 70] fall into this category. Different researchers include different information in the state. The spectrum ranges from the inclusion of every (relevant) predicate that is currently observed by the agent (the pure form of reflexive planning), over every (relevant) predicate of the current world state (the pure form of universal planning) to every (relevant) predicate of the current world state plus

some information about the history (e.g. by using a "note pad" which is always in the visual field of the agent and on which the agent can record information[5]).

"In the classical approach to planning, an agent computes a plan once, and commits to the plan to achieve a goal. This can be advantageous ... if the agent has an effective method for predicting future states; in such cases, the work done in generating a plan need only be done once. In a more dynamic environment, committing to a plan can be problematic," argue Dean and Kanazawa [24] in favor of online-planning (to be exact: revolving planning). A universal plan consists of state-action rules that compile the domain knowledge needed for on-line planning. (Plan compilation was introduced in [37]. For a general account of compilation see [86].) A compiled plan allows the agent to execute plan steps extremely quickly [68] (under the assumption that the sensing operations that are required to determine the current state can be executed fast), even if the outcome of an action was not the expected one. Repeatedly determining the state of the world provides a form of execution monitoring. Thus, the plan must assign an action to every state, i.e. even to states that are not expected to be encountered. The advantage in reactivity leads to a disadvantage in space. Ginsberg [42] points out that storing an action for every state uses up a lot of space, if a state-action table is used to represent the universal plan. Rules whose premises partially match the states, decision trees or neural nets cannot be used to save space for domains in which different actions have to be chosen in similar situations (such as in chess, for example).

Reactive systems demonstrate that it is possible to achieve goals without lookahead (i.e. without predicting the future during execution time) by compiling domain knowledge into state-action rules, even if the goals can only be achieved by a sequence of several actions. The field of reinforcement learning studies how reflexive agents can learn and adjust a universal plan by executing actions and receiving feedback in form of costs for the actions executed (e.g. tiredness) and rewards for the goals achieved (e.g. pain or joy). ([3] and [7] provide a good overview.) This allows reinforcement learning systems to adjust incrementally to a previously unknown or dynamic environment. (Thus, reinforcement learning provides a remedy to the problem mentioned by Ginsberg [42] that the inflexibility of universal plans makes it difficult for an agent that executes such a plan to expend its limited resources to improve its performance.) Reinforcement learning systems have been implemented that

[5]For example, after having found a parking lot in the morning, the agent can write down the location of the lot, so that it will be able to locate the car in the evening.

use state-action tables [7], or neural networks to generalize the state-action function from examples and to encode it more compactly [65, 113]. They can use either Sutton's temporal difference method [105], or Markovian decision algorithms [104, 6]. (For a comparison of some of these methods see [5].) In the latter case, the value-iteration method or a policy-iteration method are used to update the plan. A cognitively more adequate Markovian decision algorithm was developed by Watkins [107], who realized that one requirement for the applicability of these methods is unlikely to hold for animals, namely that the agent can keep track of the topology of the state space of the domain, i.e. that it can remember for every state-action pair all possible outcomes together with their probabilities and execution costs. He proposed a reinforcement learning procedure, called q-learning, that requires the agent only to remember (and update) one value for each state-action pair. Since all of the Markovian decision algorithms are anytime algorithms, learning is possible by executing the plan that is currently believed to be best, and updating the plan from time to time by running the anytime algorithm for a time slice. (This might require to experiment, i.e. to deviate from the plan that is currently believed to be optimal.)

The theory of making decisions in a probabilistic environment with costs and rewards was developed in mathematics and operations research, mainly in the subfields of statistics, decision theory, and Markovian decision theory. The dominating application areas were decision making in medicine and business administration. The non-adhoc AI approaches to probabilistic and decision-theoretic planning fall into two camps: One that still uses conventional planning methods, but that evaluates possible plans by taking the probabilities into account (for example [46] or [110, 108, 111]), and one that directly uses the information about the probabilities for planning. The latter camp usually relies on some form of propagation of the probabilities through the search tree (see [80] for details) using either Bayes' nets to keep track of the dependencies between conditional probabilities, or utility/decision trees, influence diagrams [58] or other methods (e.g. Russell and Wefald's method [90, 89, 88, 87]), but usually heuristics are chosen over exact methods to cope with the complexity of planning. Examples of such planning methods include Russell and Wefald [90], Drummond and Bresina [15], Hansson and Mayer [49], Dean and Kanazawa [24], and Smith [98]. (Overviews are given in [23] and [45].)

For example, the Bayesian Problem Solver (BPS) (see [47] for BPS and search, and [49] for BPS and planning) provides a domain-independent framework for a heuristic search system based on Bayes' nets that model the search tree. In order to apply BPS to a specific domain,

one has to encode the domain knowledge that is used to control the search. It mainly consists of heuristics that estimate the goal distances for nodes in the search tree. The possible outcomes of a heuristic are probabilistically related to the real goal distances and perhaps to the outcomes of other heuristics either for the same node or a different node (usually the predecessor of the node) in the search tree. The value of a particular heuristic is mapped into the probability distribution over the values of the correct goal distance or the values of other heuristics. Thus, rules express (probabilistic or deterministic) conditional constraints on a value given a different value. This framework does not need to be changed for probabilistic domains. When adapting the BPS framework for a new (in this case probabilistic) domain, one only has to do two things: to come up with good, i.e. discriminative, heuristics and to relate their outcomes to the correct goal distances.

Besides becoming interested in probabilistic planning, planning researchers also started to realize that there are trade-offs between planning time, execution time, and quality of the solution. Planning time is a valuable resource [14]. Thus, the planner must be able to decide whether it should continue to plan and which of the planning subtasks it should schedule next (for example, which node in the search tree to expand), or whether it should stop planning and start executing the plan. Instead of making such a decision directly, the planner can also construct a plan for deriving the decision (meta-planning, deliberation scheduling). Agents that take their resource constraints into account are called limited rational agents (see [30, 29, 28]).

We will briefly mention one decision-theoretic approach to planning: anytime algorithms. Dean and Boddy [11, 12] characterize an anytime algorithm as an algorithm that can be asked for a solution at any point in time, where the quality of the answer improves the longer the algorithm is allowed to run, and for which it is possible to characterize the trade-off between the quality of the solution and the run-time of the algorithm. This concept is an idealization: Algorithms that are characterized as anytime algorithms need a certain start-up time, during which no solution is available. Then, the solution is updated at discrete time points, see figure 6. Anytime algorithms help to cope with the deliberation scheduling problem. Many of the well-known optimization algorithms from operations research (such as the Simplex algorithm for solving linear programs) or statistics (such as sampling) are anytime algorithms. It is not yet clear, whether a complex planner can be realized as a collection of anytime algorithms [45] and whether it is tractable at all to use decision theoretic techniques in complex planners. (For complexity results see [35].)

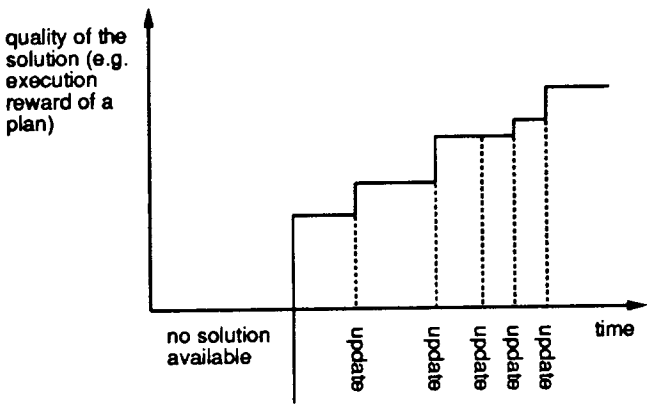


Figure 6: (Time, Solution Quality) Graph of an Anytime Algorithm with Start-Up Time

3. A Formal Model of the Planning Problem

In this chapter, we will state the planning problem more precisely than in the introduction. This will provide us with a basis for formalizing the problem, and enables us to define the terms more precisely. At the same time, we will explain the components of the planning model and show its flexibility.

Our probabilistic and decision-theoretic planning model uses the (GPS) state space representation (rather than the partial plan representation [91, 92]). Given is a finite state space, and for each state a finite set of applicable actions. During the execution, the agent is in exactly one state at every point in time (measured on a discrete time scale). When executing an (applicable) action in a state, the agent again reaches a state in the state space. The probability distribution over the successor states is known and depends only on the state that the agent was in and the action that it executed in that state. An execution cost (in "currency" units or utilities, see below) is associated with each (state, action, outcome) triple. The planner also knows the probability distribution over the states that will be used to determine the start state, and it knows the set of goal states. With each goal state there is a (positive or negative) reward associated.[6] The environment is assumed to be static, i.e. the probabilities,

[6]In the following, we will use the convention that "costs" are negative values, whereas "rewards" can be any (i.e. positive or negative) values. Although the costs are associated with (state, action, outcome) triples, i.e. depend on the state, the action executed, and the successor state reached, one can calculate from this information the expected cost for executing an action in a state as the sum of the costs of the (state, action, outcome) triples weighted by the probability of the action outcome. This value is meant when we talk about

costs and rewards do not change over time.

Planning is done off-line, i.e. there is a planning phase followed by a separate execution phase. During the planning phase it is unknown which of the possible outcomes of an action will occur. But the transition probabilities are only relevant during planning, since we assume that the agent exactly knows which state it is in at all times during the execution. At the beginning of the execution phase, the start state is randomly chosen according to the given probability distribution. Then, the agent selects an action from the set of applicable actions for that state according to its instructions (the plan) and executes it. Next, it determines the new state that it has reached and again selects an action, etc. If the current state is a goal state, then the plan can require the agent to execute a stop action to stop the plan execution. The (total) execution reward (of the plan) is the sum of the costs of the outcomes of the executed actions plus the reward of the goal state in which the agent stopped.

The planning task is to find the best plan among the plans that guarantee that the execution eventually stops in a goal state. If the planning time is limited, we want to find an as good plan as possible that satisfies the above restriction. The goodness of a plan is determined by its total execution reward.

We have seen that GPS-like planning problems are a special case of our more general planning problem. The implicit assumptions of our planning model are made explicit in appendix 1. Most of these assumptions are relaxed restrictions that are usually imposed by GPS-like planning models.

A formal model of our planning problem consists of the following items:

1. S, a non-empty, finite set of states;
2. for all $s \in S$: $A(s)$, the finite set of actions that are applicable in state s;
3. for all $s, s' \in S$ and $a \in A(s)$: $p(s'|s,a) \in \Re_0^+$, the conditional probability that the successor state is s' when the action a is executed in state s; (since $p(\cdot|s,a)$ is a probability distribution over S, it must hold that $\sum_{s' \in S} p(s'|s,a) = 1$);
4. for all $s, s' \in S$ and $a \in A(s)$, such that $p(s'|s,a) > 0$: $c(s,a,s') \in \Re^-$, the execution

---

the cost of an action and the outcome is not known. If the outcome is known, we will talk about the cost of the outcome of the action and mean the cost of the corresponding (state, action, outcome) triple.

cost of the transition to state s' when having executed action a in state s;

5. for all $s \in S$: $b(s) \in \Re_0^+$, the probability that state s is the start state; (since $b(s)$ is a probability distribution over S, it must hold that $\sum_{s \in S} b(s) = 1$);

6. $G \subseteq S$, the set of goal states; and

7. for all $s \in G$: $r(s) \in \Re$, the reward for stopping in goal state s.

A state s is called a start state, iff $b(s) > 0$. A state s' is called a (potential) outcome of action $a \in A(s)$, iff $p(s'|s,a) > 0$. The probability of this outcome is $p(s'|s,a)$ and its execution cost is $c(s,a,s')$. The execution cost of action $a \in A(s)$ is $\sum_{s' \in S} p(s'|s,a)c(s,a,s')$.

STRIPS-like operators do no longer exist in the planning model: Operators are replicated for every state in which they are applicable and for every instantion of variables that has different effects. For example, if an operator is applicable in state s_1 and deterministically leads to state s_2 or s_3 depending on its instantiation, and it is also applicable in state s_3 and leads to state s_4 with 50 percent probability and to state s_5 with 50 percent probability, then it is described as three different actions, two for state s_1 and one for state s_3.

The set of applicable actions depends on the state. Actions that are applicable in one state need not be applicable in another. This does not cause problems, because the agent always knows which state it is in during plan execution. The probability of an outcome and its execution cost depend on the action chosen, the state it is executed in and the resulting successor state, but not on the history of the execution. For example, painting a block might be more expensive than unstacking it from a 10-block stack, which might cost more than unstacking it from a 2-block stack. If we detect a failure early during an attempt to execute an action and are able to cancel its execution with a failure report, we might need less work than a successful execution of the action would need. On the other hand, if we are not able to cancel its execution, the failing action might take more effort to execute.

Costs and rewards could be expressed in some "currency" unit (e.g. seconds, dollars etc.). For example, if one had to pay a certain amount of money for the execution of an action, one would use dollar amounts to express the costs. If one wanted to minimize the expected execution time of the plan, one would use the (negative of the) time it takes to execute an action as its cost. But one usually prefers "utilities" over "currency" units. Utilities express the preferences of the agent between otherwise possibly incompatible goods. Using utilities,

one can model not only risk-neutral behavior, but also risk-aversive and gambling behavior. Then, the reward of a goal state represents the satisfaction of having accomplished this particular goal, and the execution costs for the actions represent the displeasure about the effort that the execution of the plan caused. Researchers in decision theory have developed methods for measuring such preferences and converting them into utility functions. (For example, see [71, 82, 55, 56]. For an early account in AI see [36], and for a current overview of their application in AI see [27]. Decision theoretic work in an AI context can be found in [44, 48].)

In general, a plan is any specification of commands that completely determines the behavior of the agent during the execution phase. This does not imply that the behavior of the agent executing the plan is predictable in advance, since plans are allowed to contain randomness. For example, a plan could contain coin-flipping actions, if it specifies the actions to take for each of the possible outcomes of the coin flip. At every point the time, the agent exactly knows what to do. But without knowing what the coin flip will be, an external observer is not able to predict the behavior of the agent. It will turn out in chapter 6 that for every planning problem there exists a stationary plan that is optimal. Thus, we can restrict our attention to universal plans, and we do not need to worry about randomness.

A universal plan is a function f from S into $\bigcup_{s \in S} A(s) \cup \{stop\}$ such that $f(s) \in A(s)$ (if $s \notin G$) or $f(s) \in A(s) \cup \{stop\}$ (if $s \in G$). In other words, a universal plan is a state-action assignment: Whenever the agent is in state s, the plan requires it to execute the action $f(s)$. The function f is total, i.e. the universal plan assigns an action to every state, even to states that are not reachable from a start state. Since the action assignment for such states does not matter, they can get an arbitrary (applicable) action assigned.

We expect a plan at least to direct the agent in such a way that it will eventually achieve one of the goal states. Let us make this notion of an admissible plan more precise: A state s' is called directly reachable from a state s with action $a \in A(s)$, iff $p(s'|s,a) > 0$. s' is called directly reachable from s under plan f, iff s' is directly reachable from s with $f(s)$. s' is called reachable from s under plan f, iff there exist states $s_1, s_2, ..., s_n$ ($n \in \{1,2,...\}$) such that $s = s_1$, $s' = s_n$, and s_2 is directly reachable from s_1 under plan f, etc. (The "reachable" relation is the reflexive, transitive closure of the "directly reachable" relation.) s' is called reachable (directly reachable) from s, iff there exists a plan f such that s' is reachable (directly reachable) from s under f.

A plan f is called admissible, iff under f from every state that is reachable from a start state a state can be reached in which the plan requires to execute the stop action. An admissible plan is called a solution of the planning problem. An admissible plan has the following property: If we view probabilistic planning as a game against nature in which the planner chooses an action and afterwards nature chooses the outcome of the action, then an agent executing an admissible plan can never get trapped during execution. No matter what the outcome of an action is, the agent will eventually reach a goal state and stop in it. The probability of not having executed a stop action in the first n steps approaches (at least asymptotically) zero, when n approaches infinity. (In order to be able to guarantee these properties, it is not enough to know that from every start state at least one goal state can be reached in which the plan requires to execute the stop action, since it might well be that states can also be reached from which the task will no longer be completed successfully.)

If the domain (or an external supervisor) requires the agent to stop after having reached a particular goal state (or to stop in the first goal state that it reaches), we simply allow in that particular goal state (or in all goal states) only one action, namely the stop action thereby leaving the planner no choice which action to choose.

If a plan is admissible then we define the total (execution) reward of the plan as the expected total reward of the plan execution (i.e. the expected sum of the execution costs of all (state, action, outcome) triples encountered during plan execution plus the reward for the goal state in which the execution is stopped) if the start state is chosen according to the probability distribution b. This value is well defined, since the plan is admissible. A plan is called optimal or an optimal solution of the planning problem, iff it is admissible and no other plan yields a larger total execution reward.

One could make every state in the planning model a goal state and influence the goodness of the plans only with the costs and rewards. A purely decision-theoretic planning problem is defined to be a planning problem for which every state is a goal state. States one does not intuitively consider to be goal states get (depending on the domain) attached a reward of zero or a highly negative reward as punishment for stopping the execution in them, thereby making it highly undesirable. This is a special case of our more general framework, that distinguishes between goal states and non-goal states. (In a purely decision-theoretic planning model, non-goal states could be viewed as goal states that have a reward of minus infinity attached, but then one needed to deal not only with numbers, but with the object "minus infinity" as

well.) The reason for the introduction of goal states was our desire to generate plans that guarantee that their execution will eventually accomplish the given task, i.e. stop in one of the designated goal states. If it turns out that a given planning problem is not solvable, because there is no plan that guarantees that its execution eventually stops, one could turn enough additional states (e.g. all unsolvable states) into goal states. However, we will regard such a planning problem as unsolvable.

4. The Blocks-World and the Augmented Blocks-World

In this chapter, we will describe the two example domains in greater detail that we use in this report: the classical blocks-world and the augmented blocks-world. The augmented blocks-world generalizes the classical blocks-world in four aspects: First, an action to paint blocks is added. This creates a large variety of alternative plans that achieve the same goal. Second, the actions have probabilistic outcomes. Third, the actions have costs and the goal states rewards attached. Fourth, blocks do no longer have unique names, but can only be referred to by their properties (e.g. location or color). We will describe augmented STRIPS operators for probabilistic and decision-theoretic domains and show how the actions of the augmented blocks-world can be expressed by these operators.

One classical example domain for a planner is the blocks-world (see e.g. [40]). We describe a version that consists of four blocks, named *bl*1, *bl*2, *bl*3, and *bl*4, and a horizontal surface, called *table*. The blocks are uniquely identifiable cubes of equal volume. They can be stacked, subject to the following restrictions: Every block has to be directly supported by either exactly one other block or the table. At most one other block can be placed directly on top of a block, but an arbitrary number of blocks can be placed directly on the table. No block can directly or indirectly support itself. We are only interested in the way the stacks are assembled from the blocks, not in the spatial relationships among the stacks. As an example, it does not matter whether block *bl*1 is to the left or to the right of block *bl*2. But *bl*1 being on top of *bl*2 is a different state from *bl*2 being on top of *bl*1.

The following relations are needed to describe the operators: (Variables start with uppercase letters and are untyped. For example, *BlockA* is a variable and every object, even the *table*, can be bound to it.)

- $BLOCK(BlockA) \equiv BlockA$ is one of the four blocks;
  $BLOCK := \{bl1, bl2, bl3, bl4\}$;
- $ON(BlockA, BlockB) \equiv BlockA$ is directly on top of $BlockB$;
- $CLEAR(BlockA) \equiv$ the top of $BlockA$ is clear;
  $\neg\exists BlockB : ON(BlockB, BlockA)$;
- $UNEQUAL(BlockA, BlockB) \equiv BlockA$ and $BlockB$ are different blocks;
  $UNEQUAL := \{(bl1, bl2), (bl1, bl3), (bl1, bl4), (bl2, bl1), \ldots\}$;

Then, the operators can be stated as follows:

- "move block $BlockA$ from top of block $BlockB$ to block $BlockC$"

```
move(BlockA,BlockB,BlockC)
  precond: on(BlockA,BlockB),clear(BlockA),clear(BlockC),
           block(BlockB),unequal(BlockA,BlockC)
  delete:  on(BlockA,BlockB),clear(BlockC)
  add:     on(BlockA,BlockC),clear(BlockB)
```

- "stack block $BlockA$ on block $BlockB$"

```
move(BlockA,table,BlockB)
  precond: on(BlockA,table),clear(BlockA),clear(BlockB),
           unequal(BlockA,BlockB)
  delete:  on(BlockA,table),clear(BlockB)
  add:     on(BlockA,BlockB)
```

- "unstack block $BlockA$ from block $BlockB$"

```
move(BlockA,BlockB,table)
  precond: on(BlockA,BlockB),clear(BlockA),block(BlockB)
  delete:  on(BlockA,BlockB)
  add:     on(BlockA,table),clear(BlockB)
```

The start state (and thus every state) contains the $BLOCK$ and the $UNEQUAL$ relation. The $UNEQUAL$ relation is needed, because we allow different variables to unify with the same constant (for example object name), which is ruled out by most STRIPS-like planners. Therefore, we have to provide a predicate with which one can assure that two variables are bound to different values. It follows from the operator definitions that we do not need to include block-table pairs into the $UNEQUAL$ relation.

The blocks-world has four drawbacks that we remedy by augmenting the domain to the augmented blocks-world:

- Small Variety of Actions

  In the classical blocks-world, there is only a small number of plans that achieve a given goal and all of them achieve the goal in the same way, namely by moving blocks around. To create a larger variety of plans, we add an operator that paints a block either white or black, regardless of its position. Now every block has a color. It is always colored either totally black or totally white.

- Deterministic Actions

  In the classical blocks-world, actions have deterministic effects. In the augmented blocks-world, we introduce probabilistic outcomes of actions by distinguishing two effects of an action: the expected effect and an unexpected effect (either a failure or the expected effect accompanied by a side effect). The move action will fail with a certain probability (smaller than one). In this case, the block that the agent wanted to move drops onto the table. The paint action will have an unwanted side effect with a certain probability (smaller than one). If the block that the agent wants to paint is supported by another block, the supporting block will be accidently painted as well (in the same color). (Note that for many domains it is impossible to classify the outcomes of actions as either expected outcomes or failures. In the artificial augmented blocks-world, this classification makes it easier to describe the action outcomes. Although we use this terminology, we do not treat "expected outcomes" and "failures" differently, which is what traditional planning approaches do.)

- Length of Plan as Goodness Measure

  In the classical blocks-world, the goodness of a plan is determined by its length or a similar measure. In the augmented blocks-world, different (state, action, outcome)

triples can have different costs, and different goal states can have different rewards associated. We require the costs of actions to be negative. Then, the goodness of a plan can be measured by its expected total execution reward.

- Block Names

  In the classical blocks-world, blocks are identified by unique names. Since a block in the augmented blocks-word has enough properties that can be used to identify it, we refer to blocks in the augmented blocks-world no longer by names, but by their properties (such as their $COLOR$ or ON relationships). This is a simple generalization, that is not needed by probabilistic and decision-theoretic planning methods, but that makes sense, since in non-artificial domains objects usually do not have (unique) names. This generalization decreases the number of states: Block configurations that are different if the blocks have names can collapse into the same configuration. (For example, if in a two-block blocks-world block $b1$ is white and block $b2$ were black, it would matter whether $b1$ is on top of $b2$ or vice versa. But if both blocks were white, it would not make a difference which block is on top of the other, because in both cases the state is characterized by a stack of two white blocks.) On the other hand, the generalization complicates the representation of the states as sets of predicates and the test whether two states are equal.

The following additional relations are needed to describe the operators:

- $COLOR(ColorA) \equiv ColorA$ is one of the two colors;
  $COLOR := \{white, black\}$;
- $COLORED(BlockA, ColorA) \equiv$ the block $BlockA$ is colored in color $ColorA$;

In the following, we state the operators of the augmented blocks-world domain using an augmented STRIPS-like operator notation. Since the transition probabilities and execution costs are allowed to depend on the state the action is executed in, the outcome and the variable bindings of the operator, we represent these values below by the placeholders "num" and state (as meta notation) in parentheses the restrictions that these values must satisfy. The nums can be either numerical constants or function calls that are parameterized with the state and the variable bindings.

- "move block *BlockA* from top of block *BlockB* to block *BlockC*"

```
move(BlockA,BlockB,BlockC)
  precond:  on(BlockA,BlockB),clear(BlockA),clear(BlockC),
            block(BlockB),unequal(BlockA,BlockC)
  outcome /* the expected outcome */
    prob:   num1 (num1>=0)
    cost:   num2 (num2<0)
    delete: on(BlockA,BlockB),clear(BlockC)
    add:    on(BlockA,BlockC),clear(BlockB)
  outcome /* failure: BlockA falls onto the table */
    prob:   num4 (num4>=0, num1+num4=1)
    cost:   num5 (num5<0)
    delete: on(BlockA,BlockB)
    add:    clear(BlockB),on(BlockA,table)
```

- "stack block *BlockA* on block *BlockB*"

```
move(BlockA,table,BlockB)
  precond:  on(BlockA,table),clear(BlockA),
            clear(BlockB),unequal(BlockA,BlockB)
  outcome /* the expected outcome */
    prob:   num7 (num7>=0)
    cost:   num8 (num8<0)
    delete: on(BlockA,table),clear(BlockB)
    add:    on(BlockA,BlockB)
  outcome /* failure: BlockA remains on the table */
    prob:   num10 (num10>=0, num7+num10=1)
    cost:   num11 (num11<0)
    delete: ---
    add:    ---
```

- "unstack block *BlockA* from block *BlockB*"

```
move(BlockA,BlockB,table)
```

```
precond:  on(BlockA,BlockB),clear(BlockA),block(BlockB)
outcome /* the expected outcome
           (the "failure" has the same effect) */
  prob:   1
  cost:   num13 (num13<0)
  delete: on(BlockA,BlockB)
  add:    on(BlockA,table),clear(BlockB)
```

- "color block *BlockA*, that is on the table, with color *ColorA*"

```
color(BlockA,ColorA)
  precond:  on(BlockA,table),colored(BlockA,ColorB),
            color(ColorA)
  outcome /* the expected outcome; (nothing can fail, since
             block BlockA is not supported by another block) */
    prob:   1
    cost:   num15 (num15<0)
    delete: colored(BlockA,ColorB)
    add:    colored(BlockA,ColorA)
```

- "color block *BlockA*, that is supported by another block, with color *ColorA*"

```
color(BlockA,ColorA)
  precond:  on(BlockA,BlockB),block(BlockB),color(ColorA),
            colored(BlockB,ColorC),colored(BlockA,ColorB),
  outcome /* the expected effect */
    prob:   num17 (num17>=0)
    cost:   num18 (num18<0)
    delete: colored(BlockA,ColorB)
    add:    colored(BlockA,ColorA)
  outcome /* failure: the supporting block is painted as well */
    prob:   num20 (num20>=0, num17+num20=1)
    cost:   num21 (num21<0)
    delete: colored(BlockA,ColorB),colored(BlockB,ColorC)
    add:    colored(BlockA,ColorA),colored(BlockB,ColorA)
```

In figure 7, we show part of the state space of the augmented blocks-world, namely all actions that are applicable in a particular state together with their outcomes. (The state space is too large to show all of it.) This augmented blocks-world has the following properties: The actions achieve their intended effects with probability 0.6, and fail with probability 0.4. Painting a block always costs exactly one, whereas moving a block costs at least one. If a block is moved upward, then the level difference it is moved counts as additional cost. Thus, the nums have the following values: num1 = num7 = num17 = 0.6, num4 = num10 = num20 = 0.4, and num2 = num5 = num9 = num11 = num13 = -1 – max(0,level of the moved block afterwards–level of the moved block before), i.e. num5 = num11 = num13 = -1. Note that we have labeled the blocks in the source configuration of figure 7. This allows us to state the actions concisely. It does not imply that the blocks have unique names; permuting the names does not create a different configuration. For example, one of the successor states of the initial state after executing the $move(b2, b1, b3)$ action is the initial configuration itself, although the block names are permuted by the action.

5. Properties of the Planning Problem

In this chapter, we will give a brief introduction to different methods that are used for probabilistic planning, and their properties. The idea behind this introduction is to give the reader a feeling for the problems that one has to solve when designing a probabilistic and decision-theoretic planner. We will also introduce the idea behind Markovian decision algorithms and show why they are useful for planning. The authors of operations research textbooks usually choose to give a formal introduction to dynamic programming [8], explore its properties, and then proceed to Markovian decision algorithms. In this chapter, we use a different approach: we introduce Markovian decision algorithms in an informal way in the context of planning. First, we show how non-cyclic planning problems can be represented as decision trees (from utility theory) and solved using the roll-back method. Then, we show how the solution method has to be altered to be able to handle cyclic planning problems as well. We discuss two such ways, that lead to the value-iteration method and the policy-iteration method, respectively. We discuss the advantages and disadvantages of these planning methods and include a discussion of probabilistic planning problems with a deadline (time horizon). Then, we will give an overview of the properties of Markovian decision algorithms in chapter 6. Our presentation is aimed at readers that have no background in operations research,

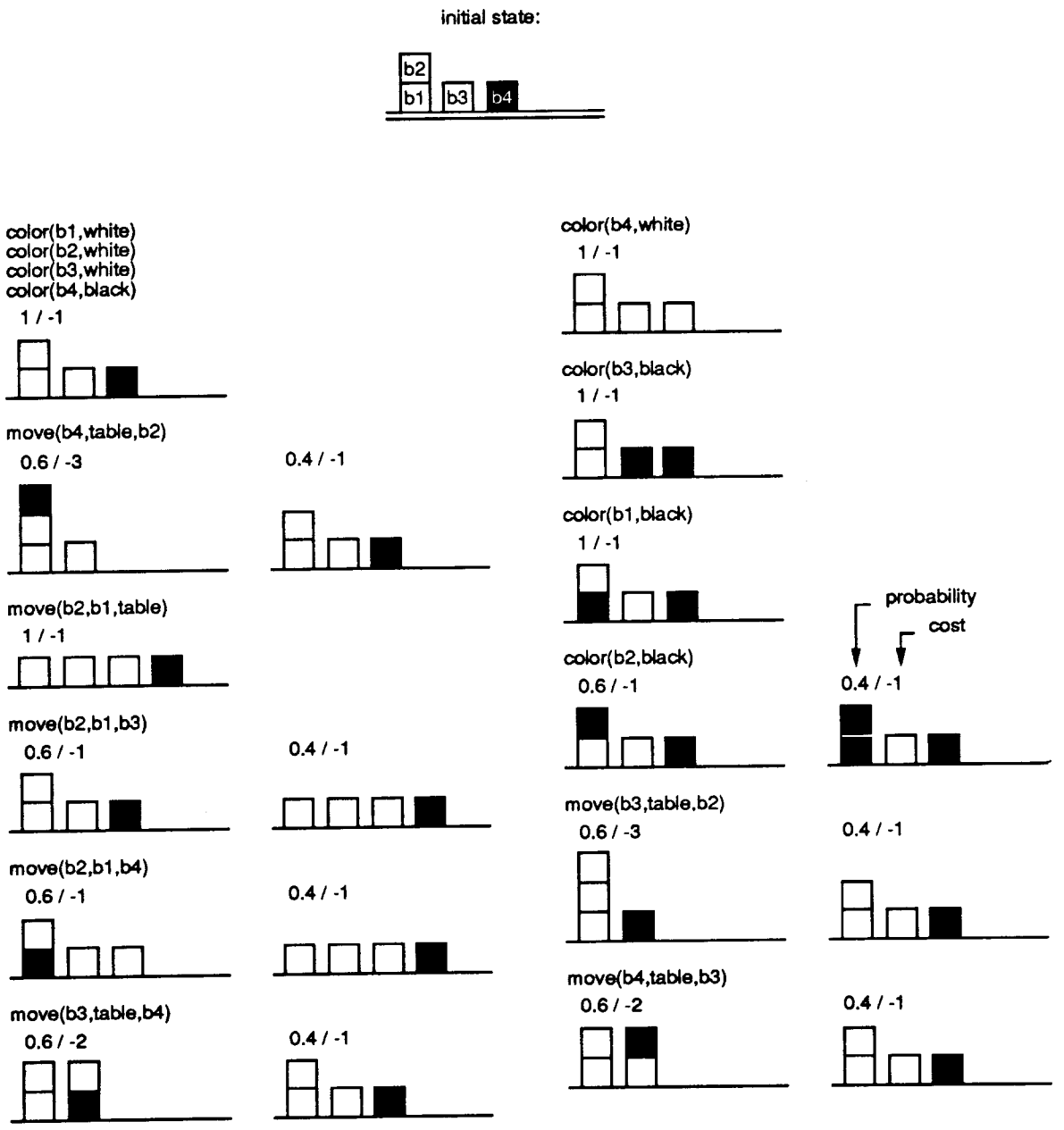


Figure 7: An Example for the Augmented Blocks-World

but in decision theory and utility theory. The more theoretical reader might want to skip chapters 5 and 6 and instead read a formal introduction to Markovian decision problems such as [84], chapter 6, and [54].

As we have already stated, there always exists an optimal plan for a probabilistic planning problem (without deadlines) that is universal (if a solution exists at all). This shows that the probability distributions are not needed during the execution phase: The compiled plan is deterministic. We will use this justification for universal plans in probabilistic domains to reduce the size of the set of plans that we have to search in order to find an optimal plan. There are only a finite number of universal plans for any given planning domain, since the number of states and the number of applicable actions in each state are finite. This guarantees that an optimal plan exists. This plan could be found by enumerating all universal plans, determining their goodness values, and then choosing the universal plan with the largest value. But the number of plans can be exponential in the number of states, and the number of states is already exponential in the number of predicates used to describe them in a STRIPS-like notation. We are interested in methods that find an optimal plan faster.

In deterministic domains, we want to avoid universal plans that have cycles. A cycle is a set of states such that for any two states s and s' in that set there is a positive probability for the agent to reach s' when it is started in s and acts according to the given plan. Once an agent has entered a cycle in a deterministic domain, it is no longer able to leave it. Thus, it is no longer able to stop the execution. In other words, once the agent reaches a state again during plan execution, it cycles and will never complete the execution. In probabilistic domains, cycles are only a problem, if the agent cannot leave the cycle. The repetition of a state during plan execution does not imply that the agent cannot complete its task successfully. These cycles make probabilistic planning harder than deterministic planning.

It is important to understand that a universal plan that contains cycles can indeed be optimal for a probabilistic planning problem. One could be fooled to believe that the optimal universal plan has to be acyclic by an argument such as the following: "Since every action has a negative cost, the number of actions executed during plan execution must be bounded. (Otherwise the total reward of such an execution and thus the total execution reward of the plan would be minus infinity, which cannot be optimal.) If a universal plan contains a cycle, it can lead to an unbounded execution sequence and thus cannot be optimal." This is not a valid argument, because the probability that the number of executed actions exceeds a given

(arbitrary) limit, can converge towards zero, when the limit approaches infinity. Thus, the total execution reward of a plan can be bounded, although the total rewards of the executions are not bounded. (A consequence is that cyclic interdependencies do not allow an early pruning of partial solutions for probabilistic planning domains, as utilized in branch-and-bound methods.)

Assume that we want to find the optimal plan for a probabilistic domain in which every universal plan is acyclic. Then we can represent the planning problem as a decision tree (from utility theory [82], not from classification theory [81] as used by Schoppers to represent universal plans) or as an acyclic decision graph. Pearl [80] describes decision trees as an explicit representation of all scenarios that can possibly emanate from a given initial situation. This situation is represented by the root of the tree. Each path from the root corresponds to one possible scenario, involving a sequence of actions chosen by the agent (such choice points are represented by quadratic nodes, called decision nodes), probabilistic events that may occur (represented by circular probability nodes), and finally diamond-shaped value nodes that express the utility of the situation created by the entire path. (Acyclic decision graphs and decision trees are equally powerful, but acyclic decision graphs make the repetition of identical subtrees unnecessary.)

We represent every state of the planning problem by a decision node and every action as a probability node. We also create one value node, that has a value of zero. The successors of a decision node are the probability nodes that represent the actions that are applicable in the state represented by the decision node. These connections have a cost of zero (since choosing which action to execute is free). The successors of a probability node are the decision nodes that represent the states that can be reached when executing the action represented by the probability node. These connections have the probability and the cost of the corresponding (state, action, outcome) triple. Every decision node that represents a goal state additionally has a probability node as successor that represents the stop action. This connection has a cost of zero. The only successor of this probability node is the value node. This connection has a probability of one and a cost that is equal to the reward that the agent receives when it executes the stoppaction.

Acyclic decision graphs can be solved using the roll-back method. This method is a backward search method from dynamic programming that starts with the value nodes and proceeds in reverse topological ordering with the nodes. The value of a value node is the value with

which the node is labeled. The value of a probability node is the sum over its successor nodes of the transition probability times the sum of the value of the node and the transition cost. The optimal decision for a decision node is the successor node that maximizes the sum of the value of the node and the transition cost. The value of a decision node is the sum of the value of its optimal decision and the transition cost. The roll-back method calculates the value of every node once. After a finite amount of time, the values of all states and the optimal decision for the decision nodes are found.

An example is given in figure 8. In this example, there are three states. In state 1, two actions can be executed. Action 1 leads with probability 1 and cost -1 to state 2. Action 2 leads with probability 0.4 and cost -1 to state 2 and with probability 0.6 and cost -2 to state 3. In state 2, two actions can be executed, both leading with probability 1 to state 3. Action 3 has cost -2, and action 4 has cost -3. State 3 is a goal state with reward 10. In state 3, only the stop action can be executed. The acyclic decision graph is annotated with the results of the roll-back algorithm. The optimal plan is to choose action 2 in state 1, action 3 in state 2 and the stop action in state 3.

Until now we have assumed that every universal plan in the planning domain is acyclic. If this is not the case, we can still represent the planning problem with a decision graph, but the decision graph will have a cycle. Cyclic decision graphs are uncommon in the decision theory literature, because there they are interpreted as representing static decision making. Then, a cycle in a decision graph implies that a decision directly or indirectly depends on itself. Such decisions are not very interesting. We interpret decision graphs as a representation for dynamic decision making. Thus, a cycle implies that a certain decision can be made at several points in time.

The roll-back method can be used to solve cyclic decision graphs as well. Unfortunately, the nodes can no longer be sorted topologically. Which decision to choose for a decision node (and thus its value) can now depend directly or indirectly on its own value. Thus, it is no longer true that the values of its successor nodes have already been calculated, before the value of a node is to be determined. To solve this dilemma, we note that the decision (and thus the value) of a decision node can only depend on past decisions, not future ones. Therefore, we transform the cyclic decision graph into an infinite acyclic decision graph by replicating every state for every point in time. A transition between state s and state s' in the cyclic decision graph is transformed into transitions between the state that represents s

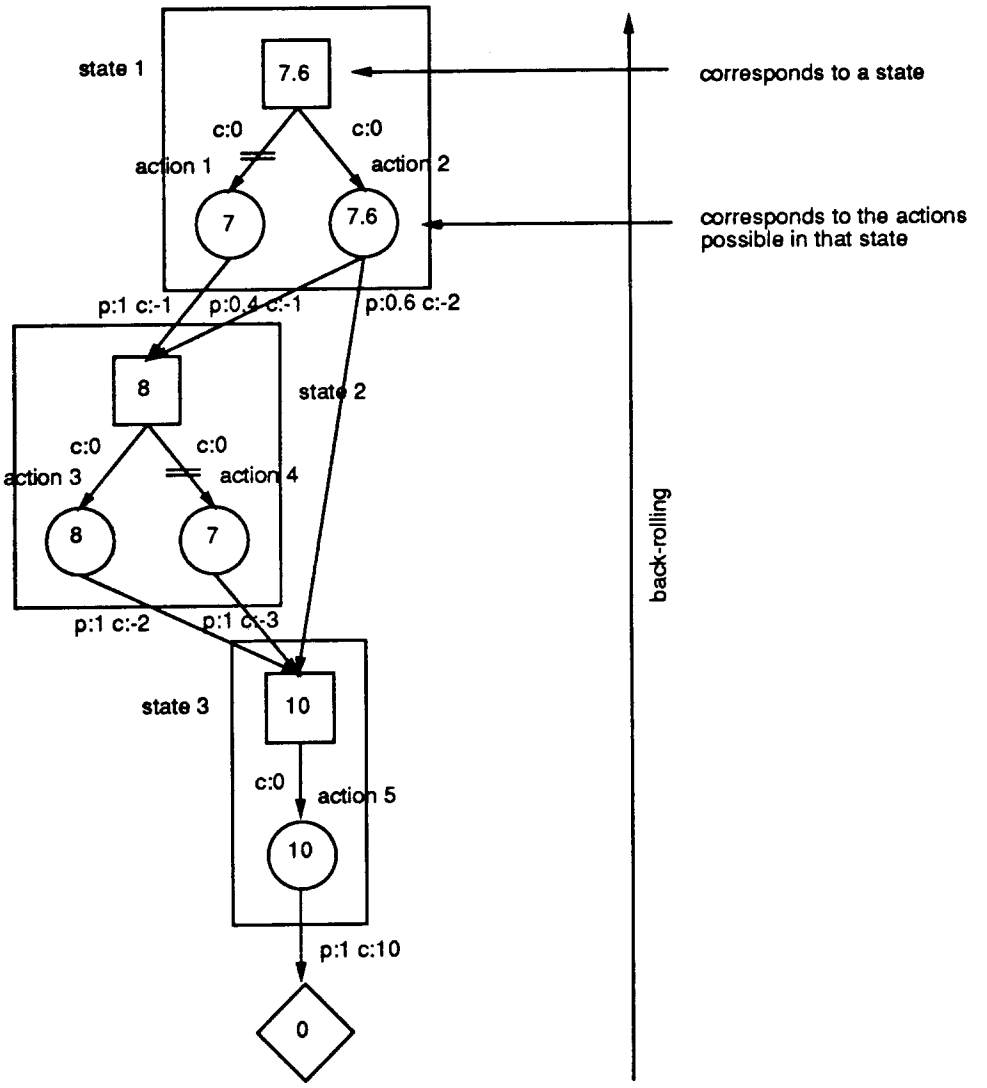


Figure 8: The Decision Graph for an Acyclic Planning Problem

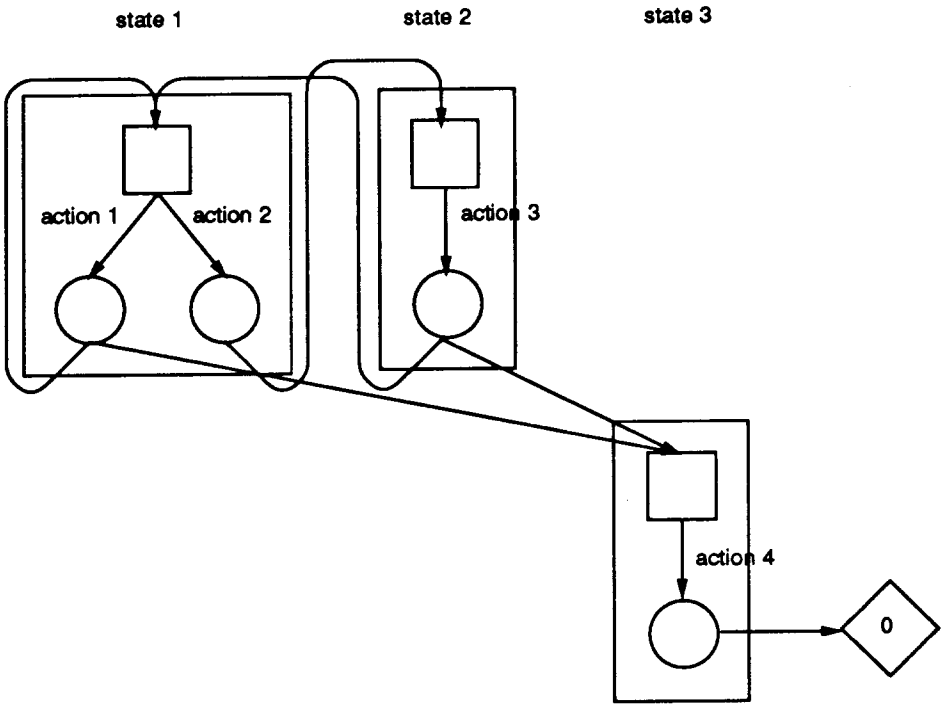


Figure 9: The Decision Graph for a Cyclic Planning Problem

at time t and the state that represents s' at time $t+1$ (for every $t \in \{1,2,...\}$). We call this transformation "unrolling" the cyclic decision graph. As an example, we show a cyclic decision graph in figure 9 and the corresponding unrolled decision graph in figure 10.

To show how we solve the cyclic decision graph using the corresponding unrolled decision graph, let us assume for a moment that a time horizon of n is given, and that the additional restriction is imposed that the execution has to be stopped after n actions (not counting stop actions) or earlier. If no stop action has been executed when the deadline is reached, the agent is forced to stop and does not receive the reward for stopping in a goal state. The planning problem remains to find the plan with the largest total execution reward. The decision graph for this planning problem is acyclic and finite. We can construct it by unrolling the cyclic decision graph and replacing all probability nodes at depth n (i.e. all nodes that correspond to actions executed at time n), with the exception of probability nodes that correspond to stop actions, by a value node labeled with zero. This decision graph can be solved using the roll-back method. As an example, we show the decision graph in figure 11 that corresponds to the decision graph in figure 9 when a time horizon of 3 is imposed.

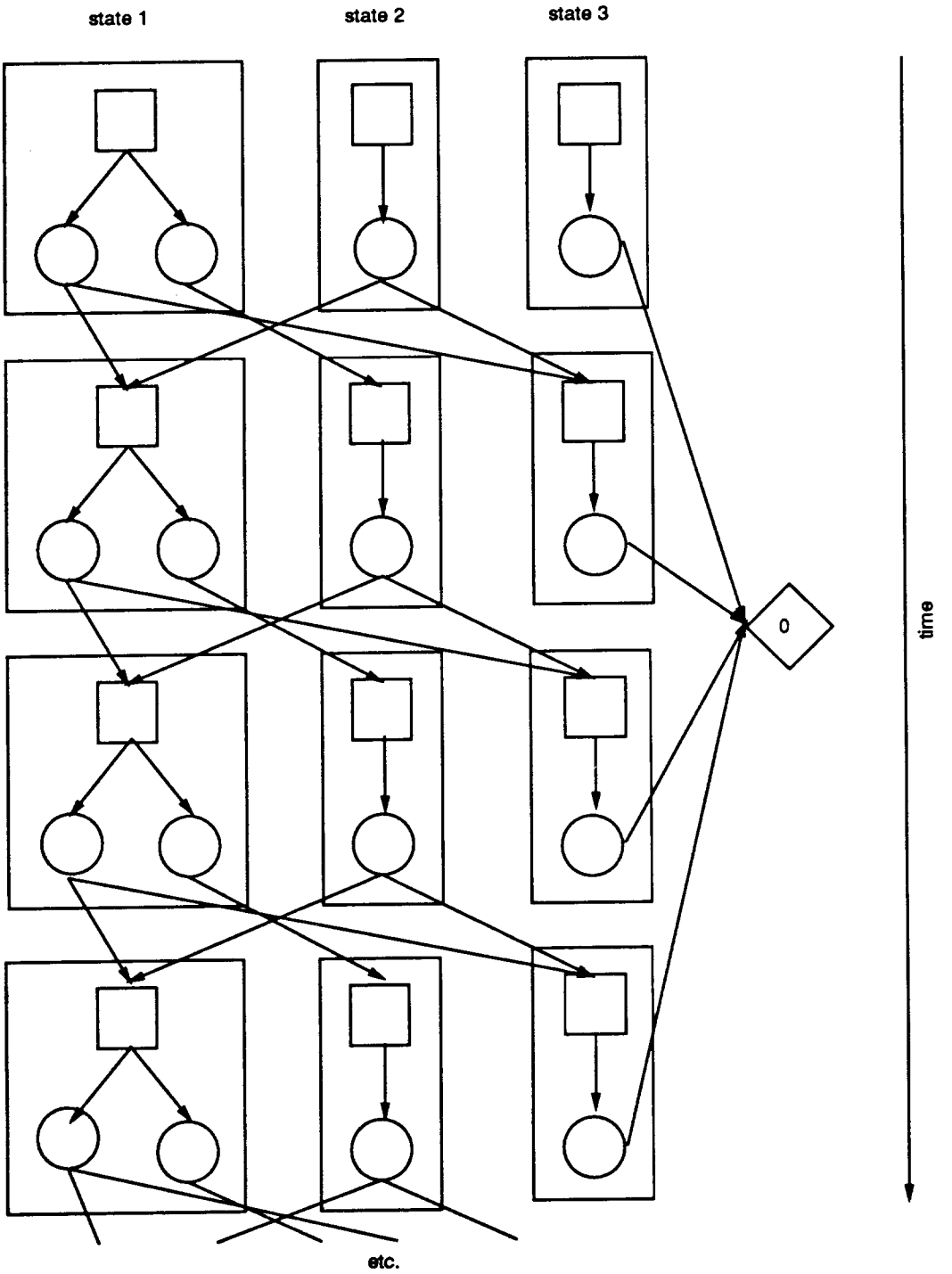


Figure 10: The Unrolled Decision Graph for the Acyclic Planning Problem from Figure 9

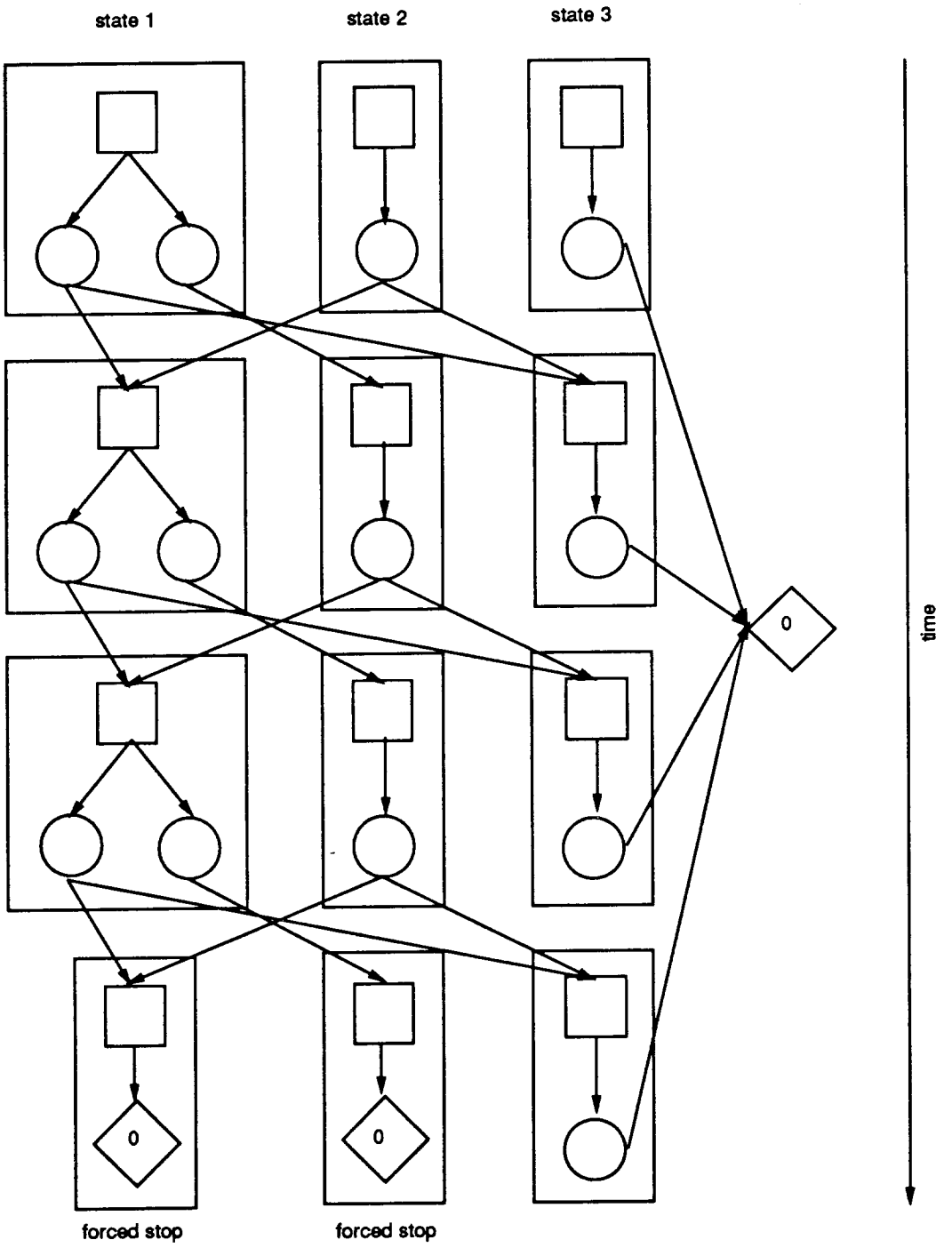


Figure 11: The Decision Graph from figure 9 with a Time Horizon of 3

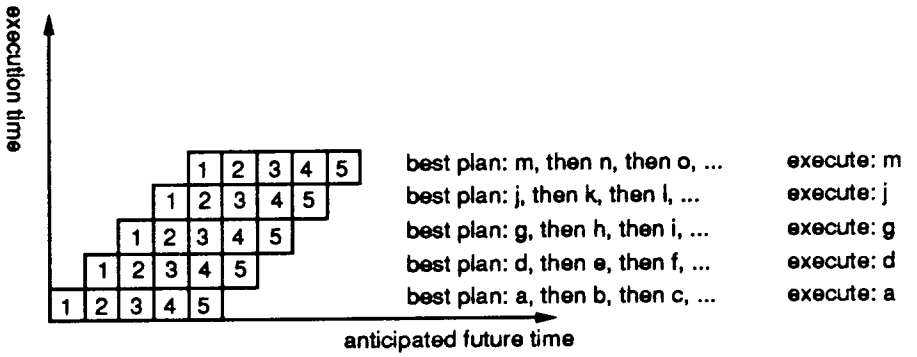


Figure 12: Revolving Planning with a Time Horizon of 5

Finding the optimal plan for a planning problem with a time horizon of n is equivalent to revolving planning, see figure 12 and the discussion of Dean and Kanazawa's work in chapter 2: A revolving planner determines the optimal plan for the planning problem with a time horizon of n, and executes the first action of the plan (i.e. the action assigned to the (current state, first time step) pair). Then, it repeats the process.

It is a hill-climbing approach to impose a time horizon to approximate the optimal solution of a planning problem without a time horizon. The limited lookahead of n time periods can be used to model the behavior of animals, because they are only capable of short-term planning. Consequences of their actions that are beyond the time horizon are not taken into account by them. By choosing the time horizon appropriately, we can build planners with a varying look-ahead thereby trading off planning time and the quality of the plan. For example, choosing the action that maximizes the immediate reward (sometimes misleadingly called "reactive planning") is a special case of revolving planning with $n = 1$. In our task-driven planning model, the limited lookahead presents a problem: A positive feedback is only received at the end of the plan execution. Before that, only costs occur. If the goal cannot be reached within the time horizon, revolving planning chooses a plan that minimizes the costs of the actions executed within the time horizon, regardless of whether that plan can achieve a goal, which goal it can achieve, and how costly achieving the goal is. Therefore, the horizon effect ,that is known from game playing domains [9], is a problem for planning as well. We need a heuristic evaluation function to evaluate intermediate states, but it is unclear how such a function looks like.

For planning problems with deadlines, it is no longer true that the optimal plan can always

be stated as a universal plan, i.e. state-action rules. Usually, one has to use state-time-action rules (specifying which action to choose when a certain state is reached at a certain point in time), since it can be optimal to execute different actions in the same state at different points in time.

Unfortunately, a time horizon does not exist for our planning problem. But we have seen that planning problems with a time horizon can be solved using the roll-back method, even if they are cyclic. If the time horizon approaches infinity, the solution of the planning problem with the time horizon approaches the solution of the planning problem without the time horizon. That is, the value of a state for a sufficiently late point in time approaches the value of the state under the optimal plan in the planning problem without a time horizon. (When the deadline approaches infinity, the value will asymptotically converge towards the correct value, but it might never be equal to tit) Likewise, the action assigned to a state for a sufficiently late point in time equals the optimal action for that state in the planning problem without a time horizon. (There exists a time point such that for all later time points the correct action is assigned to the state.) Thus, we could pick a time horizon that is large enough and solve the planning problem with the time horizon in order to approximate the solution of the planning problem without the time horizon. The disadvantage of this method is its inflexibility: If we have already approximated the solution, but want to approximate it even closer, we have to transform the cyclic decision graph again into an acyclic decision graph (this time using a larger time horizon) and solve that graph, thereby duplicating work we have done before.

A better method is to operate directly on the cyclic decision graph. At the beginning, the value of every node is initialized to zero. Then, we iteratively update each node of the cyclic decision graph using the current values of its successors. Each such iteration corresponds to extending the time horizon for one time step. The planner stops iterating when it runs out of planning time or when the approximation seems to be close enough to the exact solution. Such a method is called a value-iteration method. When we refer to *the* value-iteration method, we mean a similar method that first determines for every decision node its new decision using the old values of the nodes. Then, it simultaneously updates the values of all nodes using the decisions made. (There exist other derivations of the value-iteration method, that differ for example in the order in which the nodes are scheduled to get their values updated, see [2].) We will show in chapter 17 that some form of the value-iteration method (often combined with heuristics) is used by most of the common probabilistic planners. Since there are error

bounds known for the value-iteration method [10, 84, 85], it is possible to test whether the approximation is already close to the exact solution. They could be used to determine how large the time horizon has to be in order to achieve a good performance for the common probabilistic planners. They could also be used to trade off between planning cost and the quality of the plan. However, they are not very tight. For example, they cannot be used to immediately detect that the optimal plan has been found. Thus, if the value-iteration algorithm runs for a sufficiently long time, the optimal plan will eventually be found, but this will not be immediately detectable. (Some of these statesments are simplified, see chapter 6 for details.)

A different view on the value-iteration method is that it solves our problem that the decision to choose for a decision node in a cyclic decision graph (and thus its value) can depend directly or indirectly on its own value. The solution is to first assign initial values to the nodes and then to update the values iteratively: First, new decisions are determined for the decision nodes, then the values of all nodes are updated to reflect the change of the decisions.

A small change of the value-iteration algorithm leads to the policy-iteration algorithm. The policy-iteration algorithm first assigns initial values to the nodes and then updates the nodes iteratively, like the value-iteration algorithm: First, new decisions are determined for the decision nodes, then the values of all nodes are updated. (The policy-iteration algorithm can also be used by first assigning initial decisions to the decision nodes, and then iteratively determining the values of the nodes and updating the decisions.) The difference between the value-iteration algorithm and the policy-iteration algorithm is the way in which the values of the nodes are updated. The value-iteration algorithm does that locally. For example, remember that the value of a decision node is determined as the sum of the value of its optimal decision and the transition cost. The policy-iteration algorithm calculates the *correct* values under the given plan by solving a system of linear equations: There is one equation and one variable for each state. If action $f(s) \in A(s)$ is assigned to state s, then the corresponding equation is $v(s) = \sum_{s' \in S} p(s'|s, f(s))(c(s, f(s), s') + v(s')) = c(s, f(s)) + \sum_{s' \in S} p(s'|s, f(s))v(s')$. If the stop action is assigned to s, then the equation is $v(s) = r(s)$.

An example is given in figure 13. There, a decision graph and a plan (i.e. a decision for each decision node) are given, and it is shown how a repeated application of the value-determination operation of the value-iteration algorithm and a single application of the value-determination operation of the policy-iteration algorithm determine the values of the states.

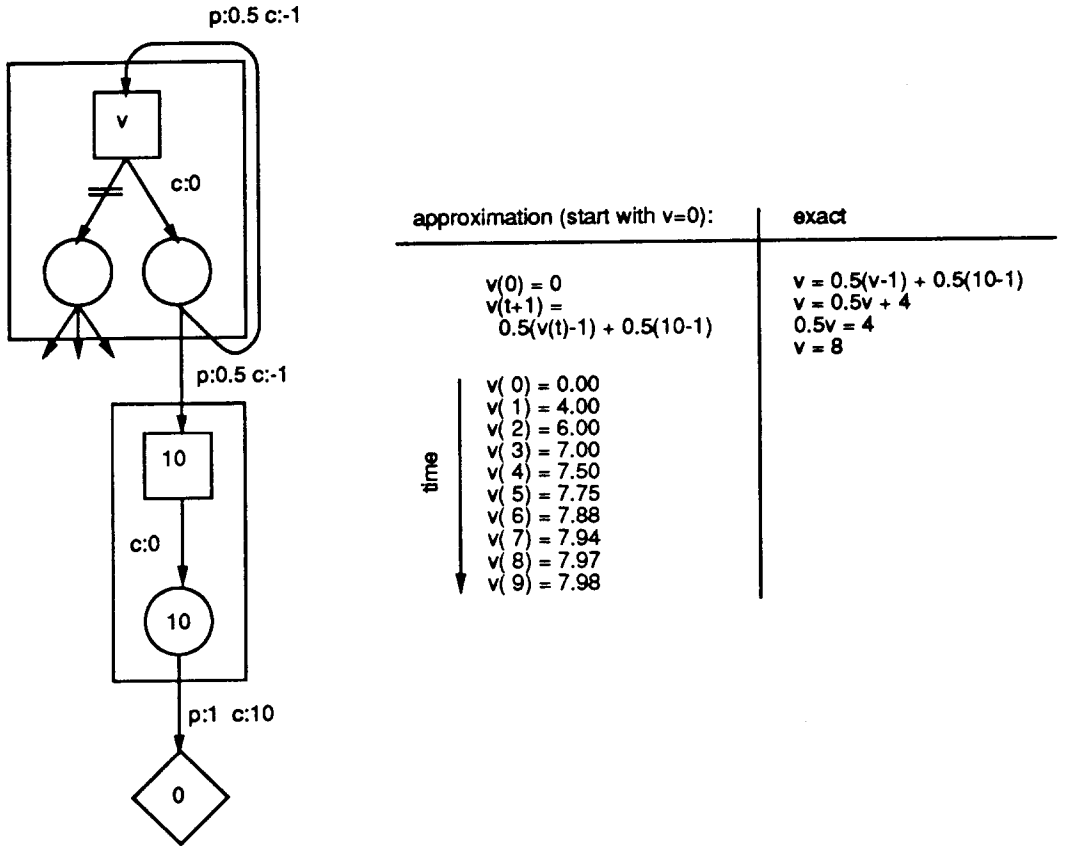


Figure 13: The Results of the Approximation Method and the Exact Method

The value-iteration algorithm can only approximate the value of state v, the policy-iteration algorithm calculates the exact value. (The value-iteration algorithm overlaps the approximation of the values under a given plan with the optimization of the plan. The policy-iteration algorithm keeps these two activities distinct.)

The policy-iteration algorithm finds the optimal plan for the planning problem without a time-horizon in finite time, and immediately detects that it has found the optimal plan. (This statement is simplified, see chapter 6 for details.) It is not able to solve planning problems with time horizons optimally.

The value-iteration algorithm and the policy-iteration algorithm both have strengths and weaknesses:

A drawback of the value-iteration algorithm is that the plan representation as (state, time, action) rules is less compact than its representation as (state, action) rules. Furthermore, the

(state, time, action) rules will usually only approximate the optimal plan for the planning problem without a time horizon. Given the (state, time, action) rules of an optimal plan for a planning problem with a manageable (i.e. reasonably small) time horizon, it is not at all clear how the (state, action) rules of the optimal plan for the planning problem without a time horizon look like. The approximation of the optimal solution can be very slow: It can take an arbitrary long time to reach a value for a node which is within an epsilon interval around the correct value. Such a case can easily be constructed and is depicted in figure 13, when we make the probability to reach the goal state arbitrarily small, but positive (for example 0.0001 instead of 0.5). The policy-iteration algorithm avoids all of these problems: It needs only a few iterations to converge, finds the optimal (state, action) rules, and detects immediately when it has found the optimal plan.

But the value-iteration algorithm also has advantages. The more local the steps of the value-determination operation are, the faster they are and the easier they can be parallelized and run on a multi-processor system. The value-iteration algorithm updates the value of a node by only looking at the values of its successor nodes. The policy-iteration algorithm has to determine the values for all nodes simultaneously by solving a linear programming problem. This is a global operation. Furthermore, since the value-iteration algorithm approximates the optimal solution, it can be used to trade off planning time and the quality of the plan.

In conclusion, it depends on the properties that are needed for a particular application whether to prefer the value-iteration algorithm or the policy-iteration algorithm. We will concentrate on the policy-iteration algorithm, since it enables us to find the optimal plan easily (as opposed to an approximation of the optimal plan).

6. Markovian Decision Problems

In this chapter, we will introduce the notion of a Markovian decision problem, its properties, and algorithms that can be used to solve it. Markovian decision problems provide a means to formalize probabilistic and decision-theoretic planning problems. We have already motivated Markovian decision algorithms in the last chapter, namely the value-iteration algorithm and the policy-iteration algorithm. We will study some variations of these algorithms and will state the properties that they require a Markovian decision problem to have. The notion of "ergodicity" will be important for us, because this property of a plan will play an important

role in finding simple algorithms that solve Markovian decision problems. The definitions and results presented in this chapter are well known in the operations research literature. We will only introduce results that are needed in the following chapters and will state them in the specific form that we need, not in their full generality. The proofs will be left out, but can be found in book on dynamic programming, such as [84], [85] or [10].

Markovian decision problems have been studied thoroughly in the operations research literature for over three decades and have become a well-established subfield of operations research. They provide a general framework that can be used to model many dynamic programming problems (for example, optimal stopping problems). The basic algorithm that we will discuss in the following can easily be adapted or extended to more complex decision problems, such as decision problems that involve discounting [54], risk-sensitive decision-makers [57, 64], time-lags , policy restrictions [72], or varying transition time [54]. Markovian decision problems have recently become of interest for the AI community in the context of reinforcement learning. New algorithms have been invented (e.g. q-learning [107, 22]) and the properties of the old algorithms have been investigated in the new context [2].

A Markovian decision problem consists of a finite set of states $\bar{S}$. Associated with each state $s \in \bar{S}$ is a set of alternatives $\bar{A}(s)$. The number of alternatives of each state has to be finite, but the number can differ from state to state, as can the available alternatives. An alternative $a \in \bar{A}(s)$ is a finite set of transition links (s, a, s'), each of which originates in state s. Each transition link (s, a, s') $(s, s' \in \bar{S}, a \in \bar{A}(s))$ is labeled with a probability $\bar{p}(s'|s, a)$ and a real number $\bar{c}(s, a, s')$, the so-called reward. Since an alternative is a probability distribution over the states, the probabilities of its transitions must add up to one. A Markov process is a dynamically evolving sequence of states s_i $(i \in \{1, 2, 3, ...\})$. A policy is a set of instructions that state for each index i of the sequence which alternative $a \in A(s_i)$ to choose depending on the subsequence $s_1, s_2 \ldots, s_i$. (It turns out that only the state s_i is important for making the decision which action to execute.) If $(s_i, a, s) \in a$ then state s_{i+1} will be equal to state s with probability $\bar{p}(s|s_i, a)$, and in this case a reward of $\bar{c}(s_i, a, s)$ will be received. The expected immediate reward $\bar{c}(s, a)$ of an action $a \in \bar{A}(s)$ can then be calculated as $\sum_{s' \in S} \bar{p}(s'|s, a)\bar{c}(s, a, s')$. The gain $g(s)$ of a state s under a given policy is defined to be the expected reward per transition received by an infinite Markov process that is started in s.

The problem is to find a policy (called an optimal policy) that maximizes the gain of every state. (Note that this optimization criterion is different from the optimization criterion of the

planning problem.) The gain of any state is always finite under every policy. The maximal gain of a state is also finite and can be realized under a policy. Furthermore, there exists a policy that maximizes the gains of all states at the same time, i.e. in order to increase the gain of one state it is never necessary to decrease the gain of another state. Thus, the problem is well defined.

A policy f is called stationary iff it assigns every state $s \in \bar{S}$ one alternative $f(s) \in \bar{A}(s)$, the so-called decision for that state. This alternative is to be executed when the Markov process is in state s. A stationary policy uniquely determines a Markov chain. One can show that there always exists a stationary policy that maximizes the gain for every state. Thus, we can restrict our attention to stationary policies. In the following, we will use "stationary policy" and "policy" as synonyms.

An ergodic set is a minimal set of states of a Markov chain such that a Markov process cannot leave this set once it has entered it. Obviously, every Markov chain has at least one ergodic set, and there exist Markov chains that have more than one. A state is called ergodic if it is a member of an ergodic set. An ergodic state is called absorbing, iff it is the only member of an ergodic set. States that are not ergodic are called transient. A Markov chain is called ergodic, iff it has only one ergodic set. (This definition allows an ergodic Markov chain to have transient states. Other definitions in the literature are stricter. They require every state of an ergodic Markov chain to belong to the same ergodic set, which rules out transient states.) Every Markov process of a finite, ergodic Markov chain has the same limiting state probability distribution, no matter in which state the Markov process was started. (The limiting state probability distribution specifies for each state the probability that the Markov process is in that state when it has run for a large number of steps.) A policy is called ergodic, iff the Markov chain that it determines is ergodic. A Markovian decision problem is called completely ergodic iff every possible policy is ergodic.

The value-iteration algorithm can be used to approximate the solutions of arbitrary Markovian decision problems. It can be stated as follows:

1. Set $i := 1$, and for each $s \in \bar{S}$: set $G(s)$ to zero.
2. For each $s \in \bar{S}$: set $f(s) := a : \max_{a \in \bar{A}(s)} (\bar{c}(s,a) + \sum_{s' \in \bar{S}} \bar{p}(s'|s,a)G(s))$. If the old decision $f(s)$ of a state s yields as large a value as any other alternative $a \in \bar{A}(s)$, leave the old decision unchanged.

3. For each $s \in \bar{S}$: simultaneously set $G(s) := \bar{c}(s, f(s)) + \sum_{s' \in \bar{S}} \bar{p}(s'|s, f(s))G(s)$, and $g(s) := G(s)/i$.

4. Determine whether to continue. If so, set $i := i + 1$ and go to step 2.

5. Output f as an approximation of the optimal policy, and stop.

The idea behind the algorithm was sketched in the previous chapter. Here, we leave unspecified when to continue. The g values of the states converge towards their g values under the optimal policy, but the convergence might only be asymptotic. It is not guaranteed that the g values do not decrease from one iteration to the next: In step 2, the G values of the successor states of s can have decreased since the last calculation of $G(s)$. Although the optimal policy is found in finite time, the problem is to detect whether the current policy is optimal. Another problem is that the G values will grow without bounds the more iterations the algorithm is run, and thus will eventually exceed the memory limits of the computer that executes the algorithm. Remedies for this problem exist for special structures of the Markovian decision problem, see for example [10].

In his Sc.D. thesis and subsequent publications, R.A. Howard developed a dynamic programming method for finding an optimal policy for a given arbitrary Markovian decision problem. His multiple-chain policy-iteration algorithm works as follows:

1. Choose an arbitrary policy, i.e. for each $s \in \bar{S}$ choose an $a \in \bar{A}(s)$ and set $f(s) := a$.

2. (value-determination operation; policy-evaluation routine)
   Use the current policy f to solve the double set of equations

$$\begin{aligned} g(s) &= \sum_{s' \in \bar{S}} \bar{p}(s'|s, f(s))g(s') \\ g(s) + v(s) &= \bar{c}(s, f(s)) + \sum_{s' \in \bar{S}} \bar{p}(s'|s, f(s))v(s') \end{aligned}$$

   for all $v(s)$ and $g(s)$ ($s \in \bar{S}$), after the value of one $v(s)$ in each ergodic set has been set to zero.

3. (policy-improvement routine)
   For each state $s \in \bar{S}$, determine the alternative $a \in \bar{A}(s)$ that maximizes

$$\sum_{s' \in \bar{S}} \bar{p}(s'|s, a)g(s')$$

using the current g values determined in the value-determination operation, and set $f(s) := a$. If

$$\sum_{s' \in \bar{S}} \bar{p}(s'|s,a)g(s')$$

is the same for all alternatives, or if several alternatives are equally good according to this test, break the tie and choose the alternative $a \in \bar{A}(s)$ among the equally good alternatives that maximizes

$$\bar{c}(s,a) + \sum_{s' \in \bar{S}} \bar{p}(s'|s,a)v(s')$$

using the current v values determined in the value-determination operation, and set $f(s) := a$. Regardless of whether the policy-improvement test is based on the first or second criterion, if the old decision $f(s)$ in a state s yields as large a value of the test quantity as any other alternative $a \in \bar{A}(s)$, leave the old decision unchanged.

4. If the new policy is different from the previous one, go to step 2, otherwise output f as the optimal policy, and stop.

In each iteration, first the value-determination operation determines the "goodness" of every state s under the current policy. The goodness is measured with two values: the g value (or gain) of s, $g(s)$, and the v value of s, $v(s)$. Remember that the gain of a state s under a given policy is defined to be the average reward received per transition by an infinite Markov process that is started in s. If it holds for two states s and s' that they are in the same ergodic set, then $v(s) - v'(s)$ is the advantage in the expected total reward received by an infinite Markov process that is started in s instead of s'. Both goodness values are calculated at the same time for every state.

The following policy-improvement routine calculates for every alternative of a state the expected gain of the successor states when applying the alternative in the state. The alternative that leads to the largest gain is the new decision for the state. If two alternatives tie, then the alternative is chosen for which the expected v value of the successor states minus its expected execution cost is largest. However, regardless of whether the policy-improvement test is based on the first or second criterion, if the old decision for the state qualifies again, because it yields as large a value of the test quantity as every other alternative, we will keep it and not assign a different action with the same value. This rule assures convergence in the case of equivalent policies.

The multiple-chain policy-iteration algorithm has the following properties: The values $v(s)$ and $g(s)$ are the unique solution of the double set of equations of the value-determination operation. The gain of any state can never decrease from iteration to iteration. If none of the gains could be increased during an iteration, then none of the v values will have been decreased. If in this case all v values remained the same, the algorithm stops. Otherwise it will iterate again. In a finite number of iterations an optimal policy is found and the algorithm stops. For this policy it does not only hold that no policy exists for which the gain of a state is larger, but also that no policy exists for which the gain of a state is the same, but its v value is larger.

In case the Markovian decision problem is completely ergodic, the multiple-chain policy-iteration algorithm collapses to the simple policy-iteration algorithm (which is also called the single-chain policy-iteration algorithm):

1. Choose an arbitrary policy, i.e. for each $s \in \bar{S}$ choose an $a \in \bar{A}(s)$ and set $f(s) := a$.

2. (value-determination operation; policy-evaluation routine)
   Use the current policy f to solve the set of equations

$$g + v(s) = \bar{c}(s, f(s)) + \sum_{s' \in \bar{S}} \bar{p}(s'|s, f(s))v(s')$$

   for g and all $v(s)$ $(s \in \bar{S})$, after the value of one $v(s)$ has been set to zero.

3. (policy-improvement routine)
   For each state s, determine the alternative $a \in \bar{A}(s)$ that maximizes

$$\bar{c}(s, a) + \sum_{s' \in \bar{S}} \bar{p}(s'|s, a)v(s')$$

   using the current v values determined in the value-determination operation, and set $f(s) := a$. If the old decision $f(s)$ in a state s yields as large a value of the test quantity as any other alternative $a \in \bar{A}(s)$, leave the old decision unchanged.

4. If the new policy is different from the previous one, go to step 2, otherwise output f as the optimal policy, and stop.

The simple policy-iteration algorithm works exactly like the multiple-chain one, only the expressions in the value-determination operation and the policy-improvement routine could

be simplified using the assumption that every possible policy is ergodic. (We could utilize that every Markov chain determined by a policy has one ergodic set. Furthermore, it holds that $g(s) = g'(s)$ for every policy and every pair of states s and s', so that we used g instead.) The algorithm has the same properties as the multiple-chain policy-iteration algorithm.

Completely ergodic Markovian decision problems can also be stated as linear programs:

$$\min g$$
$$g + v(s) \geq \bar{c}(s,a) + \sum_{s' \in \bar{S}} p(s'|s,a)v(s') \qquad \forall s \in \bar{S}, a \in \bar{A}(s)$$

in which g is unconstrained in sign. Note that the absolute values of the v values do not matter, only their difference is important.

We are interested in a special type of Markovian decision problems, namely completely ergodic Markovian decision problems with one (absorbing) state $\bar{s}$ that has only one applicable action $\bar{a}$ such that $\bar{p}(\bar{s}|\bar{s},\bar{a}) = 1$ and $\bar{c}(\bar{s},\bar{a}) = 0$. Then, $\{\bar{s}\}$ is an ergodic set, and it is the only one no matter what the chosen policy is, since the Markovian decision problem is completely ergodic. Thus, every possible Markov process will finally become absorbed in $\bar{s}$. From then on, the gain received per transition is zero. Thus, the gain of every state is zero under every policy, i.e. $g = 0$. In this case, the Markovian decision algorithms maximize the v values of the states. If we set $v(\bar{s}) := 0$, then $v(s) - v(\bar{s}) = v(s)$ specifies the advantage in the expected total reward received by an infinite Markov process that is started in s instead of $\bar{s}$. This value is equal to the expected total reward when the Markov process is started in state s, and it is equal to the expected total reward until absorption when the Markov process is started in state s. Since it is guaranteed that every Markov process eventually becomes absorbed, the Markovian decision problem behaves like a discounted, completely ergodic Markovian decision problem. (There the problem is to find a policy that maximizes the total discounted reward, which is defined to be the expected sum of the discounted rewards received.)

This problem can be solved using the simple policy-iteration algorithm, that can be slightly simplified after the substitution $v(\bar{s}) = 0$ and $g = 0$. In this case, the v values are always non-decreasing and at least one can be strictly increased, if the algorithm does not stop after the iteration.

The problem can also be solved using the value-iteration algorithm. If we are interested in

using the value-iteration algorithm to determine the gains of the states for a particular policy f instead of finding the optimal policy, we set $i := 1$, and execute step 3 and 4 of the value-iteration algorithm iteratively (jumping back to step 3). If f is ergodic, then $G_i(s) = ig + v(s)$ holds for large i. ($G_i(s)$ denotes the value of $G(s)$ during the ith iteration.) This allows us to reformulate the value-iteration algorithm:

1. For each $s \in \bar{S}$ set $v(s)$ to zero.
2. For each $s \in \bar{S}$: set $f(s) := a : \max_{a \in \bar{A}(s)} (\bar{c}(s,a) + \sum_{s' \in \bar{S}} \bar{p}(s'|s,a)v(s))$. If the old decision $f(s)$ in a state s yields as large a value as any other alternative $a \in \bar{A}(s)$, leave the old decision unchanged.
3. For each $s \in S$: simultaneously set $v(s) := \bar{c}(s, f(s)) + \sum_{s' \in \bar{S}} \bar{p}(s'|s, f(s))v(s')$.
4. Determine whether to continue. If so, go to step 2.
5. Output f as an approximation of the optimal policy, and stop.

For each state the alternative is chosen that maximizes the expected v value of the successor states minus the expected execution cost. However, if the old decision of the state qualifies again, because it yields as large a value as every other alternative, we will keep it and not assign a different action with the same value. (This rule assures convergence in the case of equivalent policies.) At the same time, this value is used to update the v value of the state.

7. The Transformation of the Planning Problem

In this chapter, we will show how the probabilistic and decision-theoretic planning problem can be transformed into a Markovian decision problem. In the last chapters, we have introduced both problems independently. It became clear, that the states of the search space correspond to the states of the Markovian decision problem, the actions to the alternatives, and a plan to a policy. Thus, we can (and will) use the terms introduced for our planning problem in the context of Markovian decision problems and vice versa. Being able to transform a planning problem into a Markovian decision problem enables us to use the Markovian decision algorithms to solve the planning problem.

We transform a probabilistic and decision-theoretic planning problem into a Markovian decision problem by first creating a state of the Markovian decision problem for every state of the planning problem. For every action that is applicable in a state s of the planning problem and that leads with certain probabilities and costs to its outcomes, we create an alternative that is applicable in the state of the Markovian decision problem that corresponds to s, and that leads with the same probabilities and costs to the states that correspond to the outcomes. Since the Markovian decision algorithms need only the expected cost for executing an action a in a state s and not the costs for each (state, action, outcome) triple, we can collapse the cost information into one value for every action by calculating the expected value $\bar{c}(s,a) := \sum_{s' \in \bar{S}} \bar{p}(s'|s,a)c(s,a,s')$. Additionally, we need to show how to translate the (probability distribution over the) start states, the goal states, and the stop actions:

- The Probability Distribution over the Start States

  In order to model the probability distribution b over the start states, we could introduce a new state $\overline{\overline{s}}$ as a unique start state, and a new virtual action $\overline{\overline{a}}$ (i.e. an action that does not correspond to a physical action performed by the agent, and therefore has an execution cost of zero), for which we define $A(\overline{\overline{s}}) = \{\overline{\overline{a}}\}$, and for all $s' \in S$: $p(s'|\overline{\overline{s}}, \overline{\overline{a}}) = b(s')$ and $c(\overline{\overline{s}}, \overline{\overline{a}}, s') = 0$. Then, the gain of a plan is equal to the gain of $\overline{\overline{s}}$. Likewise, the execution reward of a plan is equal to the v value of $\overline{\overline{s}}$ (if the gain is the same for every start state, otherwise it is undefined). But the notion of a start state is not needed for optimizing a policy: The optimality of a policy does not depend on which state the Markov process is started in, since the gains and (for equals gains) the v values are optimized for every state.

  Thus, we do not introduce a unique start state $\overline{\overline{s}}$, but keep in mind that the gain of a policy is calculated as $\sum_{s \in S} b(s)g(s)$ (i.e. it is 0 for admissible plans), and its v value as $\sum_{s \in S} b(s)\left(v(s) - g(s)\right)$ (i.e. it is $\sum_{s \in S} b(s)v(s)$ for admissible plans).

- The Goal States and the Stop Actions

  The goal states of the planning problem are treated in the same way every other state is treated: They are transformed into states of the Markovian decision problem. Goal states are not different from other states, except that the agent is allowed to stop the execution when it is in a goal state. We model this stopping as executing a stop action. Since it is not required to stop in a goal state, there can be other alternatives as well. Thus, iff $s \in G$, then $stop \in \bar{A}(s)$. Every stop action deterministically leads to the same

new state $\bar{s}$, that represents that the plan execution is stopped (i.e. $\bar{p}(\bar{s}|s, stop) = 1$ for all $s \in G$). Stop actions are virtual actions. Since the agent receives a reward for stopping, we add that reward to the execution cost (which is zero for virtual actions) and get $\bar{c}(s, stop, \bar{s}) = r(s)$. (If the stop actions correspond to physical actions that the agent has to execute in order to stop the execution, we use their execution costs instead of zero.)

Once it has stopped the execution, the agent remains in the state $\bar{s}$ and does no longer receive costs or rewards. Therefore, we make $\bar{s}$ an absorbing state by creating a new virtual action $\bar{a}$, for which we define $A(\bar{s}) = \{\bar{a}\}$, $p(\bar{s}|\bar{s}, \bar{a}) = 1$, and $c(\bar{s}, \bar{a}, \bar{s}) = 0$. Thus, $\{\bar{s}\}$ is an ergodic set.

We can formalize the transformation along the lines outlined above. Given a planning problem, we transform it into the following Markovian decision problem:

1. $\bar{S} := S \cup \{\bar{s}\}$, the non-empty, finite set of states that contains a distinguished element $\bar{s}$; (states that are not reachable from at least one start state can be deleted together with all actions applicable in them);

2. for all $s \in S$: if $s \notin G$ then $\bar{A}(s) := A(s)$, otherwise $\bar{A}(s) := A(s) \cup \{stop\}$, the finite set of actions applicable in state s; $\bar{A}(\bar{s}) := \{\bar{a}\}$;

3. for all $s, s' \in S$ and $a \in A(s)$: $\bar{p}(s'|s, a) := p(s'|s, a)$, the conditional probability that the successor state is s' when action a is executed in state s; if $s \in G$ then $\bar{p}(\bar{s}|s, stop) := 1$ and $\bar{p}(s'|s, stop) := 0$; $\bar{p}(\bar{s}|\bar{s}, \bar{a}) := 1$; $\bar{p}(s'|\bar{s}, \bar{a}) := 0$;

4. for all $s, s' \in S$ and $a \in A(s)$, such that $p(s'|s, a) > 0$: $\bar{c}(s, a, s') := c(s, a, s')$, the execution cost of the transition to state s' when having executed action a in state s; if $s \in G$ then $\bar{c}(s, stop, \bar{s}) := r(s)$; $\bar{c}(\bar{s}, \bar{a}, \bar{s}) := 0$; additionally, we define for all $s \in S$ and $a \in A(s)$: $\bar{c}(s, a) := \sum_{s' \in \bar{S}, p(s'|s,a)>0} p(s'|s, a)c(s, a, s')$, the expected execution cost when having executed action a in state s;

We call Markovian decision problems with this structure (i.e. Markovian decision problems that can be used to solve our probabilistic and decision-theoretic planning problem) pdt (probabilistic and decision-theoretic) Markovian decision problems. (In the following we will often use "Markovian decision problem" in the sense of "pdt Markovian decision problem".)

We depict such problems similar to Markov chains. A state is represented as a circle, and a transition link (s, a, s') as a directed edge from s to s' that is labeled with $\bar{p}(s'|s, a)$ and $c(s, a, s')$ (in this order). The transition links belonging to the same alternative are bundled together. For an example see figure 20.

8. Ergodicity and Admissible Plans

In this chapter, we will explore the properties of plans that can be used to devise an optimizing planner. Since our planning problem can be transformed into a Markovian decision problem, it can be solved using one of the Markovian decision algorithms. Furthermore, since planning problems and pdt Markovian decision problems are different ways of expressing the same problem, properties that hold for Markovian decision problems can be translated into properties that hold for planning problems. (For example, we can conclude that among all optimal plans there will be at least one universal plan.) Most of the following paragraphs informally state lemmas that we will use in later chapters and the reason why it holds.

A pdt Markovian decision problem (as defined in chapter 7) has the property that the execution costs of all actions are negative with the possible exception of the execution costs of stop actions. The stop actions lead with probability 1 to the state $\bar{s}$, which is an absorbing state with a gain of zero. Remember that an ergodic set is a minimal set of states such that a Markov process cannot leave this set once it has entered it. If ergodic sets other than $\{\bar{s}\}$ (and thus not containing $\bar{s}$) exist under some plan, every state contained in them must have a negative gain, since the only transitions that can have a non-negative reward are transitions that belong to stop actions and these transitions can never connect two states of an ergodic set other than $\{\bar{s}\}$.

Every state is either an ergodic state or a transient state. Ergodic states from the same ergodic set have the same gain. The gain of a transient state is the average over the gains of the ergodic sets weighted with the probability distribution over the ergodic sets that determines the probability that a Markov process started in the given transient state will eventually become absorbed by a particular ergodic set. Thus, every state has either a negative gain or a gain of zero.

If a state s has a gain of zero under a plan f, we call s solved by f. In this case, $\bar{s}$ can be reached under f from every state that is reachable from s. If this were not the case, then

there would be a possibility that a Markov process determined by f and started in s became absorbed by an ergodic set unequal to $\{\bar{s}\}$. Thus, the gain of s would be negative. This is a contradiction. Thus, an execution of plan f with a start state s will eventually stop in a goal state.

We call s solvable if a plan f exists that solves s. (In other words, s is solvable iff there exists a plan such that a goal state can be reached under that plan from every state that is reachable from s.) Goal states are always solvable, because a plan that assigns the stop action to a goal state makes $\bar{s}$ the only reachable state from that goal state. The definition of a solvable state only requires that for every such state a plan exists that solves the state. Different states are allowed to be solved by different plans. We will show that there exists at least one plan that simultaneously solves every solvable state. Since Markovian decision algorithms maximize the gain of every state, the gain of a state under the optimal plan found is zero iff that state is solvable. If the state is solvable, there exists a plan that solves it. Under this plan, the gain of the state is zero. Since this is the largest possible gain, the gain of the state under the optimal plan that is found by the Markovian decision algorithms cannot be larger: it will also be zero. Therefore, the state is solved under the optimal plan found as well. On the other hand, if the gain of a state under the optimal plan found is zero, then the state is solved by the optimal plan (per definition). Thus, it is solvable. In conclusion, all solvable states are simultaneously solved under the optimal plan found.

If s is solvable, there exists a plan that solves s. Then, $\bar{s}$ can be reached from every state that is reachable from s under this plan. Since $\bar{s}$ can only be reached by executing a stop action in a goal state, it holds that a goal state can be reached under that plan from every state that is reachable from s (not necessarily the same goal state, though). On the other hand, if there exists a plan f such that a goal state can be reached under that plan from every state that is reachable from s, then $\bar{s}$ is reachable under f from every state that is reachable from s, if we change the actions that are assigned to the goal states to stop actions.

Per definition, a plan is admissible iff $\bar{s}$ can be reached under the plan from every state that is reachable from at least one of the start states. In other words, a plan is admissible iff it solves every start state. Thus, iff every start state can be solved, an admissible plan exists. Furthermore, iff an admissible plan exists, the planning problem is solvable. Thus, in order to determine whether a planning problem is solvable or not, we have to find out whether every start state is solvable or not.

Iff a planning problem contains at least one unsolvable state, none of the plans is ergodic. (Recall that an ergodic plan is a plan that has only one ergodic set.) For every ergodic plan, the execution of that plan with an unsolvable state as start state would inevitably lead to the only ergodic set $\bar{s}$. But then, the plan would solve the unsolvable state. This is a contradiction. On the other hand, if every state were solvable, then there would exist a plan that solved all of them. Thus, this plan would be ergodic. Again, this is a contradiction.

As a simple consequence, every ergodic plan is admissible (no matter what the start states are), but not vice versa. Since $\{\bar{s}\}$ is an ergodic set, it is the only one of an ergodic plan. Thus, an ergodic plan solves every state. Therefore, it also solves the start states and every state that is reachable from them. On the other hand, an admissible plan does not need to be ergodic, since it can contain unsolvable states that cannot be reached from a start state under the plan.

9. A Straight-Forward One-Step Planner

In this chapter, we will show how Markovian decision algorithms can directly be utilized to solve the planning problem by translating the planning problem into a Markovian decision problem and solving it. We call this approach the one-step planner, and will analyze its search behavior.

Unless the planning problem is known to be completely ergodic, the multiple-chain policy-iteration algorithm must be used instead of the simple one. Since we assume that the state space and the transition probabilities of the planning domain are completely known at planning time, the decision whether to use the simple or the multiple-chain policy iteration algorithm can be made at the beginning of the planning phase. Then, the straight-forward approach to solving the planning problem leads to the following one-step planner:

1. Transform the planning problem into a Markovian decision problem (see chapter 7).
2. Choose an initial policy by randomly assigning one of the applicable actions to every state.
3. (planning phase)
   Run the multiple-chain policy-iteration algorithm using the policy determined in step 2 as the initial policy.

4. Stop. If every start state has a gain of zero, then the plan found is an optimal solution of the planning problem, otherwise the planning problem is not solvable.

We know that an iteration of the multiple-chain policy-iteration algorithm never decreases the gain of a state. It also never decreases the v value of a state, unless the gain of some state could be increased. If a state has a gain of zero, it has the same gain again under the new plan, since this is the largest possible gain for a state in a pdt Markovian decision problem. If the old plan is admissible, then it solves the start states. So, they have a gain of zero, which they retain under the new plan. Since the start states are still solved under the new plan, it is admissible as well. Since the execution reward of an admissible plan is a weighted average over the v values of the start states, it is also guaranteed not to decrease.

Since the gain of the start states can not be increased, their v values cannot be decreased. We know that for at least one state either its gain or its v value can be increased (except during the last iteration of the algorithm). But we cannot guarantee that the value of a start state can be changed, unless every state is a start state. Thus, it is not guaranteed that the execution reward of the old plan can be strictly increased. In conclusion, an admissible plan is transformed into a new admissible plan which is at least as good as the old plan, but not necessarily better.

Assume that a given planning problem is solvable. Then, it might take the one-step planner some iterations to find an initial sub-optimal solution, i.e. an admissible plan. (If the planning problem is not solvable, then no admissible plan can be found, i.e. at least one of the start states has always a negative gain.) Thus, the multiple-chain policy-iteration algorithm has a weak anytime property with a set-up time. The length of the set-up time is determined by the number of iterations that are needed to find the first solution of the planning problem. From this time on, all policies produced are admissible and every plan has at least the same execution reward as its predecessor. After the first ergodic policy is found, all following policies will remain ergodic. Finally, the optimal policy is found. (A graphical representation is given in figure 14 of chapter 10.)

To be able to describe the behavior of the parallel search that is performed by the multiple-chain policy-iteration algorithm in *deterministic* domains better, let us introduce a distance measure between the states of the Markov chain: Let the longest distance between two states x and y of the Markov chain be the length of the longest sequence of states minus one such

that the states are all pairwise different, the first state is x, the last state is y, and every state in the sequence is directly reachable from its predecessor. If under the optimal plan the longest distance from a state to a goal state for which the stop action has been selected is n, then the optimal decision for that state will be found by the policy-iteration algorithm in a maximum of n iterations. This implies that the number of iterations is bound by the maximum of the distances between a state and a goal state under the optimal plan which is smaller than the maximum of the distances between a state and a goal state under any plan. Sometimes it is easy to determine this value from the structure of the domain (e.g. for the grid-world, that is introduced in chapter 16). The policy-iteration algorithm degenerates to a backward parallel search that starts from the goal states and proceeds breadth-first until the optimal decision for every state has been found. Since the decision for every state is checked for a possible revision at every point in time, a "closed" node is automatically "reopened" if a better path to a goal state could be found for that node.

10. A Two-Step Planner

In this chapter, we will show that under certain circumstances the simple policy-iteration algorithm can be used for planning, even if the planning problem is not completely ergodic. This is the case, if the initial plan is ergodic. We call this approach the two-step planner, because first an ergodic plan has to be found, which is then optimized using the simple policy-iteration algorithm. We will discuss the motivation behind this two-step approach, and show its differences to the one-step planning approach.

If all unsolvable states are deleted from a planning problem that has a solution (together with all actions applicable in them and all actions having an unsolvable state as outcome), then the new planning problem is still solvable and has the same solutions as the old planning problem. Since the old planning problem is solvable, the start states are solvable and thus will not be deleted. Under an admissible plan, the unsolvable states can never be reached from a start state. Thus, if we ignore the state-action rules of the old plan that refer to unsolvable states, we will get a solution plan of the new problem. On the other hand, if we have a solution plan of the new problem, then it is also a solution plan of the old problem, since the new problem is a restriction of the old one. (To make the plan universal, we can assign every unsolvable state an arbitrary action that is applicable in that state.)

Every state of the new planning problem is solvable. There might still be plans that are not admissible or admissible, but not ergodic. But since the optimal plan found by the Markovian decision algorithms (i.e. by both policy-iteration algorithms and the value-iteration algorithm) solves every solvable state, it will be ergodic and thus admissible. This shows that at least one ergodic plan exists for the new planning problem. It does not imply, however, that the new planning problem is completely ergodic, i.e. that every possible plan is ergodic. So, we cannot blindly apply the simple policy-iteration algorithm to solve the new planning problem.

However, if the simple policy-iteration algorithm is given an ergodic plan as initial plan, it will continue to produce only ergodic plans until it finds the optimal plan, see appendix 2. (The intuitive reason is that the policy-iteration algorithm never decreases the goodness of the initial plan. If a state is solved by the initial plan, it is also solved by every subsequent plan. Thus, if every state is solved by the initial plan (i.e. the initial plan is ergodic), then this property holds for the subsequent plans as well.) On the other hand, the simple policy-iteration algorithm needs an initial plan that is ergodic. Thus, if we are able to find an ergodic plan for a planning problem, we can use it as an initial plan for the simple policy-iteration algorithm and we do not need to use the multiple-chain policy-iteration algorithm.

We showed in chapter 8 that finding an ergodic plan is only possible, if the unsolvable states are deleted. (If the unsolvable states are deleted, only solvable states remain. We have shown that a plan exists that simultaneously solves all solvable states. This plan is ergodic by definition. On the other hand, if a planning problem has at least one unsolvable state, an ergodic plan cannot exist.) Unsolvable states do not pose a problem for the one-step planner, since the multiple-chain policy-iteration algorithm does not require the initial plan, an intermediate plan, or the optimal plan to be ergodic.

The above statements allow us to propose a two-step approach to probabilistic and decision-theoretic planning: During the first planning phase, the planner determines the unsolvable states. If one or more start states are not solvable, the planning problem is not solvable and the planner can stop. Otherwise, it deletes the unsolvable states, which does not change the solution. Then, it finds an ergodic policy for the remaining states. During the second planning phase, the planner uses this policy as initial policy for the simple policy-iteration algorithm, that then finds the optimal plan. We remarked before that certain types of Markovian decision problems can be formulated as linear programming problems. The first

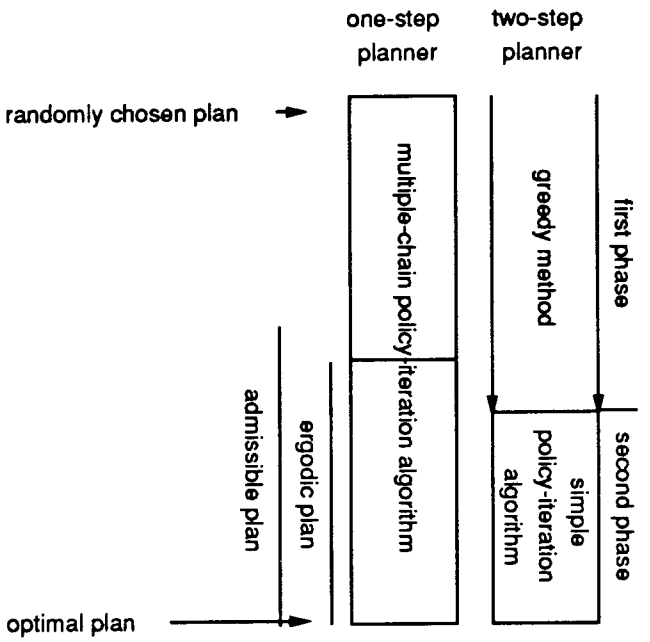


Figure 14: The One-Step and the Two-Step Planner

planning step corresponds to finding an initial basis, the second planning step to optimizing the solution using the Simplex algorithm. For the reference of the reader, figure 14 gives an overview of both the one-step and the two-step planner. The two-step planner is implemented by the following algorithm:

1. Transform the planning problem into a Markovian decision problem (see chapter 7).
2. (first planning phase)
   Find all unsolvable states, delete them from the Markovian decision problem together with all actions applicable in them and all actions having an unsolvable state as outcome, and determine an ergodic policy for the remaining states.
3. If at least one start state was deleted, then stop. The planning problem is not solvable.
4. (second planning phase)
   Run the simple policy-iteration algorithm on the Markovian decision problem that was determined in step 2 using the policy that was determined in the same step as the initial policy.
5. Stop. The plan found is the optimal solution of the original planning problem. (Any applicable action can be assigned to each of the states deleted in step 2.)

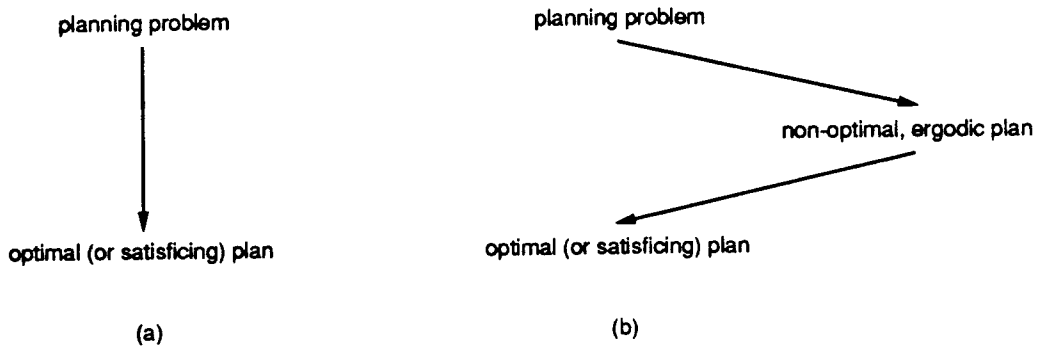


Figure 15: (a) Direct Approach to Planning, (b) Indirect Approach to Planning

The two-step planner differs from the one-step planner most notably in its division into two steps. The first step has the task of finding an ergodic plan that is not necessarily optimal. The second phase then optimizes this plan. Thus, the one-step planner directly tries to find an optimal plan, whereas the two-step planner has the intermediate goal to find a suboptimal, but ergodic plan. This is depicted in figure 15. Note that the intermediate plan has to be more than a (possibly suboptimal) solution, i.e. an admissible plan. It has to solve every solvable state, not only the start states. Thus, the intermediate plan has to be not only a "working" plan, but a plan that is guaranteed to work even if the start states are not known in advance.

The probability distribution b that is used to determine the start states is needed by neither the one-step nor the two-step planner. The planners only need the set of start states in order to be able to determine whether the planning problem is solvable. The numerical values of the probabilities do not matter. The reason is that the Markovian decision algorithms optimize the gain and v value of every state, regardless of whether the state is a start state or not. Only if we want to calculate the total execution reward of a plan, we need the exact probabilities b. (The start states are also needed to construct the relevant part of the state space by applying STRIPS rules to them and repeatedly to every state obtained in this way until the state space can no longer be expanded.)

Traditional planners usually try to find a good (i.e. satisficing or optimal) plan directly. When traditional planners take the approach to find an arbitrary suboptimal solution first, the reason usually is that directly finding a good plan seems to be difficult or even impossible. In those cases, first a working plan is constructed, that is later "postoptimized" (for example by peephole optimization techniques). Since local optimization techniques are only able to

fine-tune a plan and can not change the overall structure of the plan (i.e. the general approach taken to reach a goal), such techniques will usually not be able to transform the initial plan into one that approximates an optimal plan unless the initial plan is already close to optimal. The two-step planner, on the other hand, can totally change a plan from one iteration to the next. For example, it is possible that all states that were reachable from a start state before an iteration (except for the start states themselves) are no longer reachable afterwards. The reason is that during an iteration the decisions of *all* states are checked for revision, not only the decisions of states that are reachable from a start state under the current plan. If this were not the case, then a state that is not reachable under the current plan would not be updated. If such a state had a highly negative v value, it would remain undesirable (although it might have a better v value under a different plan) and actions leading to such a state would probably be never considered. This is the problem with traditional post-optimization techniques.

The first planning phase of the two-step planner is qualitative, whereas the second one is quantitative. The reason is that the planner is not concerned with optimization in the first planning phase. Thus, costs, rewards and the numerical values of the transition probabilities are unimportant, although the planner has to take the uncertainty of the action outcomes into account. But this can be done by restricting the value of a transition probability $\tilde{p}(s'|s,a)$ to the information whether the probability is positive (indicating that the state s' is a possible outcome when executing action a in state s) or zero (indicating that the state s' is not a possible outcome when executing action a in state s). Thus, arithmetic can be replaced by symbolic reasoning about the reachability relation between states, and AND-OR search methods, graph-theoretic search methods or deterministic planning methods can be utilized to solve the first planning step.

The two-step planning approach mediates between the two camps of AI researchers in probabilistic planning: The ones that determine plans using deterministic planning methods, evaluate them by taking the probabilities into account, and then reject them if they are not good enough, and the researchers that directly utilize the probabilities for finding good plans. The former approach has a poor performance if there are a lot of plans with very different quality, the latter approach has a complexity problem. Our approach clearly belongs to the second camp, but it shows that purely symbolic reasoning and reasoning that resembles planning in deterministic domains can be part of solving probabilistic planning problems.

In the following two chapters, we will discuss the properties of the first and the second planning phase of the two-step planner in more detail. The second planning phase uses the policy-iteration method and thus is similar to the one-step planner. Therefore, we will explain the second planning phase of the two-step planner before its first planning phase.

11. The Second Planning Phase

In this chapter, we will compare the second step of the two-step planner with the one-step planner, i.e. the simple policy-iteration algorithm with the multiple-chain one.

If both the simple policy-iteration algorithm and the multiple-chain policy-iteration algorithm are given the same initial ergodic policy, they behave the same, i.e. they produce the same policy after each iteration. Thus, both of them need the same number of iterations to find the optimal policy, and the second step of the two-step planner resembles the behavior of the one-step planner. Thus, we can immediately conclude that the second planning phase of the two-step planner has the weak anytime property and will eventually find the optimal plan. Since it takes the first planning phase some time to produce an initial (suboptimal) ergodic policy, the two-step planner is an anytime algorithm with start-up time.

But the simple policy-iteration algorithm is simpler than the multiple-chain one: The multiple-chain algorithm has some overhead, because it has to identify the different ergodic sets in the value-determination procedure. The single-chain algorithm assumes that there is only one ergodic set. Furthermore, all states that were deleted in the first phase of the two-step planner do not need to be considered during the second phase, which also speeds up the computation time of one iteration. This gain in speed is the reason why we prefer the simple policy-iteration algorithm over the multiple-chain one.

12. The First Planning Phase

In this chapter, we will discuss different methods for implementing the first planning phase of the two-step planner, and we will investigate which structural properties of the planning domain favor a small time complexity of the first planning phase.

It is misleading to compare only the speed of the second phase of the two-step planner with

the speed of the one-step planner. A major difference between the planners is that the two-step planner transfers some work from the Markovian decision algorithm (i.e. its second phase) to its first phase. Thus, we expect the second phase to need fewer iterations than the one-step planner. Another difference between the planners is that the two-step planner already detects the unsolvability of the given planning problem after the first phase, whereas the one-step planner will notice it only at the very end. Then, the advantage of the two-step planner over the one-step planner depends on the planning time and the other resources that are needed by its first phase. For example, it depends on the speed of the first planning phase whether the one-step planner or the two-step planner finds a (suboptimal) solution first. Also, it is important how good the plan is that the first phase produces. This depends not only on the algorithm that is used to implement the first planning phase, but also on the structure of the planning domain, and thus needs to be empirically investigated. For example, we have already seen that the implementation of the first planning phase is trivial for purely decision-theoretic planning problems (i.e. domains in which every state is a goal state).

Three of the following methods that implement the first planning phase of the two-step planner are applicable only for domains with special structural properties, one is a general method. We suggest to use always the fastest algorithm that can be used to solve the problem, i.e. to prefer the domain-specific and thus faster algorithms over the general algorithm. We suppose that such algorithms can be developed for a broad range of planning domains. Ordered by increasing complexity, the domain characteristics needed for the methods discussed in the following are: First, the planning domain is completely ergodic, or at least one action of every non-goal state has at least one goal state as outcome. Second, domain specific heuristics can be utilized to accomplish the task of the first planning phase. Third, the task can be solved by restricting the planning problem to its (quasi-)deterministic actions.

- Completely Ergodic Domains, Purely Decision-Theoretic Domains, Leaking Actions

  It is helpful for finding all unsolvable states and determining an ergodic plan for the remaining states, if the planning domain is completely ergodic or if at least one action of every non-goal state has at least one outcome that is a goal state.

  If a planning problem is completely ergodic, then every plan is ergodic and every state is solved under any plan. Thus, no state has to be deleted, and the first planning phase degenerates to the assignment of an (arbitrary) applicable action to every state.

Unfortunately, planning problems are usually not completely ergodic. For example, if at least one of the actions in a deterministic domain is reversible, a second ergodic set can be created by assigning this action to a state in which it is applicable and its counteraction to the successor state. Another example is a planning domain in which the agent is never forced to stop in a goal state, because every goal state allows it to execute at least one non-stop action. Then, a plan exists that assigns the stop action to no state. This plan cannot be ergodic. A third example is a planning domain that has unsolvable states and therefore only non-ergodic plans.

Some weak criteria exist in the operations research literature that can be utilized to guarantee that a planning domain has almost the properties of completely ergodic planning domains (see for example [51]). These criteria usually utilize the "leakage" that is present in the domain. A (state, action) pair leaks, iff one of its action outcomes is a goal state. (The term "leakage" refers to the assumption that for most actions the probability to reach a goal state is extremely small, but positive.) For example, if every non-goal state has a leaking action, then one can construct an ergodic plan by assigning every goal state the stop action and every other state its leaking action. Unfortunately, most domains do not satisfy this criterion, with one exception: purely decision-theoretic planning domains, i.e. domains in which every state is a goal state. In this case, none of the states has to be excluded from the planning problem, and an ergodic policy can be obtained by assigning the stop action to every state.

- Domain-Specific Heuristics

  Domain-specific knowledge can be helpful for finding all unsolvable states and determining an ergodic plan for the remaining states. We will show how to utilize domain-specific knowledge to find an initial ergodic plan for a deterministic planning domain. Deterministic domains can be solved by traditional AI search methods. Nevertheless, we chose a deterministic domain as example domain, because probabilistic domains have not yet been carefully analyzed in the AI literature and the type of domain-specific knowledge that is useful for solving them is not well understood. Thus, it is easier to see how domain knowledge can be used for deterministic domains. But domain knowledge can be utilized for probabilistic domains as well, for example in travel planning.

  The classical blocks-world domain is a deterministic domain in which every state can be reached from every other state. Thus, every state is solvable and no state has to be excluded from the planning problem, no matter what the goal states are. We will show

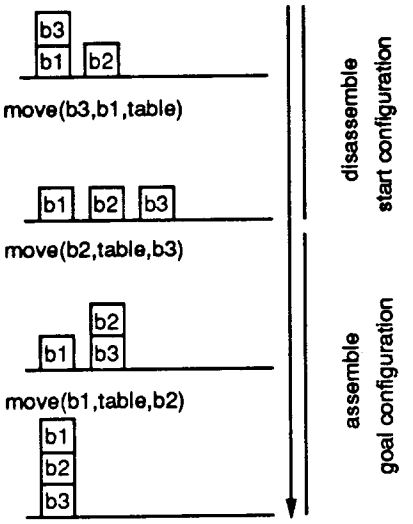


Figure 16: Optimally Solving the Sussman Anomaly

how to augment a simple, domain-specific planner in order to find an ergodic policy.

The classical blocks-world planning problem is to find a plan that leads from a given block configuration (the start state) to another given block configuration (the goal state). It is easy to find a plan for this planning problem: One can first disassemble the start configuration by placing every block on the table (disassembly phase), and then assemble the goal configuration bottom up (assembly phase). The domain-specific knowledge utilized in this case is, first, that one can immediately pick up every block if there are no stacks, and second, that stacks should be erected bottom up. This heuristic determines which of the goal conjuncts should be achieved next, and the planner does not need to backtrack: It starts with the start state and finds the action to apply by analyzing the structure of the state (i.e. it does not need to look ahead). Then, it determines the successor state and repeats this procedure until the goal state is reached. This algorithm, more formally stated below, always finds an admissible plan (if the goal configuration does not contain blocks that are not present in the start configuration). Figure 16 shows how the algorithm optimally solves the Sussman anomaly in 3 steps. The plan found is not guaranteed to be optimal, though. For example, figure 17 shows a planning problem that is solved by the algorithm in 3 steps. Here, block $b2$ could be placed directly on top of block $b3$ (instead of first putting it on the table), thereby reducing the number of actions to 2, the minimal plan length.

1. Test whether the descriptions of the start state and the goal state are valid con-

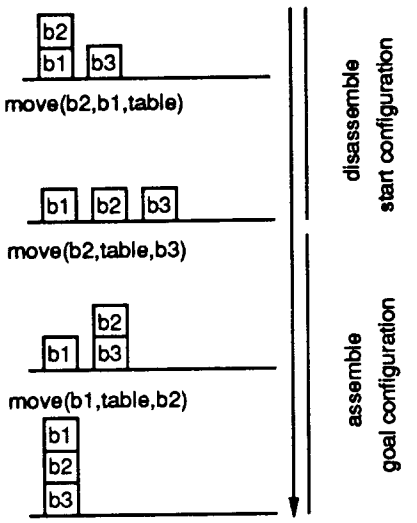


Figure 17: Suboptimally Solving a Planning Problem

figurations of the classical blocks-world problem. (For example, test that no block supports itself or more than one other block. This is possible without search by examining the structure of the state.) If not, stop with an error message.

2. Set the current state to the start state.
3. (disassembly phase: put every block onto the table)
   If the predicates $CLEAR(X)$, $ON(X,Y)$, $UNEQUAL(Y,table)$ match the current state, update the current state by unstacking X from Y, and record the unstacking action for the old state. Go to step 3.
4. (assembly phase: build the stacks of the goal configuration bottom up)
   If the current state does not equal the goal state, match the predicates $ON(X,Y)$, $ON(Y,Z)$ against the goal state, and $CLEAR(Y)$, $ON(Y,Z)$ against the current state, update the current state by stacking X on Y, and record the stacking action for the old state. Go to step 4.
5. Record the stop action for the goal state, output the actions in the order they were recorded, and stop.

This algorithm exhibits three deficiencies when it is used for finding an ergodic policy for the blocks-world problem:

- One Goal State

First, it assumes that the planning problem has only one goal state. If the goal state is partially described, we can pick an arbitrary configuration that satisfies the goal description and use it as the only goal. This works, because in the blocks-world every state is reachable from every other state. Thus, every goal state but one can be deleted from the planning problem.

- Non-Stationary Plans

  Second, the algorithm has to find a stationary plan, but it can erroneously assign two different actions to the same state. When a configuration is encountered during the disassembly phase, a block is unstacked. However, if the same configuration is encountered during the assembly phase, a block is stacked, or (if the configuration is the goal state) the stop action is executed. Thus, the plan is not stationary.

  This problem can be remedied in the following way: If the same state appears more than once in the action sequence that leads from the start state to the goal state, then the first and last occurrence of that state are merged and the intermediate states between them are skipped. Figure 18 shows an example. This plan transformation (or plan repair) is a form of post-optimization. There can be more than one way to repair a non-stationary plan depending on the order in which duplicate states are merged, and the resulting patched plans can have different lengths. Since this cannot happen for the blocks-world, we give a domain-independent example in figure 19: Each letter represents a state, and the sequence of letters represents the sequence of states that are encountered when executing a deterministic plan that leads from state A to state E. If the states B are merged before the states C, sequence a will result, otherwise sequence b will result. Sequence b is shorter than sequence a. However, the different lengths are no problem, since the initial plan that the first planning phase has to find does no need to be optimal. Thus, the states can be merged in any order.

  We do not need to patch plans for the blocks-world explicitly. Instead, we can use the following algorithm: First, the stop action is assigned to one of the goal states (or to every goal state, it does not matter). Second, every goal state is iteratively disassembled until every block is on the table: We test whether the first goal state is the configuration in which every block is on the table. If so, we are done with this goal state and repeat the procedure for the next goal state, if there is one left. If not, we choose a stacked block that is clear and unstack it. If the new state

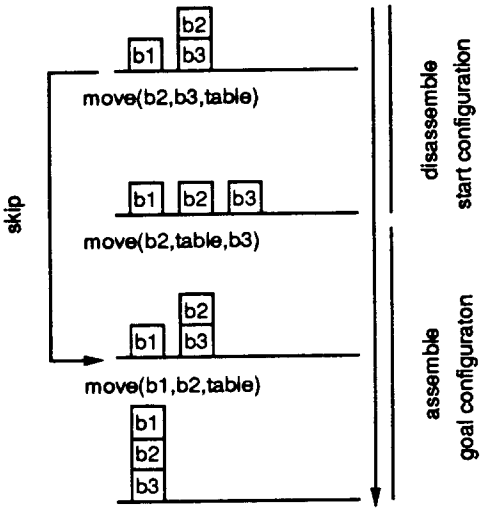


Figure 18: An Example for a Non-Stationary Policy

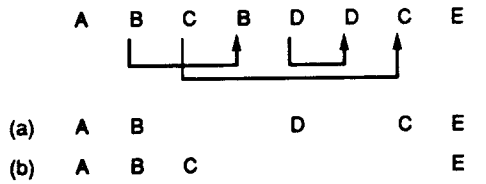


Figure 19: Patching Plans

has already an action assigned, we are done with this goal state and repeat the procedure for the next goal state. If it does not, we assign the new state the reverse of the unstacking action that we executed last (i.e. the action that neutralizes the unstacking) and repeat the procedure for the new state. Third, every state that has still not been assigned an action gets an action assigned that unstacks one of the blocks. This algorithm produces stationary plans for the blocks-world.

- Not Every State is Assigned an Action

  Third, the algorithm has to find a plan that solves every state, not only a given start state. We can run the algorithm for every state as start state. When the planner reaches a state that it has already assigned an action, it can stop planning for that state, since this state (and every of its potential successor states) has already been solved.

Thus, all of the three deficiencies can easily be remedied.

- Restriction to Deterministic or Quasi-Deterministic Actions

  In order to find all unsolvable states and determine an ergodic plan for the remaining states, one can try to restrict the planning problem to its deterministic (or quasi-deterministic) actions.

  If one can assign actions to the states of the planning problem such that the number of actions executed until a goal state is reached is bounded from above by a constant, then we can construct an ergodic plan as follows: We assign every goal state the stop action and every non-goal state an action such that the above property holds [67].

  One way to utilize this knowledge is to restrict the actions of a probabilistic domain to its deterministic actions, i.e. actions that have only one outcome, or its quasi-deterministic actions, i.e. actions whose effects under the optimal plan can be modeled using deterministic actions. After the other actions are removed from the description of the planning problem, the new planning problem is solved. Since the new problem is deterministic, one can use a traditional AI search method to find an admissible policy for the remaining states. Such a policy is also a solution of the original planning problem.

  A domain for which every solvable state is also solvable if the domain is restricted to its deterministic and quasi-deterministic actions is called almost deterministic. (The reserve implication always holds: If a state is solvable for the restricted problem, it is also solvable for the original domain.) Thus, it holds for almost deterministic domains

that if no admissible plan can be found for the restricted domain, the original planning problem is unsolvable as well.

The notion of a quasi-deterministic action is explained in figure 20. The part of a Markov chain that is shown in (a) represents the assignment of a non-deterministic action to a state. With probability $p_1 \neq 0$ and cost c_1 the action succeeds. With probability p_2 ($p_1 + p_2 = 1$) and cost c_2 the action fails, i.e. does not change the state. This action is quasi-deterministic, because we can construct a corresponding deterministic Markov chain as shown in (b): We know that for Markovian decision problems one of the optimal plans is stationary. If the agent executes the action under such a plan and it fails, the agent will remain in the same state. Thus, it will try the action again. In other words, it will try the action repeatedly until it eventually succeeds. Using this knowledge, we can change the structure of the Markov chain in a way that the old one and the new one are equivalent for the purpose of finding an optimal plan. If one repeatedly executes the action until it succeeds, one will eventually succeed with probability $p = 1$ and cost $c = c_1 + \frac{p_2}{1-p_2}c_2$. ($\frac{p_2}{1-p_2}$ is the average number of failures before the execution succeeds.)

Other ways for avoiding nondeterminism by transforming the actions of a planning problem can easily be imagined. For example, in (c) of figure 20, state D is deterministically reachable from states A, B, and C. Thus, this Markov chain can be transformed to the one shown in (d). Note that the problem of finding such transformations is just a local version (i.e. a version that involves only a limited number of states) of the overall task of the first planning phase. The task of the first planning phase is to transform actions into deterministic ones, namely to determine the states from which $\bar{s}$ can be reached deterministically.

For example, the augmented blocks-world has the following deterministic or quasi-deterministic actions:

- $MOVE(BlockA, table, BlockB)$
  ("stacking a block")
- $MOVE(BlockA, BlockB, table)$
  ("unstacking a block")
- $COLOR(BlockA, ColorA)$ if $ON(BlockA, table)$
  ("coloring a block that is on the table")

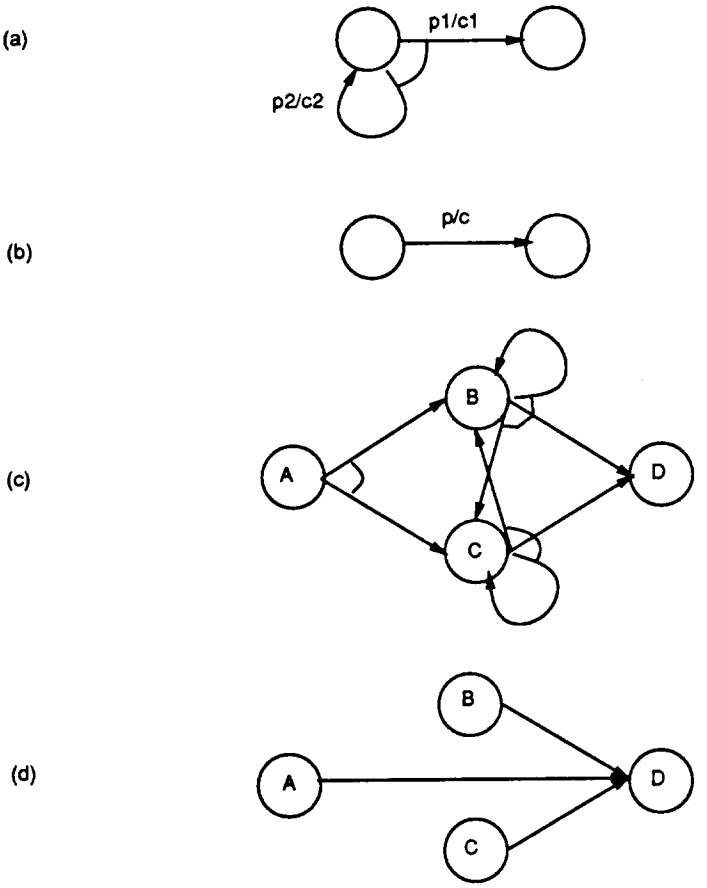


Figure 20: Quasi-Deterministic Actions

- $COLOR(BlockA, ColorA)$ if
  $ON(BlockA, BlockB)$ and $COLORED(BlockB, ColorA)$
  ("coloring a block with the color of the block that it is supported by")

The last three actions are deterministic, whereas the first one is not, since stacking a block might fail. Stacking a block is a quasi-deterministic action, because the configuration of blocks on the table is not changed, if the stacking fails.

The restricted version of the augmented blocks-world problem is created by removing every action except the deterministic and quasi-deterministic ones from the domain specification of the augmented blocks-world problem. For example, all actions that move a stacked block onto another block have to be deleted.

In the augmented blocks-world every state can be reached from every other state. Thus, every state is solvable. The same holds for its restricted version, since we can always disassemble a configuration until every block is on the table, then paint the blocks (if needed), and finally assemble the goal configuration. Thus, if one can reach one state from another state in the augmented blocks-world domain, so can one in its restricted version.

- A General Method

In order to find all unsolvable states and determine an ergodic plan for the remaining states, one can use a general greedy algorithm that works for every pdt Markovian decision problem. Basically, this algorithm is a breadth-first search algorithm that iteratively eliminates all unsolvable states and assigns an action to every remaining state. It is similar to an AND-OR search method.

An AND-OR search [79] determines the nodes of an AND-OR graph that can be proved (i.e. that are definitely true) and the ones that can be disproved (i.e. that are definitely false). Nodes that can neither be proved or disproved are usually assumed to be false. A terminal node is proved if it is true. It is disproved if it is false. An AND node is proved, if all of its successors are proved. It is disproved, if one of its successors is disproved. An OR node is proved, if one of its successors is proved. It is disproved, if all of its successors are disproved.

We can transform the problem of finding an initial ergodic policy to a search problem on an AND-OR graph with a special topology. For each state $s \in S$ there is one OR node, named n_s. For each state $s \in S$ and each action $a \in \bar{A}(s)$, there is one AND

node, named $n_{s,a}$. The only terminal node is $n_{\bar{s}}$ and it is labeled true. The set of successor nodes for an OR node n_s is $\{n_{s,a}|a \in \bar{A}(s)\}$, and for an AND node $n_{s,a}$ it is $\{n_{s'}|\bar{p}(s'|s,a) > 0\}$. Thus, a syntactic element consists of one OR node (the planner makes the decision which action to choose in a state) and its successor nodes, that are AND nodes (nature chooses the outcome of the chosen action). This structure is shown in figure 21. The AND-OR graph is a non-numerical version of a cyclic decision graph, as shown for example in figure 9. Thus, it exactly resembles the structure of a cyclic decision graph: AND nodes correspond to probability nodes, and OR nodes to decision nodes.

Every OR node that can be proved corresponds to a solvable state. The proof shows which of its successors can be proved, i.e. which action to assign to that state: If $n_{s,a}$ can be proved, then we always execute action a in state s. (If several successors can be proved, we choose an arbitrary one.) However, the implication cannot be reversed: It is not true that every solvable state has a corresponding OR node that can be proved. The reason is that the semantics of the planning problem differs in a crucial point from the semantics of the AND-OR search problem. An AND node in the planning problem can in be labeled true in some cases in which an AND node in the AND-OR graph cannot. These cases involve recursive pointers. For example, figure 22 shows a quasi-deterministic action a that leads from state s to a goal state s'. State s is solvable, but cannot be proved: We get for state s' an equation of $n_{s',stop} = n_{s'} = \mathit{true}$ and for state s an equation of $n_{s,a} = n_s \wedge n_{s'}, n_s = n_{s,a}$, which has two possible solutions: $n_{s,a} = n_s = \mathit{true}$ and $n_{s,a} = n_s = \mathit{false}$. Since the solution is not unique, s cannot be proved.

To account for the different semantics of the planning problem and the AND-OR search, we use the following greedy algorithm. Given a planning problem, this algorithm outputs the set of solvable states and an ergodic plan for these states (that is not guaranteed to be optimal):

1. Set $X := \bar{S}$, and $f(\bar{s}) := \bar{a}$.
2. Set $X' := \{\bar{s}\}$.
3. Try to find an $s \in X \setminus X'$ and an $a \in \bar{A}(s)$ such that $\exists s' \in X' : \bar{p}(s'|s,a) > 0$, and $\forall s' \in \bar{S} \setminus X : \bar{p}(s'|s,a) = 0$.
4. If such an s and a could be found then set $X' := X' \cup \{s\}$, set $f(s) := a$ and go to

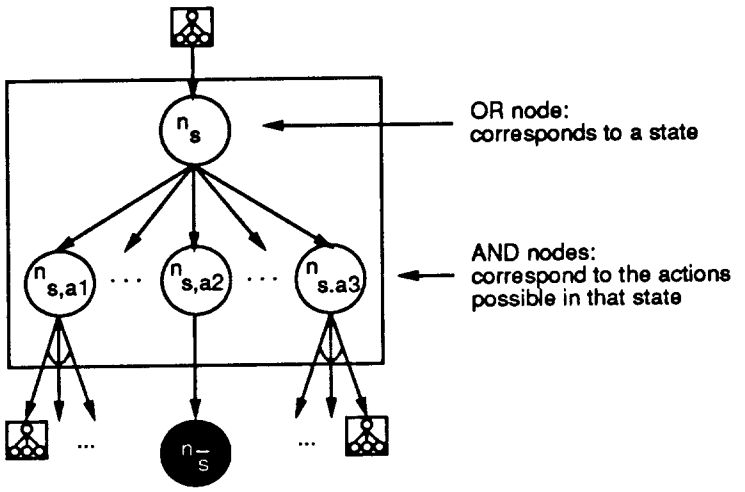


Figure 21: An AND-OR Search Graph for an Acyclic Planning Problem

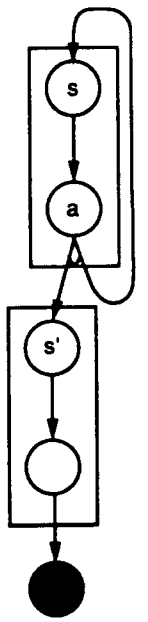


Figure 22: An Example for the Difference between Planning and AND-OR Search

3.

5. If $X \neq X'$ then set $X := X'$, and go to 2.

6. Output X, output $f(s)$ for all $s \in X$, and stop.

At the beginning of an iteration of the outer loop, $\bar{S} \setminus X$ contains the states that have already been found to be unsolvable. Thus, X contains the states that might still turn out to be solvable. X' is set to $\{\bar{s}\}$. Then, the algorithm iteratively collects states in X' in which an action is applicable that has at least one outcome in X' but none in $\bar{S} \setminus X$. This collection is continued until X' cannot be expanded further. $X \setminus X'$ contains the states that have been found to be unsolvable during the iteration of the inner loop. If this set is empty, the algorithm stops, otherwise the outer loop is iterated again. This re-iteration is necessary, because every state that has been assigned an action that has a state in $X \setminus X'$ as outcome has to get a different action assigned. (We could easily support the localization of such states by doing some bookkeeping. This would make the algorithm more efficient, but also more space consuming.) After the algorithm stops, it has collected all solvable states in X.

A proof of the correctness of the algorithm is given in appendix 4. The idea behind the proof is that there must be at least one state from which no goal state can be reached if the planning problem has unsolvable states. This state is unsolvable and so is every state for which every of its applicable actions has at least one unsolvable outcome.

Figure 23 illustrates for an abstract example how the algorithm works. We do not use the blocks-world, since every state in the blocks-world is solvable. Thus, we could not demonstrate how the algorithm deals with unsolvable states. Although the algorithm works directly on sets of states, we visualize the process using the AND-OR graph representation. The planning problem is shown in (a). The next pictures show the intermediate results after each iteration of the algorithm. Finally, (f) shows the resulting ergodic plan after the unsolvable states have been deleted.

An OR node is darkened iff the corresponding state is in X'. An AND node is darkened iff it corresponds to an action that could be assigned to the state corresponding to its predecessor. The AND node that corresponds to the action that is actually assigned to this state is marked with a white cross. (The node that corresponds to the state $\bar{s}$ is always darkened.) Boxes that are shaded gray cover OR nodes that correspond to states in $\bar{S} \setminus X'$ and their immediate successor nodes.

Before the algorithm is started, no box is shaded gray. At the beginning of an iteration of the outer loop, every node is unmarked, except for the terminal node, that is marked. Boxes shaded gray stay shaded. Then, the nodes are colored according to the following rules until no new node can be colored any longer: An AND node can be colored, iff at least one of its immediate successor nodes is colored and none of its immediate successor nodes is in a gray box. An OR node can be colored, iff at least one of its immediate successor nodes is colored. One of these successor nodes is chosen and marked with a white cross. After no node can be colored any longer, every box that contains an uncolored OR node is shaded gray. If at least one box could be shaded, the outer loop is iterated again. Otherwise, the algorithm stops. Every OR node that is covered by a gray box is unsolvable. Every other OR node is solvable. The AND nodes that are marked with a white cross correspond to the actions that determine an ergodic policy for the solvable states.

The example demonstrates that sometimes the chosen action for a state has to be changed, although the state remains potentially solvable, see state 1 in (d). It also demonstrates why several iterations might be required to find every unsolvable state. The substructure composed of the boxes 3, 4, 5, 6, and 7 is an example. During the first iteration, the boxes 5 and 6 can be eliminated. In the following iterations, only one box can be eliminated per iteration. However, if every state of the planning problem is solvable, then the algorithm stops after the first iteration.

13. On-line Planning and Reinforcement Learning

In this chapter, we will show the advantages that an anytime algorithm has both for off-line and on-line planning. On-line planning can be done by interleaving the second planning phase of the two-step planner with plan execution. We will discuss this approach in greater detail and show the assumptions under which the planner can still guarantee that the plan execution eventually stops, or, in other words, that the plan is always ergodic.

We have seen that both the one-step and the two-step planner have the anytime property. This has the following advantages for off-line planning: A plan is available, if unexpected external events require the planner to deliver a plan immediately and it has not found the optimal plan yet. The planner could also trade off planning time and the quality of the

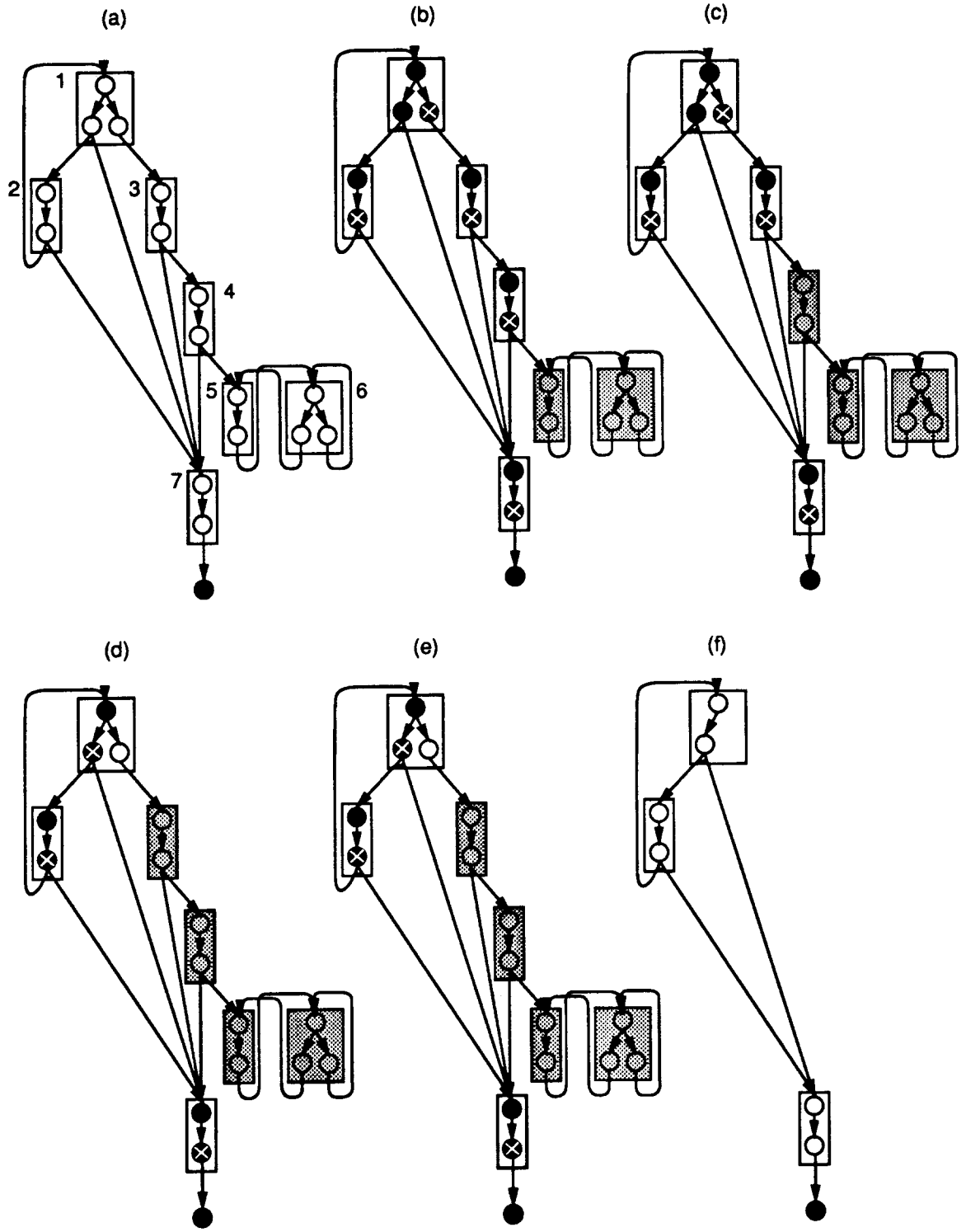


Figure 23: An Example Illustrating the General Algorithm for the First Planning Phase

resulting plan by stopping to plan before the optimal plan is found. Since during each iteration of the policy-iteration algorithm the goodness measure of the current plan (i.e. its total execution reward) is computed, the planner could use decision theory to optimize this trade-off according to its utility function.

We can also use the anytime property of the Markovian decision algorithms to interleave the second planning phase of the two step planner with plan execution and thus to transform the off-line planning approach to an on-line planning approach. We start the execution with the solution found by the first planning phase and then incrementally improve the plan during its execution. This has the following advantages: The plan execution can be started early, since the agent does not need to wait until the optimal plan is found. If the agent by chance reaches a goal state with a large reward early during execution, it can stop not only to execute the plan, but also to refine the plan. (This assumes that planning is primarily used to increase the one-time performance of the plan execution. Then, the plan execution can stop before the optimal plan is found.) Then, some planning effort has been saved. If the agent learns the transition probabilities, execution costs, and rewards during execution, we can drop the assumption that these values are known ahead of execution time or that these values do not change over time. In this case, the on-line planning approach becomes a reinforcement learning approach. (If the planning domain is not static, we can no longer guarantee that the plan converges to the optimal one, of course. For problems that arise when combining planning and reinforcement learning see for example [4, 112]).

The restriction imposed on our original off-line planning task applies for on-line planning as well: We want to be able to guarantee that the agent eventually stops in a goal state. Thus, we restrict the plans that the agent is allowed to execute to ergodic plans. The first planning phase of the two-step planner finds the initial ergodic plan. Therefore, it still has to be run off-line. The second planning phase only produces ergodic plans and thus can be arbitrarily interleaved or run in parallel with the plan execution. The decision whether to refine the plan or to execute a step of the plan could be made with decision-theoretic methods. If the current plan is known to be optimal, there is no need to try to improve the plan and thus the refinement step could be disabled.

Thus, the two-step planning algorithm is changed in the following way:

1. Transform the planning problem into a Markovian decision problem.

2. (first planning phase)
   Find all unsolvable states, delete these states from the Markovian decision problem together with all actions applicable in them and all actions having an unsolvable state as outcome, and determine an ergodic policy for the remaining states. This policy is used as an initial policy for the following phase.

3. If at least one start state was deleted, then stop. The planning problem is not solvable.

4. (Start the execution.)

5. (mixed planning and plan execution phase)
   Execute one of the following steps:

   - (step of the plan execution phase)
     Sense the state that the agent is in and execute the action assigned to that state by the current policy. If the action is a stop action, then stop the plan execution successfully.
   - (step of the second planning phase)
     Refine the current policy by executing the simple policy-iteration algorithm on the current policy for one iteration.

6. Go to step 5.

The policy-iteration method, that we advocated to use for the second planning phase, has a disadvantage for on-line planning: It takes a long time to complete one iteration compared to the value-iteration algorithm. During one iteration, the policy-iteration algorithm usually improves the plan more than the value-iteration algorithm. We show this trade-off between the time that is needed to update the plan and its improvement in figure 24. (Note that this figure is for the purpose of illustration only. We do not make any claims about the exact values of the time intervals or the increase in the goodness of the plan. For a first analysis in this direction see [5].) Since steadily improving the plan a little bit leads to better execution results than improving it in large time intervals a lot, we favor in general the value-iteration algorithm for on-line planning. (The trade-off depends on the planning domain as well. So, for certain domains it might be better to use the policy-iteration algorithm, for others to use the value-iteration algorithm.)

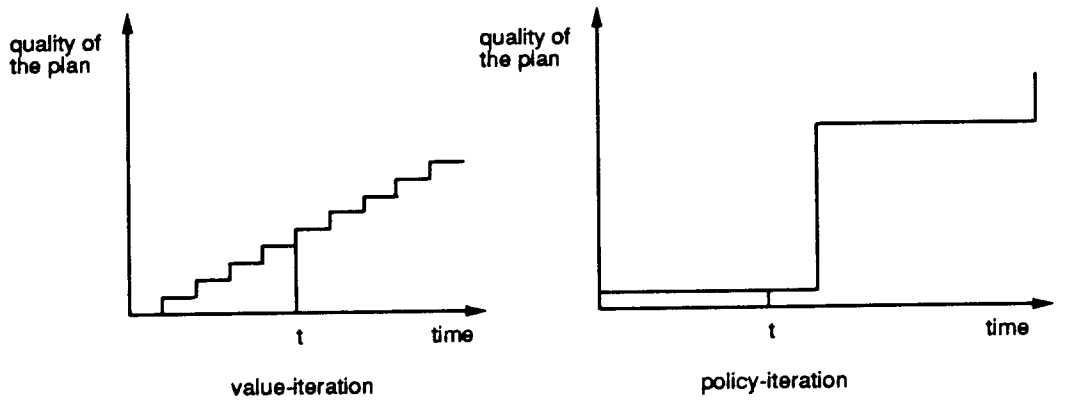


Figure 24: Behavior of the Value-Iteration and the Simple Policy-Iteration Algorithm

When we use the value-iteration algorithm instead of the policy-iteration algorithm, we want to make sure that the planner still produces only ergodic plans and that it still has the anytime property. Since the action assigned to a state is only a byproduct of updating the v value of that state, we need to find initial values (instead of an ergodic plan) that guarantee that the plans produced remain ergodic. Then, we can use the simple version of the value-iteration algorithm, that solves completely ergodic Markovian decision problems with one absorbing state. In appendix 3, we state a sufficient condition for the v values of the states such that the value-iteration algorithm shows this kind of behavior: after all unsolvable states are deleted, the v values for the remaining states $s \in S$ must satisfy that $v(s) \leq \bar{c}(s, f(s)) + \sum_{s' \in \bar{S}} \bar{p}(s'|s, f(s))v(s')$ for some policy f. (To make the inequalities satisfiable, f must be ergodic.) One way to find such values is to find an ergodic policy f (for example by using one of the methods discussed in chapter 12) and afterwards to calculate the v values of the states under this policy (for example by using the value-determination operation once). Then, it holds for all states s that $v(s) = \bar{c}(s, f(s)) + \sum_{s' \in \bar{S}} \bar{p}(s'|s, f(s))v(s')$.

This on-line planning algorithm can also be used as a reinforcement learning system. However, there are two additional problems to consider:

- Known and Static Structure of the Domain

  First, the structure of the domain must be known and remain static. The structure is the information about a domain that is needed by the first planning phase. It only consists of the reachability relation between states and does not include numerical values. So, we assume that a probability that initially was positive remains positive,

and a probability that initially was zero remains zero. Thus, the solvability of states and the ergodicity of plans do not change over time: States are solvable iff the planner assumes that they are solvable, and plans are ergodic iff the planner assumes that they are ergodic. In this case, the initial estimates of the probabilities can be arbitrarily bad and the numerical values of the probabilities can change over time. If the structure of a domain is not static, however, a plan that the planner believes to be ergodic might not be ergodic, and the agent might reach a state during the execution from which it can no longer reach a goal state.

- Experimentation

  Second, we need to include experimentation in the on-line planner. If the agent is not allowed to experiment, i.e. to sample states that are not reachable under the current plan, it might never find the optimal plan. Instead, it will locally optimize its plan. How to optimally integrate experimentation during the solution of a given problem task is still an open research problem in reinforcement learning. Often, it is assumed that there is a separate learning phase before the execution phase in which the successor state reached and the cost incurred is presented to the agent for randomly selected (state, action) pairs. However, if the agent has to experiment itself such an approach is not feasible. Since the domain structure does not change over time, the set of states that can be reached from a start state remains the same. Furthermore, the set of actions applicable in all such states remains constant. These are the actions that the agent has to sample. All of the approaches that have been developed by reinforcement researchers can be integrated into our on-line planner: If we assume that the agent will get a new planning problem after it has solved a given problem, then the agent can execute every action with probability one that it needs to sample (perhaps after several trials to reach the state in which the action is applicable), because this state is reachable from a start state. Since the outcome of that action is a solvable state, the agent can not become trapped by executing the action. (However, one must be careful to avoid that the agent learns a plan that optimizes the execution reward per action instead of the total execution reward, when it *repeatedly* solves planning problems during the learning phase. See chapter 14 for an explanation.)

14. Optimizing the Reward per Unit Time

In this chapter, we will explore a different goodness measure of plans, namely the average execution reward per unit time (i.e. time step). We will discuss when it is appropriate to use the new goodness measure and when it is better to use the old one, namely the total execution reward of a plan. Depending on the goodness measure, the planning problem has to be transformed into different Markovian decision problems. We will discuss the new transformation, and the problems that arise with the new structure of the Markovian decision problem, if one still wants to guarantee that the execution will eventually stop in a goal state.

Until now, we have not cared about the time that it takes to execute an action. Usually, (state, action, outcome) triples will not only have an execution cost associated with them, but also an execution time. We will denote the execution time of the transition to state s' when having executed action a in state s with $t(s,a,s') \in \mathcal{R}^+$ (for all $s,s' \in S$ and $a \in A(s)$ such that $p(s'|s,a) > 0$) in the representation of the planning problem from chapter 3. The stop actions are virtual actions and thus have an execution time of zero. In the pdt Markovian decision problem to which the planning problem is translated, we get for all $s,s' \in S$ and $a \in A(s)$, such that $p(s'|s,a) > 0$: $\bar{t}(s,a,s') := t(s,a,s')$, the execution time of the transition to state s' when having executed action a in state s. If $s \in G$ then $\bar{t}(s,stop,\bar{s}) := 0$. $\bar{t}(\bar{s},\bar{a},\bar{s}) := 1$. Additionally, we define for all $s \in S$ and $a \in A(s)$: $\bar{t}(s,a) := \sum_{s' \in \bar{S}, p(s'|s,a)>0} p(s'|s,a)t(s,a,s')$, the expected execution time when having executed action a in state s.

Actions that have an expected execution time of zero can be eliminated. For example, the stop actions could be eliminated from the planning problem from chapter 3 in the following way: Every action $a \in A(s)$ for which one outcome is a goal state s' is duplicated. In its copy, the transition (s,a,s') is replaced by the transition $(s,a,\bar{s})$, that has an execution cost of $\bar{c}(s,a,s') + r(s')$ and an execution time of $\bar{t}(s,a,s')$. If such an action has n goal states as outcomes (in which the planner is not forced to stop), it will eventually be split into 2^n different actions. Thus, the advantage of not eliminating actions that have an execution time of zero is that the model is not blown up and information does not have to be repeated. Thus, the model is potentially smaller and easier to understand.

For the planning problem from chapter 3, it is rational to maximize the expected total reward of the plan execution, since we assumed that time was an unlimited resource. A different

situation arises if a plan is to be executed over and over again: After the agent stopped the execution in a goal state, it is immediately started again in a start state selected according to the distribution over the start states. In this case, it is rational to maximize the expected reward of the plan execution per unit time, because time no longer is an unlimited resource. (Note that the goal here is different from the goal to minimize the execution time. In the latter case, we could set the cost of each (state, action, outcome) triple to the negative of its execution time and the rewards of the goal states to zero. Then, we could use the planning problem from chapter 3 to maximize the total execution reward.)

As an example for the difference between maximizing the total reward and maximizing the reward per unit time assume that there are two different plans that can achieve the only goal, that has a reward of 10: One takes 2 hours to execute and costs 6 dollars, the other one takes 6 hours to execute and costs 4 dollars. If time is an unlimited resource, we prefer the 6-hour plan, because it is the cheapest way to reach the goal. If the plan is to be executed repeatedly, time no longer is a limited resource: Assume we have a limited time of 60 hours. If we repeatedly execute the 6-hour plan, we can execute it 10 times and gain 60 dollars. If we repeatedly execute the 2-hour plan, we can execute it 30 times and gain 120 dollars. Thus, we prefer the 2-hour plan over the 6-hour plan.

The Markovian decision algorithms treat every transition equally and thus assume that all execution times are the same. However, it is not necessary to include the different execution times in the planning problem from chapter 3, since the optimal policy does not depend on them. The reason is that the expected total reward of the plan execution is not influenced by the execution times.

If we want to execute a plan repeatedly and thus optimize the execution reward per unit time, we need to change our model in the following way:

- Changed Structure of the Markovian Decision Model

  We need to redefine the behavior of the state $\bar{s}$. After the agent stopped in a goal state, the execution phase no longer is over, but the execution is started again in a start state. Thus, $\bar{s}$ can no longer be an absorbing state, instead it must restart the execution. So, there is only one action possible in $\bar{s}$: a virtual action $\bar{a}$ that succeeds with probability one, has no execution cost and time, and leads to the possible start states according to the probability distribution for the start states. Therefore, we set $\bar{p}(s|\bar{s},\bar{a}) := b(s)$,

$\bar{p}(\bar{s}|\bar{s},\bar{a}) = 0$, $\bar{c}(\bar{s},\bar{a}) := 0$, and $\bar{t}(\bar{s},\bar{a}) := 0$ for all $s \in S$. (If stopping in a goal state or restarting the execution costs something or takes time, the values $\bar{c}(\bar{s},\bar{a})$ or $\bar{t}(\bar{s},\bar{a})$ are set to the appropriate values.) Note that we could drop the state $\bar{s}$ if we changed the stop actions to have the effect of the action $\bar{a}$, i.e. to lead back to the start states.

- Changed Markovian Decision Algorithm

  In the new model, we can no longer ignore the execution times of the actions. The Markovian decision algorithms assume that all execution times are identical. But different execution times can easily be included into the Markovian decision algorithms. For the value-iteration algorithm, it is obvious how to proceed: For each state, the continuous time scale is discretized such that the v value of every state under the optimal plan is the same for every point in the time interval. Depending on the execution times, these intervals can be very small. Then, the computation becomes very expensive. The policy-iteration algorithms can be used as well, if the equations for the value-determination procedure are slightly changed, and do not have this problem. For example, the equations for the simple policy-iteration algorithm become:

$$\bar{t}(s,a)g + v(s) = \bar{c}(s,f(s)) + \sum_{s' \in S} \bar{p}(s'|s,f(s))v(s')$$

  The $\bar{t}(s,a)$ can be arbitrary values, including zero. However, if there exists a cycle with a total execution time of zero or less, the set of equations no longer is uniquely solvable or solvable at all.

We must make sure that there are no cycles of non-positive execution time in the new Markovian decision model. Since all $\bar{t}(s,a)$ values are either positive (for physical actions) or zero (for virtual actions, namely the stop actions and the restart action), we only need to worry about cycles that have an execution time of zero. Then, the average gain is undefined (positive or negative infinity). Such cycles can only occur if it is possible to stop in a start state. We do not allow such cycles. If the time for stopping or restarting is non-zero, such cycles do not exist. If the reward for stopping in a state is negative, we can delete the stop action under the assumption that the agent prefers to abstain from executing a plan whose reward is smaller than zero.

For the model from chapter 3, we were able to guarantee that every execution of the optimal plan will eventually stop, if the state $\bar{s}$ can be reached from every start state. (We used as

trick that the action $\bar{a}$ has a cost of 0. Every other cycle must have a smaller cost, since all actions have negative execution costs. Thus, reaching the state $\bar{s}$ with probability 1 yields the largest reward per unit time.) This is no longer true for the modified model: There can be cycles that include a start state, but do not include the action $\bar{a}$. If a plan that contains such a cycle has the largest execution reward per unit time, the Markovian decision algorithms will return that plan as optimal plan. Although such a plan might be adequate for behaving in the world, we would like to ensure that it is rewarding for the agent to repeatedly reach a goal state and stop in it.

To test whether the state $\bar{s}$ will be reached under the plan found by Markovian decision algorithm for the new Markovian decision model, we can determine this plan and check it: We run the multiple-chain policy-iteration algorithm on the new model and then check whether there is only one ergodic set reachable from the start states and whether it contains $\bar{s}$. If this is not the case, we could force the agent to periodically stop in a goal state by either changing the structure of the domain appropriately or by making it attractive to reach a goal. The latter can be achieved by decreasing the costs of the actions or increasing the rewards of the goal states. However, even when increasing all rewards by the same constant, the optimal plan depends on the value of the constant. Thus, this method seems to be very arbitrary.

We can also utilize the two-step planner to test whether the state $\bar{s}$ will be reached under the plan found by a Markovian decision algorithm for the new Markovian decision model, if we make the assumption that the agent prefers to abstain from executing a plan whose reward per transition is negative: First, we solve the model that maximizes the total execution reward, i.e. the model from chapter 3. If the optimal total execution reward is negative, the optimal execution reward per unit time is also negative. In this case, it might or might not hold that the execution of the plan that maximizes the expected reward per unit time cycles without ever stopping in a goal state, but (according to our assumption) the agent prefers not to execute the plan in any case. If the optimal total execution reward is positive, the optimal execution reward per unit time is also positive. In this case, the plan that maximizes the execution reward per unit time has only cycles that contain the state $\bar{s}$ (after all unsolvable states have been removed). Thus, it is ergodic. The plan that we have determined so far maximizes the total execution reward. Now, we remove the unsolvable states and then use this plan as initial plan for the second planning phase of the two-step planner to solve the model that maximizes the execution reward per unit time. The second proof sketch in appendix 2

outlines why this method works: If the total gain of a policy is non-negative in the model that maximizes the total execution reward, then this policy is ergodic and every state has a non-negative gain in the model that maximizes the execution reward per unit time. Since the multiple-chain policy-iteration algorithm never decreases the gain of a state, the new policy after one iteration will again be ergodic. Therefore, the simple policy-iteration algorithm can be used instead of the multiple-chain policy-iteration algorithm.

15. Computational Complexity

In this chapter, we will study the time and space complexity of the one-step and the two-step planner. It will turn out that both the planning time and the space needed to store the universal plan prevent the planners to solve large planning problems optimally. We will present first ideas to decrease the time complexity using a hierarchical planning approach.

The space that is needed to represent a Markovian decision problem is on the order of the number of states times the number of actions: For every action, the number of its outcomes is bounded from above by the number of states. For each action and outcome, the probability of the outcome, the execution cost of the action, and its execution time have to be stored. (The number of actions is the sum of the cardinalities of the sets $\bar{A}(s)$ for all states s.) A plan is a compiled solution of the planning problem and needs, if stored in form of a table, one entry for each state, that contains the action assigned to the state. Thus, the total space needed is linear in the number of states times the logarithm of the number of actions.

The time complexity of planning and execution is determined by the planning phase. The decision which action to execute is very inexpensive and takes virtually no time, since it is only a fetch operation (using a table indexed by the current state) followed by the execution of the retrieved action. The worst-case run-time of the planning phase depends on the implementation of the Markovian decision algorithm: We know that the Markovian decision problem can be reformulated as a linear programming problem with a variable for each state and one for the gain. Then, the simple policy-iteration algorithm behaves very similar to the Simplex algorithm. Thus, its run-time can be exponential. But linear programming problems can be solved in polynomial run-time [39] by using an ellipsoid algorithm. (See for example [77].) Thus, the Markovian decision problem can be solved in polynomial run-time. If we replace the second phase of the two-step planner by an ellipsoid algorithm, the worst-case run-

time of the second phase is polynomial, and so is the total worst-case run-time of the planner. This compares favorably to the exponential run-time of the straight-forward approach of enumerating every possible plan, calculating its goodness value, and finally choosing the best plan. Unfortunately, the planning problem still becomes quickly infeasible when the state space grows. Note that the complexity of planning is usually measured as a function of the number of predicates that are used to describe the states, and that the number of possible states is exponential in the number of predicates. One way to cope with the time complexity of optimal probabilistic planning is to interleave planning with execution.

There is another, related complexity problem: To solve the planning problem using Markovian decision algorithms one needs to expand the whole state space. As already pointed out, the number of states is potentially exponential in the number of predicates that are used to characterize the states. This leads to an exponential explosion for a planning problem that could be stated in STRIPS-like notation in very limited space. Planning domains with infinitely many states cannot be handled at all. When representing a planning problem as a Markovian decision problem, one also loses knowledge, namely which set of predicates is represented by a state, thereby making techniques such as subgoaling impossible.

Usually, planning methods do not explore the state space completely, but use heuristics to guide the search, such as abstraction levels, macro operators, subgoaling, or model reduction. Therefore, they cannot guarantee optimality, but they are able to handle a large state space in a feasible way. (For a complexity analysis of traditional planning methods see [16].)

One obvious way to implement the model reduction approach for Markovian decision problems is to modularize the problem by grouping together states such that the newly formed sets of states have only one entering and one exiting state. Such a cluster of states can be used as a meta-state for an abstracted Markovian decision problem. For highly interconnected planning problems, such as the blocks-world, it is impossible to isolate or even approximate meta-states, because of the many interactions among the actions. For example, the state space of the three-block blocks-world is shown in figure 25. The more blocks exist in the blocks-world, the more interactions are there. But non-artificial planning domains, such as travel planning, often allow for hierarchical planning or a close approximation thereof, since there is less interdependence among the effects of actions. However, in many domains, there will be at least *some* interactions among the actions. (For example, such interactions lead to the Sussman anomaly.) Even then, one can use Markovian decision algorithms (such as

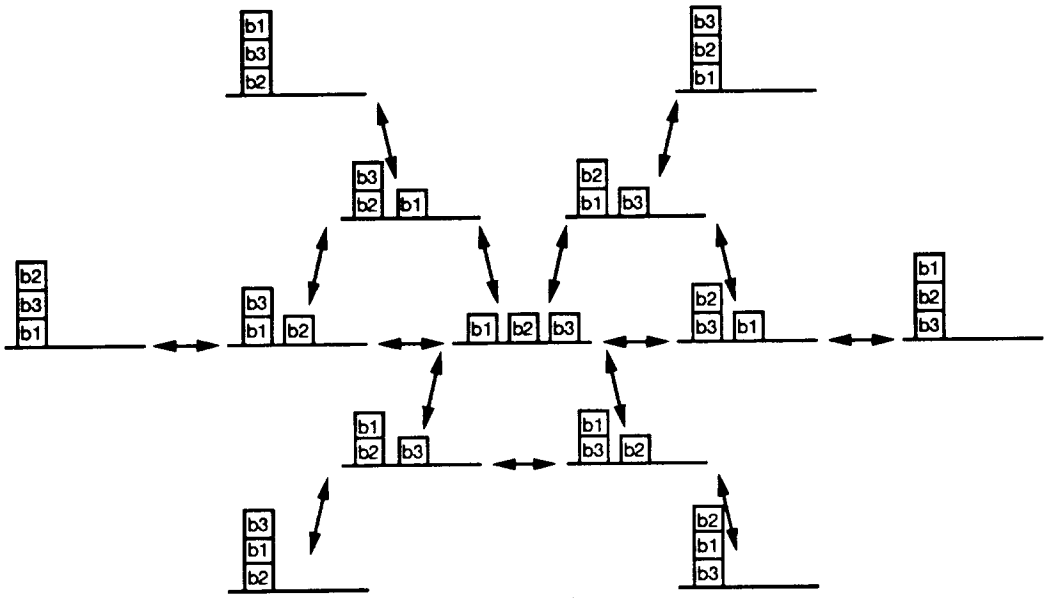


Figure 25: The State Space of the Three-Block Blocks-World

Bertsekas's modification of the policy-iteration algorithm [10]) for hierarchical planning, but one might lose the optimality guarantee.

An ideal domain for hierarchical planning would resemble a hierarchy of "actions" (i.e. operators, macro operators etc.). Every action could be hierarchically decomposed into a new planning problem, namely how to implement the action using smaller actions, as shown in figure 26. There are no interactions in such a domain and the subproblems can be solved independently. Thus, the planner first solves the smallest (i.e. non-decomposable) planning problems. Then, for each of these planning problems, it deletes all states of the planning problem except its start state and absorbing state, and adds an action applicable to the start state that deterministically leads to the absorbing state and has the v value of the start state under the optimal plan as cost attached. (Depending on the rewards, this "cost" could be non-negative.) Once every subproblem of a larger planning problem is solved, the planner starts solving the larger problem, until the top-most planning problem is solved.

It is a topic of further research how to use the normative planning theory to develop satisficing planning methods in order to be able to handle large-scale planning problems.

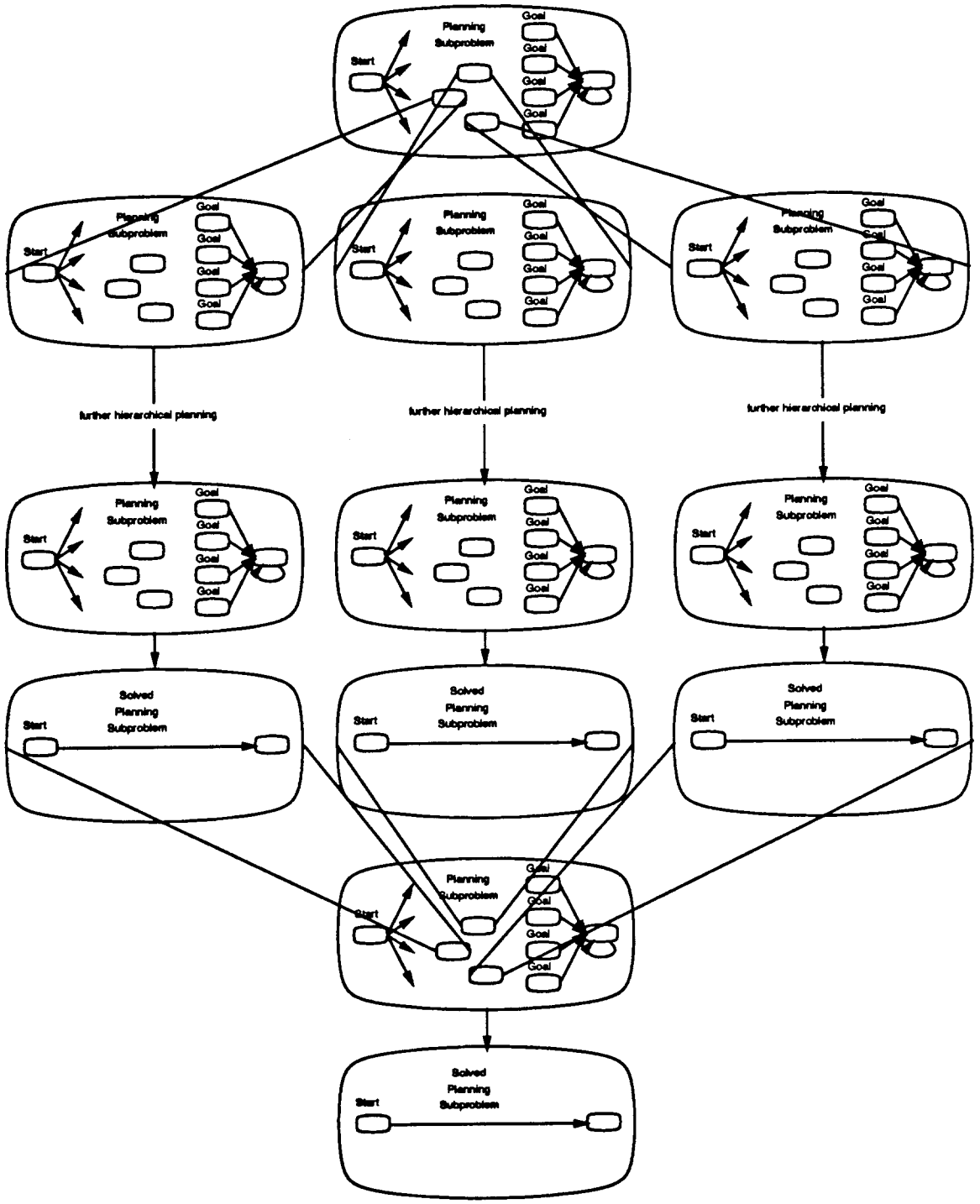


Figure 26: Hierarchical Planning

16. Implementation Details

In this chapter, we will describe the architecture of a software system that implements our ideas and sketch the outcomes of preliminary experiments with the system. The system is constructed in a hierarchical way as shown in figure 27: The STRIPS interpreter translates augmented STRIPS operators into a state-action representation (a directed graph with hyperedges). This representation and the rest of the specification of the planning task are translated into a Markovian decision problem by the model generator and used to find an initial plan by the plan generator (either by calling a procedure specified by the user or by using one of the algorithms discussed in chapter 12). Then, both the Markovian decision problem and the initial plan are used by the implementation of one of the Markovian decision algorithms (policy iteration, in the future also value iteration and q-learning) to find the optimal (or a close-to-optimal) policy. The plan generator and the implementation of the Markovian decision algorithm are set up in a way that both the one-step planner and the two-step planner can be simulated. The final plan and the statistics gathered during the planning phase are used by the result evaluator to provide the user with information about the planning process in terms of his original domain representation. The Markovian decision algorithm is coded in C for efficiency reasons, the other modules are coded in PROLOG [18].

In the following, we will describe the components of the software system in greater detail:

- The Model Generator

  The model generator needs the following data and procedures: The type of the problem (maximization of average or total gain), a probability distribution over the start states, a list of goal-reward pairs, and a procedure that returns the names of all applicable actions in a given state (via backtracking) and their transition lists (i.e. lists of (successor state, transition probability, execution cost, execution time) quadruples). The user also has to provide a procedure that determines whether two states in his representation are equal. The model generator translates the state-action representation provided by the user into a canonical one, namely into a directed graph with hyperedges. This graph includes all states that are reachable from the start states by a sequence of actions. Actions that have no effect, or actions that exactly duplicate the effects of other actions are deleted. The resulting graph and a translation table from the external to the internal state-action representation are written to a file as PROLOG clauses. The graph is then

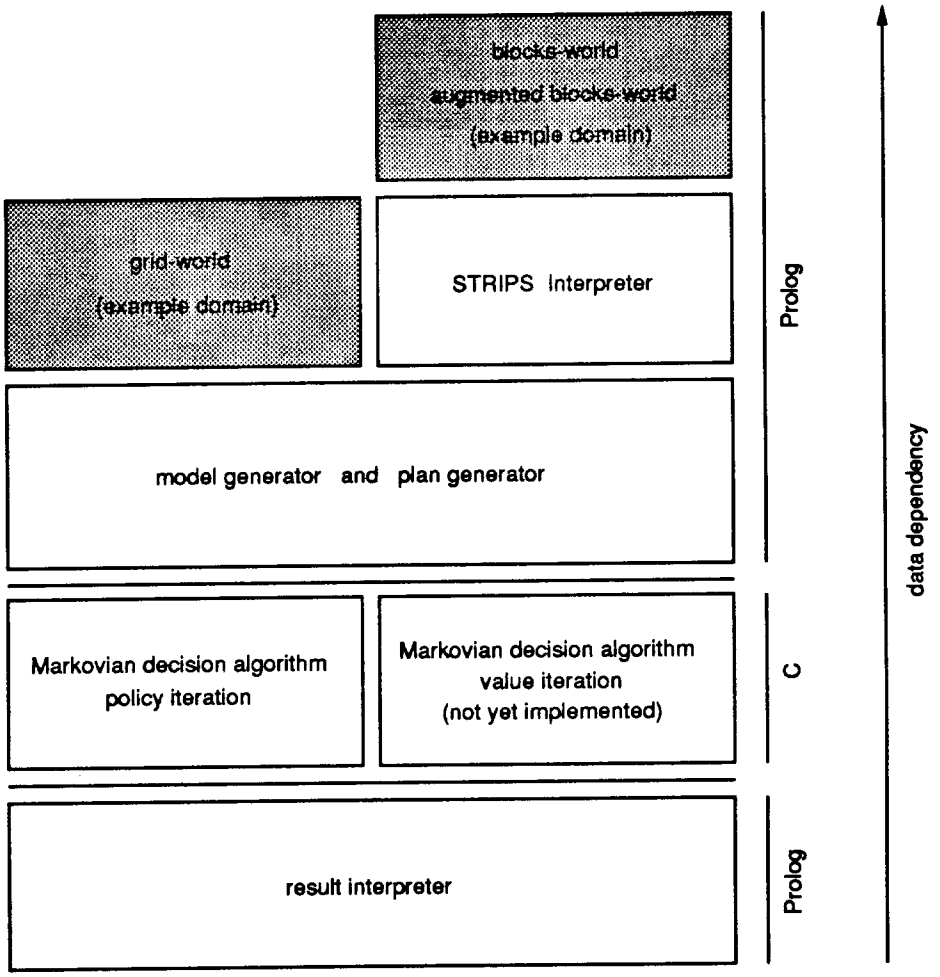


Figure 27: The Architecture of the System

converted to a pdt Markovian decision problem according to the type of the problem, which is further processed by the plan generator.

- The STRIPS Interpreter

  In order to simplify the description of the domain, the user can use the STRIPS interpreter to describe the states as sets of predicates and the actions as augmented STRIPS operators. Every operator has a head and a body. The head is the name of the operator parameterized with constants or variables used in the body. Once the variables are instantiated, this name must uniquely identify the action among all actions of a state. The body of an operator specifies a precondition list and for each action outcome its probability, execution cost, execution time, an add list and a delete list. The three numeric values can be functions of the state in which the operator is executed, the outcome, and the bindings of the operator. In the following, we give an example of an operator definition that shows how the functions can be implemented in PROLOG. The operator defines a stacking operation in the augmented blocks-world. (The execution cost of a successful move is -1 if the block was not moved upward, otherwise it is -1 plus the the negative of the level difference the block was moved. The execution cost of an unsuccessful move is -1.)

```
move(BlockA,table,BlockB) ::
  precond([on(BlockA,table),clear(BlockA),clear(BlockB),
           unequal(BlockA,BlockB)]),
  [ ( 0.6, X^Y^Z^movecost(X,Y,BlockA,Z), 1,
    dellist([on(BlockA,table),clear(BlockB)]),
    addlist([on(BlockA,BlockB)])),
    ( 0.4, -1, 1, dellist([]), addlist([]) )
  ].

movecost(A,B,C,D) :-
  blockheight(A,C,E),
  blockheight(B,C,F),
  (E < F -> D is -1 + E - F ; D = -1).

blockheight(A,B,0) :-
```

```
  member(on(B,table),A), !.

blockheight(A,B,C) :-
  member(on(B,D),A),
  blockheight(A,D,E),
  C is E+1.
```

The STRIPS interpreter implements the two procedures that are needed by the model generator, namely the procedure that generates all applicable actions in a given state and the procedure that determines whether two states are equal. To determine the applicable actions, the STRIPS interpreter tries to apply every operator to the state in every possible way. Two states could be regarded as being equal, iff every predicate in one state has an equal counterpart in the other state, but we implemented a more general notion of state equivalence. Usually objects are described by their intrinsic properties (such as "the *black* block") and their relations to other objects (such as "the block *directly on* the table"). The identity of an object can only be determined by using such properties, not by its name. For example, in the augmented blocks-world it is misleading to give blocks fixed names (such as $b1$), we can only refer to a block using its properties (such as "the black block supported by a stack of three white blocks"). So, a stack of four black blocks constitutes one configuration, and not 24 of them, since these 24 configurations are indistinguishable. Still, references to objects ("names") are needed to describe states and actions, because an object is part of several relations (such as in $ON(b1, b2)$ and $COLORED(b1, black)$, the simple $ON(black, white)$ will not do), and because we want to refer to an object as parameter of an operator (such as in $MOVE(b1, b2, table)$). These object-name bindings are useful only when referring to a particular state. They are not maintained for the successor state. The user can provide a list of symbolic references ("deictic names") used for reference purposes, i.e. for indexing objects (such as $b1$, $b2$, etc.). To determine whether two states are equal, the STRIPS interpreter determines whether a permutation of the symbolic references exists such that after the permutation every predicate in one state has an equal counterpart in the other state. If the list provided by the user is empty, this is equivalent to the more restricted traditional interpretation. (The idea of symbolic references was taken from languages for semantic networks, for example [13].)

- The Plan Generator

The plan generator can be run in either of two modes. The first mode specifies that for each state one of its applicable actions should be chosen at random. In this case, the initial plan is not guaranteed to be ergodic. This option is used to make the Markovian decision algorithm step simulate the one-step planner. The second mode specifies that the unsolvable states should be deleted from the pdt Markovian decision problem and that an ergodic policy should be found for the remaining states. If the user has defined a procedure to accomplish this task, this procedure is called. Otherwise, the general greedy algorithm is used. After the mode-specific processing has been done, the pdt Markovian decision problem and the initial plan are written to a file in form of a numerical table that is suitable as input for the Markovian decision algorithm.

- The Markovian Decision Algorithm (Policy-Iteration)

  The implementation of the Markovian decision algorithm uses the pdt Markovian decision problem and the initial plan that was produced by the plan generator. Then, it uses the multiple-chain policy-iteration algorithm to solve the planning problem. Thus, it can cope with non-ergodic initial policies. If the policy is ergodic, the multiple-chain policy-iteration algorithm collapses to the simple one, and therefore the number of iterations is the same for both algorithms. So, the number of iterations needed by the Markovian decision algorithm step is equal to the number of iterations needed by the one-step planner, if the plan generator was run in mode one. Otherwise, it equals the number of iterations of the second phase of the two-step planner. Therefore, the performance of the two planners can be compared in terms of the number of required iterations. Note that the first planning phase of the two-step planner is not accounted for in the number of iterations. Thus, the number of iterations cannot be the only evaluation criterion for the two planners. A fairer evaluation criterion is the run-time of both planners on the same set of planning problems. The run-time also takes into account that an iteration of the second phase of the two-step planner is faster if we implement the simple policy-iteration algorithm directly, instead of using the multiple-chain policy iteration algorithm to achieve the effect of the simple one. After every iteration, the Markovian decision algorithm writes for each state its v value, gain and the assigned action to a file. Due to the nature of the algorithm, all v values, gains, and actions, respectively, will be identical for the last two iterations and denote the values of an optimal plan.

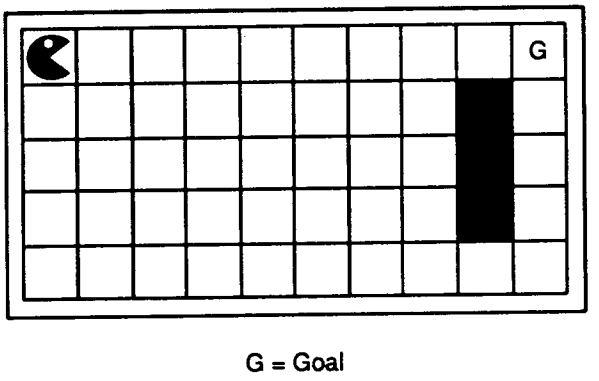


Figure 28: The Grid-World

Experiments using this software system are still under way. So far, three domains have been implemented: The blocks-world and the augmented blocks-world, both using the STRIPS interpreter, and the grid-world [12], that resembles Sutton's reinforcement learning test bed [103], see figure 28. The grid-world is a rectangular collection of squares, some of which are blocked. An agent can move from a square to one of the four neighboring squares, iff that square is inside the grid and unblocked. Its task is to reach a designated goal square from a given start square. The grid-world is an interesting domain, because it is easy to determine initial plans for the grid-world with a varying degree of goodness.

Preliminary experiments confirm the intuition that the number of iterations needed by the second phase of the two-step planner depends on how close to optimal the plan found by the first planning phase is. The better it is, the less iterations are required. An unexpected result is that a random assignment of applicable actions to states, i.e. an initial random plan, can lead to less iterations of the two-step planner than a bad, but ergodic, initial plan. Another unexpected observation can be made when the one-step planner is used to solve a planning problem that has only solvable states: The intermediate policies stay non-admissible during the majority of iterations before they finally become admissible.[7] This means that the one-step planner usually needs a lot of iterations before it finds the first solution. The longer the start-up time of an anytime algorithm is, the less useful is its anytime property. If a (not

[7]This can be explained as follows: When the one-step planner is used, the transformation of the random, initial plan to an admissible plan is only a by-product of the optimization of the initial plan. When the first time an admissible plan is found, this plan is already optimized to a large degree. Therefore, the intermediate plans are non-admissible for most of the iterations. But once an admissible plan has been found, it takes only a few additional iterations to transform it to the optimal plan.

necessarily optimal) solution is needed early and the first phase of the two-step planner finds a suboptimal, but ergodic plan fast, the two-step planner should be preferred. Of course, these results depend on the structure of the domain. Further research will concentrate on the impact of the initial plan on the second planning phase and a quantification of the trade-off between the first and second phase of the two-step planner.

17. Related Work

In this chapter, we will discuss work that is related to our approach to planning in probabilistic and decision-theoretic domains. We have already pointed out that the two-step planner uses Markovian decision algorithms and ideas from universal planning, anytime algorithms, and reinforcement learning.

The idea of planning in two steps resembles Simmons' generate, test and debug approach to planning [21]. In the first step, his planner uses heuristics to find a plan that is not necessarily correct. In the second phase, the initial plan is incrementally debugged until it solves the given task. The final plan is correct, but not necessarily optimal, since Simmons is not concerned with optimality.

The v values used by the Markovian decision algorithms resemble the notion of potential used by some path planning methods, e.g. by Koditschek's navigation functions [62]. There, the plan consists of potentials for the states. During execution, the agent always chooses the action that maximizes the expected increase in potential. (If the potential cannot be increased, the agent chooses the action that minimizes the expected decrease in potential.) The difference between our approach and planning with potentials is that in our case the state space is not continuous and less structured. In potential fields, states that are close to each other usually have a similar value (smoothness condition). This allows one to state the function that maps states into values in closed form. However, many potential field approaches discretize the continuous state space for complexity reasons.

The AI approaches that are closely related to the planning problem presented in this report are the one's by Dean and Kanazawa, Drummond and Bresina, and Smith. Their three planners use different approaches to solve almost the same problem. We will discuss them in greater detail to show that approaches that are currently pursued in the literature fit into the same framework. All three models suffer from the limited time horizon problem, because

they unroll the underlying Markovian decision problem. One has to keep in mind, though, that the goal of these planners is to make planning in probabilistic domains tractable, not to study its properties.

- Dean and Kanazawa's Planner

  Dean and Kanazawa [24, 60, 25, 26] use a revolving planning approach. They view a plan as a sequence of actions ("behaviors"). At each point in time, the agent chooses the available plan with the largest expected reward ("utility") over the next n time periods and executes its first behavior. As soon as the agent has executed this action, it again chooses the plan with the largest expected utility etc. The utility is calculated by predicting the future within a limited time horizon. This prediction is based on some predicates of the current state ("current sensory information") and perhaps the history ("past information"). States are represented in STRIPS-tradition as sets of predicates. Dean and Kanazawa use an influence diagram to model the dependencies among the probabilities, and Shachter's method [94] to select the action that optimizes the reward. The influence diagram has a number of probability nodes for the input, that correspond to the current sensor information and perhaps some sort of state information. It also contains one decision node, that represents the choice of the behavior, a value node, that represents the reward of the plan, and some intermediate (probability) nodes. See figure 29 for the structure of the influence diagram. Dean and Kanazawa call such an influence diagram a "causal model in probabilistic, temporal reasoning". Its structure requires the probability distribution over the action outcomes to depend only on the value of the input nodes and the chosen action (Markov property). The current joint probability distribution over the predicates (i.e. the probability distribution over the states) must be uniquely determined by the inputs. (Unless the predicates are pairwise independent, it is not enough to know only the probability distributions over the values of the predicates.) Dean and Kanazawa also assume that the total reward is the sum of the rewards that are received at each point on the discrete time scale (time-separable, additive value function).

  An advantage of their model is that revolving planning allows them to vary the amount of lookahead during planning by choosing different time horizons. This allows them to trade off the quality of the solution and the planning time for on-line planning. Dean and Kanazawa also show how the influence diagram can be compiled thereby decreasing

the amount of time spent on on-line planning at the expense of time spent on off-line planning and memory space used. Compiling the influence diagram is equivalent to finding input-action rules.

If sensory information is uncertain or missing, an optimal planner has to represent the state that is believed to be the current state as a probability distribution over the states. Dean and Kanazawa's model could easily be extended to this case by maintaining the layers of the Bayes' net that correspond to a past point in time. (This is equivalent to using partially observable Markovian decision problems with a finite time horizon.) Unfortunately, this would make the computation of the Bayes' net quickly infeasible, and Shachter's method could no longer be used to compile the influence diagram, because then the topology of the influence diagram changes over time. To circumvent this intractability problem, Dean and Kanazawa rely on structural properties of the domain and have input nodes that can represent history information.

This model has been applied to a task of behaving, namely for selecting actions for a mobile robot that moves around in a simple environment and tries to stay in operation by replenishing its energy at locations known to have energy sources. It was also used for a task of tracking a moving object. Since Dean and Kanazawa's approach allows them to obtain tractability for certain domains, they concentrate their research on structuring the domain representation for their planning domains to make it suitable for their planner.

Dean and Kanazawa recognize that temporal belief networks that satisfy the Markov property, and Markovian decision problems with a finite time horizon are equivalent, but they see two advantages of temporal belief networks: First, they are less space demanding, because a layer of an unrolled Markov chain contains one state for every state of the world, whereas a layer of their model only needs one for every predicate. (In the worst case, the number of states grows exponentially with the number of predicates.) Second, temporal belief networks facilitate the computation of the probability distributions over the predicates. We would like to note that there is not much difference in the spatial and computational effort that is needed in the worst case: For example, the table that is needed to implement the decision node grows in the worst case linearly in the number of states. (Thus, it is confusing to count only the number of states, but not the size of the data structures that are needed to implement them.) Furthermore, Markovian decision processes do not need to replicate states for

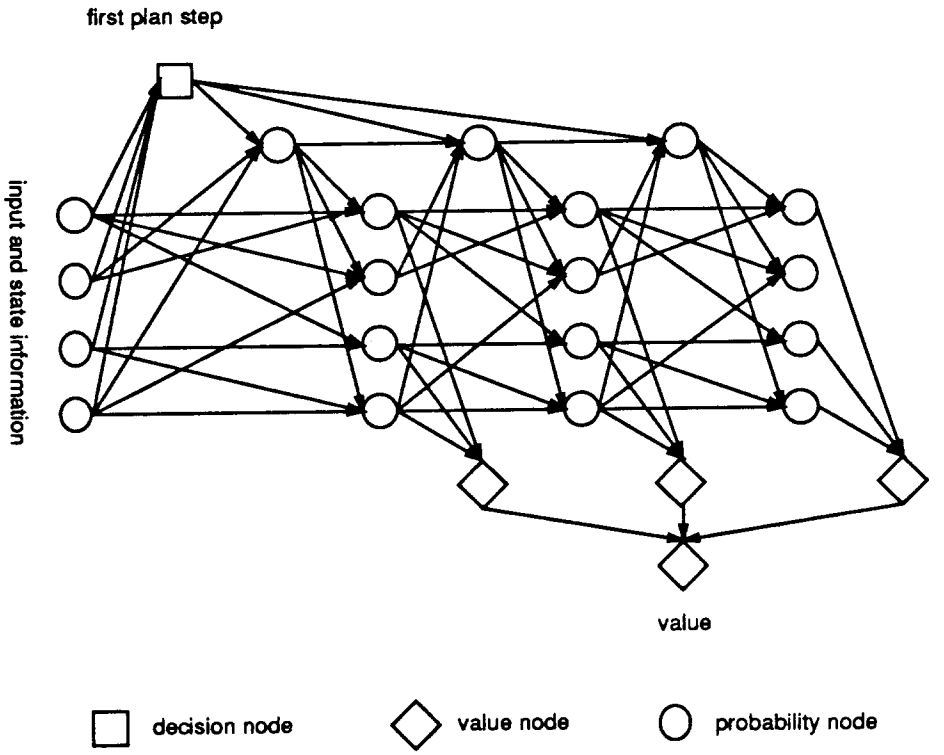


Figure 29: Planning Model of Dean and Kanazawa [from their paper with only minor changes]

each point in time within the planning horizon, but for Dean and Kanazawa's model this is unavoidable. Using Shachter's method to determine the maximal reward for causal models is equivalent to using the value-iteration method to propagate the values of the states using Bayes' law for Markovian decision problems. Using causal models will be slightly less time efficient than using Markovian decision problems, because of the overhead introduced by influence diagrams. Off-line planning is possible, since the structure of the model and the solution method (the roll-back method) are known in advance. A compilation of the decision model as suggested by Dean and Kanazawa is equivalent to finding a universal plan with the roll-back method. It is important to note that Dean and Kanazawa's model can avoid the worst case situation by taking advantage of the domain structure: The smaller the number of actions and the smaller their precondition, add and drop lists, the better are the time and space demands of the causal model.

- Drummond and Bresina's Planner

  Drummond and Bresina's probabilistic planner [15, 31] tries to maximize the probability of satisfying behavioral (temporal) constraints heuristically. Since we are not interested in such constraints, we will phrase our summary in terms of goals and subgoals. A plan consists of a set of (subgoal, state, action) rules ("situated control rules") which will usually not be exhaustive (total). It can be viewed as a subset of the union of several universal plans, one for each subgoal. Drummond and Bresina's planner uses a heuristic search method in unrolled Markovian decision problems. Thus, the Markov property must hold for the domain.

  In the first phase of Drummond and Bresina's planner, an initial set of control rules is determined. First, the goal is split into a sequence of subgoals that are to be achieved in order. Starting with the start state and the first subgoal, the planner repeatedly finds actions that are applicable to the active nodes and applies them. The outcomes are predicted using the conditional probability of the successor state given an action and the state it is executed in ("domain causal theory"). This way, a directed acyclic graph (the "projection graph") is created, that contains (state, time) pairs as nodes and action applications as edges. (Thus, it is part of an unrolled Markovian decision problem representing the domain.) Not all possible action outcomes are considered: a filter is used to select a subset of the most probable action outcomes and only these are used to produce the successor nodes. A heuristic value is calculated for each of the

successor nodes based on the probability with which this node can be reached from the start node and an estimate of the remaining work that is required to satisfy the current subgoal from this node. Another filter is used to restrict the set of newly activated nodes to those with large heuristic values. The parameters of these two filters need to be tuned for every application domain. Once a path is found that leads from the start node to a node that satisfies the first subgoal, for each node on the path a (subgoal, state, action) rule ("If you are in state s and the (sub)goal is g, then execute action a") is created. The endpoint of the path is used as the new start state for the search of a path leading to a node that satisfies the second subgoal ("cut and commit strategy") etc. When the last subgoal is finally achieved, a lower bound for the probability of achieving the overall goal can be computed as the probability of traversing the path leading from the overall start node to the node that satisfies the last subgoal.

If planning time is still left, the second phase of the planner tries to incrementally increase the probability of satisfying the goal by synthesizing additional (subgoal, state, action) rules. Thus, the second planning phase has the anytime property. It detects additional paths by first finding highly probable deviations from the existing solution path and then determining a path that recovers from each deviation. A deviation is a potential successor of a node on a goal path for which a (subgoal, state, action) rule that is part of a goal path has not yet been computed. Note that all paths (even the deviations from the solution path that was found first) have to pass through the same subgoals, although this might not be optimal. After such a path has been determined, a (subgoal, state, action) rule is created for each node on the path. As long as planning time is left, the second phase can be repeated. (It can also be run in parallel with the execution.)

- Smith's Planner

  Smith's decision theoretic planner [97, 98, 99] produces partial, flow-diagram like plans in a probabilistic STRIPS-like domain. Goals can be decomposed into subgoals. This decomposition is stated with (possibly variablized) PROLOG-like rules that show all the ways a goal can be decomposed. Plan fragments are created for each conjunctive subgoal of the given goal and then assembled into an overall plan. There is only one goal, which does not need to be achieved if stopping the execution without having achieved it leads to a lower total cost (the cost of executing the plan plus the product of the probability of failing to achieve the goal and the price of failure) than trying to

achieve it. Thus, the planner has to decide whether to try to achieve the goal, and if so, what the cost minimal way of achieving the goal is. Main considerations are in which order to try to achieve the subgoals, in which order to plan for the subgoals (i.e. how to bind variables that appear in the STRIPS-like rules), and how to achieve a subgoal. Smith uses a best-first search procedure guided by non-admissible heuristics to decide the first two problems. The last problem includes the decision whether to assume that the subgoal will hold, to coerce the world into a state in which the subgoal holds (and which action to choose in this case, see also [43]), to plan separately for the cases that the subgoal holds and that it does not hold, to predetermine at planning time whether the subgoal holds, or to defer planning until the state of the world is better known. In Smith's model, these decisions depend on the execution costs of the actions, the probability that an action will achieve the desired goal, the extent to which each action would damage the world in case of failure (by making the application of other actions impossible and thus the goal harder or impossible to achieve), and the importance of achieving the overall goal. The way plans are ranked is depicted in figures 30 and 31.

This approach has several implicit assumptions, all of which are stated by Smith himself (except for the last one): First, it must be easy to approximate the probabilities ($P(p,s)$, $A(p,s)$) and the cost ($C(p,s)$) for all states and all predicates (and their negations). Furthermore, the same must be true for $P(p,s|B(q,s)), C(p,s|B(q,s))$, and $A(p,s|B(q,s))$ without knowing how the optimal plan to achieve q in s looks like. Smith did not present empirical evidence that this is indeed possible, but stated that such values belong to the common-sense knowledge of human problem solvers and could be learned with machine learning techniques. (And, indeed, the Bayesian Problem Solver shows how to utilize heuristics to guess these values.) Second, the overall goal must be achievable by linearly realizing the subgoals. Thus, "nonlinear" problems such as the Sussman anomaly [102] cannot be solved optimally. Third, during plan execution it must be obvious at every point in time which branch of the plan to take. Thus, the execution mechanism must be able to recognize immediately the failure of an action to achieve the desired results. Fourth, the system of equations for evaluating a plan is a linear programming problem, but is not treated as such. (For example, to define $E(p,g,s)$, i.e. the cost for achieving a simple predicate in a state in order to eventually achieve a given goal, Smith needs $C(g,s|p)$, the cost of the best plan to achieve the goal from the successor state. Instead of assuming like Smith that $C(g,s|p)$ can be

approximated easily, we can determine its correct value: $E(g,g,s)$. Substituting this value into the equation can introduce recursion: The value of $E(g,g,s)$ is defined by referring to it own value. To solve the equation, we must solve a linear programming problem.)

An advantage of Smith's planning model is that it allows one to reason about the instantiation of STRIPS-like rules, whereas many approaches only allow rules that consist of ground clauses. Smith's approach also seems to provide a good basis for modeling abstraction hierarchies, since the subgoals can be chosen on an arbitrary level, but no such usage has been reported yet.

18. Conclusion

In this report, we developed a theory for modeling and optimally solving simple probabilistic and decision-theoretic planning problems in a domain and application independent way. The model is more general than GPS-like planning models, but can be stated in a STRIPS-like notation that is augmented with probabilities, costs, and rewards. It provides a normative theory for probabilistic planning models that are currently discussed in the literature.

We showed how to utilize results from the operations research literature for planning. This resulted in optimal probabilistic and decision-theoretic planners. We developed approaches for off-line planning, on-line planning, and on-line planning with reinforcement learning (to be able to adapt to an unknown or changing environment) and gained knowledge about the properties of optimal plans and planning algorithms. For example, we showed that one can restrict the search for an optimal plan to universal plans, and that one can trade off planning time and the quality of the resulting plan without sacrificing the admissibility of the plan (anytime property).

A one-step planner for optimal probabilistic and decision-theoretic planning could be developed by transforming the planning problem into a Markovian decision problem, that is then solved by the value-iteration or the policy-iteration method. In this report, we also proposed a new two-step planning approach: The first planning phase determines an initial ergodic plan for every solvable state. We provided a general algorithm to solve this task, but also showed how the structure of the planning domain can be utilized to develop faster domain-specific algorithms, for example by taking advantage of either leaking actions or deterministic

$I(g)$	the price of failure to achieve the goal predicates g;
$C(a,s)$	the cost of executing action or plan a in state s;
$P(q,s)$	the probability that predicate q holds in state s;
$s\|a$	the (probability distribution over the) successor state(s) when executing action or plan a in state s;
$B(q,s)$	the best (cost minimal) plan in state s to achieve predicate q;

Define $\bar{P}(q,s) := P(\neg q,s)$, $C(q,s) := C(B(q,s),s)$ (the cost of the best plan in state s to achieve predicate q), $s|q := s|B(q,s)$ (the probability distribution over the successor states when executing the best plan in state s to achieve predicate q), $A(q,s) := P(q,s|q)$ (the probability that the best plan in state s to achieve predicate q will succeed), and $\bar{A}(q,s) := P(\neg q,s|q)$. Then

$$B(q,s) := p : \min_p(C(p,s) + \bar{P}(q,s|p)I(q))$$

Let e be a conjunctive predicate. For a conjunct p in e we write $p \in e$. Let $\tau(p)$ be the predicates preceding predicate p in the execution order, and $\varphi(p)$ be the sequence of predicates preceding p in the planning order (i.e. the sequence of predicates that get their variables bound before p). Let $p\backslash q$ denote a predicate that is derived from p by binding all unbound variables that p shares with q to the values they have in q. Then, the cost for achieving a simple predicate $p \in e$ in state s in order to achieve the goal predicates g eventually is

$$E(p,g,s) := C(p,s) + \bar{A}(p,s)(C(g,s|p) + \bar{A}(g,s|p)I(g))$$

and the cost for achieving the conjunctive predicate e in state s in order to achieve the goal predicates g eventually is

$$E(e,g,s) := \min_{\tau,\varphi} \sum_{p \in e} A(\tau(p),s)E(c\backslash\varphi(p),g,s|\tau(p))$$

Thus, the cost of the best plan in state s to achieve the goal predicates g is $E(g,g,s)$.

Figure 30: Smith's Method for Constructing and Evaluating Plans (1) [my figure]

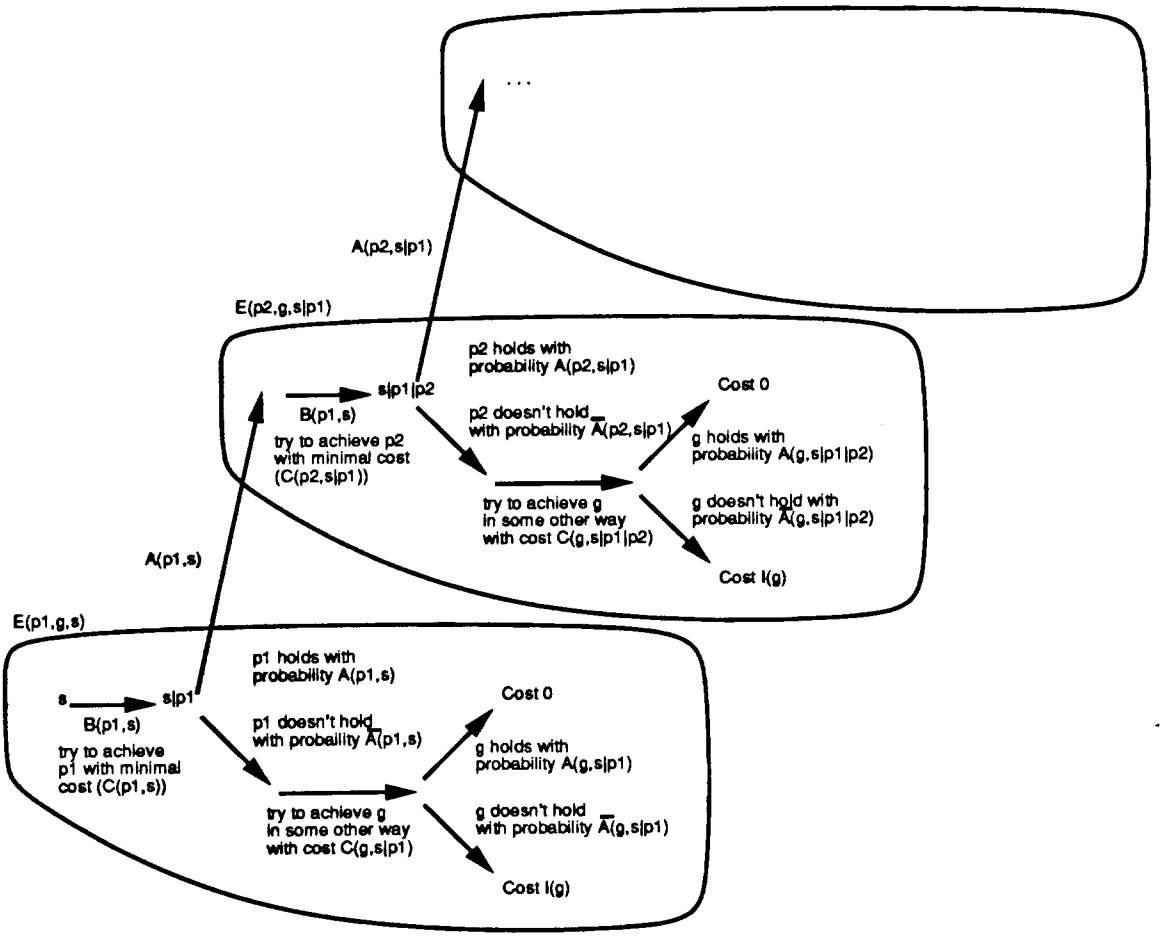


Figure 31: Smith's Method for Constructing and Evaluating Plans (2) [my figure]

(and quasi-deterministic) actions. Since the numerical values of the transition probabilities are unimportant in the first planning phase, AND-OR search methods or slightly augmented deterministic planners can be used, thereby narrowing the gap between deterministic and probabilistic planning. In the second planning phase a Markovian decision algorithm is used to refine the initial plan incrementally and derive increasingly better plans, until the optimal plan is finally found. Thus, we deviated from the traditional approaches that try to find an optimal or satisficing plan directly.

We stated that the complexity of the planning problem is polynomial in the number of states. This is no problem for a normative theory, as long as it is only used as a theoretical tool to guide the search for suboptimal theories or heuristics. For example, our planning model provided the basis for a theory that compares the quality of visual procedures (under restricted assumptions) using Markovian decision models, thereby circumventing the complexity of partially observable Markovian decision models [17]. The planning method presented here can be implemented directly, if the state space is small enough. The implementation can then be used as a planner on its own, or to supply parameters to heuristic planning methods (for example BPS) or to evaluate and compare them. Low-level planning domains for robots can be suitable domains, since they are often highly probabilistic, but have only a small state space.

To summarize, we have shown connections between probabilistic planning and the following fields: Markovian decision theory (operations research), universal planning, anytime algorithms, and reinforcement learning. It is a topic of further research how to use the normative theory to develop satisficing planning methods in order to be able to handle large-scale planning problems, e.g. by integrating abstraction levels (e.g. by modularizing the Markovian decision problem), macro operators, subgoaling, or model reduction. Another topic that the author pursues is to integrate physical actions and perceptual actions, see the short outline in appendix 5. The work described here is only a first step towards a theory of probabilistic and decision-theoretic planning.

19. Acknowledgments

My work on this report was started at the University of California at Berkeley, continued at Carnegie Mellon University, and finally turned into a project report of the University

of California at Berkeley. It was advised by Stuart Russell, who I am indebted to for his continuing support and advice. Lotfi Zadeh was so kind to accept the second readership. My other two advisors at Berkeley, Robert Wilensky and Peter Norvig, helped me by guiding my search for interesting areas in the jungle of Artificial Intelligence. At Carnegie Mellon University, I am indebted to my advisor Reid Simmons for his encouragement to continue the report and many discussions and comments about this approach to probabilistic planning. At the University of Hamburg (Germany), I am indebted to my two advisors Bernd Neumann and Matthias Jantzen, who made me interested in Artificial Intelligence, encouraged me to study in the US, and maintained contact all the time. Matthias Jantzen often made me wish I were more of a theoretical computer scientist than I actually am. Lonnie Chrisman, a colleague at Carnegie Mellon University, provided valuable feedback by showing how one can utilize planning with Markovian decision models as proposed in this report to focus attention in selective sensing. I would also like to thank my former colleagues at Berkeley for helpful discussions. This includes Jeff Conroy, Othar Hansson, Sonia Marx, Andrew Mayer, Gary Ogasawara, Ron Musick, Sudeshna Sarkar, Shlomo Zilberstein, and especially Marie desJardins.

A Restrictions of the Planning Model

Since the probabilistic and decision-theoretic planning model is based on the GPS domain model, it has as implicit assumptions most of the assumptions usually made when using the state space approach in connection with the situation calculus [50]. The only additional assumptions that we use are that the state space is finite and that the state space can be easily constructed. The major assumptions are listed in the following (see also [41]):

- Finiteness

  We assume that the state space of the domain is finite, and that the number of applicable actions in every state is finite. In reality, the states and the actions are often parameterized with real values. As an example, imagine a three-dimensional blocks-world with real-valued locations of the blocks on the table. Then, the positions of the blocks are determined by real values, namely the x, y, and z coordinates. Thus, neither the state space of the domain nor the number of different grasping actions is finite. In order to fit the model, continuous domains can be finitely approximated.

- No External Effects

  We also assume that nothing happens unless the agent makes it happen. Thus, no external events are allowed: The environment cannot act on its own, and there are no other agents. Actions can only be executed one after the other, and their effects occur immediately. Therefore, there is neither parallelism, delays, nor scheduled events. So, the outcome of an action is not influenced by remaining effects of actions that were executed earlier, or effects of actions executed in parallel.

- Markov Property

  Furthermore, we assume that only the state and the action executed in that state determine the (probability distribution over the) successor state(s). The history of executed actions and action outcomes that led to the source state have no influence on the outcome of the action. Hence, they are not important for choosing an action. This independence assumption (Markov property) is restrictive, because it abstracts away from cause and effect. If a certain action failed to achieve the desired effect in a certain state for a certain reason, the action will again fail to work in the same state if the cause for the failure is still present. However, it can be argued that in such a case the state representation is not adequate and therefore needs to be refined.

- Task-Driven Planning

  The task of the agent is to achieve only *one* goal state. If the agent reaches a goal state during the execution, but decides not to stop, then it does not receive the reward for that goal state. It only receives the reward for the goal state in which it finally stops the execution. (This is like buying ice cream to cheer up oneself, but then deciding not to eat it and instead trading it for a bar of chocolate. The pleasure results only from eating the chocolate bar, not from both the chocolate bar and the ice cream, since the ice cream is no longer available.) This model is adequate for task-driven planning problems, for which the "mission" of the robot is externally given. This is a different planning task than "behaving" in the world. Solving the latter task usually requires multiple-goal planning. (For example, if one passes the optometrist when going to the ice cream parlor and one needs one's glasses fixed, it might be a good idea to save some effort by combining the trips. Then, the pleasure results from both the ice cream and the fixed glasses.) Multiple goal planning can easily be included in Markovian decision models by adding the reward of the goal state to the costs of all (state, action, outcome)

triples that lead to the goal state. The "cost" of an action can be positive in this case. Another problem is that the rewards of the goal states are usually interdependent. That is, the reward of a goal state depends on which of the other goal states have already been achieved. (For example, it might be that the agent only receives pleasure from achieving a goal for the first time, and not every time it reaches the goal state.) In this case, one has to blow up the state space to include in the states the information which of the goals has already been achieved by the agent. We excluded these problems from our consideration and focussed our attention on the single-goal planning task, because we are interested in guaranteeing that a given task will eventually be achieved.

- Planning Time and Fetch Operations during Execution not Included

  The goodness of a plan is determined by the expected total reward of its execution. We do not take the time and other resources that are used for planning, metaplanning etc. into account, which could be done, for example, by optimizing the sum of the planning cost and the expected total execution reward (see for example [20, 19, 33, 52, 53, 61]). That is, we do not trade off the cost of additional planning and the expected utility that could be gained (the increase of the expected total reward of the plan execution). The reason is that planning is performed off-line in our model. Once the planner has compiled a plan, this plan can be used repeatedly in the execution phase. The execution time will usually not be dominated by the plan fetch operations, since it takes virtually no time to determine which action of the universal plan to execute.

- Time Separable Value Function and Risk Neutrality

  We defined the expected reward of the execution phase as the sum of the costs of the actions executed plus the reward of the goal state in which the execution is stopped. This is the measure of plan goodness that has been used so far by almost all AI researchers in the area of decision theoretic planning. Its simplicity allows for elegant planning methods, but it has some inherent assumptions, that are rather restrictive: It is assumed that all costs and rewards are expressible as (one-dimensional) real values in the same unit, and that they additively determine the utility of the plan ("time separable value function"). This implies that the agent is risk neutral and that time constraints (deadlines) and other resource constraints do not exist. With regard to the special resource time, for example, it is assumed that the agent is indifferent at which point in time a certain cost occurs and that it does not have to obey deadlines. All

of these restrictions could be relaxed by using methods from decision theory [82] (for example opportunity costs or indifference curves from multi-attribute utility theory), thereby making the planning problem much harder. One could easily incorporate risk-aversity [57] or a very restricted form of opportunity costs (by using discounting [54]) into Markovian decision algorithms. Using a discount factor for modeling risk-aversity is sometimes justified with the argument that a future return is uncertain and therefore should not receive as much weight as the same return at an earlier point in time. (The real reason for this ad-hoc rule seems to be that the mathematics becomes easier for discounted Markovian decision problems, since then the total reward is always finite, which is in general not true for undiscounted Markovian decision problems.) This justification is not applicable for our model: If a discount factor is used, not only the rewards will decrease, but also the absolute values of the costs, although a risk-aversive decision maker would lower the rewards and increase the absolute values of the costs. In business administration, a discount factor is the inverse of the interest rate (the price for a scare resource, e.g. money, in the equilibrium of supply and demand) plus one. It models the possibility for the decision maker to let someone else utilize the resource temporarily (in case of money, for example by depositing it in a savings account) and receive interest instead of using the resource himself. Thus, the discount factor corresponds to the opportunity cost of a resource. The analogue of a discount factor for our planning model is difficult to imagine.

B The Simple Policy-Iteration Algorithm Maintains Ergodicity

The simple policy-iteration algorithm can be used to solve completely ergodic pdt Markovian decision problems. In this chapter, we will show that it can also be used to find an optimal policy for an arbitrary pdt Markovian decision problem if the initial policy is ergodic. This will be done by proving that every iteration will again produce an ergodic policy, if the old policy is ergodic. Thus, every policy will be ergodic, and the non-ergodic policies that might exist will never be encountered. This result depends on the special structure of pdt Markovian decision problems and does not hold in general for Markovian decision problems.

We prove by contradiction that one iteration of the simple policy-iteration algorithm (i.e. the execution of the value-determination operation followed by the policy-improvement routine) again produces an ergodic policy for a pdt Markovian decision problem if it is started with

an ergodic policy. The theorem then follows by induction.

Given an ergodic policy f. Assume that the policy determined by the policy-improvement routine, call it f', is not ergodic. Then, there must exist an ergodic set that does not contain the state $\bar{s}$. Call this set E. Since E is finite, there exists (at least) one element of E, call it e, such that the v value of e as determined by the value-determination operation is at least as large as the v values of the other elements in E. Since only states in E are reachable from state e under policy f', it holds that

$$\begin{aligned}
\bar{c}(e, f'(e)) + \sum_{s' \in \bar{S}} \bar{p}(s'|e, f'(e))v(s') &\leq \bar{c}(e, f'(e)) + \sum_{s' \in \bar{S}} \bar{p}(s'|e, f'(e))v(e) \\
&= \bar{c}(e, f'(e)) + v(e) \sum_{s' \in \bar{S}} \bar{p}(s'|e, f'(e)) \\
&= \bar{c}(e, f'(e)) + v(e) \\
&< v(e) \\
&= \bar{c}(e, f(e)) + \sum_{s' \in \bar{S}} \bar{p}(s'|e, f(e))v(s')
\end{aligned}$$

But then the policy-improvement routine would have chosen $f(e)$ over $f'(e)$ as the action to execute in state e, which is a contradiction.

(An alternative proof is to show that the theorem is a simple corollary of the correctness proof for the multiple-chain policy iteration algorithm [54]: If the initial policy is ergodic; then the multiple-chain policy iteration algorithm collapses to (i.e can be rewritten as) the simple policy-iteration algorithm. We know that every iteration of the multiple-chain policy-iteration algorithm never decreases the gain of any state. For every ergodic policy of a pdt Markovian decision problem it holds that the gain of every state is zero, the maximal gain possible. Thus, the gain of every state will remain zero. But there is only one ergodic set that has this gain, the set $\{\bar{s}\}$. Thus, the new policy will again be ergodic.)

C The Value-Iteration Algorithm Maintains Ergodicity

We will show how the value-iteration algorithm can be used in the second phase of the two-step planner to solve a pdt Markovian decision problem instead of the policy-iteration algorithm. We have already shown in chapter 12 how to find and eliminate all unsolvable states. Since for

the value-iteration algorithm the action assigned to a state is only a byproduct of updating the v value of that state, we need to find initial values (instead of an ergodic plan) that guarantee that the plans produced remain ergodic. We will prove that if the v values satisfy the property $v(s) \leq \bar{c}(s, f(s)) + \sum_{s' \in \bar{S}} \bar{p}(s'|s, f(s))v(s')$ $(s \in S)$ and the alternatives considered when updating the decision of a state always contain its current decision, then the v values will satisfy the same inequality after one iteration of the value-iteration algorithm and an ergodic policy will have been determined. Thus, per induction one can show that every policy determined will be ergodic. The result depends on the special structure of pdt Markovian decision problems and does not hold in general for Markovian decision problems.

The following proof is stated in a general way. It holds no matter how many states are updated during an iteration of the value-iteration algorithm or how many alternatives are looked at to update the value of a state.

We use induction on the number of iterations of the value-iteration algorithm to show the lemma that $v(s) \leq \bar{c}(s, f(s)) + \sum_{s' \in \bar{S}} \bar{p}(s'|s, f(s))v(s')$ always holds for all $s \in S$. We required it to be true for the initial v values and decisions for all $s \in S$. In the following, we will denote the v value (the decision) of a state s before the iteration with $v_0(s)$ $(f_0(s))$ and after the iteration with $v_1(s)$ $(f_1(s))$. If during some iteration every $s' \in S' \subseteq S$ is updated and for each updated state s' the alternatives considered contain $f_0(s')$, then it holds that $v_1(s') = \bar{c}(s', f_1(s')) + \sum_{s'' \in \bar{S}} \bar{p}(s''|s', f_1(s'))v_0(s'') \geq \bar{c}(s', f_0(s')) + \sum_{s'' \in \bar{S}} \bar{p}(s''|s', f_0(s'))v_0(s'') \geq v_0(s')$. For every (non-updated) $s \in S \setminus S'$, it holds that $v_1(s) = v_0(s)$. Thus, it holds in general that the v values are non-decreasing. Then, for $s' \in S'$ it holds that $v_1(s') = \bar{c}(s', f_1(s')) + \sum_{s'' \in \bar{S}} \bar{p}(s''|s', f_1(s'))v_0(s'') \leq \bar{c}(s', f_1(s')) + \sum_{s'' \in \bar{S}} \bar{p}(s''|s', f_1(s'))v_1(s'')$. Likewise, for $s \in S \setminus S'$ it holds that $v_1(s) = v_0(s) \leq \bar{c}(s, f_0(s)) + \sum_{s'' \in \bar{S}} \bar{p}(s''|s, f_0(s))v_0(s'') = \bar{c}(s, f_1(s)) + \sum_{s'' \in \bar{S}} \bar{p}(s''|s, f_1(s))v_0(s'') \leq \bar{c}(s, f_1(s)) + \sum_{s'' \in \bar{S}} \bar{p}(s''|s, f_1(s))v_1(s'')$. Thus, the lemma holds. At the same time, we have seen that $v_0(s) \leq v_1(s)$ and $v(s) \leq \max_{a \in \bar{A}(s)}(\bar{c}(s, a) + \sum_{s' \in \bar{S}} \bar{p}(s'|s, a)v(s')$ for all $s \in S$. (As an aside, per induction it follows that $v_0(s) \leq v_n(s)$ and thus $v_0(s) \leq v(s)$, where $v(s)$ is the v value of s under an optimal policy.)

Now, we are going to show by contradiction that f_1 is ergodic. Assume it is not. Then we have an ergodic set that does not contain the state $\bar{s}$. Call this non-empty set E. Since E is finite, there exists (at least) one element of E, call it e, such that the v_0 value of e is at least as large as the v_0 values of the other elements in E. Since only states in E are reachable from

state e under the new policy, it holds that

$$\begin{aligned} v_1(e) &\leq \bar{c}(e, f_1(e)) + \sum_{s' \in \bar{S}} \bar{p}(s'|e, f_1(e)) v_0(s') \\ &\leq \bar{c}(e, f_1(e)) + \sum_{s' \in \bar{S}} \bar{p}(s'|e, f_1(e)) v_0(e) \\ &= \bar{c}(e, f_1(e)) + v_0(e) \sum_{s' \in \bar{S}} \bar{p}(s'|e, f_1(e)) \\ &= \bar{c}(e, f_1(e)) + v_0(e) \\ &< v_0(e) \end{aligned}$$

This is a contradiction, since we have already shown that $v_0(e) \leq v_1(e)$. Therefore, f_1 must be ergodic. This concludes the proof of the theorem.

We showed above that the initial v value of a state is smaller than the v value of the same state under an optimal policy. In figure 32 we show that this is not a sufficient criterion for the policies to remain ergodic even if the policy determined by the iteration of the value-iteration algorithm is ergodic. (a) shows the planning problem. There are four states. Their v values are written inside the circles. State 1 is the absorbing state. State 4 is the only state which has two actions to choose from. In this example, all actions are deterministic. They are annotated with the transition probabilities and costs. The actions forming the current policy have a black arrowhead. The optimal policy and the v values for the states under this policy are shown in (d). The v values of the states in (a) are clearly smaller. (b) is the result after iterating the value-iteration algorithm on (a) once. The policy determined is ergodic. After iterating the value-iteration algorithm once more, we reach a non-ergodic policy as shown in (c).

D A Correctness Proof for the General Greedy Algorithm

We prove that the following greedy algorithm, when given a pdt Markovian decision problem, outputs all solvable states and a policy that solves them.

1. Set $X := \bar{S}$, and $f(\bar{s}) := \bar{a}$.
2. Set $X' := \{\bar{s}\}$.

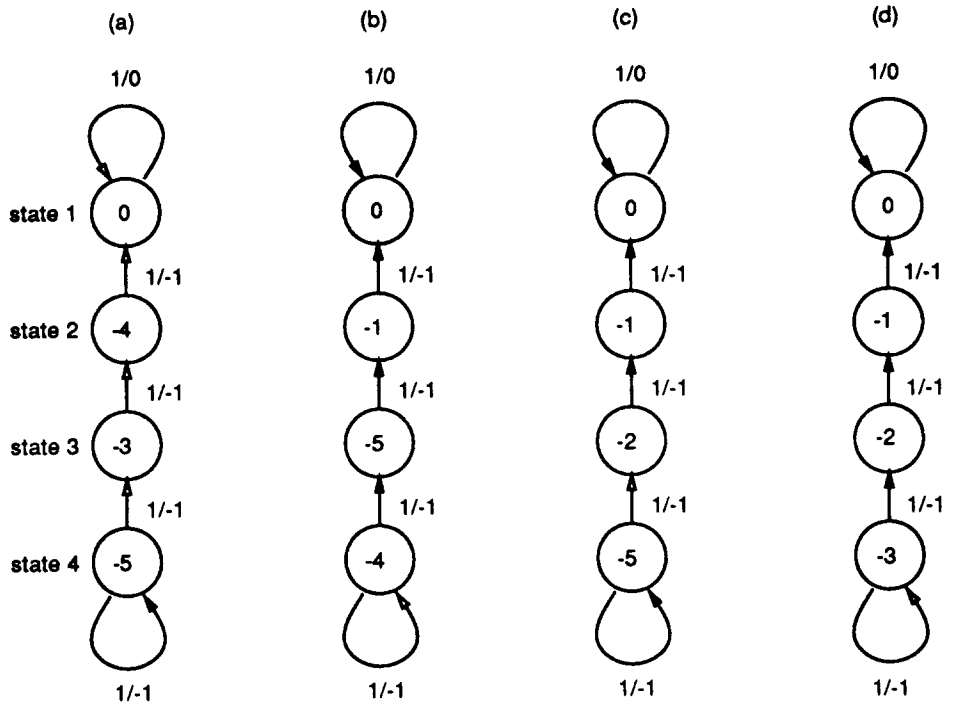


Figure 32: A Counter Example

3. Try to find an $s \in X \setminus X'$ and an $a \in \bar{A}(s)$ such that $\exists s' \in X' : \bar{p}(s'|s,a) > 0$, and $\forall s' \in \bar{S} \setminus X : \bar{p}(s'|s,a) = 0$.

4. If such an s and an a could be found, then set $X' := X' \cup \{s\}$, set $f(s) := a$ and go to 3.

5. If $X \neq X'$ then set $X := X'$, and go to 2.

6. Output X, output $f(s)$ for all $s \in X$, and stop.

First, we show that the inner loop (lines 3 and 4) terminates. X' is set to $\{\bar{s}\}$ on line 2, and whenever a backward jump from line 4 to line 3 is taken, a new element is added to X'. But X' is always a subset of X, which itself is always a subset of $\bar{S}$, a finite set. Therefore, the cardinality of X' is bounded by a constant from above. Thus, the inner loop terminates.

As a lemma we prove that whenever the beginning of line 5 is reached, X' contains a state s if and only if s is in X and there exists a policy f'_s and a sequence of states $s = s_1, s_2, \ldots, s_n = \bar{s}$ such that $n \in \{1,2,3,...\}$ and $\forall i \in \{1,2,\ldots,n-1\} : (\bar{p}(s_{i+1}|s_i, f'_s(s_i)) > 0 \wedge \forall s' \in \bar{S} \setminus X : \bar{p}(s'|s_i, f'_s(s_i)) = 0)$. Furthermore, for every state in X', f is such a policy.

We prove the "only if" direction of the lemma by induction. For $X' = \{\bar{s}\}$ it is true. Whenever we decide to add a state $s \in X$ to X' and to set $f(s) := a$ on line 4, we have checked before on line 3 that $\exists s' \in X' : \bar{p}(s'|s, f(s)) > 0$, and $\forall s'' \in \bar{S} \setminus X : \bar{p}(s''|s, f(s)) = 0$. By assumption, there exists a sequence of states $s' = s_1, s_2, \ldots, s_n = \bar{s}$ such that $n \in \{1,2,3,...\}$ and $\forall i \in \{1,2,\ldots,n-1\} : (\bar{p}(s_{i+1}|s_i, f(s_i)) > 0 \wedge \forall s'' \in \bar{S} \setminus X : \bar{p}(s''|s_i, f(s_i)) = 0)$. Then, the sequence of states $s, s', s_2, \ldots, s_n$ has the required property.

We prove the "if" direction of the lemma by contradiction. Assume that when the beginning of line 5 is reached, X' does not contain s, but s is in X and there exists a policy f'_s for which there exists a sequence of states $s = s_1, s_2, \ldots, s_n = \bar{s}$ such that $n \in \{1,2,3,...\}$ and $\forall i \in \{1,2,\ldots,n-1\}$: $(\bar{p}(s_{i+1}|s_i, f'_s(s_i)) > 0 \wedge \forall s' \in \bar{S} \setminus X$: $\bar{p}(s'|s_i, f'_s(s_i)) = 0)$. Let j be the largest index such that $s_j \notin X'$. Such an index exists, since at least one element of the sequence of states is not in X', namely s. Then, s_{j+1} is element of X'. Since $\bar{p}(s_{j+1}|s_j, f'_s(s_j)) > 0 \wedge \forall s' \in \bar{S} \setminus X : \bar{p}(s'|s_j, f'_s(s_j)) = 0$, the test on line 3 would have succeeded, and s_j would have been added to X'. This is a contradiction.

Next, we show that the outer loop (lines 2 to 5) terminates. X is set to $\bar{S}$ on line 1, and whenever the backward jump from line 5 to line 2 is taken, X is set to X', a strict subset of X, i.e. at least one element is removed from X. But X is always a superset of $\{\bar{s}\}$. Since the cardinality of X is bounded by one from below and the inner loop terminates, the outer loop terminates as well.

Now we prove the "if" direction of the main theorem, namely, that an element of X (as printed out) is solved under policy f (as printed out). Of course, this shows at the same time that these states are solvable. Combining the lemma from above with the fact that X' equals X after the last inner loop has been executed before the algorithm stops, we know that under policy f from every state in X only states in X can be reached and that for every state in X the state $\bar{s}$ is one of the reachable states. Thus, f solves every state in X.

It remains to show that if s is solvable, then s is an element of X (and thus printed out) when the algorithm stops. This is the "only if" direction of the main theorem. We show by induction that at every point in time $\bar{S} \setminus X$ contains only unsolvable states. The theorem then follows immediately. For $X = \bar{S}$ this is true, because $\bar{S} \setminus X = \emptyset$. Whenever a backward jump from line 5 to line 2 is taken, the elements of $X \setminus X'$ are eliminated from X. According to the lemma from above, for each such eliminated state s and each possible alternative for s either the state $\bar{s}$ cannot be reached from s or a state in $\bar{S} \setminus X$, which is according to the

induction assumption unsolvable, can be reached. Thus, s is unsolvable and can safely be eliminated from X.

E Further Research

In this report, we have assumed that the agent is always able to determine exactly the state it is in during plan execution, and that this operation is free and takes no execution time. Both assumptions are true for human beings if they solve an easy task such as stacking toy blocks on a table. There, a human agent receives, no matter which plan he pursues, a constant stream of visual information for free, that correctly identifies the configuration of the blocks on the table. But for more complicated tasks both assumptions do no longer hold. The human visual system can be fooled, and the person has to focus his or her attention on the pattern in the visual input stream that he or she thinks are relevant.

In robotics, this becomes even more obvious, since the robot has to execute actions to change the world ("physical actions") or to receive information about the state of the world ("perceptual actions"). Perceptual actions behave in many ways like physical actions. Both take time to execute and consume resources. Both of them can be unreliable, but in both cases we assume that the probability distribution over the successor states is known and depends only on the state the agent is in and the action it executes. (Note that in robotics many macro actions are hybrids, i.e. mixed sequences of perceptual and physical actions. For example, in order to sense a particular object at given coordinates, the camera has to be moved to new coordinates and pointed into the right direction, before the sensing can be done. The physical move action can change the configuration of the world by accident.)

The traditional approach in the AI literature was to find out enough about the current state by performing perceptual actions in order to be able to pick the correct physical action. Another approach was to reduce the uncertainty not with perceptual actions, but with physical actions that achieve a unique state no matter in which state the actions were executed [43]. A third approach was to execute physical actions that randomize the state, i.e. no matter in which state they were executed, they lead to a unique probability distribution over the states [34] (an example for this unusual approach is shaking a sieve).

We can adapt the pdt Markov model to such a planning situation:

We distinguish two kinds of states. At each point in time the world is in a certain state, namely the state that is manipulated by the physical actions. The state of the world can be changed by physical actions only. Since the agent does not know the state of the world for sure, at each point in time there is also the state of belief the agent has about the world. Such a state of belief is a probability distribution over the possible states of the world. It is changed both by perceptual and physical actions. Note that even if there are only a finite number of states in the world, there will be an infinite, uncountable number of states of belief about the world.

Since the agent does not gain information about the state of the world by executing a physical action, such an actions maps each state of belief into exactly one state of belief. (The agent does not receive feedback which of the possible successor states resulted from the execution of the physical action.) Perceptual actions map a state of belief into several successor states. For each of the possible outcomes of the perceptual action there is a different successor state. (Perceptual actions implement a conditional, because they provide the agent with new, but not necessarily accurate information.)

The new planning problem can be translated into a partially observable Markovian decision problem and then be solved using Sondik's Markovian decision algorithm, see for example [100, 96, 101, 69, 51, 66] for algorithms and [78] for their complexity.

We are interested in how the notions that we have developed in this report for (totally observable) Markovian decision problems (planning domains in which the current world state can be determined exactly without cost or time) carry over to partially observable Markovian decision problems (planning domains in which sensing of the current world state is error prone, costly and time consuming.)

References

[1] Phil Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the AAAI 1987*, pages 268–272.

[2] Leemon C. Baird and Ronald J. Williams. A mathematical analysis of actor-critic architectures for learning optimal controls through incremental dynamic programming. In *Proceedings of the Sixth Yale Workshop on Adaptive and Learning Systems 1990*, pages 96–101.

[3] Dana H. Ballard, Richard S. Sutton, and Steven D. Whitehead. Advances in reinforcement learning and their implications for intelligent control. In *Proceedings of the Fifth IEEE International Symposium on Intelligent Control 1990*, pages 1289–1297.

[4] Dana H. Ballard and Steven D. Whitehead. Learning to perceive and act. Technical Report 331, Department of Computer Science, University of Rochester, (revised) 1990.

[5] Andrew G. Barto and Satinder Pal Singh. On the computational economics of reinforcement learning. In *Connectionist Models: Proceedings of the Summer School 1990.*

[6] Andrew G. Barto and Satinder Pal Singh. Reinforcement learning and dynamic programming. In *Proceedings of the Sixth Yale Workshop on Adaptive and Learning Systems 1990*, pages 83–88.

[7] Andrew G. Barto, Richard S. Sutton, and Christopher J. Watkins. Learning and sequential decision making. Technical Report 89-95, Department of Computer and Information Science, University of Massachusetts at Amherst, 1989.

[8] Richard E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, New Jersey, 1957.

[9] Hans Berliner. On the construction of evaluation functions for large domains. In *Proceedings of IJCAI 1979*, pages 53–55.

[10] Dimitri P. Bertsekas. *Dynamic Programming.* Prentice-Hall, New York, New York, 1987.

[11] Mark Boddy and Thomas Dean. An analysis of time-dependent planning. In *Proceedings of the AAAI 1988*, pages 49–54.

[12] Mark Boddy and Thomas Dean. Solving time-dependent planning problems. In *Proceedings of the IJCAI 1989*, pages 979–984.

[13] Ronald J. Brachman. On the epistemological status of semantic networks. In R. J. Brachman and H. J. Levesque, editors, *Readings in Knowledge Representation.* Morgan Kaufmann, San Mateo, California, 1985.

[14] John S. Breese and Michael R. Fehling. A computational model for decision-theoretic control of problem-solving under uncertainty. Technical Report 837-88-5, Rockwell International Science Center, Palo Alto, 1988.

[15] John Bresina and Mark Drummond. Anytime synthetic projection: Maximizing the probability of goal satisfaction. In *Proceedings of the AAAI 1990*, pages 138–144.

[16] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–378, 1987.

[17] Lonnie Chrisman and Reid Simmons. Senseful planning: Focusing perceptual attention. In *Proceedings of the AAAI 1991*, pages 756–761, 1991.

[18] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* Springer-Verlag, New York, New York, 1981.

[19] Gregory F. Cooper, David E. Heckerman, and Eric J. Horvitz. Reflection and action under scarce resources: Theoretical principles and empirical study. In *Proceedings of the IJCAI 1989*, pages 1121–1127.

[20] Bruce D'Ambrosio and Michael R. Fehling. Constrained rational agency. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics 1990.*

[21] Randall Davis and Reid G. Simmons. Generate, test, and debug: Combining associational rules and causal models. In *Proceedings of the IJCAI 1987*, pages 1071–1078.

[22] Peter Dayan. Q-learning. unpublished manuscript, University of Edinburgh, 1990.

[23] Thomas Dean. Decision-theoretic control of inference for time-critical applications. Technical Report CS-89-44, Department of Computer Science, Brown University, (revised) 1990.

[24] Thomas Dean and Keiji Kanazawa. A model for projection and action. In *Proceedings of the IJCAI 1989*, pages 985–990.

[25] Thomas Dean and Keiji Kanazawa. Probabilistic temporal reasoning. In *Proceedings of the AAAI 1988*, pages 524–528.

[26] Thomas Dean and Keiji Kanazawa. A model for reasoning about persistence and causation. Technical Report CS89-04, Department of Computer Science, Brown University, 1989.

[27] Jon Doyle. Rationality and its roles in reasoning. In *Proceedings of the AAAI 1990*, pages 1093–1100.

[28] Jon Doyle. Artificial Intelligence and rational self-government. Technical Report CMU-CS-88-124, School of Computer Science, Carnegie Mellon University, 1988.

[29] Jon Doyle. On rationality and learning. Technical Report CMU-CS-88-122, School of Computer Science, Carnegie Mellon University, 1988.

[30] Jon Doyle. Similarity, conservatism, and rationality. Technical Report CMU-CS-88-123, School of Computer Science, Carnegie Mellon University, 1988.

[31] Mark Drummond. Situated control rules. In *Proceedings of the Rochester Planning Workshop 1989*, pages 18–33.

[32] Mark Drummond and Austin Tate. AI planning: A tutorial and review. Technical Report AIAI-TR-30, AI Applications Institute, University of Edinburgh, 1989.

[33] David Einav and Michael R. Fehling. Resource-constrained search. In *Proceedings of the IEEE Conference on Systems, Man, and Cybernetics 1990.*

[34] Michael A. Erdmann. *On Probabilistic Strategies for Robot Tasks.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachussetts Institute of Technology, 1989.

[35] Oren Etzioni. Tractable decision-analytic control. Technical Report CMU-CS-89-119, School of Computer Science, Carnegie Mellon University, 1989.

[36] Jerome R. Feldman and Robert F. Sproull. Decision theory and Artificial Intelligence II: The hungry monkey. *Cognitive Science*, 1(2):158–192, 1977.

[37] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288, 1972.

[38] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[39] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, New York, New York, 1979.

[40] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, Palo Alto, California, 1987.

[41] Michael P. Georgeff. Reasoning about plans and actions. In Howard E. Shrobe, editor, *Exploring Artificial Intelligence*, pages 173–196. Morgan Kaufmann, San Mateo, California, 1988.

[42] Matthew L. Ginsberg. Universal planning: An (almost) universally bad idea. *AI Magazine*, 10(4):40–44, 1989.

[43] Kenneth Y. Goldberg and Matthew T. Mason. Bayesian grasping. In *Proceedings of the IEEE International Conference on Robotics and Automation 1990*, pages 1264–1269.

[44] Peter Haddawy and Steve Hanks. Issues in decision-theoretic planning: Symbolic goals and numeric utilities. In *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control 1990*, pages 48–58.

[45] Steve Hanks. Controlling inference in planning systems: Who, what, when, why, and how. Technical Report 90-04-01, Department of Computer Science and Engineering, University of Washington, 1990.

[46] Steve Hanks. *Projecting Plans for Uncertain Worlds.* PhD thesis, Department of Computer Science, Yale University, 1990.

[47] Othar Hansson and Andrew Mayer. Heuristic search as evidential reasoning. In *Proceedings of the Fifth Workshop on Uncertainty in AI 1989*, pages 152–161.

[48] Othar Hansson and Andrew Mayer. The optimality of satisficing solutions. In *Proceedings of the Workshop on Uncertainty in Artificial Intelligence 1988*, pages 148–157.

[49] Othar Hansson, Andrew Mayer, and Stuart J. Russell. Decision-theoretic planning in BPS. In *Proceedings of the AAAI Spring Symposium 1990.*

[50] P. Hayes and John McCarthy. Some philosophical problems from the standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, Great Britain, 1968.

[51] Dale J. Hockstra. *Partially Observable Markov Decision Processes with Applications.* PhD thesis, Department of Operations Research and Statistics, Stanford University, 1973.

[52] Eric J. Horvitz. Reasoning about beliefs and actions under computational resource constraints. In *Proceedings of the Third Workshop on Uncertainty in Artificial Intelligence 1987.*

[53] Eric J. Horvitz. Reasoning under varying and uncertain resource constraints. In *Proceedings of the AAAI 1988*, pages 111–116.

[54] Ronald A. Howard. *Dynamic Programming and Markov Processes.* The M.I.T. Press, Cambridge, Massachusetts, 1964.

[55] Ronald A. Howard. Information value theory. *IEEE Transactions on Systems Science and Cybernetics*, SSC-2(1):22–26, 1966.

[56] Ronald A. Howard. The foundation of decision analysis. *IEEE Transactions on Systems Science, and Cybernetics*, 4:211–219, 1968.

[57] Ronald A. Howard and James E. Matheson. Risk-sensitive Markov decision processes. *Management Science*, 18(7):356–369, 1972.

[58] Ronald A. Howard and James E. Matheson. Influence diagrams. In *The Principles and Applications of Decision Analysis*, pages 721–762. Strategic Decisions Group, Menlo Park, California, 1981.

[59] Leslie P. Kaelbling and Stanley J. Rosenschein. The synthesis of machines with provable epistemic properties. In *Proceedings of the Conference on Theoretical Aspects of Reasoning about Knowledge 1986*, pages 83–98.

[60] Keiji Kanazawa. Thesis proposal: Tradeoffs in actions under uncertainty. Technical report, Department of Computer Science, Brown University, 1990. unpublished manuscript.

[61] Naiping Keng and David Y. Y. Yun. A planning/scheduling methodology for the constrained resource problem. In *Proceedings of the IJCAI 1989*, pages 998–1003.

[62] Daniel E. Koditschek. Exact robot navigation by means of potential functions. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1–6, 1987.

[63] Richard E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1), 1987.

[64] David M. Kreps. *Markov Decision Problems with Expected Utility Criteria.* PhD thesis, Department of Operations Research, Stanford University, 1975.

[65] Long-Ji Lin. Self-improving reactive agents: Case studies of reinforcement learning frameworks. Technical Report CMU-CS-90-109, School of Computer Science, Carnegie Mellon University, 1990.

[66] William S. Lovejoy. A note on exact solutions of partially observed Markov decison processes. Technical Report 1003, Graduate School of Business, Stanford University, 1988.

[67] Hisashi Mine and Shunji Osaki. *Markovian Decision Processes.* American Elsevier, New York, New York, 1970.

[68] Tom Mitchell. Becoming increasingly reactive. In *Proceedings of the AAAI 1990*, pages 1051–1058.

[69] George E. Monahan. A survey of partially observable Markov decision processes: Theory, models, and algorithms. *Management Science*, 28(1):1–16, 1982.

[70] R. Moore, Nils J. Nilsson, and M. Torrance. ACTNET: An action network language and its interpreter. Technical report, Computer Science Department, Stanford University, 1990.

[71] Oskar Morgenstern and John von Neumann. *Theory of Games and Economic Behavior.* Princeton University Press, Princeton, New Jersey, 1944.

[72] John Nafeh. *Markov Decision Processes with Policy Constraints.* PhD thesis, Department of Engineering-Economic Systems, Stanford University, 1976.

[73] Allen Newell, J. C. Shaw, and Herbert A. Simon. Report on a general problem-solving program. In *Proceedings of the International Conference on Information Processing 1960*, pages 256–264.

[74] Allen Newell and Herbert A. Simon. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[75] Nils J. Nilsson. Action networks. In *Proceedings of the Rochester Planning Workshop 1988*, pages 21–52.

[76] Nils J. Nilsson. Action networks. Technical report, Computer Science Department, Stanford University, 1989.

[77] Christos H. Papadimitriou. *Combinatorial Optimization; Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[78] Christos H. Papadimitriou and John N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.

[79] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Menlo Park, California, 1984.

[80] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems.* Morgan Kaufmann, San Mateo, California, 1988.

[81] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1, 1986.

[82] Howard Raiffa. *Decision Analysis: Introductory Lectures on Choices under Uncertainty.* Addison-Wesley, Reading, Massachusetts, 1968.

[83] Stanley J. Rosenschein. Formal theories of knowledge in AI and robotics. Technical Report CSLI-87-84, Center for the Study of Language and Information, Stanford University, 1987.

[84] Sheldon M. Ross. *Applied Probability Models with Optimization Applications.* Holden-Day, San Francisco, California, 1970.

[85] Sheldon M. Ross. *Introduction to Stochastic Dynamic Programming.* Academic Press, New York, New York, 1983.

[86] Stuart J. Russell. Execution architectures and compilation. In *Proceedings of the IJCAI 1989*, pages 15–20.

[87] Stuart J. Russell. Fine-grained decision-theoretic search control. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence 1990*, pages 436–442.

[88] Stuart J. Russell and Eric Wefald. On optimal game-tree search using rational metareasoning. In *Proceedings of the IJCAI 1989*, pages 334–340.

[89] Stuart J. Russell and Eric Wefald. Principles of metareasoning. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning 1989*, pages 400–411.

[90] Stuart J. Russell and Eric Wefald. Decision-theoretic control of reasoning: General theory and an application to game-playing. Technical Report UCB/CSD 88/435, Computer Science Department, University of California at Berkeley, 1988.

[91] Earl D. Sacerdoti. The nonlinear nature of plans. In *Proceedings of the IJCAI 1975*, pages 206–214.

[92] Earl D. Sacerdoti. *A Structure for Plans and Behavior.* American Elsevier, New York, New York, 1977.

[93] M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the IJCAI 1987*, pages 1039–1046.

[94] Ross D. Shachter. Evaluating influence diagrams. *Operations Research*, 34(6):871–882, 1986.

[95] Reid G. Simmons. A theory of debugging plans and interpretations. In *Proceedings of the AAAI 1988*, pages 94–99.

[96] Richard D. Smallwood and Edward J. Sondik. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research*, 21(5):1071–1088, 1973.

[97] David E. Smith. Controlling inference. Technical Report STAN-CS-86-1107, Department of Computer Science, Stanford University, 1986.

[98] David E. Smith. A decision-theoretic approach to the control of planning search. Technical Report LOGIC-87-11, Department of Computer Science, Stanford University, 1988.

[99] David E. Smith. Controlling backward inference. In *Artificial Intelligence*, volume 39 of *2*, 1989.

[100] Edward J. Sondik. *The Optimal Control of Partially Observable Markov Processes.* PhD thesis, Department of Electrical Engineering, Stanford University, 1971.

[101] Edward J. Sondik. The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations Research*, 26(2):282–304, 1978.

[102] Gerald J. Sussman. A computational model of skill acquisition. Technical Report AI-TR-297, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1973.

[103] Richard S. Sutton. First results with DYNA, an integrated architecture for learning, planning and reacting. In *Proceedings of the AAAI Spring Symposium 1990.*

[104] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning 1990*, pages 216–224.

[105] Richard S. Sutton. *Temporal Credit Assignment in Reinforcement Learning.* PhD thesis, Department of Computer and Information Science, University of Massachusetts at Amherst, 1984.

[106] D. H. Warren. Generating conditional plans and programs. In *Proceedings of the AISB Summer Conference 1976*, pages 344–454.

[107] Christopher J. Watkins. *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge University, 1989.

[108] Michael P. Wellman. Exploiting functional dependencies in qualitative probabilistic reasoning. In *Proceedings of the Sixth Conference on Uncertainty in AI 1990*, pages 2–9.

[109] Michael P. Wellman. The STRIPS assumption for planning under uncertainty. In *Proceedings of the AAAI Spring Symposium 1990*, pages 146–150.

[110] Michael P. Wellman. *Formulation of Tradeoffs in Planning under Uncertainty.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

[111] Michael P. Wellman. Graphical inference in qualitative probabilistic networks. *Networks*, 20(5):687–701, 1990.

[112] Steven D. Whitehead. Thesis proposal: Scaling reinforcement learning systems. Technical Report 304, Department of Computer Science, University of Rochester, 1989.

[113] Ronald J. Williams. Toward a theory of reinforcement-learning connectionist systems. Technical Report NU-CCS-88-3, College of Computer Science, Northeastern University, 1988.