Introduction to Automated Planning

(Draft)

Jussi Rintanen
Albert-Ludwigs-Universität Freiburg, Institut für Informatik
Georges-Köhler-Allee, 79110 Freiburg im Breisgau
Germany

February 15, 2005

Foreword

These are the lecture notes of the AI planning course at the Albert-Ludwigs-University Freiburg in summer term 2004, based on earlier notes for the course in winter 2002/2003.

Many thanks for comments and corrections to students who have participated the planning course, including Slawomir Grzonka, Bernd Gutmann and Martin Wehrle.

Contents

List of Figures

Chapter 1

Introduction

1.1 What is AI planning?

- modeling decision making needed by intelligent creatures acting in a complicated environment
- development of efficient algorithms for such decision making
- emphasis on general-purpose problem representation and general-purpose solution techniques; alternative would be to derive tailored algorithms for every problem separately

Impediments for the success of AI in producing genuinely intelligent beings are related to perceiving and representing knowledge concerning the world. The real world is very complicated in its all physical and geometric as well as social aspects, and representing all the knowledge required by an intelligent being may be too inflexible and complicated by the logical and symbolical means almost exclusively used in artificial intelligence and in planning. This has been criticized by many researchers [Brooks, 1991] and is a topic of continuing scientific debate as the problem is not well understood.

AI planning, and knowledge representation techniques in AI in general, are best applicable to restricted domains in which it is easy to identify what the atomic facts are and to exactly describe how the world behaves. These properties are best fulfilled by systems that are completely man-made, or systems in which planning needs to consider only at a very abstract level what is happening in the world.

Examples of completely man-made systems to which planning techniques have successfully been applied are given in the next section. This includes applications of planning in autonomous spacecraft.

A simple real-world application in which abstracting away the details of the real world would be transportation planning: how to get from Freiburg to London by public transportation, trains, airplanes and buses. If a robot were capable of finding its way between the couple of hundred of meters between the various forms of transportation and recognize the trains and buses to board it could easily travel all over the world. Planning what transportation to use is an easy problem in this case.

1.2 Where is AI planning used?

Truly intelligent robots or other artificial intelligent beings do not exist yet, and planning, like most other work on artificial intelligence, is still very much still something that takes place in research labs only.

Perhaps the most visible application of AI planning has been experimentation with autonomous spacecraft by the U. S. space agency NASA [Muscettola *et al.*, 1998].

At the level of applied AI research, AI planning is being used by many research projects that have produced autonomous but not very intelligent robots doing simple routine tasks in restricted environments, like delivering mail in an office or distributing medicine in a hospital. The uses of planning algorithms in this kind of setting, however, employ only very little from the potential of AI planning.

1.3 Types of planning problems

The word *planning* is very general, and denotes very many different things. Even in the AI and robotics context there are many types of planning, related to each other, but having different flavors.

The first problem in controlling autonomous robots, just their basic movement from one location to another, and the movement of the limbs, possibly with the ability to grip objects and move them, so called manipulators, is a very challenging problem. These problems are called *path planning* and *motion planning*, and they are not discussed in this lecture, as they require specialized representations of the geometric properties of the world, and cannot usually be efficiently represented in the general state-based model we are interested in. There is also the very well established research area of *scheduling* which is concerned with ordering and choosing a schedule for executing a number of predefined actions.

The more abstract planning that is the topic of this lecture is sometimes called task planning to distinguish it from the more geometric and physical forms of planning used in controlling the movements of robots and similar systems.

Even within task planning, there are many different types of planning problems, depending on the assumptions concerning the properties of actions and of the world that are made. Some of these are the following.

1. Determinism versus nondeterminism.

   In the simplest form of planning the state of the world at any moment is unambiguously determined by the initial state of the world and the sequence of actions that have been taken. Hence the world is completely deterministic.

   The assumption of a deterministic world holds when planning is to be applied in a sufficiently restricted setting. However, when the world is modeled in more detail and more realistically, the assumption does not hold any more: the plans have to take into account events that take place independently of the actions and also the possibility that the effects of an action are not the same every time the action is taken, even when the world appears to be the same.

   Nondeterminism comes from at least two different sources.

First, the model of the world is usually very incomplete, and things that are possible as far as our beliefs are concerned can be viewed as a form of nondeterminism: we do not know whether somebody is going to phone or visit us, and then the visit or phone call can be modeled as a nondeterministic event that may or may not take place.

Second, many actions themselves are by their nature nondeterministic, either intentionally or unintentionally. Throwing two dice has 12 possible outcomes that usually cannot be predicted (which is why throwing dice is interesting!) Throwing some object to a garbage bin from a distance may or may not succeed.

Notice that there is still the possibility that the physical universe is completely deterministic, but as long as we do not know the exact causes of events, we might just as well consider them nondeterministic.

2. Observability.

   For deterministic planning problems with one initial state there is no need to consider observations, because the goals can always be reached by one sequence of actions and the plan does not need to decide in the middle of plan execution between different courses of action.

   When the actions or the environment can be nondeterministic, or when there are several initial states, the notion of plans as a sequence of actions is not sufficient.

   There are two possibilities. Either planning is interleaved with plan execution: only one action is chosen at a time, it is executed, and based on the observations that are made the next action is chosen, and so on. Or a complete plan is generated, covering all possible events that can happen. This plan in the most general form has the structure of a program with branches (*if-then-else*) and loops.

   In both cases, what can be observed has a strong impact on how exactly the actual state of the world can be determined: the more facts can be observed, the more precisely the current state of the world can be determined, and the better the most appropriate action can be chosen. If there is a lot of uncertainty concerning the current state of the world it may be impossible to choose an appropriate action.

   If the current state can always been determined uniquely, we have *full observability*. If the current state cannot be determined uniquely we have *partial observability*, and planning algorithms are forced to consider sets of possible current states.

3. Time.

   Most work on planning uses discrete (integer) time and actions of unit duration. This means that all changes caused by an action taken at time point t are visible at time point $t + 1$. So changes in the world take only one unit of time, and what happens between two time points is not analyzed further.

   More complicated models of time and change are possible, but in this lecture we consider only discrete time. Most types of problems can be analyzed in terms of discrete time by making the unit duration sufficiently small. Rational and real time cause unnecessary conceptual difficulties. Effects of actions that are not immediate can easily be reduced to the basic case by encoding the delayed effects in the state description.

4. Control information and plan structure.

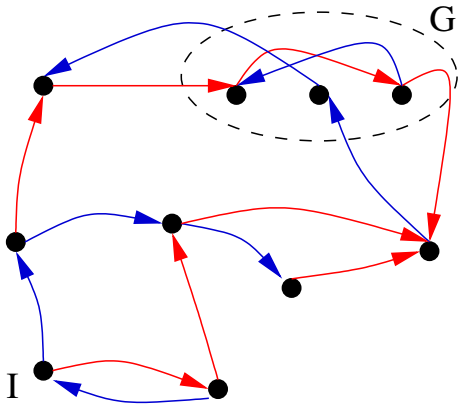


Figure 1.1: A deterministic planning problem

In the basic planning problem a plan is to be synthesized based on a generic description of how the actions affect the world.

There may be, however, further control information that may affect the planning process and the plans that are produced. In hierarchical planning, for example, information on the structure of the possible plans is given in the form of a hierarchical task network, and the plans that are produced must conform to this structure. This kind of structural information may substantially improve the efficiency of planning. Another way of restricting the structure of plans, for efficiency or other reasons, is the use of temporal logics [Bacchus and Kabanza, 2000].

5. Plan quality.

   The purpose of a plan is often just to reach one of the predefined goal states, and plans are judged only with respect to the satisfaction of this property.

   However, actions may have differing costs and durations, and plans could be assessed in terms of their time consumption or cost.

   In nondeterministic planning, because different executions of a plan produce different sequences of actions, plans can be valued in terms of their expected costs, best-case costs, worst-case costs, and probability of eventually reaching the goals.

   Plans with an infinite execution length can also be considered, and then plans may be valued according to their average cost per unit time, or according to their geometrically discounted costs.

1.4 Examples

Figure 1.1 illustrates what deterministic planning is. There is a set of states (the black dots), two actions (blue, red), an initial state I, and a set of goal states G. The task is to find a path from the initial state to one of the goal states. The planning problem is deterministic because in all states there is at most one red and at most one blue arrow going out of that state, which means that for all states the successor state is unambiguously determined by the action. In this example there are several possible plans for reaching G from I. Some of them are BRR (for blue, blue, red), RRRB,

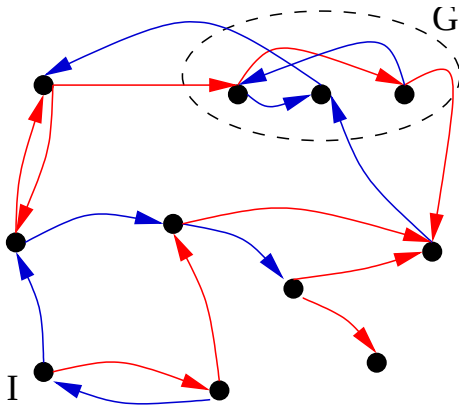


Figure 1.2: A nondeterministic planning problem

BRRRBRBRBRBRBRBRBR. The unique shortest plan is clearly BRR, as there are no plans of length 2 and not other plans of length 3.

Adding nondeterminism complicates planning, because even when there is a path from the initial state to a goal state, there is no guarantee that any single sequence of actions reaches the goal states, or even that the goals can be reached when choosing the next action can be postponed to the point when it is to be taken. A practical example is winning 1 euro in roulette given an initial capital of 5 euro. Whatever way the game is played, there is a relatively high probability of failure (not to mention that the expected outcome of the game is to lose money; don't do it!)

Figure 1.2 illustrates what a nondeterministic planning problem is. We have added a red arrow to the second state above the initial state, as well as a new state to the bottom right corner with a nondeterministic transition leading to it.

The new arrows make the red action nondeterministic, and have an impact on which action sequences are plans. First, nothing starting with BBBR or RRBR is a plan, because these action sequences might end in the bottom right state from which goal states cannot be reached. Second, if first the actions B and R are taken, the only way to proceed is to take the action R, and this either leads to a goal state, or takes us back toward the initial state. One possible plan would be to first take action B, and then repeat the sequence RR until a goal state has been reached.

Of course, even in this case there still are plans that do not involve nondeterministic steps, for example the plan RRRB.

A third possibility is planning with nondeterminism, but without the possibility of uniquely determining what the current state is. Figure 1.3 extends Figure 1.2 by restrictions on *observability*. Now during plan execution it is possible only to recognize the current state based on the color green or black. All black states are observationally indistinguishable from each other, and likewise all green states. We may be able to infer something about the current state based on the actions already taken and the observations made earlier, but these do not in general allow to infer the current state unambiguously.

The plan in the previous case consisting of the action B followed by iteration of RR until a goal state is reached cannot be executed any more, because the two states reached with the second R are indistinguishable.

However, for this problem there is a closely related plan that works also with partial observability. First take actions BR. Then repeat RR until the current state becomes black. So, it does not

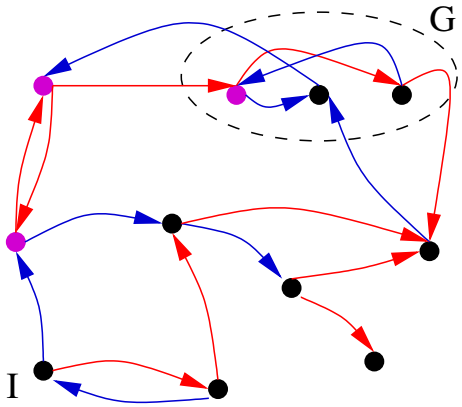


Figure 1.3: A nondeterministic planning problem with partial observability

simply suffice to reach a goal state, but one also has to be able to recognize that the goal state has been reached so that plan execution can terminate.

1.5 Related topics

Reasoning about action has emerged as a separate research topic with the goal of making inferences about actions and their effects [Ginsberg and Smith, 1988; Shoham, 1988; Sandewall, 1994a; 1994b; Stein and Morgenstern, 1994]. Important subtopics include the qualification and the ramification problems, which respectively involve deciding whether a certain action can be performed to have its anticipated effects and what are the indirect effects of an action. Both of these problems are of independent interest, both for their relations to the reasoning human beings do and for their importance in representing the world as required by any intelligent system for doing planning. In this lecture, however, we assume that a description of some actions is given, with all preconditions and direct and indirect effects fully spelled out, and concentrate on what kind of planning can be performed with these actions. The separation between planning and reasoning about actions is useful for both structuring systems that plan and act in a complicated world and for learning about these two topics.

Markov decision processes [Puterman, 1994] in operations research is essentially a formalization of planning. In contrast to AI planning, work in that area has used explicit enumerative representations of transition systems, like those used in Section 2.1, and as a consequence the algorithms have a different flavor than most planning algorithms do. However, most of the recent work on probabilistic planning, as discussed in Chapter 5, is based on Markov decision processes.

Discrete event systems (DES) in control engineering have been proposed as a model for synthesizing controllers for systems like automated factories [Ramadge and Wonham, 1987; Wonham, 1988], and this topic is closely related to planning. Again, there are differences in the problem formulation, with state spaces represented enumeratively or more succinctly, for example as Petri nets [Ichikawa and Hiraishi, 1988] or vector additions systems [Li and Wonham, 1993].

Synthesis of programs for reactive systems that work in nondeterministic and partially observable environments is similar to planning under same conditions. Program synthesis has been considered for example from specifications of their input-output behavior in different types of temporal logics [Vardi and Stockmeyer, 1985; Kupferman and Vardi, 1999].

1.6 Early research on AI planning

Research that has to current AI planning started in the 1960's in the form of programs that were meant simulate problem solving abilities of human beings.

One of the first programs of this kind was the General Problem Solver by Newell and Simon [Ernst *et al.*, 1969]. GPS performed state space search and used a heuristic that estimated differences between the current and goal states.

In the end of 1960's Green proposed the use of theorem-provers for constructing plans [Green, 1969]. However, because of the immaturity of theorem-proving techniques at that time, this approach was soon mostly abandoned in favor of specialized planning algorithms. There was theoretically oriented work on deductive planning that used different kinds of modal and dynamic logics [Rosenschein, 1981], but these works had little impact on the development of planning algorithms. Deductive and logic-based approaches to planning gained popularity again only in the end of 1990's as a consequence of the development of more sophisticated programs for the satisfiability problem of the classical propositional logic [Kautz and Selman, 1996].

The historically most influential planning system is probably the STRIPS planner from the beginning of the 1970's [Fikes and Nilsson, 1971]. The states in STRIPS are sets of formulae, and the operators change these state descriptions by adding and deleting formulae from the sets. Heuristics similar to the GPS system were used in guiding the search. The definition of operators, with a *precondition* as well as *add* and *delete* lists, corresponding to the literals that respectively become true and false, and the associated terminology, has been in common use until very recently. The add list is simply the set of state variables that the action makes true, and the delete list similarly consists of the state variables that become false.

Starting in the 1970's the dominating approach to domain-independent planning was the so-called partial-order, or causal link, or nonlinear planning, [Sacerdoti, 1975; McAllester and Rosenblitt, 1991], which remained popular until the mid-1990's and the introduction of the Graphplan planner [Blum and Furst, 1997] which started the shift away from partial-order planning to types of algorithms that had been earlier considered infeasible, even the then-notorious total-order planners. The basic idea of partial-order planning is that a plan is incrementally constructed starting from the initial state and the goals, by either adding an action to the plan so that one of the open goals or operator preconditions is fulfilled, or adding an ordering constraint on operators already in the plan in order to resolve potential conflicts between them. In contrast to the forward or backward search strategies in Chapter 3, partial-order planners tried to avoid unnecessarily imposing an ordering on operators too early. The main advantages of both partial-order planners and Graphplan are present in the SAT/CSP approach to planning which we will discuss in detail in Section 3.5.

In parallel to partial-order planning, the notion of hierarchical planning emerged [Sacerdoti, 1974], and it has been deployed in many real-world applications. The idea in hierarchical planning is that the problem description imposes a structure on solutions and restricts the number of choices the planning algorithm has to make. A hierarchical plan consists of a main task that can be decomposed to smaller tasks that are recursively solved. For each task there is a choice between solution methods. The less choice there is, the more efficiently the problem is solved. Furthermore, many hierarchical planners allow the embedding of problem-specific heuristics and problem-solvers to further speed up planning.

A collection of articles on AI planning starting from the late 1960's has been edited by Allen et al. [Allen *et al.*, 1990]. Many of the papers are mainly of historical interest, and some of them

outline ideas that are still very much in use today.

1.7 This book

My intention in writing these lecture notes was to cover planning problems of different generality and some of the most important approaches to solving each type of problem. It goes without saying that during the last several decades of planning research a lot of work has been done that are not covered by these notes.

Important differences to most textbooks and research papers on planning is that I use a unified and rather expressive syntax for representing operators, including nondeterministic and conditional effects. This has several implications on the material covered in this book. For example, many people may find it surprising that I do not use a concept viewed very central for deterministic planning by some, *the planning graphs* of Blum and Furst [1997]. This is a direct implication of the general syntax for operators I use, as discussed in more detail in Section 3.9. It seems that any graph useful graph-theoretic properties planning graphs have lose their meaning when a definition of operators more general than STRIPS operators is used.

One of the important messages of these notes is the importance of logic (propositional logic in our case) in representing many of the notions important to all forms of planning ranging from the simplest deterministic case to the most general types of planning with partial observability. As we will see, states, sets of states, belief states and transition relations associated with operators are in many cases represented most naturally as propositional formulae. This representation shows up once and again in connection of different types of planning algorithms, including backward search in classical/deterministic planning, planning as satisfiability, and in implementations of nondeterministic planning algorithms by means of binary decision diagrams.

In addition to generalizing many existing techniques to the more general definition of planning problems, many of the algorithms are either new or have been developed further from earlier algorithms. I cite the original sources in the literature sections in the end of every chapter. Some of my contributions can be singled out rather precisely. They include the following.

1. The definition of regression for conditional and nondeterministic operators in Sections 3.2.2 and 4.1.4.

2. The algorithm for computing invariants in Section 3.6. The computation of mutexes in Blum and Furst's [1997] planning graphs can be viewed as a simple special case of my algorithm, restricted to unconditional operators only.

3. The algorithm for planning with full observability in Section 4.3.2. This algorithm is based on a similar but more complicated algorithm by Cimatti et al. [2003].

4. The representation of planning without observability as quantified Boolean formulae in Section 4.4.2.

5. The framework for non-probabilistic planning with partial observability in Section 4.4.3.

6. The complexity results in Section 4.5.3, most importantly the 2-EXP-completeness result for conditional planning with partial observability.

Chapter 2

Background

In this chapter we define the formal machinery needed in the rest of the lecture for describing different planning problems and algorithms. We give the basic definitions related to the classical propositional logic, theory of computational complexity, and the definition of the transition system model that is the basis of most work on planning. The transition systems in this lecture are closely related to finite automata and transition systems in other areas of computer science.

2.1 Transition systems

The most important way of modeling the application underlying a planning problem is based on the notion of *a transition system*. A transition system consists of a set of *states*, which represent the world at a given instant, and a number of *actions* that describe the possible changes in the world that can be caused by the agent/robot/something. The states form the *state space*.

The actions are best understood as directed graphs with the states as the nodes.

Now a transition system is a 2-tuple $\langle S, O \rangle$ where O is a finite set of actions $o \subseteq S \times S$.

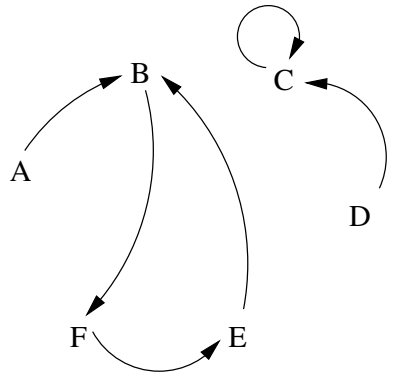
In the beginning we consider deterministic actions only. An action $o \in O$ is *deterministic* if and only if it is a (partial) function on S, that is, for every $s \in S$ there is at most one $s' \in S$ such that $(s, s') \in o$. For *nondeterministic* actions the number of successor states s' may be higher than one.

Later in Section 5.1 we will not just associate more than one successor state with a state, but a probability distribution on the states so that some of the successor states can be more likely than others.

2.1.1 Incidence matrices

Graphs can be represented graphically, or in terms of incidence matrices M (adjacency matrices) in which element $M_{i,j}$ indicates that a transition from state i to j is possible. We will later derive representations of transition systems as propositional formulae that are best understood as succinct representations of the kind of incidence matrices described here. Matrix operations like sum and product have counterparts as operations on propositional formulae, and they are used in some of the algorithms that we will discuss later.

Figure 2.1 depicts the transition graph of an action and the corresponding incidence matrix. The action can be seen to be deterministic because for every state there is at most one arrow going out of it, and each row of the matrix contains at most one non-zero element.



	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	0	0	0	1
C	0	0	1	0	0	0
D	0	0	1	0	0	0
E	0	1	0	0	0	0
F	0	0	0	0	1	0

Figure 2.1: The transition graph and the incidence matrix of a deterministic action

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	0	0	0	1
C	0	0	1	0	0	0
D	0	0	1	0	0	0
E	0	1	0	0	0	0
F	0	0	0	0	1	0

$\times$

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	0	0	0	1
C	1	0	0	0	0	0
D	0	0	0	1	0	0
E	0	0	0	0	1	0
F	0	0	0	1	0	0

$=$

	A	B	C	D	E	F
A	0	0	0	0	0	1
B	0	0	0	**1**	0	0
C	1	0	0	0	0	0
D	1	0	0	0	0	0
E	0	0	0	0	0	1
F	0	0	0	0	1	0

Figure 2.2: Matrix product corresponds to sequential composition.

For matrices $M_1, \ldots, M_n$ that represent the transition relations of actions $a_1, \ldots, a_n$, the combined transition relation is $M = M_1 + M_2 + \cdots + M_n$. The matrix M now tells whether a state can be reached from another state by at least one of the actions.

Here $+$ is the usual matrix addition that uses the Boolean addition for integers 0 and 1, which is defined as $0 + 0 = 0$, and $b + b' = 1$ if $b = 1$ or $b' = 1$. Later in Chapter 5 we will use normal addition and interpret the matrix elements as probabilities of nondeterministic transitions.

Boolean addition is used because later in the presence of nondeterminism we could have 1 for both of two transitions from A to B and from A to C. Later, when the matrix elements represent transition probabilities, we will be using the ordinary arithmetic addition for real numbers.

2.1.2 Reachability as product of matrices

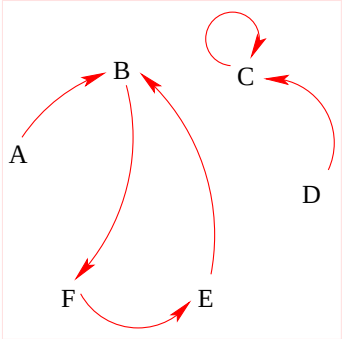
The incidence matrix corresponding to first taking action a_1 and then a_2 is $M_1 M_2$. This is illustrated by Figure 2.2 The inner product of two vectors in the definition of matrix product corresponds to the reachability of a state from another state through all possible intermediate states.

Now we can compute for all pairs s, s' of states whether s' is reachable from s by a sequence of actions.

Let M be the matrix that is the (Boolean) sum of the matrices of the individual actions. Then define

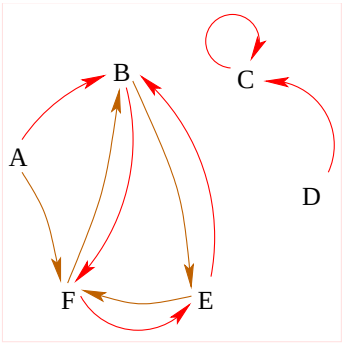
$$\begin{aligned} R_0 &= I_{n\times n} \\ R_i &= R_{i-1} + M R_{i-1} \text{ for all } i \geq 1 \end{aligned}$$

Here n is the number of states and $I_{n\times n}$ is the unit matrix of size n.



	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	0	0	0	1
C	0	0	1	0	0	0
D	0	0	1	0	0	0
E	0	1	0	0	0	0
F	0	0	0	0	1	0

Figure 2.3: A transition graph and the corresponding matrix M



	A	B	C	D	E	F
A	0	1	0	0	0	1
B	0	0	0	0	1	1
C	0	0	1	0	0	0
D	0	0	1	0	0	0
E	0	1	0	0	0	1
F	0	1	0	0	1	0

Figure 2.4: A transition graph extended with composed paths of length 2 and the corresponding matrix $M + M^2$

This computation ends because every element that is 1 for some i, is 1 also for all $j > i$, and because of this monotonicity property there is a fixpoint by Tarski's fixpoint theorem. Matrix R_i represents reachability by i actions.

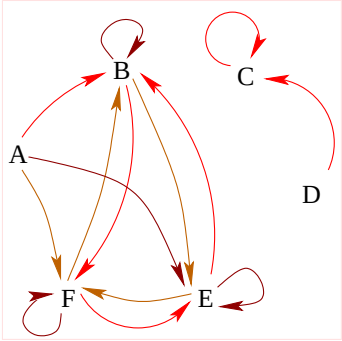
Matrix $R_i = M^0 \cup M^1 \cup \cdots \cup M^i$ represents reachability by i actions or less.

$R_i = R_j$ for some $i \in \{1, \ldots, n\}$ and all $j \geq i$.

2.2 Classical propositional logic

Let P be a set of atomic propositions. We define the set of propositional formulae inductively as follows.

1. For all $p \in P$, p is a propositional formula.
2. If ϕ is a propositional formula, then so is $\neg\phi$.
3. If ϕ and ϕ' are propositional formulae, then so is $\phi \vee \phi'$.
4. If ϕ and ϕ' are propositional formulae, then so is $\phi \wedge \phi'$.
5. Th symbols $\bot$ and $\top$, respectively denoting truth-values false and true, are propositional formulae.



	A	B	C	D	E	F
A	0	1	0	0	1	1
B	0	1	0	0	1	1
C	0	0	1	0	0	0
D	0	0	1	0	0	0
E	0	1	0	0	1	1
F	0	1	0	0	1	1

Figure 2.5: A transition graph extended with composed paths of length 3 and the corresponding matrix $M + M^2 + M^3$

We define the implication $\phi \rightarrow \phi'$ as an abbreviation for $\neg\phi \vee \phi'$, and the equivalence $\phi \leftrightarrow \phi'$ as an abbreviation for $(\phi \rightarrow \phi') \wedge (\phi' \rightarrow \phi)$.

A valuation on P is a function $v : P \rightarrow \{0, 1\}$. Here 0 denotes false and 1 denotes true. For propositions $p \in P$ we define $v \models p$ if and only if $v(p) = 1$. Given a valuation of the propositions P, we can extend it to a valuation of all propositional formulae over P as follows.

1. $v \models \neg\phi$ if and only if $v \not\models \phi$
2. $v \models \phi \vee \phi'$ if and only if $v \models \phi$ or $v \models \phi'$
3. $v \models \phi \wedge \phi'$ if and only if $v \models \phi$ and $v \models \phi'$
4. $v \models \top$
5. $v \not\models \bot$

Computing the truth-value of a formula under a given valuation of propositional variables is polynomial time in the size of the formula by the obvious recursive procedure.

A propositional formula ϕ is *satisfiable* (*consistent*) if there is at least one valuation v so that $v \models \phi$. Otherwise it is *unsatisfiable* (*inconsistent*). A propositional formula ϕ is *valid* or a *tautology* if $v \models \phi$ for all valuations v. We write this as $\models \phi$. A propositional formula ϕ is *a logical consequence* of a propositional formula ϕ', written $\phi' \models \phi$, if $v \models \phi$ for all valuations v such that $v \models \phi'$. A propositional formula that is a proposition p or a negated proposition $\neg p$ for some $p \in P$ is *a literal*. A formula that is a disjunction of literals is *a clause*.

2.2.1 Quantified Boolean formulae

There is an extension of the satisfiability and validity problems of the classical propositional logic that introduces quantification over the truth-values of propositional variables. Syntactically, *quantified Boolean formulae* (QBF) are defined like propositional formulae, but there are two new syntactic rules for the quantifiers.

6. If ϕ is a formula and $p \in P$, then $\forall p\phi$ is a formula.
7. If ϕ is a formula and $p \in P$, then $\exists p\phi$ is a formula.

The truth-value of these formulae is defined if the following two conditions are fulfilled.

- For every $p \in P$ occurring in ϕ, there is exactly one occurrence of $\exists p$ or $\forall p$ in ϕ.
- All occurrences of $p \in P$ are inside $\exists p$ or $\forall p$.

Define $\phi[\psi/x]$ as the formula obtained from ϕ by replacing occurrences of the propositional variable x by ψ.

We define the truth-value of QBF by reducing them to ordinary propositional formulae without occurrences of propositional variables. The atomic formulae in these formulae are the constants $\top$ and $\bot$. The truth-value of these formulae is independent of the valuation, and is recursively computed by the Boolean function associated with the connectives $\vee$, $\wedge$ and $\neg$.

Definition 2.1 (Truth of QBF) *A formula $\exists x\phi$ is true if and only if $\phi[\top/x] \vee \phi[\bot/x]$ is true. (Equivalently, if $\phi[\top/x]$ is true or $\phi[\bot/x]$ is true.)*

A formula $\forall x\phi$ is true if and only if $\phi[\top/x] \wedge \phi[\bot/x]$ is true. (Equivalently, if $\phi[\top/x]$ is true and $\phi[\bot/x]$ is true.)

A formula ϕ with an empty prefix (and consequently without occurrences of propositional variables) is true if and only if ϕ is satisfiable (equivalently, valid: for formulae without propositional variables validity coincides with satisfiability.)

Example 2.2 The formulae $\forall x \exists y(x \leftrightarrow y)$ and $\exists x \exists y(x \wedge y)$ are true.

The formulae $\exists x \forall y(x \leftrightarrow y)$ and $\forall x \forall y(x \vee y)$ are false. ■

Notice that a QBF with only existential quantifiers is true if and only if if the formula stripped from the quantifiers is satisfiable. Similarly, truth of QBF with only universal quantifiers coincides with the validity of the corresponding formulae without quantifiers.

Changing the order of two consecutive variables quantified by the same quantifier does not affect the truth-value of the formula. It is often useful to ignore the ordering in these cases, and view each quantifier as quantifying a set of formulae, for example $\exists x_1 x_2 \forall y_1 y_2 \phi$.

Quantified Boolean formulae are interesting because evaluating their truth-value is PSPACE-complete [Meyer and Stockmeyer, 1972], and several computational problems that presumably cannot be translated to the satisfiability of the propositional logic in polynomial time (assuming that NP≠PSPACE) can be efficiently translated to QBF.

2.2.2 Binary decision diagrams

Propositional formulae can be transformed to different normal forms. The most well-known normal forms are the conjunctive normal form (CNF) and the disjunctive normal form (DNF). Formulae in conjunctive normal form are conjunctions of disjunctions of literals, and in disjunctive normal form they are disjunctions of conjunctions of literals. For every propositional formula there is a logically equivalent one in both of these normal forms. However, the formula in normal form may be exponentially bigger.

Normal forms are useful for at least two reasons. First, certain types of algorithms are easier to describe when assumptions of the syntactic form of the formulae can be made. For example, the resolution rule which is the basis of many theorem-proving algorithms, is defined for formulae in the conjunctive normal form only (the clausal form). Defining resolution for non-clausal formulae is more difficult.

The second reason is that certain computational problems can be solved more efficiently for formulae in normal form. For example, testing the validity of propositional formulae is in general co-NP-hard, but if the formulae are in CNF then it is polynomial time: just check whether every conjunct contains both p and $\neg p$ for some proposition p.

Transformation into a normal form in general is not a good solution to any computationally intractable problem like validity testing, because for example in the case of CNF, polynomial-time validity testing became possible only by allowing a potentially exponential increase in the size of the formula.

However, there are certain normal forms for propositional formulae that have proved very useful in various types of reasoning needed in planning and other related areas, like model-checking in computer-aided verification.

In this section we discuss (ordered) binary decision diagrams (BDDs) [Bryant, 1992]. Other normal forms of propositional formulae that have found use in AI and could be applied to planning include the decomposable negation normal form [Darwiche, 2001] which is less restricted than binary decision diagrams (formulae in DNNF can be viewed as a superclass of BDDs) and are sometimes much smaller. However, smaller size means that some of the logical operations that can be performed in polynomial time for BDDs, like equivalence testing, are NP-hard for formulae in DNNF.

The main reason for using BDDs is that the logical equivalence of BDDs coincides with syntactic equivalence: two BDDs are logically equivalent if and only if they are the same BDD. Propositional formulae in general, or formulae in CNF or in DNF do not have this property. Furthermore, computing a BDD that represents the conjunction or disjunction of two BDDs or the negation of a BDDs also takes only polynomial time.

However, like with other normal forms, a BDD can be exponentially bigger than a corresponding unrestricted propositional formula. One example of such a propositional formulae is the binary multiplier: Any BDD representation of n-bit multipliers has a size exponential in n. Also, even though many of the basic operations on BDDs can be computed in polynomial time in the size of the component BDDs, iterating these operations may increase the size exponentially: some of these operator may double the size of the BDD, and doubling n times is exponential in n and in the size of the original BDD.

A main application of BDDs has been model-checking in computer-aided verification [Burch *et al.*, 1994; Clarke *et al.*, 1994], and in recent years these same techniques have been applied to AI planning as well. We will discuss BDD-based planning algorithms in Chapter 4.

BDDs are expressed in terms of the ternary Boolean operator if-then-else $ite(p, \phi_1, \phi_2)$ defined as $(p \wedge \phi_1) \vee (\neg p \wedge \phi_2)$, where p is a proposition. Any Boolean formula can be represented by using this operator together with propositions and the constants $\top$ and $\bot$. Figure 2.6 depicts a BDD for the formula $(A \vee B) \wedge (B \vee C)$. The normal arrow coming from a node for P corresponds to the case in which P is true, and the dotted arrow to the case in which P is false. Note that BDDs are graphs, not trees like formulae, and this provides a further reduction in the BDD size as a subformula never occurs more than once.

There is an ordering condition on BDDs: the occurrences of propositions on any path from the root to a leaf node must obey a fixed ordering of the propositions. This ordering condition together with the graph representation is required for the good computational properties of BDDs, like the polynomial time equivalence test.

A BDD corresponding to a propositional formula can be obtained by repeated application of an

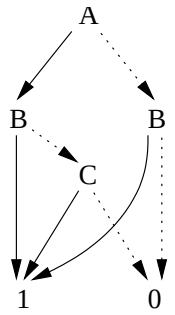


Figure 2.6: A BDD

equivalence called the Shannon expansion.

$$\phi \equiv (p \wedge \phi[\top/p]) \vee (\neg p \wedge \phi[\bot/p]) \equiv ite(p, \phi[\top/p], \phi[\bot/p])$$

Example 2.3 We show how the BDD for $(A \vee B) \wedge (B \vee C)$ is produced by repeated application of the Shannon expansion. We use the variable ordering A, B, C and use the Shannon expansion to eliminate the variables in this order.

$$\begin{array}{ll} & (A \vee B) \wedge (B \vee C) \\ \equiv & ite(A, (\top \vee B) \wedge (B \vee C), (\bot \vee B) \wedge (B \vee C)) \\ \equiv & ite(A, B \vee C, B) \\ \equiv & ite(A, ite(B, \top \vee C, \bot \vee C), ite(B, \top, \bot)) \\ \equiv & ite(A, ite(B, \top, C), ite(B, \top, \bot)) \\ \equiv & ite(A, ite(B, \top, ite(C, \top, \bot)), ite(B, \top, \bot)) \end{array}$$

The simplifications in the intermediate steps are by the equivalences $\top \vee \phi \equiv \top$ and $\bot \vee \phi \equiv \phi$ and $\top \wedge \phi \equiv \phi$ and $\bot \wedge \phi \equiv \bot$. When

$$ite(A, ite(B, \top, ite(C, \top, \bot)), ite(B, \top, \bot))$$

is first turned into a tree and then equivalent subtrees are identified, we get the BDD in Figure 2.6. The terminal node 1 corresponds to $\top$ and the terminal node 0 to $\bot$. ■

There are many operations on BDDs that are computable in polynomial time. These include forming the conjunction $\wedge$ and the disjunction $\vee$ of two BDDs, and forming the negation $\neg$ of a BDD. However, conjunction and disjunction of n BDDs may have a size that is exponential in n, as adding a new disjunct or conjunct may double the size of the BDD.

An important operation in many applications of BDDs is the existential abstraction operation $\exists p.\phi$, which is defined by

$$\exists p.\phi = \phi[\top/p] \vee \phi[\bot/p]$$

where $\phi[\psi/p]$ means replacing all occurrences of p in ϕ by ψ. Also this is computable in polynomial time, but existentially abstracting n variables may result in a BDD that has size exponential in n, and hence may take exponential time. Existential abstraction can of course be used for any propositional formulae, not only for BDDs.

The formula ϕ' obtained from ϕ by existentially abstracting p is in general not equivalent to ϕ, but has many properties that make the abstraction operation useful.

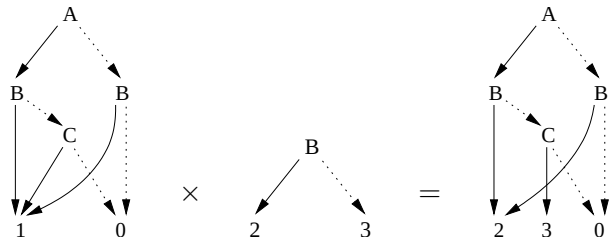


Figure 2.7: Three ADDs, the first of which is also a BDD.

Lemma 2.4 *Let ϕ be a formula and p a proposition. Let $\phi' = \exists p.\phi = \phi[\top/p] \vee \phi[\bot/p]$. Now the following hold.*

1. *ϕ is satisfiable if and only if ϕ' is.*
2. *ϕ is valid if and only if ϕ' is.*
3. *If χ is a formula without occurrences of p, then $\phi \models \chi$ if and only if $\phi' \models \chi$.*

Example 2.5

$$\exists B.((A \rightarrow B) \wedge (B \rightarrow C))$$
$$= ((A \rightarrow \top) \wedge (\top \rightarrow C)) \vee ((A \rightarrow \bot) \wedge (\bot \rightarrow C))$$
$$\equiv C \vee \neg A \equiv A \rightarrow C$$

$$\exists AB.(A \vee B) = \exists B.(\top \vee B) \vee (\bot \vee B) = ((\top \vee \top) \vee (\bot \vee \top)) \vee ((\top \vee \bot) \vee (\bot \vee \bot))$$

■

2.2.3 Algebraic decision diagrams

Algebraic decision diagrams (ADDs) [Fujita *et al.*, 1997; Bahar *et al.*, 1997] are a generalization of binary decision diagrams that has been applied to many kinds of probabilistic extensions of problems solved by BDDs. BDDs have only two terminal nodes, 1 and 0, and ADDs generalize this to a finite number of real numbers.

While BDDs represent Boolean functions, ADDs represent mapping from valuations to real numbers. The Boolean operations on BDDs, like taking the disjunction or conjunction of two BDDs, generalize to the arithmetic operations to take the arithmetic sum or the arithmetic product of two functions. There are further operations on ADDs that have no counterpart for BDDs, like constructing a function that on any valuation equals the maximum of two functions.

Figure 2.7 depicts three ADDs, the first of which is also a BDD. The product of ADDs is a generalization of conjunction of BDDs: if for some valuation/state ADD A assigns the value r_1 and ADD B assigns the value r_2, then the product ADD $A \cdot B$ assigns the value $r_1 \cdot r_2$ to the valuation.

The following are some of the operations typically available in implementations of ADDs. Here we denote ADDs by f and g and view them as functions from valuations x to real numbers.

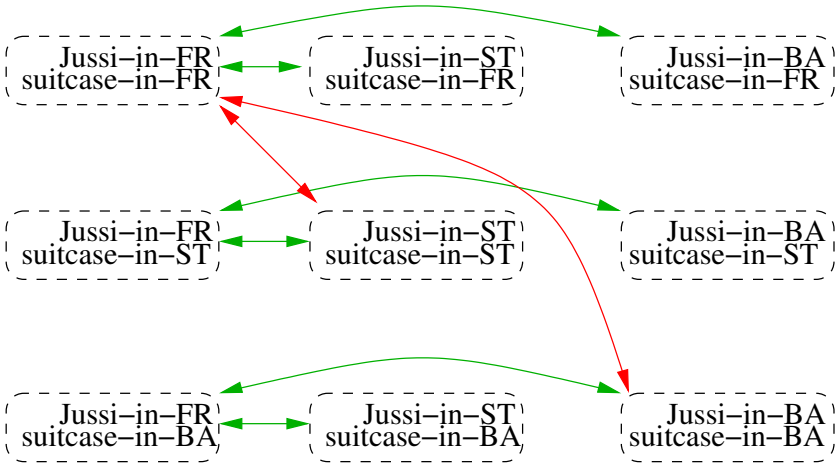


Figure 2.8: A simple transition system based on state variables

operation	notation	meaning
sum	$f + g$	$(f + g)(x) = f(x) + g(x)$
product	$f \cdot g$	$(f \cdot g)(x) = f(x) \cdot g(x)$
maximization	$\max(f, g)$	$(\max(f, g))(x) = max(f(x), g(x))$

There is an operation for ADDs that corresponds to the existential abstraction operation on BDDs, and that is used in multiplication of matrices represented as ADDs, just like existential abstraction is used in multiplication of Boolean matrices represented as BDDs.

Let f be an ADD and p a proposition. Then *arithmetic existential abstraction* of f, written $\exists p.f$, is an ADD that satisfies the following.

$$(\exists p.f)(x) = (f[\top/p])(x) + (f[\bot/p])(x)$$

2.3 Operators and state variables

Transition systems are widely used in AI planning and other areas of computer science, and it is a model that can very well be used for describing all kinds of systems, especially man-made systems and abstractions of the real-world used by human beings.

However, describing a transition system by giving a set of states and then relations representing the actions is usually not the most natural nor the most concise description. This is because the individual states usually have a certain meaning, which determines which actions are possible in the state and what the possible successor states of the state under the given action are.

The common type of description of states is based on *state variables*. Let A be a finite set of state variables. Each state variable $a \in A$ can have a finite number of different values R. Now a state s can be understood as a valuation $s : A \to R$ that assigns a value to each state variable.

In this lecture we will restrict to Boolean state variables with $R = \{0, 1\}$, but almost everything in the lecture directly generalizes to any finite set R of values.

The state space S is now the set of all valuations of A.

Example 2.6 Figure 2.8 illustrates a small transition system induced by state variables. We have

depicted each state by enumerating the state variables that have the value *true* in it (exactly two in each of the states in the figure), and left out states that do not correspond to the intuitive meaning of the states. Each state variable indicates whether one of the two objects is in one of the three locations (Freiburg, Strassburg, Basel.)

The two actions respectively correspond to traveling with and without the suitcase.

Clearly, if we were using many-valued state variables, it would suffice to have only two of them, each having three possible values corresponding to the three locations. ■

It remains to give a description of the set of actions in terms of state variables. Intuitively, we have to say whether an action is applicable in a given state s, and what the successor state s' of that state under the given action is.[1] Actions are represented as *operators* $\langle c, e\rangle$, where c is a propositional formula over A that has to be satisfied by the valuation s for the action to be possible, and e describes how s' is obtained by changing the values of state variables in s.

Atomic effects in general are of the form $a := r$ for $a \in A$ and $r \in R$. In the Boolean case it is common to simply write a for $a := 1$ and $\neg a$ for $a := 0$, and also we will do so. Be careful to avoid confusion with an effect like $e = a_1 \wedge \neg a_2$ and exactly the same looking formula $\phi = a_1 \wedge \neg a_2$. After the effect e the formula ϕ will be true, but this is the only direct relationship between formulae and effects; in particular, there are no disjunctions $\vee$ in effects and there is nothing in the propositional logic that corresponds to $\rhd$.

Definition 2.7 *Let A be a set of state variables. An operator is a pair $\langle c, e\rangle$ where c is a propositional formula over A describing the precondition, and e is an effect over A. Effects are recursively defined as follows.*

1. *$\top$ is an effect (the dummy effect).*
2. *a and $\neg a$ for state variables $a \in A$ are effects.*
3. *$e_1 \wedge \cdots \wedge e_n$ is an effect if $e_1, \ldots, e_n$ are effects over A (the special case with $n = 0$ is the empty conjunction $\top$.)*
4. *$c \rhd e$ is an effect if c is a formula over A and e is an effect over A.*

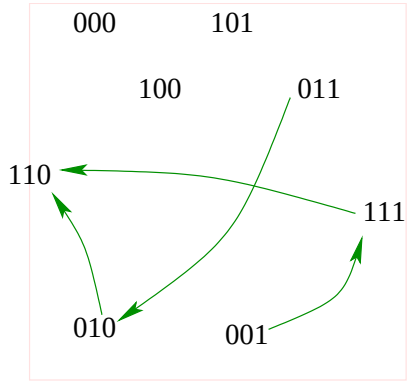
Notice that the representation of transition systems in terms of state variables and operators opens the possibility that the size of the transition system, the number of states in the transition system, may be exponential in the size of the set of operators. This idea of *succinct representations* of various objects is present in very many areas of computer science. As we will see later in this lecture, succinctness usually means that algorithms for reasoning about the objects in question increases: if a computational problem, like finding shortest paths in transition systems represented as graphs, is solvable in polynomial time, solving the same problem for succinctly represented transition systems will be much higher.

Definition 2.8 (Operator application) *Let $\langle c, e\rangle$ be an operator over A. Let s be a state, that is an assignment of truth values to A. The operator is applicable in s if $s \models c$ and the set $[e]_s$, defined below, does not contain a and $\neg a$ for any $a \in A$.*

Recursively assign each effect e a set $[e]_s$ of literals a and $\neg a$ for $a \in A$ (the active effects.*)*

1. *$[\top]_s = \emptyset$*

[1] We discuss nondeterministic actions in Chapter 4.



$\langle (B \wedge C) \vee (\neg A \wedge B \wedge \neg C) \vee (\neg A \wedge C),$
$((B \wedge C) \rhd \neg C) \wedge (\neg B \rhd (A \wedge B)) \wedge (\neg C \rhd A) \rangle$

Figure 2.9: A transition graph with valuations of A, B and C as states and, a corresponding operator

2. $[a]_s = \{a\}$ *for* $a \in A$.
3. $[\neg a]_s = \{\neg a\}$ *for* $a \in A$.
4. $[e_1 \wedge \cdots \wedge e_n]_s = [e_1]_s \cup \ldots \cup [e_n]_s$.
5. $[c' \rhd e]_s = [e]_s$ *if* $s \models c'$ *and* $[c' \rhd e]_s = \emptyset$ *otherwise.*

The successor state of s under the operator is the one that is obtained from s by making the literals in $[e]_s$ true and retaining the truth-values of state variables not occurring in $[e]_s$. This state is denoted by $app_o(s)$. We call the process of computing the successor state of a state with respect to an operator as progression.

Example 2.9 Consider the operator $\langle a, e \rangle$ where $e = \neg a \wedge (\neg c \rhd \neg b)$ and a state s with a, b and c all true. The operator is applicable because $s \models a$. Now $[e]_s = \{\neg a\}$ and $app_{\langle a,e \rangle}(s) \models \neg a \wedge b \wedge c$. ■

Example 2.10 Figure 2.9 depicts a transition graph with valuations of three state variables A, B and C as nodes, and a corresponding operator. ■

2.3.1 Extensions

The basic language for effects could be extended with further constructs. A natural construct would be *sequential composition* of effects. If e and e' are effects, then also $e; e'$ is an effect that corresponds to first executing e and then e'. We do not discuss this topic further in this lecture. Definition 3.11 and Theorem 3.12 show how sequential composition can be eliminated from effects.

2.3.2 Normal forms

We introduce a normal form for effects that will be used in later sections for defining operations on propositional formulae describing sets of states.

Table 2.1 lists a number of equivalences on effects. Their proofs of correctness with Definition 2.8 are straightforward. An effect e is equivalent to $\top \wedge e$, and conjunctions of effects can be

$$
\begin{aligned}
c \rhd (e_1 \wedge \cdots \wedge e_n) &\equiv (c \rhd e_1) \wedge \cdots \wedge (c \rhd e_n) && (2.1)\\
c \rhd (c' \rhd e) &\equiv (c \wedge c') \rhd e && (2.2)\\
(c_1 \rhd e) \wedge (c_2 \rhd e) &\equiv (c_1 \vee c_2) \rhd e && (2.3)\\
e \wedge (c \rhd e) &\equiv e && (2.4)\\
e &\equiv \top \rhd e && (2.5)\\
e_1 \wedge (e_2 \wedge e_3) &\equiv (e_1 \wedge e_2) \wedge e_3 && (2.6)\\
e_1 \wedge e_2 &\equiv e_2 \wedge e_1 && (2.7)\\
c \rhd \top &\equiv \top && (2.8)\\
e \wedge \top &\equiv e && (2.9)\\
& && (2.10)
\end{aligned}
$$

Table 2.1: Equivalences on effects

arbitrarily reordered without affecting the meaning of the operator. These trivial equivalences will later be used without explicitly mentioning them, for example in the definitions of the normal forms and when applying equivalences.

The normal form corresponds to moving the conditionals inside so that their consequents are atomic effects, and it is useful for example in the computation of properties satisfied by predecessor states by regression in Section 3.2.2.

Definition 2.11 *An effect e is in* normal form *if it is $\top$ or a conjunction of one or more effects of the form $c \rhd a$ and $c \rhd \neg a$ where a is a state variable, and there is at most one occurrence of atomic effects a and $\neg a$ for any state variable a. An operator $\langle c, e\rangle$ is in normal form if e is in normal form.*

Theorem 2.12 *For every operator there is an equivalent one in normal form. There is one that has a size that is polynomial in the size of the operator.*

Proof: We can transform any operator into normal form by using the equivalences 2.1, 2.2, 2.3, 2.6, 2.7, and 2.8 in Table 2.1.

The proof is by structural induction on the effect e of the operator $\langle c, e\rangle$.

Induction hypothesis: the effect e can be transformed to normal form.

Base case 1, $e = \top$: This is already in normal form.

Base case 2, $e = a$ or $e = \neg a$: An equivalent effect in normal form is $\top \rhd e$ by Equivalence 2.5.

Inductive case 1, $e = e_1 \wedge e_2$: By the induction hypothesis e_1 and e_2 can be transformed into normal form, so assume that they already are. If one of e_1 and e_2 is $\top$, by Equivalence 2.9 we can eliminate it.

Assume e_1 contains $c_1 \rhd l$ for some literal l and e_2 contains $c_2 \rhd l$. We can reorder $e_1 \wedge e_2$ with Equivalences 2.6 and 2.7 so that one of the conjuncts is $(c_1 \rhd l) \wedge (c_2 \rhd l)$. Then by Equivalence 2.3 this conjunct can be replaced by $(c_1 \vee c_2) \rhd l$. Because this can be done repeatedly for every literal l, we can transform $e_1 \wedge e_2$ into normal form.

Inductive case 1, $e = z \rhd e_1$: By the induction hypothesis e_1 can be transformed to normal form, so assume that it already is.

If e_1 is $\top$, e can be replaced with the equivalent effect $\top$.

If $e_1 = z' \rhd e_2$ for some z' and e_2, then e can be replaced by the equivalent (by Equivalence 2.2) effect $(z \wedge z') \rhd e_2$ in normal form.

Otherwise, e_1 is a conjunction of effects $z \rhd l$. By Equivalence 2.1 we can move z inside the conjunction. Applications of Equivalences 2.2 transform the effect into normal form.

In this transformation the conditions c in $c \rhd e$ are copied into front of the atomic effects. Let m be the sum of the sizes of all the conditions c, and let n be the number of occurrences of atomic effects a and $\neg a$ in the effect. An upper bound on size increase is $\mathcal{O}(nm)$, which is polynomial in the size of the original operator. □

A further reduction in the size of the descriptions of transition systems is obtained by using *schematic operators* instead of operators as described above.

There are often regularities in the set of operators and corresponding regularities in the transition system. A common regularity is that there are several almost identical *objects* that behave in the same way. For example, operators describing driving car 1 and car 2 between cities are otherwise identical except that in one case a reference to state variables about car 1 are used and in the other state variables about car 2. This kind of regularities are ubiquitous, and operators allowing easy expression of such sets of operators are used by almost all implementations of planning algorithms.

Example 2.13 Consider the schematic operator

$$\langle \text{in}(x, t_1), \text{in}(x, t_2) \wedge \neg \text{in}(x, t_1) \rangle$$

where the schema variables x, t_1 and t_2 take values as follows.

$$\begin{array}{l} x \in \{\text{car1}, \text{car2}\} \\ t_1 \in \{\text{Freiburg}, \text{Strassburg}, \text{Basel}\} \\ t_2 \in \{\text{Freiburg}, \text{Strassburg}, \text{Basel}\} \\ t_1 \neq t_2 \end{array}$$

This schematic operator corresponds to the following set of operators.

$$\begin{array}{ll} \{ & \langle \text{in}(\text{car1}, \text{Freiburg}), \text{in}(\text{car1}, \text{Basel}) \wedge \neg \text{in}(\text{car1}, \text{Freiburg}) \rangle, \\ & \langle \text{in}(\text{car1}, \text{Freiburg}), \text{in}(\text{car1}, \text{Strassburg}) \wedge \neg \text{in}(\text{car1}, \text{Freiburg}) \rangle, \\ & \langle \text{in}(\text{car1}, \text{Strassburg}), \text{in}(\text{car1}, \text{Freiburg}) \wedge \neg \text{in}(\text{car1}, \text{Strassburg}) \rangle, \\ & \langle \text{in}(\text{car1}, \text{Strassburg}), \text{in}(\text{car1}, \text{Basel}) \wedge \neg \text{in}(\text{car1}, \text{Strassburg}) \rangle, \\ & \langle \text{in}(\text{car1}, \text{Basel}), \text{in}(\text{car1}, \text{Freiburg}) \wedge \neg \text{in}(\text{car1}, \text{Basel}) \rangle, \\ & \langle \text{in}(\text{car1}, \text{Basel}), \text{in}(\text{car1}, \text{Strassburg}) \wedge \neg \text{in}(\text{car1}, \text{Basel}) \rangle, \\ & \langle \text{in}(\text{car2}, \text{Freiburg}), \text{in}(\text{car2}, \text{Basel}) \wedge \neg \text{in}(\text{car2}, \text{Freiburg}) \rangle, \\ & \langle \text{in}(\text{car2}, \text{Freiburg}), \text{in}(\text{car2}, \text{Strassburg}) \wedge \neg \text{in}(\text{car2}, \text{Freiburg}) \rangle, \\ & \langle \text{in}(\text{car2}, \text{Strassburg}), \text{in}(\text{car2}, \text{Freiburg}) \wedge \neg \text{in}(\text{car2}, \text{Strassburg}) \rangle, \\ & \langle \text{in}(\text{car2}, \text{Strassburg}), \text{in}(\text{car2}, \text{Basel}) \wedge \neg \text{in}(\text{car2}, \text{Strassburg}) \rangle, \\ & \langle \text{in}(\text{car2}, \text{Basel}), \text{in}(\text{car2}, \text{Freiburg}) \wedge \neg \text{in}(\text{car2}, \text{Basel}) \rangle, \\ & \langle \text{in}(\text{car2}, \text{Basel}), \text{in}(\text{car2}, \text{Strassburg}) \wedge \neg \text{in}(\text{car2}, \text{Basel}) \rangle \quad \} \end{array}$$

■

Schematic operators may also allow *existential* and *universal* quantification over sets of objects for encoding disjunctions and conjunctions more concisely. For example, $\exists x \in \{A, B, C\}\text{in}(x, \text{Freiburg})$ is a short-hand for $\text{in}(A, \text{Freiburg}) \vee \text{in}(B, \text{Freiburg}) \vee \text{in}(C, \text{Freiburg})$.

Non-schematic operators are often called *ground operators*, and the process of producing a set of ground operators from a schematic operator is called *grounding*. In this lecture we will be using ground operators only. Most planning programs take schematic operators as input, and have a preprocessor that grounds them.

2.3.3 Sets of states as propositional formulae

Because we identified states with valuations of state variables, we can now identify sets of states with propositional formulae over the state variables. This allows us to perform set-theoretic operations on sets as logical operations, and test relations between sets by inference in the propositional logic.

operation on sets	operation on formulae
$A \cup B$	$A \vee B$
$A \cap B$	$A \wedge B$
$A \backslash B$	$A \wedge \neg B$
question about sets	question about formulae
$A \subseteq B$?	$\models A \rightarrow B$?
$A \subset B$?	$\models A \rightarrow B$ and not $\models B \rightarrow A$?
$A = B$?	$\models A \leftrightarrow B$?

Any inconsistent formula, like $A \wedge \neg A$ or $\bot$, is not true in any state, and therefore represents the empty set. Similarly, any valid formula, for instance $\top$ or $A \vee \neg A$, represents the set of all states (all valuations of the state variables.)

2.4 Computational complexity

In this section we discuss deterministic, nondeterministic and alternating Turing machines (DTMs, NDTMs and ATMs) and define several complexity classes in terms of them. For a detailed introduction to computational complexity see any of the standard textbooks [Balcázar *et al.*, 1988; 1990; Papadimitriou, 1994].

The definition of ATMs we use is like that of Balcázar et al. [1990] but without a separate input tape. Deterministic and nondeterministic Turing machines (DTMs, NDTMs) are a special case of a alternating Turing machines.

Definition 2.14 *An* alternating Turing machine *is a tuple* $\langle \Sigma, Q, \delta, q_0, g \rangle$ *where*

- Q *is a finite set of states (the internal states of the ATM),*
- Σ *is a finite alphabet (the contents of tape cells),*
- δ *is a transition function* $\delta : Q \times \Sigma \cup \{|, \Box\} \rightarrow 2^{\Sigma \cup \{|\} \times Q \times \{L,N,R\}}$,
- q_0 *is the initial state, and*
- $g : Q \rightarrow \{\forall, \exists, \textit{accept}, \textit{reject}\}$ *is a labeling of the states.*

The symbols | and □, the end-of-tape symbol and the blank symbol, in the definition of δ respectively refer to the beginning of the tape and to the end of the tape. It is required that $s = |$ and $m = R$ for all $\langle s, q', m\rangle \in \delta(q, |)$ for any $q \in Q$, that is, at the left end of the tape the movement is always to the right and the end-of-tape symbol | may not be changed. For $s \in \Sigma$ we restrict s' in $\langle s', q', m\rangle \in \delta(q, s)$ to $s' \in \Sigma$, that is, | gets written onto the tape only in the special case when the R/W head is on the end-of-tape symbol. Notice that the transition function is a total function, and the ATM computation terminated upon reaching an accepting or a rejecting state.

A configuration of an ATM is $\langle q, \sigma, \sigma'\rangle$ where q is the current state, σ is the tape contents left of the R/W head with the rightmost symbol under the R/W head, and σ' is the tape contents strictly right of the R/W head. This is a finite representation of the finite non-blank segment of the tape of the ATM.

The computation of an ATM starts from the initial configuration $\langle q_0, |a, \sigma\rangle$ where $a\sigma$ is the input string of the Turing machine. Below ϵ denotes the empty string.

The configuration of an ATM changes as follows.

1. From $\langle q, \sigma a, \sigma'\rangle$ to $\langle q', \sigma, a'\sigma'\rangle$ when $\delta(q, a) = \langle a', q', L\rangle$.

2. From $\langle q, \sigma a, \sigma'\rangle$ to $\langle q', \sigma a', \sigma'\rangle$ when $\delta(q, a) = \langle a', q', N\rangle$.

3. From $\langle q, \sigma a, b\sigma'\rangle$ to $\langle q', \sigma a' b, \sigma'\rangle$ when $\delta(q, a) = \langle a', q', R\rangle$.

4. From $\langle q, \sigma a, \epsilon\rangle$ to $\langle q', \sigma a'\square, \epsilon\rangle$ when $\delta(q, a) = \langle a', q', R\rangle$.

A configuration $\langle q, \sigma, \sigma'\rangle$ of an ATM is *final* if $g(q) =$ accept or $g(q) =$ reject.

The acceptance of an input string by an ATM is defined recursively starting from final configurations. A final configuration is accepting if $g(q) =$ accept. Non-final configurations are accepting if the state is universal ($\forall$) and all the immediate successor configurations are accepting, or if the state is existential ($\exists$) and at least one of the immediate successor configurations is accepting. Finally, the ATM accepts a given input string if the initial configuration is accepting.

A nondeterministic Turing machine is an ATM without universal states. A deterministic Turing machine is an ATM with $|\delta(q, s)| = 1$ for all $q \in Q$ and $s \in \Sigma$.

The complexity classes used in this lecture are the following. PSPACE is the class of decision problems solvable by deterministic Turing machines that use a number of tape cells bounded by a polynomial on the input length n. Formally,

$$\text{PSPACE} = \bigcup_{k \geq 0} \text{DSPACE}(n^k).$$

Similarly other complexity classes are defined in terms of the time consumption (DTIME($f(n)$) on a deterministic Turing machine, time consumption (NTIME($f(n)$) on a nondeterministic Turing machine, or time or space consumption on alternating Turing machines (ATIME($f(n)$) or

ASPACE($f(n)$)) [Balcázar *et al.*, 1988; 1990].

$$\begin{aligned}
\text{P} &= \bigcup_{k\geq 0} \text{DTIME}(n^k) \\
\text{NP} &= \bigcup_{k\geq 0} \text{NTIME}(n^k) \\
\text{EXP} &= \bigcup_{k\geq 0} \text{DTIME}(2^{n^k}) \\
\text{NEXP} &= \bigcup_{k\geq 0} \text{NTIME}(2^{n^k}) \\
\text{EXPSPACE} &= \bigcup_{k\geq 0} \text{DSPACE}(2^{n^k}) \\
\text{2-EXP} &= \bigcup_{k\geq 0} \text{DTIME}(2^{2^{n^k}}) \\
\text{2-NEXP} &= \bigcup_{k\geq 0} \text{NTIME}(2^{2^{n^k}}) \\
\\
\text{APSPACE} &= \bigcup_{k\geq 0} \text{ASPACE}(n^k) \\
\text{AEXPSPACE} &= \bigcup_{k\geq 0} \text{ASPACE}(2^{n^k})
\end{aligned}$$

There are many useful connections between complexity classes defined in terms of deterministic and alternating Turing machines [Chandra *et al.*, 1981], for example

$$\begin{aligned}
\text{EXP} &= \text{APSPACE} \\
\text{2-EXP} &= \text{AEXPSPACE}.
\end{aligned}$$

Roughly, an exponential deterministic time bound corresponds to a polynomial alternating space bound.

We have defined all the complexity classes in terms of Turing machines. However, for all purposes of this lecture, we can equivalently use conventional programming languages (like C or Java) or simplified variants of them for describing computation. The main difference between conventional programming languages and Turing machines is that the former use random-access memory whereas memory access in Turing machines is local and only the current tape cell can be directly accessed. However, these two computational models can be simulated with each other with a polynomial overhead and are therefore for our purposes equivalent. The differences show up in complexity classes with very strict (subpolynomial) restrictions on time and space consumption.

Later in this lecture, in some of the proofs that a given computational problem belongs to a certain class we will usually give a program in a simple programming language comparable to a small subset of C or Java, instead of giving a formal description of a Turing machine, because the latter would usually be very complicated and difficult to understand.

A problem L is *C-hard* (where C is any of the complexity classes) if all problems in the class C are polynomial time *many-one reducible* to it; that is, for all problems $L' \in C$ there is a function $f_{L'}$ that can be computed in polynomial time on the size of its input and $f_{L'}(x) \in L$ if and only if $x \in L'$. We say that the function $f_{L'}$ is a translation from L' to L. A problem is *C-complete* if it belongs to the class C and is C-hard.

In complexity theory the most important distinction between computational problems is that between *tractable* and *intractable* problems. A problem is considered to be tractable, efficiently solvable, if it can be solved in polynomial time. Otherwise it is intractable. Most planning problems are highly intractable, but for many algorithmic approaches to planning it is important than certain basic steps in these algorithms can be guaranteed to be tractable.

In this lecture we analyze the complexity of many computational problems, showing them to be complete problems for some of the classes mentioned above. The proofs consist of two parts. We show that the problem belongs to the class. This is typically by giving an algorithm for the

problem, possibly a nondeterministic one, and then showing that the algorithm obeys the resource bounds on time or memory consumption as required by the complexity class. Then we show the hardness of the problem for the class, that is, we can reduce any problem in the class to the problem in polynomial time. This can be either by simulating all Turing machines that represent computation in the class, or by reducing a complete problem in the class to the problem in question in polynomial time (a many-one reduction).

For almost all commonly used complexity classes there are more or less natural complete problems that often have a central role in proving the completeness of other problems for the class in question. Some complete problems for the complexity classes mentioned above are the following.[2]

class	complete problem
P	truth-value of formulae in the propositional logic in a given valuation
NP	satisfiability of formulae in the propositional logic (SAT)
PSPACE	truth-value of quantified Boolean formulae

Complete problems for classes like EXP and NEXP can be obtained from the P-complete and NP-problems by representing propositional formulae succinctly in terms of other propositional formulae [Papadimitriou and Yannakakis, 1986]. We will not discuss this topic further in this lecture.

2.5 Exercises

2.1 Show that for any transition system $\langle S, \{o_1, \ldots, o_n\}\rangle$ in which the states $s \in S$ are valuations of a set A of propositional variables (as in Example 2.10), the actions $o_1, \ldots, o_n$ can be represented in terms of operators.

2.2 Show that conditional effects with $\triangleright$ are necessary, that is, find a transition system where states are valuations of a set of state variables and the actions cannot be represented as operators without conditional effects with $\triangleright$. *Hint*: There is an example with two states and one state variable.

[2] For definition of P-hard problems we have to use more restricted many-one reductions that use only logarithmic space instead of polynomial time. Otherwise all non-trivial problems in P would be P-hard and P-complete.

Chapter 3

Deterministic planning

In this chapter we describe a number of algorithms for solving the historically most important and most basic type of planning problem. Two rather strong simplifying assumptions are made. First, all actions are deterministic, that is, under every action every state has at most one successor state. Second, there is only one initial state.

Under these restrictions, whenever a goal state can be reached, it can be reached by a fixed sequence of actions. With more than one initial state it would be necessary to use a different sequence of actions for every initial state, and with nondeterministic actions the sequence of actions to be taken is not simply a function of the initial state, and for producing appropriate sequences of actions a more general notion of plans with branches/conditionals becomes necessary. This is because after executing an action, even when the starting state was known, the state that is reached cannot be predicted, and the way plan execution continues depends on the new state. In Chapter 4 we relax both of these restrictions, and consider planning with more than one initial state and with nondeterministic actions.

The structure of this chapter is as follows. First we discuss the two ways of traversing the transition graphs without producing the graphs explicitly. In forward traversal we repeatedly compute the successor states of our current state, starting from the initial state. In backward traversal we must use sets of states, represented as formulae, because we must start from the set of goal states, and further, under a given action a state may have several predecessor states.

Then we discuss the use of heuristic search algorithms for performing the search in the transition graphs and the computation of distance heuristics to be used in estimating the value of the current states or sets of states. Further improvements to plan search are obtained by recognizing symmetries in the transition graphs, and for backward search, restricting the search by invariants that are formulae describing which states are reachable from the initial state.

A complementary approach to planning is obtained by translating the planning problem to the classical propositional logic and then finding plans by algorithms that test the satisfiability of formulae in the propositional logic. This is called satisfiability planning. We discuss two translations of deterministic planning to the propositional logic. The second translation is more complicated but also more efficient as it avoids considering all interleavings of a set of mutually independent operators.

We conclude the chapter by presenting the main results on the computational complexity of deterministic planning.

3.1 Problem definition

We formally define the deterministic planning problem.

Definition 3.1 *A 4-tuple $\langle A, I, O, G\rangle$ consisting of a set A of state variables, a state I (a valuation of A), a set O of operators over A, and a propositional formula G over A, is* a problem instance in deterministic planning.

The state I is the *initial state* and the formula G describes the set of *goal states*.

Definition 3.2 *Let $\Pi = \langle A, I, O, G\rangle$ be a problem instance in deterministic planning. A sequence $o_1, \ldots, o_n$ of operators is a* plan *for Π if and only if $app_{o_n}(app_{o_{n-1}}(\cdots app_{o_1}(I)\cdots)) \models G$, that is, when applying the operators $o_1, \ldots, o_n$ in this order starting in the initial state, one of the goal states is reached.*

3.2 State-space search

The simplest planning algorithm just generates all states (valuations of A), constructs the transition graph, and then finds a path from the initial state I to a goal state $g \in G$ for example by a shortest-path algorithm. The plan is then simply the sequence of actions corresponding to the edges on the shortest path from the initial state to a goal state.

However, this algorithm is in general not feasible when the number of state variables is higher than 20 or 30, as the number of valuations is very high: $2^{20} = 1048576 \sim 10^6$ for 20 Boolean state variables, and $2^{30} = 1073741824 \sim 10^9$ for 30.

Instead, it will often be much more efficient to avoid generating most of the state space explicitly, and just to produce the successor or predecessor states of the states currently under consideration. This is how many of the modern planning algorithms work.

There are two main possibilities in finding a path from the initial state to a goal state: traverse the transition graph forward starting from the initial state, or traverse it backwards starting from the goal states.

The main difference between these is caused by the fact that there may be several goal states (and even one goal state may have several possible predecessor states with respect to one operator) but only one initial state: in forward traversal we repeatedly compute the unique successor state of the current state, whereas with backward traversal we are forced to keep track of a possibly very high number of possible predecessor states of the goal states.

Again, it is difficult to say which one is in general better. Backward search is slightly more complicated to implement, but when the number of goal states is high, it allows to simultaneously consider a high number of potential suffixes of a plan, each leading to one of the goal states.

3.2.1 Progression and forward search

We already defined progression for single states s as $app_o(s)$, and the definition of the deterministic planning problem in Section 3.1 suggests a simple algorithm that does not require the explicit representation of the transition graph: generate a search tree starting from the initial state as the root node, and generate the children nodes by computing successor states by progression. Any node corresponding to a state s such that $s \models G$ corresponds to a plan: the plan is simply the sequence of operators from the root node to the node.

Later in this chapter we discuss more sophisticated ways of doing plan search with progression, as well as computation of distance estimates for guiding heuristic search algorithms.

3.2.2 Regression and backward search

With backward search the starting point is a propositional formula G that describes the set of goal states. An operator is selected, and the set of possible predecessor states is computed, and this set again is described by a propositional formula. One step in this computation, called *regression*, is more complicated than computing unique successor states of deterministic operators by progression. Reasons for this are that a state and an operator do not in general determine the predecessor state uniquely (one state may have several predecessors), and that we have to handle arbitrary propositional formulae instead of single states.

Definition 3.3 *We define the condition $EPC_l(o)$ of literal l becoming true when the operator $\langle c, e\rangle$ is applied as $EPC_l(e)$ defined recursively as follows.*

$$\begin{aligned} EPC_l(\top) &= \bot \\ EPC_l(l) &= \top \\ EPC_l(l') &= \bot \text{ when } l \neq l' \text{ (for literals } l') \\ EPC_l(e_1 \wedge \cdots \wedge e_n) &= EPC_l(e_1) \vee \cdots \vee EPC_l(e_n) \\ EPC_l(c \rhd e) &= EPC_l(e) \wedge c \end{aligned}$$

For effects e, the truth-value of the formula $EPC_l(e)$ indicates whether l is one of the literals that the effect e assigns the value true. The connection to the earlier definition of $[e]_s$ is explained by the following lemma.

Lemma 3.4 *Let A be the set of state variables, s be a state on A, l a literal on A, and e an effect on A. Then $l \in [e]_s$ if and only if $s \models EPC_l(e)$.*

Proof: Proof is by induction on the structure of the effect e.

Base case 1, $e = \top$: By definition of $[\top]_s$ we have $l \notin [\top]_s = \emptyset$, and by definition of $EPC_l(\top)$ we have $s \not\models EPC_l(\top) = \bot$, so the equivalence holds.

Base case 2, $e = l$: $l \in [l]_s = \{l\}$ by definition, and $s \models EPC_l(l) = \top$ by definition.

Base case 3, $e = l'$ for some literal $l' \neq l$: $l \notin [l']_s = \{'l\}$ by definition, and $s \not\models EPC_l(l') = \bot$ by definition.

Inductive case 1, $e = e_1 \wedge \cdots \wedge e_n$:

$l \in [e]_s$ if and only if $l \in [e']_s$ for some $e' \in \{e_1, \ldots, e_n\}$
if and only if $s \models EPC_l(e')$ for some $e' \in \{e_1, \ldots, e_n\}$
if and only if $s \models EPC_l(e_1) \vee \cdots \vee EPC_l(e_n)$
if and only if $s \models EPC_l(e_1 \wedge \cdots \wedge e_n)$.

The second equivalence is by the induction hypothesis, the other equivalences are by the definitions of $EPC_l(e)$ and $[e]_s$ as well as elementary facts about propositional formulae.

Inductive case 2, $e = c \rhd e'$:

$l \in [c \rhd e']_s$ if and only if $l \in [e']_s$ and $s \models c$
if and only if $s \models EPC_l(e')$ and $s \models c$
if and only if $s \models EPC_l(c \rhd e')$.

The second equivalence is by the induction hypothesis.

This completes the proof. □

Notice that any operator $\langle c, e\rangle$ can be expressed in normal form in terms of $EPC_a(e)$ as

$$\left\langle c, \bigwedge_{a\in A} (EPC_a(e) \rhd a) \wedge (EPC_{\neg a}(e) \rhd \neg a) \right\rangle.$$

The formula $(a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)$ expresses the truth-value of $a \in A$ after applying o in terms of truth-values of formulae before applying o: either a was true before and did not become false, or a became true.

Lemma 3.5 *Let $a \in A$ be a state variable and $o = \langle c, e\rangle \in O$ an operator. Let s be a state and $s' = app_o(s)$. Then $s \models (a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)$ if and only if $s' \models a$.*

Proof: Assume that $s \models (a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)$. We perform a case analysis and show that $s' \models a$ holds in both cases.

Case 1: Assume that $s \models a \wedge \neg EPC_{\neg a}(e)$. By Lemma 3.4 $\neg a \notin [e]_s$. Hence a remains true in s'.

Case 2: Assume that $s \models EPC_a(e)$. By Lemma 3.4 $a \in [e]_s$, and hence $s' \models a$.

For the other half of the equivalence, assume that $s \not\models (a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)$. Hence $s \models (\neg a \vee EPC_{\neg a}(e)) \wedge \neg EPC_a(e)$.

Assume that $s \models a$. Now $s \models EPC_{\neg a}(e)$ because $s \models \neg a \vee EPC_{\neg a}(e)$, and hence by Lemma 3.4 $\neg a \in [e]_s$ and hence $s' \not\models a$.

Assume that $s \not\models a$. Because $s \models \neg EPC_a(e)$, by Lemma 3.4 $a \notin [e]_s$ and hence $s' \not\models a$.

Therefore $s' \not\models a$ in all cases. □

The formulae $EPC_l(o)$ can now be used in defining regression for operators o.

Definition 3.6 (Regression) *Let ϕ be a propositional formula. Let $\langle p, e\rangle$ be an operator. The* regression *of ϕ with respect to $o = \langle p, e\rangle$ is $regr_o(\phi) = \phi_r \wedge p \wedge f$ where ϕ_r is obtained from ϕ by replacing every proposition $a \in A$ by $(a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)$, and $f = \bigwedge_{a\in A} \neg(EPC_a(e) \wedge EPC_{\neg a}(e))$. We also define $regr_e(\phi) = \phi_r \wedge f$.*

The conjuncts of f say that none of the state variables may simultaneously become true and false.

Because $regr_e(\phi)$ often contains many occurrences of $\bot$ and $\top$, it is useful to simplify it by applying equivalences like $\top \wedge \phi \equiv \phi$, $\bot \wedge \phi \equiv \bot$, $\top \vee \phi \equiv \top$, $\bot \vee \phi \equiv \phi$, $\neg\bot \equiv \top$, and $\neg\top \equiv \bot$.

Regression can equivalently be defined in terms of the conditions the state variables stay or become false, that is, we could use the formula $(\neg a \wedge \neg EPC_a(e)) \vee EPC_{\neg a}(e)$ which tells when a is false. The negation of this formula, which can be written as $(a \wedge \neg EPC_{\neg a}(e)) \vee (EPC_a(e) \wedge \neg EPC_{\neg a}(e))$, is not equivalent to $(a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)$. However, if $EPC_a(e)$ and $EPC_{\neg a}(e)$ are never simultaneously true, we do get equivalence, that is,

$$\begin{aligned} \neg(EPC_a(e) \wedge EPC_{\neg a}(e)) \quad \models \quad & ((a \wedge \neg EPC_{\neg a}(e)) \vee (EPC_a(e) \wedge \neg EPC_{\neg a}(e))) \\ & \leftrightarrow ((a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)) \end{aligned}$$

because $\neg(EPC_a(e) \wedge EPC_{\neg a}(e)) \models (EPC_a(e) \wedge \neg EPC_{\neg a}(e)) \leftrightarrow EPC_a(e)$.

Concerning the worst-case size of the formula obtained by regression with operators $o_1, \ldots, o_n$ starting from ϕ, the obvious upper bound on its size is the product of the sizes of $\phi, o_1, \ldots, o_n$, which is exponential in n. However, because of the many possibilities of simplifying the formulae and the typically simple structure of the operators, the formulae can often be simplified a lot. For unconditional operators $o_1, \ldots, o_n$ (with no occurrences of $\rhd$), an upper bound on the size of the formula (after the obvious simplifications that eliminate occurrences of $\top$ and $\bot$) is the sum of the sizes of $o_1, \ldots, o_n$ and ϕ.

The reason why regression is useful for planning is that it allows to compute the predecessor states by simple formula manipulation. The same is not possible for progression, that is, there does not seem to be a simple definition of successor states of a *set* of states expressed in terms of a formula: simple syntactic progression is restricted to individual states only.

The important property of regression is formalized in the following lemma.

Lemma 3.7 *Let ϕ be a formula over A. Let o be an operator with effect e. Let s be any state and $s' = app_o(s)$. Then $s \models regr_e(\phi)$ if and only if $s' \models \phi$.*

Proof: The proof is by structural induction over subformulae ϕ' of ϕ. We show that the formula ϕ_r obtained from ϕ by replacing propositions $a \in A$ by $(a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)$ has the same truth-value in s as ϕ has in s'.

Induction hypothesis: $s \models \phi'_r$ if and only if $s' \models \phi'$.

Base case 1, $\phi' = \top$: Now $\phi'_r = \top$ and both are true in the respective states.

Base case 2, $\phi' = \bot$: Now $\phi'_r = \bot$ and both are false in the respective states.

Base case 3, $\phi' = a$ for some $a \in A$: Now $\phi'_r = (a \wedge \neg EPC_{\neg a}(e)) \vee EPC_a(e)$. By Lemma 3.5 $s \models \phi'_r$ if and only if $s' \models \phi'$.

Inductive case 1, $\phi' = \neg\psi$: By the induction hypothesis $s \models \psi_r$ iff $s' \models \psi$. Hence $s \models \phi'_r$ iff $s' \models \phi'$ by the truth-definition of $\neg$.

Inductive case 2, $\phi' = \psi \vee \psi'$: By the induction hypothesis $s \models \psi_r$ iff $s' \models \psi$, and $s \models \psi'_r$ iff $s' \models \psi'$. Hence $s \models \phi'_r$ iff $s' \models \phi'$ by the truth-definition of $\vee$.

Inductive case 3, $\phi' = \psi \wedge \psi'$: By the induction hypothesis $s \models \psi_r$ iff $s' \models \psi$, and $s \models \psi'_r$ iff $s' \models \psi'$. Hence $s \models \phi'_r$ iff $s' \models \phi'$ by the truth-definition of $\wedge$. □

Operators for regression can be selected arbitrarily, but there is a simple property all useful regression steps satisfy. For example, regressing a with the effect $\neg a$ is not useful, because the new formula $\bot$ describes the empty set of states, and therefore the operators leading to it from the goal formula are not the suffix of any plan. Another example is regressing a with the operator $\langle b, c\rangle$, yielding $regr_{\langle b,c\rangle}(a) = a \wedge b$, which means that the set of states becomes smaller. This does not rule out finding a plan, but finding a plan is more difficult than it was before the regression step, because the set of possible prefixes of a plan leading to the current set of states is smaller than it was before. Hence it would be better not to take this regression step.

Lemma 3.8 *Let there be a plan $o_1, \ldots, o_n$ for $\langle A, I, O, G\rangle$. If $regr_{o_k}(regr_{o_{k+1}}(\cdots regr_{o_n}(G)\cdots)) \models regr_{o_{k+1}}(\cdots regr_{o_n}(G)\cdots)$ for some $k \in \{1, \ldots, n-1\}$, then also $o_1, \ldots, o_{k-1}, o_{k+1}, \cdots, o_n$ is a plan for $\langle A, I, O, G\rangle$.*

Proof: □

Hence any regression step that makes the set of states smaller in the set-inclusion sense is unnecessary. However, testing whether this is the case may be computationally expensive.

Lemma 3.9 *The problem of testing that* $\mathit{regr}_o(\phi) \not\models \phi$ *is NP-hard.*

Proof: We give a reduction from the NP-complete satisfiability problem of the propositional logic.

Let ϕ be any formula. Let a be a propositional variable not occurring in ϕ. Now $\mathit{regr}_{\langle \neg\phi \rightarrow a, a\rangle}(a) \not\models a$ if and only if $(\neg\phi \rightarrow a) \not\models a$, because $\mathit{regr}_{\langle \neg\phi \rightarrow a, a\rangle}(a) = \neg\phi \rightarrow a$. $(\neg\phi \rightarrow a) \not\models a$ is equivalent to $\not\models (\neg\phi \rightarrow a) \rightarrow a$ that is equivalent to the satisfiability of $\neg((\neg\phi \rightarrow a) \rightarrow a)$. Further, $\neg((\neg\phi \rightarrow a) \rightarrow a)$ is logically equivalent to $\neg(\neg(\phi \vee a) \vee a)$ and further to $\neg(\neg\phi \vee a)$ and $\phi \wedge \neg a$.

Satisfiability of $\phi \wedge \neg a$ is equivalent to the satisfiability of ϕ as a does not occur in ϕ: if ϕ is satisfiable, there is a valuation v such that $v \models \phi$, we can set a false in v to obtain v', and as a does not occur in ϕ, we still have $v' \models \phi$, and further $v' \models \phi \wedge \neg a$. Clearly, if ϕ is unsatisfiable also $\phi \wedge \neg a$ is.

Hence $\mathit{regr}_{\langle \neg\phi \rightarrow a, a\rangle}(a) \not\models a$ if and only if ϕ is satisfiable. □

The problem is also in NP, but we do not show it here. Also the following problem is in NP, but we just show the NP-hardness. The question is whether an empty set of states is produced by a regression step, that is, whether the resulting formula is unsatisfiable.

Lemma 3.10 *The problem of testing that* $\mathit{regr}_o(\phi)$ *is satisfiable is NP-hard.*

Proof: By a reduction from satisfiability in the propositional logic. Let ϕ be a formula. $\mathit{regr}_{\langle \phi, a\rangle}(a)$ is satisfiable if and only if ϕ is satisfiable because $\mathit{regr}_{\langle \phi, a\rangle}(a) \equiv \phi$.

The problem is NP-hard even if we restrict to operators that have a satisfiable precondition: ϕ is satisfiable if and only if $(\phi \vee \neg a) \wedge a$ is satisfiable if and only if $\mathit{regr}_{\langle \phi \vee \neg a, b\rangle}(a \wedge b)$ is satisfiable. Here a is a proposition not occurring in ϕ. Clearly, $\phi \vee \neg a$ is true when a is false, and hence $\phi \vee \neg a$ is satisfiable. □

Of course, testing that $\mathit{regr}_o(\phi) \not\models \phi$ or that $\mathit{regr}_o(\phi)$ is satisfiable is not necessary for the correctness of backward search, but avoiding useless steps improves its efficiency.

Early work on planning restricted to goals and operator preconditions that are conjunctions of state variables, and to unconditional operator effects (STRIPS operators.) In this special case both goals G and operator effects e can be viewed as sets of literals, and the definition of regression is particularly simple: regressing G with respect to $\langle c, e\rangle$ is $(G \backslash e) \cup c$. If there is $a \in A$ such that $a \in G$ and $\neg a \in e$, then the result of regression is $\bot$, that is, the empty set of states. We do not use this restricted type of regression in this lecture.

Some planners that use backward search and have operators with disjunctive preconditions and conditional effects eliminate all disjunctivity by branching: for example, the backward step from g with operator $\langle a \vee b, g\rangle$, producing $a \vee b$, is handled by producing two branches in the search tree, one for a and another for b. Disjunctivity caused by conditional effects can similarly be handled by branching. However, this branching leads to a very high branching factor for the search tree and thus to poor performance.

In addition to being the basis of backward search, regression has many other useful applications in reasoning about actions and formal manipulation of operators.

Definition 3.11 (Composition of operators) *Let $o_1 = \langle p_1, e_1 \rangle$ and $o_2 = \langle p_2, e_2 \rangle$ be two operators on A. Then their* composition $o_1 \circ o_2$ *is defined as*

$$\left\langle p, \bigwedge_{a \in A} \left(\begin{array}{l} ((\mathit{regr}_{e_1}(\mathit{EPC}_a(e_2)) \vee (\mathit{EPC}_a(e_1) \wedge \neg \mathit{regr}_{e_1}(\mathit{EPC}_{\neg a}(e_2)))) \rhd a) \wedge \\ ((\mathit{regr}_{e_1}(\mathit{EPC}_{\neg a}(e_2)) \vee (\mathit{EPC}_{\neg a}(e_1) \wedge \neg \mathit{regr}_{e_1}(\mathit{EPC}_a(e_2)))) \rhd \neg a) \end{array} \right) \right\rangle$$

where $p = p_1 \wedge \mathit{regr}_{e_1}(p_2) \wedge \bigwedge_{a \in A} \neg (\mathit{EPC}_a(e_1) \wedge \mathit{EPC}_{\neg a}(e_1))$.

Notice that in $o_1 \circ o_2$ first is o_1 is applied and then o_2, so the ordering is opposite to the usual notation for the composition of functions.

Theorem 3.12 *Let o_1 and o_2 be operators and s a state. Then $\mathit{app}_{o_1 \circ o_2}(s)$ is defined if and only if $\mathit{app}_{o_1;o_2}(s)$ is defined, and $\mathit{app}_{o_1 \circ o_2}(s) = \mathit{app}_{o_1;o_2}(s)$.*

Proof: □

The above construction can be used in eliminating *sequantial composition* from operator effects (Section 2.3.1).

3.3 Planning by heuristic search algorithms

Plan search can be performed in the forward or in the backward direction respectively with progression or regression, as described in Sections 3.2.1 and 3.2.2. There are several obvious algorithms that could be used for the purpose, including depth-first search, breadth-first search and iterative deepening, but without informed selection of branches of search trees these algorithms perform poorly.

The use of additional information for guiding search is essential for achieving efficient planning with general-purpose search algorithms. Algorithms that use heuristic estimates on the values of the nodes in the search space for guiding the search have been applied to planning very successfully. Some of the more sophisticated search algorithms that can be used are A$*$ [Hart *et al.*, 1968], WA$*$ [Pearl, 1984], IDA$*$ [Korf, 1985], simulated annealing [Kirkpatrick *et al.*, 1983].

The effectiveness of these algorithms is dependent on good heuristics for guiding the search. For planning with progression and regression the main heuristic information is in the form of estimates on the distance between states. The distance is the minimum number of operators needed for reaching a state from another state. In Section 3.4 we present techniques for estimating the distances between states and sets of sets. In this section we discuss how heuristic search algorithms are applied in planning assuming that we have a useful heuristics for guiding these algorithms

When plan search proceeds by progression in forward direction starting from the initial state, we estimate the distance between the current state and the set of goal states. When plan search proceeds by regression in backward direction starting from the goal states, we estimate the distance between the initial state and the current set of goal states as computed by regression.

For progression, the search tree nodes are sequences of operators (prefixes of plans.)

$$o_1, o_2, \ldots, o_n$$

The initial node for search is the empty sequence. The children nodes are obtained by progression with respect to an operator or by dropping out some of the last operators.

Definition 3.13 (Children for progression) *Let $\langle A, I, O, G\rangle$ be a problem instance. For progression, the children of a search tree node $o_1, o_2, \ldots, o_n$ are the following.*

1. $o_1, o_2, \ldots, o_n, o$ *for any* $o \in O$ *such that* $app_{o_1;\ldots;o_n;o}(I)$ *is defined*
2. $o_1, o_2, \ldots, o_i$ *for any* $i < n$

When $app_{o_1;o_2\ldots,o_n}(I) \models G$ then $o_1, \ldots, o_n$ is a plan.

For regression, the nodes of the search tree are also sequences of operators (suffixes of plans.)

$$o_n, \ldots, o_1$$

The initial node for search is the empty sequence. The children of a node are those obtained by prefixing the current sequence with an operator or by dropping out some of the first actions and associated formulae.

Definition 3.14 (Children for regression) *Let $\langle A, I, O, G\rangle$ be a problem instance. For regression, the children of node $\phi_n, \ldots, o_1$ are the following.*

1. $o, o_n, \ldots, o_1$ *for any* $o \in O$
2. $o_i, \ldots, o_1$ *for any* $i < n$

When $I \models regr_{o_n;\ldots;o_1}(G)$ the sequence $o_n, \ldots, o_1$ is a plan.

For both progression and regression the neighbors that are obtained by removing some operators from the incomplete plans are needed with local search algorithms only. The systematic search algorithms can be implemented to keep track of the alternative extensions of an incomplete plan, and therefore the backup steps are not needed. Further, for these algorithms it suffices to keep track of the results of the state obtained by progression or the formula obtained by regression.

The states generated by progression from the initial state, and the formulae generated by regression are not the only possibilities for defining the search space for a search algorithm. In partial-order planning [McAllester and Rosenblitt, 1991], the search space consists of incomplete plans that are partially ordered multisets of operators. The neighbors of an incomplete plan are those obtained by adding or removing an operator, or by adding or removing an ordering constraint. Another form of incomplete plans is fixed length sequences of operators, with zero or more of the operators missing. This has been formalized as planning with propositional satisfiability as discussed in Section 3.5.

3.4 Distance estimation

Using progression and regression with just any search algorithm does not yield efficient planning. Critical for the usefulness of the algorithms is the selection of operators for the progression and regression steps. If the operators are selected randomly it is unlikely that search in possibly huge transition graphs is going to end quickly.

Operator selection can be substantially improved by using estimates on the distance between the initial state and the current goal states, for backward search, or the distance between the current state and the set of goal states, for forward search. Computing exact distances is computationally just as difficult as solving the planning problem itself. Therefore in order to speed up planning by distance information, its computation should be inexpensive, and this means that only inaccurate estimates of the distances can be used.

We present a method for distance estimation that generalizes the work of Bonet and Geffner [2001] to operators with conditional effects and arbitrary propositional formulae as preconditions.

The set makestrue(l, O), consisting of formulae ϕ such that if ϕ is true then applying an operator $o \in O$ can make the literal l true, is defined on the basis of $EPC_l(o)$ from Definition 3.3.

$$\text{makestrue}(l, O) = \{EPC_l(o) | o \in O\}$$

Example 3.15 Let $\langle A \wedge B, R \wedge (Q \rhd P) \wedge (R \rhd P) \rangle$ be an operator in O. Then $A \wedge B \wedge (Q \vee R) \in$ makestrue(P, O) because for P to become true it suffices that the precondition $A \wedge B$ of the operator and one of the antecedents Q or R of a conditional effect is true. ■

The idea of the method for estimating distances of goal states is based on the estimation of distances of states in which given state variables have given values. The estimates are not accurate for two reasons. First, and more importantly, distance estimation is done one state variable at a time and dependencies between values of different state variables are ignored. Second, tests whether a formula is true in a set of states described by a set of literals is performed by an algorithm that approximates NP-hard satisfiability testing. Of course, because we are interested in computing distance estimates efficiently, that is in polynomial time, the inaccuracy is an acceptable compromise.

We give a recursive procedure that computes a lower bound on the number of operator applications that are needed for reaching from a state s a state in which given state variables $a \in A$ have a certain value. This is by computing a sequence of sets D_i of literals. The set D_i is a set of such literals that must be true in any state that has a distance $\leq i$ from the state s. If a literal l is in D_0, then l is true in s. If $l \in D_i \backslash D_{i+1}$, then l is true in all states with distance $\leq i$ and l may be false in some states having distance $> i$.

Definition 3.16 *Let $L = A \cup \{\neg a | a \in A\}$ be the set of literals on A. Define the sets D_i for $i \geq 0$ as follows.*

$$\begin{array}{rcl} D_0 & = & \{l \in L | s \models l\} \\ D_i & = & D_{i-1} \backslash \{l \in L | o \in O, \textit{canbetruein}(EPC_{\bar{l}}(o), D_{i-1})\} \end{array}$$

Because we consider only finite sets A of state variables and $|D_0| = |A|$ and $D_{i+1} \subseteq D_i$ for all $i \geq 0$, necessarily $D_i = D_{i+1}$ for some $i \leq |A|$.

Above canbetruein(ϕ, D) is a function that tests whether there is a state in which ϕ and the literals D are true, that is, whether $\{\phi\} \cup D$ is satisfiable. This algorithm does not accurately test satisfiability, and may claim tht $\{\phi\} \cup D$ is satisfiable even when it is not. Hence it only approximates the NP-complete satisfiability problem. The algorithm runs in polynomial time and is defined as follows.

$$\begin{array}{rcl} \textit{canbetruein}(\bot, D) & = & \textit{false} \\ \textit{canbetruein}(\top, D) & = & \textit{true} \\ \textit{canbetruein}(a, D) & = & \textit{true iff } \neg a \notin D \ \ (\textit{for state variables } a \in A) \\ \textit{canbetruein}(\neg a, D) & = & \textit{true iff } a \notin D \ \ (\textit{for state variables } a \in A) \\ \textit{canbetruein}(\neg\neg\phi, D) & = & \textit{canbetruein}(\phi, D) \\ \textit{canbetruein}(\phi \vee \psi, D) & = & \textit{canbetruein}(\phi, D) \textit{ or canbetruein}(\psi, D) \\ \textit{canbetruein}(\phi \wedge \psi, D) & = & \textit{canbetruein}(\phi, D) \textit{ and canbetruein}(\psi, D) \\ \textit{canbetruein}(\neg(\phi \vee \psi), D) & = & \textit{canbetruein}(\neg\phi, D) \textit{ and canbetruein}(\neg\psi, D) \\ \textit{canbetruein}(\neg(\phi \wedge \psi), D) & = & \textit{canbetruein}(\neg\phi, D) \textit{ or canbetruein}(\neg\psi, D) \end{array}$$

The reason why the satisfiability test is not accurate is that for formulae $\phi \wedge \psi$ (respectively $\neg(\phi \vee \psi)$) we make recursively two satisfiability tests that do not assume that the component formulae ϕ and ψ (respectively $\neg\phi$ and $\neg\psi$) are *simultaneously* satisfiable.

We give a lemma that states the connection between canbetruein(ϕ, D) and the satisfiabilty of $\{\phi\} \cup D$.

Lemma 3.17 *Let ϕ be a formula and D a consistent set of literals (it contains at most one of a and $\neg a$ for every $a \in A$.) If $D \cup \{\phi\}$ is satisfiable, then canbetruein(ϕ, D) returns true.*

Proof: The proof is by induction on the structure of ϕ.

Base case 1, $\phi = \bot$: The set $D \cup \{\bot\}$ is not satisfiable, and hence the implication trivially holds.

Base case 2, $\phi = \top$: canbetruein$(\top, D)$ always returns true, and hence the implication trivially holds.

Base case 3, $\phi = a$ for some $a \in A$: If $D \cup \{a\}$ is satisfiable, then $\neg a \notin D$, and hence canbetruein(a, D) returns true.

Base case 4, $\phi = \neg a$ for some $a \in A$: If $D \cup \{\neg a\}$ is satisfiable, then $a \notin D$, and hence canbetruein$(\neg a, D)$ returns true.

Inductive case 1, $\phi = \neg\neg\phi'$ for some ϕ': The formulae are logically equivalent, and by the induction hypothesis we directly establish the claim.

Inductive case 2, $\phi = \phi' \vee \psi'$: If $D \cup \{\phi' \vee \psi'\}$ is satisfiable, then either $D \cup \{\phi'\}$ or $D \cup \{\psi'\}$ is satisfiable and by the induction hypothesis at least one of canbetruein(ϕ', D) and canbetruein(ψ', D) returns true. Hence canbetruein$(\phi' \vee \psi', D)$ returns true.

Inductive case 3, $\phi = \phi' \wedge \psi'$: If $D \cup \{\phi' \wedge \psi'\}$ is satisfiable, then both $D \cup \{\phi'\}$ and $D \cup \{\psi'\}$ are satisfiable and by the induction hypothesis both canbetruein(ϕ', D) and canbetruein(ψ', D) return true. Hence canbetruein$(\phi' \wedge \psi', D)$ returns true.

Inductive cases 4 and 5, $\phi = \neg(\phi' \vee \psi')$ and $\phi = \neg(\phi' \wedge \psi')$: Like cases 2 and 3 by logical equivalence. □

The other direction of the implication does not hold because for example canbetruein$(A \wedge \neg A, D)$ returns true even though the formula is not satisfiable. The procedure is a polynomial-time approximation of the logical consequence test from a set of literals: canbetruein(ϕ, D) always returns true if $D \cup \{\phi\}$ is satisfiable, but it may return true also when the set is not satisfiable.

Now we define the distances of states in which a literal l is true by $\delta_s(l) = 0$ if and only if $l \in D_0$, and for $d \geq 1$, $\delta_s(l) = d$ if and only if $\bar{l} \in D_{d-1} \backslash D_d$. For formulae ϕ we similarly define $\delta_s(\phi) = 0$ if canbetruein(ϕ, D_0), and for $d \geq 1$, $\delta_s(\phi) = d$ if canbetruein(ϕ, D_d) and not canbetruein(ϕ, D_{d-1}).

Lemma 3.18 *Let s be a state and $D_0, D_1, \ldots$ the sets given in Definition 3.16 for s. If s' is the state reached from s by applying the operator sequence $o_1, \ldots, o_n$, then $s' \models D_n$.*

Proof: By induction on n.

Base case $n = 0$: The length of the operator sequence is zero, and hence $s' = s$. The set D_0 consists exactly of those literals that are true in s, and hence $s' \models D_0$.

Inductive case $n \geq 1$: Let s'' be the state reached from s by applying $o_1, \ldots, o_{n-1}$. Now $s' = app_{o_n}(s'')$. By the induction hypothesis $s'' \models D_{n-1}$.

Let l be any literal in D_n. We show it is true in s'. Because $l \in D_n$ and $D_n \subseteq D_{n-1}$, also $l \in D_{n-1}$, and hence by the induction hypothesis $s'' \models l$.

Let ϕ be any member of makestrue$(\bar{l}, \{o_n\})$. Because $l \in D_n$ it must be that canbetruein(ϕ, D_{n-1}) returns false (Definition of D_n). Hence $D_{n-1} \cup \{\phi\}$ is by Lemma 3.17 not satisfiable, and $s'' \not\models \phi$. Hence applying o_n in s'' does not make l false, and consequently $s' \models l$.

□

Theorem 3.19 *Let s be a state, ϕ a formula, and $D_0, D_1, \ldots$ the sets given in Definition 3.16 for s. If s' is the state reached from s by applying the operators $o_1, \ldots, o_n$ and $s' \models \phi$ for any formula ϕ, then canbetruein(ϕ, D_n) returns true.*

Proof: By Lemma 3.18 $s' \models D_n$. By assumption $s' \models \phi$. Hence $D_n \cup \{\phi\}$ is satisfiable. By Lemma 3.17 canbetruein(ϕ, D_n) returns true. □

Corollary 3.20 *Let s be a state and ϕ a formula. Then for any sequence $o_1, \ldots, o_n$ of operators such that executing them in s results in state s' such that $s' \models \phi$, $n \geq \delta_s(\phi)$.*

Example 3.21 Consider the blocks world with three blocks and the initial state in which A is on B and B is on C.

$$\begin{aligned} D_0 \;=\; & \{\text{A-CLEAR}, \text{A-ON-B}, \text{B-ON-C}, \text{C-ON-TABLE}, \neg\text{A-ON-C}, \neg\text{B-ON-A}, \neg\text{C-ON-A}, \\ & \neg\text{C-ON-B}, \neg\text{A-ON-TABLE}, \neg\text{B-ON-TABLE}, \neg\text{B-CLEAR}, \neg\text{C-CLEAR}\} \end{aligned}$$

There is only one operator applicable, that moves A onto the table. Applying this operator makes the literals B-CLEAR and A-ON-TABLE and ¬A-ON-B true, and consequently their complementary literals do not occur in D_1, because it is possible after at most 1 operator application that these complementary literals are false.

$$\begin{aligned} D_1 \;=\; & \{\text{A-CLEAR}, \text{B-ON-C}, \text{C-ON-TABLE}, \neg\text{A-ON-C}, \neg\text{B-ON-A}, \neg\text{C-ON-A}, \\ & \neg\text{C-ON-B}, \neg\text{B-ON-TABLE}, \neg\text{C-CLEAR}\} \end{aligned}$$

In addition the operator applicable in the initial states, now there are three more operators applicable (their precondition does not contradict D_1), one moving A from the table on top of B (returning to the initial state), one moving B from the top of C onto A, and one moving B from the top of C onto the table. Hence D_2 is as follows.

$$D_2 \;=\; \{\text{C-ON-TABLE}, \neg\text{A-ON-C}, \neg\text{C-ON-A}, \neg\text{C-ON-B}\}$$

Now there are three further operators applicable, those moving C from the table onto A and onto B, and the operator moving A onto C. Consequently,

$$D_3 \;=\; \emptyset$$

■

The next two examples demonstrate the best-case and worst-case scenarios for distance estimation.

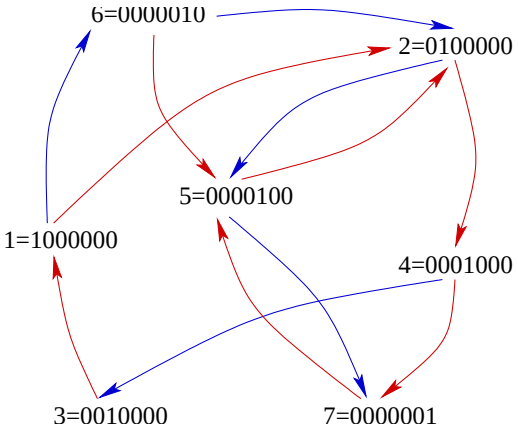


Figure 3.1: A transition system on which distance estimates are very accurate

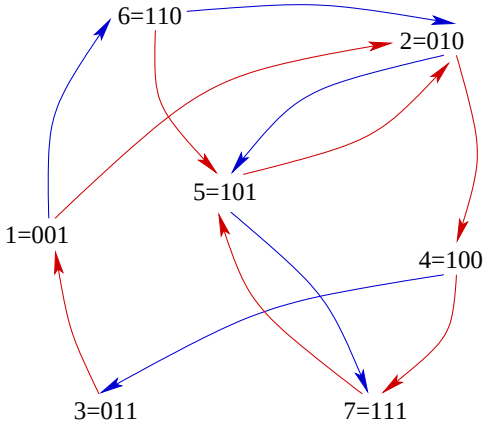


Figure 3.2: A transition system for which distance estimates are very inaccurate

Example 3.22 Figure 3.1 shows a transition system on which the distance estimates from state 1 are very accurate. The accuracy of the estimates is caused by the fact that for each state one can determine the distance accurately just on the basis of one of the state variables. Let the state variables be A, B, C, D, E, F, G.

$$\begin{aligned} D_0 &= \{A, \neg B, \neg C, \neg D, \neg E, \neg F, \neg G\} \\ D_1 &= \{\neg C, \neg D, \neg E, \neg G\} \\ D_2 &= \{\neg C, \neg G\} \\ D_3 &= \emptyset \\ D_4 &= \emptyset \end{aligned}$$

■

Example 3.23 Figure 3.2 shows a transition system on which the distance estimates from state 1 are very poor. The inaccuracy is caused by the fact that all possible values of state variables are possible after taking just one action, and this immediately gives distance estimate 1 for all states

and sets of states.

$$\begin{aligned} D_0 &= \{\neg A, \neg B, C\} \\ D_1 &= \emptyset \\ D_2 &= \emptyset \end{aligned}$$

■

The way Bonet and Geffner [2001] express the method differs from our presentation. Their definition is based on two mutually recursive equations that cannot be directly understood as executable procedures. The basic equation for distance computation for literals (negated and unnegated) state variables l ($l = a$ or $l = \neg a$ for some $a \in A$) is as follows (the generalization of this and the following definitions to arbitrary preconditions and conditional effects are due to us.)

$$\delta_s(l) = \begin{cases} 0 & \text{if } s \models l \\ \min_{\phi \in \text{makestrue}(l,O)} (1 + \delta_s(\phi)) & \text{otherwise} \end{cases}$$

The equation gives a cost/distance estimate of making a proposition $a \in A$ true starting from state s, in terms of s and cost $\delta_s(\phi)$ of reaching a state that satisfies ϕ. This cost $\delta_s(\phi)$ is defined as follows in terms of the costs $\delta_s(a)$.

$$\begin{aligned} \delta_s(\bot) &= \infty \\ \delta_s(\top) &= 0 \\ \delta_s(a) &= \delta_s(a) \text{ for state variables } a \in A \\ \delta_s(\neg a) &= \delta_s(\neg a) \text{ for state variables } a \in A \\ \delta_s(\neg\neg\phi) &= \delta_s(\phi) \\ \delta_s(\phi \vee \psi) &= \min(\delta_s(\phi), \delta_s(\psi)) \\ \delta_s(\phi \wedge \psi) &= \max(\delta_s(\phi), \delta_s(\psi)) \\ \delta_s(\neg(\phi \vee \psi)) &= \delta_s(\neg\phi \wedge \neg\psi) \\ \delta_s(\neg(\phi \wedge \psi)) &= \delta_s(\neg\phi \vee \neg\psi) \end{aligned}$$

This representation of the estimation method is useful because Bonet and Geffner [2001] have also considered another way of defining the cost of achieving a conjunction $\phi \wedge \psi$. Instead of taking the maximum of the costs of ϕ and ψ, Bonet and Geffner suggest taking the sum of the costs, which is simply obtained by replacing $\max(\delta_s(\phi), \delta_s(\psi))$ in the above equations by $\delta_s(\phi) + \delta_s(\psi)$. They call this the *additive heuristic*, in contrast to the definition given above for the *max heuristic*. The justification for this is that the max heuristic assumes that it is the cost of the more difficult conjunct that alone determines the difficulty of reaching the conjunction and the cost of the less difficult conjuncts are ignored completely. The experiments Bonet and Geffner conducted showed that the additive heuristic may lead to much more efficient planning. However, one should notice that the additive heuristic is not admissible, and indeed, Bonet and Geffner have used the max heuristic and the additive heuristic in connection with the non-optimal best-first search algorithm.

3.5 Planning as satisfiability in the propositional logic

A very powerful approach to deterministic planning emerged starting in 1992 from the work by Kautz and Selman [1992; 1996]: translate problem instances to propositional formulae $\phi_0, \phi_1, \phi_2, \ldots$ so that every valuation that satisfies formula ϕ_i corresponds to a plan of length i. Now an algorithm for testing the satisfiability of propositional formulae can be used for finding a plan: test the

satisfiability of ϕ_0, if it is unsatisfiable, continue with ϕ_1, ϕ_2, and so on, until a satisfiable formula ϕ_n is found. From the valuation the satisfiability algorithm returns we can now construct a plan of length n.

3.5.1 Actions as propositional formulae

First we need to represent all our actions in the propositional logic. We can view arbitrary propositional formulae as actions, or we can translate operators into formulae in the propositional logic. We discuss both of these possibilities.

Given a set of state variables $A = \{a_1, \ldots, a_n\}$, one could describe an action directly as a propositional formula ϕ over propositions $A \cup A'$ where $A' = \{a'_1, \ldots, a'_n\}$. Here the propositions A represent the values of state variables in the state s in which an action is taken, and propositions A' the values of state variables in a successor state s'.

A pair of valuations s and s' can be understood as a valuation of $A \cup A'$ (the state s assigns a value to propositions A and s' to propositions A'), and a transition from s to s' is possible if and only if $s, s' \models \phi$.

Example 3.24 Let there be state variables a_1 and a_2. The action that reverses the values of both state variables is described by $(a_1 \leftrightarrow \neg a'_1) \wedge (a_2 \leftrightarrow \neg a'_2)$.

This action is represented by the following matrix.

	$a'_1a'_2$ = 0 0	$a'_1a'_2$ = 0 1	$a'_1a'_2$ = 1 0	$a'_1a'_2$ = 1 1
$a_1a_2 = 00$	0	0	0	1
$a_1a_2 = 01$	0	0	1	0
$a_1a_2 = 10$	0	1	0	0
$a_1a_2 = 11$	1	0	0	0

The matrix can be equivalently represented as the following truth-table.

$a_1a_2a'_1a'_2$	
0 0 0 0	0
0 0 0 1	0
0 0 1 0	0
0 0 1 1	1
0 1 0 0	0
0 1 0 1	0
0 1 1 0	1
0 1 1 1	0
1 0 0 0	0
1 0 0 1	1
1 0 1 0	0
1 0 1 1	0
1 1 0 0	1
1 1 0 1	0
1 1 1 0	0
1 1 1 1	0

Of course, this is the truth-table of $(a_1 \leftrightarrow \neg a'_1) \wedge (a_2 \leftrightarrow \neg a'_2)$. ■

Example 3.25 Let the set of state variables be $A = \{a_1, a_2, a_3\}$. The formula $(a_1 \leftrightarrow a'_2) \wedge (a_2 \leftrightarrow a'_3) \wedge (a_3 \leftrightarrow a'_1)$ represents the action that rotates the values of the state variables a_1, a_2 and a_3 one position right. The formula can be represented as the following adjacency matrix. The rows correspond to valuations of A and the columns to valuations of $A' = \{a'_1, a'_2, a'_3\}$.

	000	001	010	011	100	101	110	111
000	1	0	0	0	0	0	0	0
001	0	0	0	0	1	0	0	0
010	0	1	0	0	0	0	0	0
011	0	0	0	0	0	1	0	0
100	0	0	1	0	0	0	0	0
101	0	0	0	0	0	0	1	0
110	0	0	0	1	0	0	0	0
111	0	0	0	0	0	0	0	1

A more conventional way of depicting the valuations of this formula would be as a truth-table with one row for every valuation of $A \cup A'$, a total of 64 rows. ■

This kind of propositional formulae are the basis of a number of planning algorithms that are based on reasoning in propositional logics. These formulae could be input to a planning algorithm, but describing actions in that way is usually more tricky than as operators, and these formulae are usually just automatically derived from operators.

The action in Example 3.25 is deterministic. Not all actions represented by propositional formulae are deterministic. A sufficient (but not necessary) condition for the determinism is that the formula is of the form $(\phi_1 \leftrightarrow a'_1) \wedge \cdots \wedge (\phi_n \leftrightarrow a'_n)$ with exactly one equivalence for every $a' \in A'$ and formulae ϕ_i not having occurrences of propositions in A'. This way the truth-value of every state variable in the successor state is unambiguously defined in terms of the truth-values of the state variables in the predecessor state, and hence the operator is deterministic.

3.5.2 Translation of operators into propositional logic

We first give the simplest possible translation of deterministic planning into the propositional logic. In this translation every operator is separately translated into a formula, and the choice between the operators can be represented by the disjunction connective of the propositional logic.

The formula τ_o that represents operator $o = \langle z, e \rangle$ is the conjunction of the precondition z and the formulae

$$((EPC_a(e) \vee (a \wedge \neg EPC_{\neg a}(e))) \leftrightarrow a') \wedge \neg(EPC_a(e) \wedge EPC_{\neg a}(e))$$

for every $a \in A$. Above the first conjunct expresses the value of a in the successor state in terms of the values of the state variables in the predecessor state. This is like in the definition of regression in Section 3.2.2. The second conjunct says that applying the operator is not possible if it assigns both the value 1 and 0 to a.

Example 3.26 Consider operator $\langle A \vee B, ((B \vee C) \rhd A) \wedge (\neg C \rhd \neg A) \wedge (A \rhd B) \rangle$.

The corresponding propositional formula is

$$\begin{array}{rl} (A \vee B) & \wedge(((B \vee C) \vee (A \wedge \neg\neg C)) \leftrightarrow A') \wedge \neg((B \vee C) \wedge \neg C) \\ & \wedge((A \vee (B \wedge \neg\bot)) \leftrightarrow B') \wedge \neg(A \wedge \bot) \\ & \wedge((\bot \vee (C \wedge \neg\bot)) \leftrightarrow C') \wedge \neg(\bot \wedge \bot) \\ \equiv & \\ (A \vee B) & \wedge(((B \vee C) \vee (A \wedge C)) \leftrightarrow A') \wedge \neg((B \vee C) \wedge \neg C) \\ & \wedge((A \vee B) \leftrightarrow B') \\ & \wedge(C \leftrightarrow C') \end{array}$$

■

Applying any of the operators $o_1, \ldots, o_n$ or none of the operators is now represented as the formula

$$\mathcal{R}_1(A, A') = \tau_{o_1} \vee \cdots \vee \tau_{o_n} \vee ((a_1 \leftrightarrow a'_1) \wedge \cdots \wedge (a_k \leftrightarrow a'_k))$$

where $A = \{a_1, \ldots, a_k\}$ is the set of all state variables. The last disjunct is for the case that no operator is applied.

The valuations that satisfy this formula do not uniquely determine which operator was applied, because for a given state two operators may produce the same successor state. However, in such cases it usually does not matter which operator is applied and one of them can be chosen arbitrarily.

3.5.3 Finding plans by satisfiability algorithms

We show how plans can be found by first translating problem instances $\langle A, I, O, G\rangle$ into propositional formulae, and then finding satisfying valuations by a satisfiability algorithm.

In Section 3.5.1 we showed how operators can be described by propositional formulae over sets A and A' of propositions, the set A describing the values of the state variables in the state in which the operator is applied, and the set A' describing the values of the state variables in the successor state of that state.

Now, for a fixed plan length n, we define sets of propositions $A_0, \ldots, A_n$ with propositions in A_i describing the values of the state variables at time point i, that is, when i operators (or sets of operators, if we have parallelism) have been applied.

Let $\langle A, I, O, G\rangle$ be a problem instance in deterministic planning.

The state at the first time point 0 is determined by I, and at the last time point n a goal state must have been reached. Therefore we include ι with time-labeling 0 and G with time-labeling n in the encoding.

$$\iota^0 \wedge \mathcal{R}_1(A_0, A_1) \wedge \mathcal{R}_1(A_1, A_2) \wedge \cdots \wedge \mathcal{R}_1(A_{n-1}, A_n) \wedge G^n$$

Here $\iota^0 = \bigwedge\{a^0 | a \in A, I(a) = 1\} \cup \{\neg a^0 | a \in A, I(a) = 0\}$ and G^n is G with propositions a replaced by a^n.

Plans are found incrementally by increasing the plan length and testing the satisfiability of the corresponding formulae: first try to find plans of length 0, then of length 1, 2, 3, and so on, until a plan is found. If there are no plans, it has to be somehow decided when to stop increasing the plan length that is tried. An upper bound on plan length is $2^{|A|} - 1$ where A is the set of state variables, but this upper bound does not provide a practical termination condition for this procedure.

The size of the encoding is linear in the plan length, and because the plan length may be exponential, the encoding might not be practical for very long plans, as runtimes of satisfiability algorithms in general grow exponentially in the length of the formulae.

Example 3.27 Consider an initial state that satisfies $I \models A \wedge B$, the goal $G = (A \wedge \neg B) \vee (\neg A \wedge B)$, and the operators $o_1 = \langle \top, (A \rhd \neg A) \wedge (\neg A \rhd A) \rangle$ and $o_2 = \langle \top, (B \rhd \neg B) \wedge (\neg B \rhd B) \rangle$.

The following formula is satisfiable if and only if $\langle A, I, \{o_1, o_2\}, G \rangle$ has a plan of length 3 or less.

$$
\begin{array}{l}
(A^0 \wedge B^0) \\
\wedge(((A^0 \leftrightarrow A^1) \wedge (B^0 \leftrightarrow \neg B^1)) \vee ((A^0 \leftrightarrow \neg A^1) \wedge (B^0 \leftrightarrow B^1)) \vee ((A^0 \leftrightarrow A^1) \wedge (B^0 \leftrightarrow B^1))) \\
\wedge(((A^1 \leftrightarrow A^2) \wedge (B^1 \leftrightarrow \neg B^2)) \vee ((A^1 \leftrightarrow \neg A^2) \wedge (B^1 \leftrightarrow B^2)) \vee ((A^1 \leftrightarrow A^2) \wedge (B^1 \leftrightarrow B^2))) \\
\wedge(((A^2 \leftrightarrow A^3) \wedge (B^2 \leftrightarrow \neg B^3)) \vee ((A^2 \leftrightarrow \neg A^3) \wedge (B^2 \leftrightarrow B^3)) \vee ((A^2 \leftrightarrow A^3) \wedge (B^2 \leftrightarrow B^3))) \\
\wedge((A^3 \wedge \neg B^3) \vee (\neg A^3 \wedge B^3))
\end{array}
$$

One of the valuations that satisfy the formula is the following.

	time i			
	0	1	2	3
A^i	1	0	0	0
B^i	1	1	0	1

This valuation corresponds to the plan that applies operator o_1 at time point 0, o_2 at time point 1, and o_2 at time point 2. There are also other satisfying valuations. The shortest plans for this problem instance are o_1 and o_2, each consisting of one operator only. ■

Example 3.28 Consider the following problem. There are two operators, one for rotating the values of bits ABC one step right, and the other for inverting the values of all the bits. Consider reaching from the initial state 100 the goal state 001 with two actions. This is represented as the following formula.

$$
\begin{array}{l}
(A^0 \wedge \neg B^0 \wedge \neg C^0) \\
\wedge(((A^0 \leftrightarrow B^1) \wedge (B^0 \leftrightarrow C^1) \wedge (C^0 \leftrightarrow A^1)) \vee ((\neg A^0 \leftrightarrow A_1) \wedge (\neg B^0 \leftrightarrow B^1) \wedge (\neg C^0 \leftrightarrow C^1))) \\
\wedge(((A^1 \leftrightarrow B^2) \wedge (B^1 \leftrightarrow C^2) \wedge (C^1 \leftrightarrow A^2)) \vee ((\neg A^1 \leftrightarrow A^2) \wedge (\neg B^1 \leftrightarrow B^2) \wedge (\neg C^1 \leftrightarrow C^2))) \\
\wedge(\neg A^2 \wedge \neg B^2 \wedge C^2)
\end{array}
$$

Because the literals describing the initial and the goal state must be true, we can replace other occurrences of these state variables by $\top$ and $\bot$.

$$
\begin{array}{l}
(A^0 \wedge \neg B^0 \wedge \neg C^0) \\
\wedge(((\top \leftrightarrow B^1) \wedge (\bot \leftrightarrow C^1) \wedge (\bot \leftrightarrow A^1)) \vee ((\neg \top \leftrightarrow A_1) \wedge (\neg \bot \leftrightarrow B^1) \wedge (\neg \bot \leftrightarrow C^1))) \\
\wedge(((A^1 \leftrightarrow \bot) \wedge (B^1 \leftrightarrow \top) \wedge (C^1 \leftrightarrow \bot)) \vee ((\neg A^1 \leftrightarrow \bot) \wedge (\neg B^1 \leftrightarrow \bot) \wedge (\neg C^1 \leftrightarrow \top))) \\
\wedge(\neg A^2 \wedge \neg B^2 \wedge C^2)
\end{array}
$$

After simplifying we have the following.

$$
\begin{array}{l}
(A^0 \wedge \neg B^0 \wedge \neg C^0) \\
\wedge((B^1 \wedge \neg C^1 \wedge \neg A^1) \vee (\neg A_1 \wedge B^1 \wedge C^1) \\
\wedge((\neg A^1 \wedge B^1 \wedge \neg C^1) \vee (A^1 \wedge B^1 \wedge \neg C^1)) \\
\wedge(\neg A^2 \wedge \neg B^2 \wedge C^2)
\end{array}
$$

Clearly, the only way of satisfying this formula is to make the first disjuncts of both disjunctions true, that is, B^1 must be true and A^1 and C^1 must be false.

The resulting valuation corresponds to taking the rotation action twice.

Consider the same problem but now with the goal state 101.

$$
\begin{array}{l}
(A^0 \wedge \neg B^0 \wedge \neg C^0) \\
\wedge(((A^0 \leftrightarrow B^1) \wedge (B^0 \leftrightarrow C^1) \wedge (C^0 \leftrightarrow A^1)) \vee ((\neg A^0 \leftrightarrow A_1) \wedge (\neg B^0 \leftrightarrow B^1) \wedge (\neg C^0 \leftrightarrow C^1))) \\
\wedge(((A^1 \leftrightarrow B^2) \wedge (B^1 \leftrightarrow C^2) \wedge (C^1 \leftrightarrow A^2)) \vee ((\neg A^1 \leftrightarrow A^2) \wedge (\neg B^1 \leftrightarrow B^2) \wedge (\neg C^1 \leftrightarrow C^2))) \\
\wedge(A^2 \wedge \neg B^2 \wedge C^2)
\end{array}
$$

We simplify again and get the following formula.

$$
\begin{array}{l}
(A^0 \wedge \neg B^0 \wedge \neg C^0) \\
\wedge((B^1 \wedge \neg C^1 \wedge \neg A^1) \vee (\neg A_1 \wedge B^1 \wedge C^1)) \\
\wedge((\neg A^1 \wedge B^1 \wedge C^1) \vee (\neg A^1 \wedge B^1 \wedge \neg C^1)) \\
\wedge(A^2 \wedge \neg B^2 \wedge C^2)
\end{array}
$$

Now there are two possible plans, to rotate first and then invert the values, or first invert and then rotate. These respectively correspond to making the first disjunct of the first disjunction and the second disjunct of the second disjunction true, or the second and the first disjunct. ■

3.5.4 Parallel plans

Plans so far always have had one operator at a time point. It turns out that it is often useful to allow *several operators in parallel*. This is beneficial for two main reasons.

First, consider a number of operators that affect and depend on disjoint state variables so that they can be applied in any order. If there are n such operators, there are $n!$ plans that are equivalent in the sense that each leads to the same state. When a satisfiability algorithm is used in showing that there is no plan of length n consisting of these operators, it has to show that none of the $n!$ plans reaches the goals. This may be combinatorially very difficult if n is high.

Second, when several operators can be applied simultaneously, it is not necessary to represent all intermediate states of the corresponding sequential plans: parallel plans require less time points than the corresponding sequential plans.

For sequences $o_1; o_2; \ldots; o_n$ of operators we define $app_{o_1;o_2;\ldots;o_n}(s)$ as $app_{o_n}(\cdots app_{o_2}(app_{o_1}(s))\cdots)$. For sets S of operators and states s we define $app_S(s)$ as the result of simultaneously applying all operators $o \in S$: the preconditions of all operators in S must be true in s and the state $app_S(s)$ is obtained from s by making the literals in $\bigcup_{\langle p,e\rangle \in S}([e]_s)$ true. Analogously to sequential plans we can define $app_{S_1;S_2;\ldots;S_n}(s)$ as $app_{S_n}(\cdots app_{S_2}(app_{S_1}(s))\cdots)$.

Definition 3.29 (Step plans) *For a set of operators O and an initial state I,* a plan *is a sequence $T = S_1, \ldots, S_l$ of sets of operators such that there is a sequence of states $s_0, \ldots, s_l$ (the execution of T) such that*

1. $s_0 = I$,
2. $\bigcup_{\langle p,e\rangle \in S_i}([e]_{s_{i-1}})$ *is consistent for every* $i \in \{1, \ldots, l\}$,
3. $s_i = app_{S_i}(s_{i-1})$ *for* $i \in \{1, \ldots, l\}$,
4. *for all* $i \in \{1, \ldots, l\}$ *and* $\langle p, e\rangle = o \in S_i$ *and* $S \subseteq S_i \backslash \{o\}$, $app_S(s_{i-1}) \models p$, *and*
5. *for all* $i \in \{1, \ldots, l\}$ *and* $\langle p, e\rangle = o \in S_i$ *and* $S \subseteq S_i \backslash \{o\}$, $[e]_{s_{i-1}} = [e]_{app_S(s_{i-1})}$.

The last condition says that the changes an operator makes would be the same also if some of the operators parallel to it would have been applied before it. This means that the parallel application can be understood as applying the operators in any order, with the requirement that the state that is reached is the same in every case.

Indeed, we can show that a parallel plan can be linearized in an arbitrary way, without affecting which state it reaches.

Lemma 3.30 *Let $T = S_1, \ldots, S_k, \ldots, S_l$ be a step plan. Let $T' = S_1, \ldots, S_k^0, S_k^1, \ldots, S_l$ be the step plan obtained from T by splitting the step S_k into two steps S_k^0 and S_k^1 such that $S_k = S_k^0 \cup S_k^1$ and $S_k^0 \cap S_k^1 = \emptyset$.*

If $s_0, \ldots, s_k, \ldots, s_l$ is the execution of T then $s_0, \ldots, s'_k, s_k, \ldots, s_l$ for some s'_k is the execution of T'.

Proof: So $s'_k = app_{S_k^0}(s_{k-1})$ and $s_k = app_{S_k}(s_{k-1})$ and we have to prove that $app_{S_k^1}(s'_k) = s_k$. We will show that the active effects of every operator $o \in S_k^1$ are the same in s_{k-1} and in s'_k, and hence the changes from s_{k-1} to s_k are the same in both plans. Let $o^1, \ldots, o^z$ be the operators in S_k^0, and let $T_i = \{o^1, \ldots, o^i\}$ for every $i \in \{0, \ldots, z\}$. We show by induction that changes caused by every operator $o \in S_k^1$ are the same when executed in s_{k-1} and in $app_{T_i}(s_{k-1})$, from which the claim follows because $s'_k = app_{T_z}(s_{k-1})$.

Base case $i = 0$: Immediate because $T_0 = \emptyset$.

Inductive case $i \geq 1$: By the induction hypothesis the changes caused by every $o \in S_k^1$ are the same when executed in s_{k-1} and in $app_{T_{i-1}}(s_{k-1})$. In $app_{T_i}(s_{k-1})$ additionally the operator o^i has been applied. We have to show that this operator application does affect the set of active effects of o. By the definition of step plans, $[e]_{app_{T_{i-1}}(s_{k-1})} = [e]_{app_{T_{i-1} \cup \{o^i\}}(s_{k-1})}$. This establishes the induction hypothesis and completes the proof. □

Theorem 3.31 *Let $T = S_1, \ldots, S_k, \ldots, S_l$ be a step plan. Then any $\sigma = o_1^1; \ldots; o_{n_1}^1; o_2^2; \ldots; o_{n_2}^2; \ldots; o_1^l; \ldots; o_{n_l}^l$ such that for every $i \in \{1, \ldots, l\}$ the sequence $o_1^i; \ldots; o_{n_i}^i$ is a total ordering of S_i, is a plan, and its execution leads to the same terminal state as that of T.*

Proof: First, all empty steps can be removed from the step plan. By Lemma 3.30 non-singleton steps can be split repeatedly to two smaller non-empty steps until every step is singleton and the singleton steps are in the desired order. The resulting plan is a sequential plan. □

Lemma 3.32 *Testing whether a sequence of sets of operators is a parallel plan is co-NP-hard.*

Proof: We can reduce the NP-complete satisfiability problem of the propositional logic to it. Let ϕ be a propositional formula in which the propositional variables $A = \{a_1, \ldots, a_n\}$ occur. Let I be an initial state in which all state variables are false. Now ϕ is valid if and only if $S_1 = \{\langle \top, \phi \rhd A \rangle, \langle \top, a_1 \rangle, \langle \top, a_2 \rangle, \ldots, \langle \top, a_n \rangle\}$ is a parallel plan that reaches the goal A. □

However, there are simple sufficient conditions that guarantee that a sequence of sets of operators satisfies the definition of parallel plans. A commonly used condition is that a state variable affected by any of the operators at one step of a plan does not occur in the precondition or in the antecedent of a conditional of any other operator in that step.

3.5.5 Translation of parallel planning into propositional logic

The second translation we give allows applying several operators is parallel. The translation differs from the one in Section 3.5.2 in that the translation is not obtained simply by combining the translations of individual operators, and that we use propositions for explicitly representing which operators are applied.

Let $o_1, \ldots, o_m$ be the operators, and $e_1, \ldots, e_m$ their respective effects. Let $a \in A$ be one of the state variables. Then we have the following formulae expressing the conditions under which the state variable p may change from false to true and from true to false.

$$(\neg a \wedge a') \rightarrow ((o_1 \wedge \mathit{EPC}_a(e_1)) \vee \cdots \vee (o_m \wedge \mathit{EPC}_a(e_m)))$$
$$(a \wedge \neg a') \rightarrow ((o_1 \wedge \mathit{EPC}_{\neg a}(e_1)) \vee \cdots \vee (o_m \wedge \mathit{EPC}_{\neg a}(e_m)))$$

Further, for every operator $\langle z, e\rangle \in O$ we have formulae that describe what values the state variables have in the predecessor and in the successor states if the operator is applied. Then the state variables $a_1, \ldots, a_n$ may be affected as follows, and the precondition z of the operator must be true in the predecessor state.

$$\begin{array}{rcl}
(o \wedge \mathit{EPC}_{a_1}(e)) & \rightarrow & a_1' \\
(o \wedge \mathit{EPC}_{\neg a_1}(e)) & \rightarrow & \neg a_1' \\
\vdots & & \\
(o \wedge \mathit{EPC}_{a_n}(e)) & \rightarrow & a_n' \\
(o \wedge \mathit{EPC}_{\neg a_n}(e)) & \rightarrow & \neg a_n' \\
o & \rightarrow & z
\end{array}$$

Example 3.33 Consider the operators $o_1 = \langle \neg LAMP1, LAMP1\rangle$ and $o_2 = \langle \neg LAMP2, LAMP2\rangle$. The application of none, one or both of these operators is described by the following formula.

$$\begin{array}{l}
(\neg LAMP1 \wedge LAMP1') \rightarrow ((o_1 \wedge \top) \vee (o_2 \wedge \bot)) \\
(LAMP1 \wedge \neg LAMP1') \rightarrow ((o_1 \wedge \bot) \vee (o_2 \wedge \bot)) \\
(\neg LAMP2 \wedge LAMP2') \rightarrow ((o_1 \wedge \bot) \vee (o_2 \wedge \top)) \\
(LAMP2 \wedge \neg LAMP2') \rightarrow ((o_1 \wedge \bot) \vee (o_2 \wedge \bot)) \\
o_1 \rightarrow LAMP1' \\
o_1 \rightarrow \neg LAMP1 \\
o_2 \rightarrow LAMP2' \\
o_2 \rightarrow \neg LAMP2
\end{array}$$

■

Finally, we have to guarantee that the last two conditions of parallel plans, that the simultaneous execution leads to the same result as executing them in any order, are satisfied. Encoding the conditions exactly is difficult, but we can use a simple encoding that provides a sufficient condition that the conditions are satisfied. We just have

$$\neg o_i \vee \neg o_j$$

whenever there is a state variable p occurring as an effect in o_i and in the precondition or the antecedent of a conditional effect of o_j.

We use

$$\mathcal{R}_2(A, A')$$

to denote the conjunction of all the above formulae.

Like $\mathcal{R}_1(A, A')$, later we use also $\mathcal{R}_2(A, A')$ with propositions labeled for different time points, and then we also have to label the propositions o for operators so that operator applications at different time points correspond to different propositions, for example o^0, o^1 and so on. For the labels for other propositions we use the superscript t in $\mathcal{R}_2^t(A, A')$.

3.5.6 Plan existence as evaluation of quantified Boolean formulae

For a more concise representation of the deterministic planning problem we need a slightly more expressive language than the propositional logic. Quantified Boolean formulae are exactly right for this purpose.

Consider the following QBF that represents the existence of transition sequences of length 2^n between two states.

$$\exists A \exists A' (\text{reach}_n(A, A') \wedge I \wedge G) \tag{3.1}$$

Here I and G are the formulae describing the initial and goal states respectively expressed in terms of variables from sets A and A'. Here $\text{reach}_i(A, A')$ means that a state represented in terms of variables from A' can be reached with $\leq 2^i$ steps from a state represented in terms of variables from A. It is recursively defined as follows.

$$\begin{array}{rcl} \text{reach}_0(A, A') & \stackrel{\text{def}}{\equiv} & \mathcal{R}_1(A, A') \\ \text{reach}_{i+1}(A, A') & \stackrel{\text{def}}{\equiv} & \exists T \forall c \exists T_1 \exists T_2 (\text{reach}_i(T_1, T_2) \\ & & \wedge (c \rightarrow (T_1 = A \wedge T_2 = T)) \\ & & \wedge (\neg c \rightarrow (T_1 = T \wedge T_2 = A'))) \end{array}$$

The sets T and A consist of propositional variables, and $A = T$ for $A = \{a_1, \ldots, a_n\}$ and $T = \{t_1, \ldots, t_n\}$ means $(a_1 \leftrightarrow t_1) \wedge \cdots \wedge (a_n \leftrightarrow t_n)$. The idea of the definition of $\text{reach}_{i+1}(A, A')$ is that the variables T describe a state halfway between A and A', and the two values for the variable c correspond to two reachability tests, one between A and T, and the other between T and A'.

This is how the PSPACE-hardness of evaluation of QBF can be proved, with $\mathcal{R}_1(A, A')$ representing the transitions of a deterministic polynomial-space Turing machine, see for example [Balcázar *et al.*, 1988].

If we eliminate all universal variables from Formula 3.1, we see that it is essentially a concise $\mathcal{O}(\log t)$ space ($t = 2^n$) representation of

$$I_0 \wedge \mathcal{R}_1(A_0, A_1) \wedge \mathcal{R}_1(A_1, A_2) \wedge \cdots \wedge \mathcal{R}_1(A_{t-1}, A_t) \wedge G_t \tag{3.2}$$

with only one occurrence of the transition relation.

The representation of deterministic planning as quantified Boolean formulae is more concise that the representation in the propositional logic, but it currently seems that the algorithms for testing the satisfiability solve the planning problem much more efficiently than algorithms for evaluating the values of QBF.

3.6 Invariants

Planning with both regression and propositional satisfiability suffer from the problem of states (valuations of state variables) that are not reachable from the initial state. Even when the number of state variables is high, the number of possible states of the world might be rather small, because not all valuations correspond to a possible world state. Hence for example regression may produce formulae that represent states that are not reachable from the initial state, and due to this backward search may spend a lot of time doing unfruitful work[1]. Clearly, search would be more efficient if backward search could be restricted to state that are indeed reachable from the initial state. Planning as propositional satisfiability suffers from the same problem.

It would be useful to eliminate those states from consideration that do not represent possible world states. However, determining whether a given state is reachable from the initial state is PSPACE-complete and equivalent to the plan existence problem of deterministic planning, and consequently computing exact information on the reachability of states could not be used for speeding up the basic forward and backward search algorithms: solving the subproblem would be just as complex as solving the problem itself, and would just lead to slow planning.

However, there is the possibility of using inexact, less expensive information about the reachability of states. In this section we present a polynomial time algorithm for computing inexact information about the reachability of states that has turned out very useful in speeding up planning algorithms based on backward search as well as other algorithms that use incomplete descriptions of sets of states, like plan search by using propositional logic in Section 3.5.

An *invariant* is a formula that holds in the initial state of a planning problem and that holds in every state that is reached by an action from a state in which it holds. A formula ϕ is *the strongest invariant* if for any invariant ψ, $\phi \models \psi$. The strongest invariant exactly characterizes the set of all states that are reachable from the initial state: For all states s, $s \models \phi$ if and only if s is reachable from the initial state. The strongest invariant is unique up to a logical equivalence.

Example 3.34 Consider a set of blocks that are on the table, and that can be stacked on top of each other so that every block can be on at most one block and on every block there can be at most one block.

We can formalize the actions that are possible in this setting as the following schematic operators.

$$\langle \text{ontable}(x) \wedge \text{clear}(x) \wedge \text{clear}(y), \text{on}(x,y) \wedge \neg\text{clear}(y) \wedge \neg\text{ontable}(x) \rangle$$
$$\langle \text{clear}(x) \wedge \text{on}(x,y), \text{ontable}(x) \wedge \text{clear}(y) \wedge \neg\text{on}(x,y) \rangle$$
$$\langle \text{clear}(x) \wedge \text{on}(x,y) \wedge \text{clear}(z), \text{on}(x,z) \wedge \text{clear}(y) \wedge \neg\text{clear}(z) \wedge \neg\text{on}(x,y) \rangle$$

When instantiated with three objects $X = \{A, B, C\}$ we get the following operators.

[1] A similar problem arises with forward search, because with progression one may reach states from which the goals cannot be reached.

$\langle \text{ontable}(A) \wedge \text{clear}(A) \wedge \text{clear}(B), \text{on}(A,B) \wedge \neg\text{clear}(B) \wedge \neg\text{ontable}(A) \rangle$
$\langle \text{ontable}(A) \wedge \text{clear}(A) \wedge \text{clear}(C), \text{on}(A,C) \wedge \neg\text{clear}(C) \wedge \neg\text{ontable}(A) \rangle$
$\langle \text{ontable}(B) \wedge \text{clear}(B) \wedge \text{clear}(A), \text{on}(B,A) \wedge \neg\text{clear}(A) \wedge \neg\text{ontable}(B) \rangle$
$\langle \text{ontable}(B) \wedge \text{clear}(B) \wedge \text{clear}(C), \text{on}(B,C) \wedge \neg\text{clear}(C) \wedge \neg\text{ontable}(B) \rangle$
$\langle \text{ontable}(C) \wedge \text{clear}(C) \wedge \text{clear}(A), \text{on}(C,A) \wedge \neg\text{clear}(A) \wedge \neg\text{ontable}(C) \rangle$
$\langle \text{ontable}(C) \wedge \text{clear}(C) \wedge \text{clear}(B), \text{on}(C,B) \wedge \neg\text{clear}(B) \wedge \neg\text{ontable}(C) \rangle$

$\langle \text{clear}(A) \wedge \text{on}(A,B), \text{ontable}(A) \wedge \text{clear}(B) \wedge \neg\text{on}(A,B) \rangle$
$\langle \text{clear}(A) \wedge \text{on}(A,C), \text{ontable}(A) \wedge \text{clear}(C) \wedge \neg\text{on}(A,C) \rangle$
$\langle \text{clear}(B) \wedge \text{on}(B,A), \text{ontable}(B) \wedge \text{clear}(A) \wedge \neg\text{on}(B,A) \rangle$
$\langle \text{clear}(B) \wedge \text{on}(B,C), \text{ontable}(B) \wedge \text{clear}(C) \wedge \neg\text{on}(B,C) \rangle$
$\langle \text{clear}(C) \wedge \text{on}(C,A), \text{ontable}(C) \wedge \text{clear}(A) \wedge \neg\text{on}(C,A) \rangle$
$\langle \text{clear}(C) \wedge \text{on}(C,B), \text{ontable}(C) \wedge \text{clear}(B) \wedge \neg\text{on}(C,B) \rangle$

$\langle \text{clear}(A) \wedge \text{on}(A,B) \wedge \text{clear}(C), \text{on}(A,C) \wedge \text{clear}(B) \wedge \neg\text{clear}(C) \wedge \neg\text{on}(A,B) \rangle$
$\langle \text{clear}(A) \wedge \text{on}(A,C) \wedge \text{clear}(B), \text{on}(A,B) \wedge \text{clear}(C) \wedge \neg\text{clear}(B) \wedge \neg\text{on}(A,C) \rangle$
$\langle \text{clear}(B) \wedge \text{on}(B,A) \wedge \text{clear}(C), \text{on}(B,C) \wedge \text{clear}(A) \wedge \neg\text{clear}(C) \wedge \neg\text{on}(B,A) \rangle$
$\langle \text{clear}(B) \wedge \text{on}(B,C) \wedge \text{clear}(A), \text{on}(B,A) \wedge \text{clear}(C) \wedge \neg\text{clear}(A) \wedge \neg\text{on}(B,C) \rangle$
$\langle \text{clear}(C) \wedge \text{on}(C,A) \wedge \text{clear}(B), \text{on}(C,B) \wedge \text{clear}(A) \wedge \neg\text{clear}(B) \wedge \neg\text{on}(C,A) \rangle$
$\langle \text{clear}(C) \wedge \text{on}(C,B) \wedge \text{clear}(A), \text{on}(C,A) \wedge \text{clear}(B) \wedge \neg\text{clear}(A) \wedge \neg\text{on}(C,B) \rangle$

Here a block being clear means that no block is on top of it.

Let all the blocks be initially on the table. Hence the initial state satisfies the formula

$$\text{clear}(A) \wedge \text{clear}(B) \wedge \text{clear}(C) \wedge \text{ontable}(A) \wedge \text{ontable}(B) \wedge \text{ontable}(C) \wedge$$
$$\neg\text{on}(A,B) \wedge \neg\text{on}(A,C) \wedge \neg\text{on}(B,A) \wedge \neg\text{on}(B,C) \wedge \neg\text{on}(C,A) \wedge \neg\text{on}(C,B)$$

that determines the truth-values of all state variables uniquely.

All the invariants in this problem instance are the following.

$$\text{clear}(A) \leftrightarrow (\neg\text{on}(B,A) \wedge \neg\text{on}(C,A))$$
$$\text{clear}(B) \leftrightarrow (\neg\text{on}(A,B) \wedge \neg\text{on}(C,B))$$
$$\text{clear}(C) \leftrightarrow (\neg\text{on}(A,C) \wedge \neg\text{on}(B,C))$$
$$\text{ontable}(A) \leftrightarrow (\neg\text{on}(A,B) \wedge \neg\text{on}(A,C))$$
$$\text{ontable}(B) \leftrightarrow (\neg\text{on}(B,A) \wedge \neg\text{on}(B,C))$$
$$\text{ontable}(C) \leftrightarrow (\neg\text{on}(C,A) \wedge \neg\text{on}(C,B))$$
$$\neg\text{on}(A,B) \vee \neg\text{on}(A,C)$$
$$\neg\text{on}(B,A) \vee \neg\text{on}(B,C)$$
$$\neg\text{on}(C,A) \vee \neg\text{on}(C,B)$$
$$\neg\text{on}(B,A) \vee \neg\text{on}(C,A)$$
$$\neg\text{on}(A,B) \vee \neg\text{on}(C,B)$$
$$\neg\text{on}(A,C) \vee \neg\text{on}(B,C)$$
$$\neg(\text{on}(A,B) \wedge \text{on}(B,C) \wedge \text{on}(C,A))$$
$$\neg(\text{on}(A,C) \wedge \text{on}(C,B) \wedge \text{on}(B,A))$$

The conjunction of these formulae describes exactly the set of states that are reachable from the initial state by the operators, and intuitively describes all the possible configurations the three blocks can be in.

We can schematically give the invariants for any set X of blocks as follows.

clear$(x) \leftrightarrow \forall y \in X \backslash \{x\}.\neg$on$(y, x)$
ontable$(x) \leftrightarrow \forall y \in X \backslash \{x\}.\neg$on$(x, y)$
$\neg$on$(x, y) \vee \neg$on(x, z) when $y \neq z$
$\neg$on$(y, x) \vee \neg$on(z, x) when $y \neq z$
$\neg(\text{on}(x_1, x_2) \wedge \text{on}(x_2, x_3) \wedge \cdots \wedge \text{on}(x_{n-1}, x) \wedge \text{on}(x_n, x_1))$ for all $n \geq 1$ and $\{x_1, \ldots, x_n\} \subseteq X$

The last schematic formula says that the *on* relation is acyclic. ■

Because testing whether a state satisfies all invariants, that is whether it is reachable from the initial state, is PSPACE-hard, the requirement that invariant computation is polynomial time leads to computing only invariants that are weaker than the strongest invariant. This kind of set of invariants only gives an upper bound (with respect to set-inclusion) on the set of reachable states.

The algorithm we present computes invariants that are disjunctions of at most n literals, for a fixed n. For representing all invariants, no finite upper bound on n may be imposed, but then also invariant computation could not be performed in polynomial time. Although the computation is polynomial time for any fixed n, the runtimes grow quickly as n is increased, and it is most useful for $n = 2$, that is, for invariants that are disjunctions of two literals.

The algorithm proceeds by first computing all n-literal clauses that are true in the initial state. Then, the algorithm removes all clauses that are not true after 1 operator application, after 2 operator applications, and so on, until the set of clauses does not change. At this point all the clauses are invariants and hold in all states that are reachable from the initial state.

3.6.1 Algorithms for computing invariants

Our algorithm for computing invariants has a similar flavor to distance estimation in Section 3.4: starting from a description of what is possible in the initial state, we inductively determine what is possible after i operator applications. In contrast to the distance estimation method, the states that are reachable after i operator applications are not characterized by sets of literals but by sets of clauses. This complicates the computation somewhat.

Let C_i be a set of clauses characterizing those states that are reachable by i operator applications. Similarly to distance computation, we consider for each operator and for each clause in C_i whether applying the operator may make the clause false. If it can, the clause could be false after i operator applications and therefore will not be in the clause set C_{i+1}.

For this basic step of invariant computation, whether an operator application may falsify a clause, we present two algorithms, first a simple one for a restricted class of operators, and then a more general for arbitrary operators.

Figure 3.3 gives an algorithm that tests whether applying an operator $o \in O$ in some state s may make a formula $l_1 \vee \cdots \vee l_n$ false assuming that $s \models \Delta \cup \{l_1 \vee \cdots \vee l_n\}$.

The algorithm performs a case analysis for every literal in the clause, testing in each case that the clause remains true: if a literal becomes false, either some other literal in the clause becomes true simultaneously or some other literal in the clause was true already and does not become false.

The algorithm is defined only for operators that have a precondition that is a conjunction of literals and an effect that is a conjunction of atomic effects (known as STRIPS operators for historical reasons). We give a similar algorithm for arbitrary operators later in Figure 3.4.

procedure simplepreserved(ϕ,Δ,o);
Now $\phi = l_1 \vee \cdots \vee l_n$ and $o = \langle l'_1 \wedge \cdots \wedge l'_{n'}, l''_1 \wedge \cdots \wedge l''_{n''} \rangle$ for some l_i, l'_j and l''_k;
if $\{\overline{l'''_1}, \cdots, \overline{l'''_m}\} \subseteq \{l'_1, \ldots, l'_{n'}\}$ for some $l'''_1 \vee \cdots \vee l'''_m \in \Delta$ **then return** true;
(Operator is not applicable. *)*
for each $l \in \{l_1, \ldots, l_n\}$ **do**
  if $\bar{l} \notin \{l''_1, \ldots, l''_{n''}\}$ **then goto** OK; *(* Literal l cannot become false. *)*
  for each $l' \in \{l_1, \ldots, l_n\} \backslash \{l\}$ **do**
    if $l' \in \{l''_1, \ldots, l''_{n''}\}$ **then goto** OK; *(* Literal l' becomes true. *)*
    if $l' \in \{l'_1, \ldots, l'_{n'}\}$ **or** $\overline{l'''_1} \vee \cdots \vee \overline{l'''_m} \vee l' \in \Delta$ for some $\{l'''_1, \ldots, l'''_m\} \subseteq \{l'_1, \ldots, l'_{n'}\}$,
    and $\overline{l'} \notin \{l''_1, \ldots, l''_{n''}\}$
    then goto OK; *(* Literal l' was true and cannot become false. *)*
  end do
  return false; *(* Truth of the clause could not be guaranteed. *)*
  OK:
end do
return true;

Figure 3.3: Algorithm that tests if applying o may falsify $l_1 \vee \cdots \vee l_n$ in a state satisfying Δ

Lemma 3.35 *Let Δ be a set of clauses, $\phi = l_1 \vee \cdots \vee l_n$ a clause, and o an operator of the form $\langle l'_1 \wedge \cdots \wedge l'_{n'}, l''_1 \wedge \cdots \wedge l''_{n''} \rangle$ where l'_j and l''_k are literals. If simplepreserved(ϕ,Δ,o) returns* true, *then $app_o(s) \models \phi$ for any state s such that $s \models \Delta \cup \{\phi\}$ and o is applicable in s. (It may under these conditions also return* false*).*

Proof: Assume s is a state such that $s \models l'_1 \wedge \cdots \wedge l'_{n'}$ and $s \models \Delta$ and $s \models \phi$ and $app_o(s) \not\models \phi$. We show that the procedure returns *false*.

Because $s \models \phi$ and $app_o(s) \not\models \phi$, there are some literals $\{l^f_1, \ldots, l^f_m\} \subseteq \{l_1, \ldots l_n\}$ such that $s \models l^f_1 \wedge \cdots \wedge l^f_m$ and $\{\overline{l^f_1}, \ldots, \overline{l^f_m}\} \subseteq \{l''_1, \ldots l''_{n''}\}$, that is, applying o makes them false, and the rest of the literals in ϕ were false and do not become true.

Choose any $l \in \{l^f_1, \ldots, l^f_m\}$. We show that when the outermost *for each* loop considers l the procedure will return *false*.

By assumption $\bar{l} \in \{l''_1, \ldots l''_{n''}\}$, and the condition of the first *if* inside the loop is not satisfied and the execution proceeds by iteration of the inner *for each* loop.

Let l' be any of the literals in ϕ except l.

Because ϕ is false in $app_o(s)$, $l' \notin \{l''_1, \ldots l''_{n''}\}$, and the condition of the first *if* statement is not satisfied.

If $l' \in \{l^f_1, \ldots, l^f_m\}$ then by assumption $\overline{l'} \in \{l''_1, \ldots, l''_{n''}\}$ and the condition of the second *if* statement is not satisfied.

If $l' \notin \{l^f_1, \ldots, l^f_m\}$ then by assumption $s \not\models l'$. Because the operator is applicable $s \models l'_1 \wedge \cdots \wedge l'_{n'}$, and hence $l' \notin \{l'_1 \wedge \cdots \wedge l'_{n'}\}$. Because s satisfies the precondition $l'_1 \wedge \cdots \wedge l'_{n'}$ and $s \models \Delta$, there is also no $\overline{l''} \vee l' \in \Delta$ for any $l'' \in \{l'_1, \ldots, l'_{n'}\}$. Hence also in this case the condition of the *if* statement is not satisfied.

Hence on none of the iterations of the inner *for each* loop is a *goto OK* executed, and as the loop exits, the procedure returns *false*. □

Figure 3.4 gives a similar algorithm for arbitrary operators. The structure of the algorithm is

procedure preserved(ϕ,Δ,o);
Now $\phi = l_1 \vee \cdots \vee l_n$ for some $l_1, \ldots, l_n$ and $o = \langle c, e \rangle$ for some c and e;
if $\Delta \models \neg c$ **then return** true; *(* Operator is not applicable. *)*
for each $l \in \{l_1, \ldots, l_n\}$ **do**
  if $\Delta \wedge \{EPC_{\bar{l}}(e)\} \models \bot$ **then goto** OK; *(* Literal l cannot become false. *)*
  for each $l' \in \{l_1, \ldots, l_n\} \backslash \{l\}$ **do**
    if $\Delta \cup \{EPC_{\bar{l}}(e), c\} \models EPC_{l'}(e)$ **then goto** OK; *(* Literal l' becomes true. *)*
    if $\Delta \cup \{EPC_{\bar{l}}(e), c\} \models l'$ **and** $\Delta \cup \{EPC_{\bar{l}}(e), c\} \models \neg EPC_{\overline{l'}}(e)$ **then goto** OK;
      (Literal l' was true and cannot become false. *)*
  end do
  return false; *(* Truth of the clause could not be guaranteed. *)*
  OK:
end do
return true;

Figure 3.4: Algorithm that tests if applying o may falsify $l_1 \vee \cdots \vee l_n$ in a state satisfying Δ

exactly the same, but the tests whether a certain literal becomes true or false or whether it was true before the operator was applied, are more complicated.

The algorithm is allowed to fail in one direction: it may sometimes return false *when $l_1 \vee \cdots \vee l_n$ actually is true after applying the operator. However, this is a necessary consequence of our requirement that our invariant computation takes only polynomial time.*

Lemma 3.36 *Let Δ be a set of clauses, $\phi = l_1 \vee \cdots \vee l_n$ a clause, and o an operator. If preserved(ϕ,Δ,o) returns* true*, then $app_o(s) \models \phi$ for any state s such that $s \models \Delta \cup \{\phi\}$ and o is applicable in s. (It may under these conditions also return* false*).*

Proof: □

Figure 3.5 gives the algorithm for computing invariants consisting of at most n literals.

Theorem 3.37 *Let A be a set of state variables, I a state, O a set of operators, and $n \geq 1$ an integer.*

Then the procedure call invariants(A, I, O, n) returns a set C' of clauses so that for any sequence $o_1; \ldots, o_m$ of operators from O $app_{o_1;\ldots,o_m}(I) \models C'$.

Proof: Let C_0 be the value first applied to the variable C in the procedure *invariants*, and $C_1, C_2, \ldots$ be the values of the variable in the end of each iteration of the outermost *repeat* loop.

Induction hypothesis: for every $\phi \in C_i$, $app_{o_1;\ldots,o_i}(I) \models \phi$.

Base case $i = 0$: $app_\epsilon(I)$ for the empty sequence is by definition I itself, and by construction C_0 consists of only formulae that are true in the initial state.

Inductive case $i \geq 1$:

□

The algorithm in Figure 3.4 does not run in polynomial time in the size of the problem instance because the logical consequence tests may take exponential time. To make the procedure run in polynomial time, we can again use an approximate logical consequence test, similar to the

procedure invariants(A, I, O, n);
C := $\{a \in A | I \models a\} \cup \{\neg a | a \in A, I \not\models a\}$;
repeat
  C' := C;
  for each $l_1 \vee \cdots \vee l_m \in C$ **do** *(* Test every clause *)*
    for each $o \in O$ **do** *(* with respect to every operator. *)*
      N := $\{l_1 \vee \cdots \vee l_m\}$;
      repeat
        N' := N;
        for each $l'_1 \vee \cdots \vee l'_{m'} \in N$ s.t. **not** preserved($l'_1 \vee \cdots \vee l'_{m'}$,$C'$,$o$) **do**
          N := $N \backslash \{l'_1 \vee \cdots \vee l'_{m'}\}$;
          if $m' < n$ **then** *(* Clause length within pre-defined limit. *)*
            begin
              N := $N \cup \{l'_1 \vee \cdots \vee l'_{m'} \vee a \mid a \in A\}$;
              N := $N \cup \{l'_1 \vee \cdots \vee l'_{m'} \vee \neg a \mid a \in A\}$;
            end
        end do
      until $N = N'$; *(* N was not weakened further. *)*
      C := $(C \backslash \{l_1 \vee \cdots \vee l_m\}) \cup N$;
    end do
  end do
until $C = C'$;
return C;

Figure 3.5: Algorithm for computing a set of invariant clauses

procedure canbetruein(ϕ, D) used in Definition 3.16. The logical consequence test is allowed to fail in one direction without invalidating the invariant algorithm in Figure 3.5: *preserved* is allowed to return *false* also when the operator would not falsify ϕ, and hence logical consequence tests may be answered *no* even when the correct answer is *yes*.

The logical consequence tests have the form $\Delta \cup S \models \phi$. The logical consequence $\Delta \cup S \models \phi$ holds if and only if $\Delta \cup \{\bigwedge S \wedge \neg\phi\}$ is not satisfiable. A correct approximation is allowed to answer *satisfiable* even when the formula is unsatisfiable.

We present a polynomial time approximation of satisfiability tests for sets of formulae $\Delta \cup S$ in the case in which Δ consists of clauses of length at most 2. It is based on the definition of sets of literals litconseqs(ϕ, Δ) given below. The idea of litconseqs(ϕ, Δ) is that this set consists of (a subset of the) literals that must be true when ϕ and Δ are true, that is, that are logical consequences of ϕ and Δ. The one-sided error litconseqs(ϕ, Δ) is allowed to make and indeed does make is how disjunction $\vee$ is handled. if $\Delta \cup \{\phi\}$ is satisfiable, then litconseqs(ϕ, Δ) does not contain $\bot$ nor a and $\neg a$ for any $a \in A$.

$$
\begin{array}{rcl}
\text{litconseqs}(\bot, \Delta) & = & \{\bot\} \\
\text{litconseqs}(\top, \Delta) & = & (\Delta \cap A) \cup (\Delta \cap \{\neg a | a \in A\}) \\
\text{litconseqs}(a, \Delta) & = & \{a\} \cup \{l | \neg a \vee l \in \Delta\} \cup (\Delta \cap A) \cup (\Delta \cap \{\neg a | a \in A\}) \\
\text{litconseqs}(\neg a, \Delta) & = & \{\neg a\} \cup \{l | a \vee l \in \Delta\} \cup (\Delta \cap A) \cup (\Delta \cap \{\neg a | a \in A\}) \\
\text{litconseqs}(\neg\neg\phi, \Delta) & = & \text{litconseqs}(\phi, \Delta) \\
\text{litconseqs}(\phi \vee \psi, \Delta) & = & \text{litconseqs}(\phi, \Delta) \cap \text{litconseqs}(\psi, \Delta) \\
\text{litconseqs}(\phi \wedge \psi, \Delta) & = & \text{litconseqs}(\phi, \Delta) \cup \text{litconseqs}(\psi, \Delta) \\
\text{litconseqs}(\neg(\phi \vee \psi), \Delta) & = & \text{litconseqs}(\neg\phi, \Delta) \cup \text{litconseqs}(\neg\psi, \Delta) \\
\text{litconseqs}(\neg(\phi \wedge \psi), \Delta) & = & \text{litconseqs}(\neg\phi, \Delta) \cap \text{litconseqs}(\neg\psi, \Delta)
\end{array}
$$

The approximation fails because the satisfiability test is too simple. Consider litconseqs$((A \vee B) \wedge \neg(A \vee B), \emptyset)$ which is the empty set of literals because litconseqs$(A \vee B, \emptyset) = \emptyset$ and litconseqs$(\neg(A \vee B), \emptyset) = \emptyset$. This formula is unsatisfiable because it has the form $\phi \wedge \neg\phi$.

There are some simple ways of strengthening this approximation. For example, conjunction could be strengthened to

$$\text{litconseqs}(\phi \wedge \psi, \Delta) = \text{litconseqs}(\phi, \Delta \cup \text{litconseqs}(\psi, \Delta)) \cup \text{litconseqs}(\psi, \Delta \cup \text{litconseqs}(\phi, \Delta))$$

and further by computing more consequences for one of the conjuncts with the literals obtained from the other until no more literals are obtained.

The function litconseqs(ϕ, Δ) can also be used as a part of slightly more powerful (???) logical consequence tests as follows.

Define

$$
\begin{array}{rcl}
\text{entailed}(\bot, D) & = & \text{false} \\
\text{entailed}(\top, D) & = & \text{true} \\
\text{entailed}(a, D) & = & \text{true iff } a \in D \text{ (for state variables } a \in A) \\
\text{entailed}(\neg a, D) & = & \text{true iff } \neg a \in D \text{ (for state variables } a \in A) \\
\text{entailed}(\neg\neg\phi, D) & = & \text{entailed}(\phi, D) \\
\text{entailed}(\phi \vee \psi, D) & = & \text{entailed}(\phi, D) \text{ or entailed}(\psi, D) \\
\text{entailed}(\phi \wedge \psi, D) & = & \text{entailed}(\phi, D) \text{ and entailed}(\psi, D) \\
\text{entailed}(\neg(\phi \vee \psi), D) & = & \text{entailed}(\neg\phi, D) \text{ and entailed}(\neg\psi, D) \\
\text{entailed}(\neg(\phi \wedge \psi), D) & = & \text{entailed}(\neg\phi, D) \text{ or entailed}(\neg\psi, D)
\end{array}
$$

Notice that the definition of entailed(ϕ, D) is similar to canbetruein(ϕ, D) in Definition 3.16 except that literals a and $\neg a$ are handled differently: entailed(ϕ, D) is about logical consequences of D, that is formulae that are guaranteed to be true when D is true, while canbetruein(ϕ, D) is about ϕ being consistent with D.

Now if entailed$(\phi, \text{litconseqs}(\psi, \Delta))$ then $\Delta \cup \{\psi\} \models \phi$.

3.6.2 Applications in planning by regression and satisfiability

The first application is in planning in the propositional logic. It has been noticed that adding the 2-literal invariants to all time points reduces runtimes of algorithms that test satisfiability. Notice that invariants do not affect the set of models of a formula representing planning: any satisfying valuation of the original formula also satisfies the invariants, because the values of propositions describing the values of state variables at any time point corresponds to a state that is reachable from the initial state, and hence this valuation also satisfies any invariant.

The second application is in planning by regression. Consider the blocks world with the goal A-ON-B $\wedge$ B-ON-C. Now we can regress with the operator that moves B onto C from the table, obtaining the new goal A-ON-B $\wedge$ B-CLEAR $\wedge$ C-CLEAR $\wedge$ B-ON-TABLE. Clearly, this does not correspond to an intended blocks world state because A-ON-B is incompatible with B-CLEAR, and indeed, $\neg$A-ON-B $\vee \neg$B-CLEAR is an invariant for the blocks world. Any regression step that leads to a goal that is incompatible with the invariants can be ignored, because that goal does not represent any of the states that are reachable from the initial state, and hence no plan can reach the goal in question.

Another application of invariants, and the intermediate sets C_i produced by our invariant algorithm, is improving the distance estimation in Section 3.4. Using v_i for testing whether an operator precondition, for example $a \wedge b$, has distance i from the initial state, the distances of a and b are used separately. But even when it is possible to reach both a and b with i operator applications, it might still not be possible to reach them both simultaneously with i operator applications. For example, for $i = 1$ and an initial state in which both a and b are false, there might be no single operator that makes them both true, but two operators, each of which makes only one of them true. If $\neg a \vee \neg b \in C_i$, we know that after i operator applications one of a or b must still be false, and then we know that the operator in question is not applicable at time point i. Therefore the invariants and the sets C_i produced during the invariant computation can improve the distance estimates.

3.7 Planning with symbolic representations of sets of states

A complementary approach to planning for planning problems represented as formulae in the propositional logic uses the formulae as a data structure. As discussed in Section 2.3.3 formulae directly provide a representation of sets of states, and in this section we show how operations on transition relations have a counterpart as operations on formulae that represent transition relations.

This yields a further planning algorithm for deterministic planning, typically implemented by means of BDDs. The algorithm in Section 3.7.3 will later be generalized to different types of nondeterministic planning.

Table 3.1 outlines a number of connections between operations on vectors and matrices, on propositional formulae, and on sets and relations.

Computing the product of two matrices that are represented as propositional formulae is based on the *existential abstraction* operation $\exists p.\phi = \phi[\top/p] \vee \phi[\bot/p]$ that takes a formula ϕ and a

matrices	formulas	sets of states
vector $V_{1\times n}$	formula over A	set of states
matrix $M_{n\times n}$	formula over $A \cup A'$	transition relation
$M_{n\times n} \times N_{n\times n}$	$\exists A'.(\phi(A, A') \wedge \psi(A', A''))$	sequential composition
$S_{1\times n} \times M_{n\times n}$	$\exists A.(\phi(A) \wedge \psi(A, A'))$	successor states of S
$S_{1\times n} + S'_{1\times n}$	$\phi \vee \psi$	set union
	$\phi \wedge \psi$	set intersection

Table 3.1: Correspondence between matrix operations, Boolean operations as well as set-theoretic and relational operations

proposition p and produces a formula ϕ' without occurrences of p.

Let ϕ be a formula over $A \cup A'$ and ψ be a formula over $A' \cup A''$. Now matrix product of matrices corresponding to ϕ and ψ' is

$$\exists A'.\phi \wedge \psi.$$

Example 3.38 Let $\phi = A \leftrightarrow \neg A'$ and $\psi = A' \leftrightarrow A''$ represent two actions, reversing the truth-value of A and doing nothing. The sequential composition of these actions is

$$\begin{array}{rcl} \exists A'.\phi \wedge \psi & = & ((A \leftrightarrow \neg\top) \wedge (\top \leftrightarrow A'')) \vee ((A \leftrightarrow \neg\bot) \wedge (\bot \leftrightarrow A'')) \\ & \equiv & ((A \leftrightarrow \bot) \wedge (\top \leftrightarrow A'')) \vee ((A \leftrightarrow \top) \wedge (\bot \leftrightarrow A'')) \\ & \equiv & A \leftrightarrow \neg A'' \end{array}$$

■

Consider the representation of planning as satisfiability in the propositional discussed in Section 3.5.3.

$$\iota^0 \wedge \underbrace{\mathcal{R}_1(A^0, A^1) \wedge \mathcal{R}_1(A^1, A^2) \wedge \cdots \wedge \mathcal{R}_1(A^{n-1}, A^n)} \wedge G^n$$

The conjunction of formulae the $\mathcal{R}_1(A^i, A^{i+1})$ representing the transition relation corresponds to the computation of the n-fold product of the corresponding adjacency matrices. Further, when the first factor in the product is the vector describing the initial state, we have the computation of the set of states reachable in n steps.

$$\underbrace{\underbrace{\iota^0 \times (\mathcal{R}_1(A^0, A^1)} \times \mathcal{R}_1(A^1, A^2) \times \cdots \times \mathcal{R}_1(A^{n-1}, A^n))}$$

Taking the intersection of this set with the set of goal states tells us whether there is a plan of length n.

In the following we discuss how this idea can be turned into a planning algorithm, in which the n-fold product of the initial state vector with the adjacency matrices is computed step by step, yielding vectors describing the sets of states reachable in $i \in \{0, \ldots, n\}$ operator applications.

3.7.1 Operations on transition relations expressed as formulae

The most basic operation is the computation of *the image* of a set of states with respect to a transition relation.

$$\mathit{img}_R(S) = \{s' | s \in S, \langle s, s' \rangle \in R\}$$

This is the set of states that can be reached from S by transition relation R. When sets of states and transition relation are represented as propositional formulae, the image computation can be performed by the existential abstraction and renaming operations as follows.

$$\mathit{img}_{\mathcal{R}(A,A')}(\phi) = (\exists A.(\phi \wedge \mathcal{R}(A, A')))[p_1/p'_1, \ldots, p_n/p'_n]$$

Similarly we can compute the product of two matrices that are represented as formulae $\mathcal{R}(A, A')$ and $\mathcal{Q}(A', A'')$ by using existential abstraction.

$$\mathcal{R}(A, A') \cdot \mathcal{Q}(A', A'') = \exists A'.(\mathcal{R}(A, A') \wedge \mathcal{Q}(A', A''))$$

The resulting formula is over state variables A and A'', from which a formula on A and A' is obtained by renaming A'' to A'.

Plan search can also be performed starting from the goal states, like done with all the algorithms in Chapter 4. In this case we must compute sets of states from which any of the states in a given set can be reached by one step. This is represented as the computation of *the preimage* of a set of states with respect to a transition relation.[2]

$$\mathit{wpreimg}_R(S) = \{s | s' \in S, \langle s, s' \rangle \in R\}$$

This is the set of states from which a state in S is reached by the transition relation R. The corresponding computation in terms of formulae is as follows. Here ϕ is a formula over A, and first it has to be renamed to a formula over A'.

$$\mathit{wpreimg}_{\mathcal{R}(A,A')}(\phi) = \exists A'.(\phi[p'_1/p_1, \ldots, p'_n/p_n] \wedge \mathcal{R}(A, A')) \tag{3.3}$$

Notice that when the relation $\mathcal{R}(A, A')$ corresponding to an operator o has been represented as discussed in Section 3.5.2, the Formula 3.3 for $\mathit{wpreimg}_{\mathcal{R}(A,A')}(\phi)$ is logically equivalent to the regression $\mathit{regr}_o(\phi)$ as given in Definition 3.6.

Example 3.39 Consider the formula $A \wedge B$ that is regressed with the operator $o = \langle C, A \wedge (A \rhd B) \rangle$. Now we have

$$\mathit{regr}_o(\phi) = C \wedge (\top \wedge (B \vee A)) \equiv C \wedge (B \vee A).$$

The transition relation of o is represented by the formula

$$\tau = C \wedge A' \wedge ((B \vee A) \leftrightarrow B') \wedge (C \leftrightarrow C').$$

The preimage of $A \vee B$ with respect to o is represented by

$$\begin{array}{rcl}
\exists A'B'C'.((A' \wedge B') \wedge \tau) & \equiv & \exists A'B'C'.((A' \wedge B') \wedge C \wedge A' \wedge ((B \vee A) \leftrightarrow B') \wedge (C \leftrightarrow C')) \\
& \equiv & \exists A'B'C'.(A' \wedge B' \wedge C \wedge (B \vee A) \wedge C') \\
& \equiv & \exists B'C'.(B' \wedge C \wedge (B \vee A) \wedge C') \\
& \equiv & \exists C'.(C \wedge (B \vee A) \wedge C') \\
& \equiv & C \wedge (B \vee A)
\end{array}$$

■

[2] This is often called the *weak preimage* to contrast it with the strong preimage operation defined in Section 4.3.

procedure planfwd(I,O,G)
$i := 0$;
$D_0 := \{I\}$;
while $G \cap D_i = \emptyset$ **and** ($i = 0$ **or** $D_{i-1} \neq D_i$) **do**
  $i := i + 1$;
  $D_i := D_{i-1} \cup \bigcup_{o \in O} img_o(D_{i-1})$; *(* Possible successors of states in D_{i-1} *)*
end
if $G \cap D_i = \emptyset$ **then** terminate; *(* There is no plan. *)*
$S := G \cap D_i$;
for j := i-1 **to** 0 **do** *(* Output plan, last operator first. *)*
  choose $o \in O$ such that $wpreimg_o(S) \cap D_j \neq \emptyset$;
  output o;
  $S := wpreimg_o(S) \cap D_j$;
end

Figure 3.6: Algorithm for deterministic planning (forward, in terms of sets)

As we will see later, computation of preimages is applicable to all kinds of operators, not only deterministic ones as required by our definition of regression, whereas defining regression for arbitrary operators is more difficult (we will give a definition of regression only for a subclass of nondeterministic operators.)

Hence our definition of regression can be viewed as a specialized method for computing preimage of formulae with respect to a transition relation corresponding to a deterministic operator. The main advantage of regression is that no existential abstraction is needed.

Notice that defining progression for arbitrary formulae (sets of states) seems to require existential abstraction. A simple syntactic definition of progression similar to that of regression does not seem to be possible because the value of state variable in a given state cannot be represented in terms of the values of the state variables in the successor state. This is because of the asymmetry of deterministic actions: the current state and an operator determine the successor state uniquely, but the successor state and the operator do not determine the current state uniquely. In other words, the changes that take place are a function of the current state, but not a function of the successor state.

3.7.2 A forward planning algorithm

The algorithm in Figure 3.6 has two phases: the computation of distance from the initial state to every reachable state, and the extraction of a plan. The set D_0 consists of the initial state, the set D_1 of those states that can be reached from the initial state by one operator, and so on.

We can express the same algorithm in terms of formulae in the propositional logic, see Figure 3.7. The plan extraction proceeds by identifying the operators in the backwards direction starting from the last one.

In the figure we give two variants of the algorithm, first expressed in terms of set-theoretic operations on sets of states and transition relations, and then expressed in terms of the propositional formulae.

Notice that in the first version of the algorithm D_i is computed as the union of D_{i-1} (reachability by $i-1$ steps or less) and the images of D_{i-1} with respect to all of the operators, and hence D_i

procedure planfwd(I,$\mathcal{R}_1(A, A')$,G)
$i := 0$;
$D_0 := I$;
while $D_i \models \neg G$ **and** ($i = 0$ **or** $\not\models D_{i-1} \leftrightarrow D_i$) **do**
  $i := i + 1$;
  $D_i := (\exists A.(D_{i-1} \wedge \mathcal{R}_1(A, A')))[p'_1/p_1, \ldots, p'_n/p_n]$; *(* Possible predecessors of states in D_{i-1} *)*
end
if $D_i \models \neg G$ **then** terminate; *(* There is no plan. *)*
$S := G \wedge D_i$;
for j := i-1 **to** 0 **do** *(* Output plan, last operator first. *)*
  choose $o \in O$ such that $D_j \not\models \neg wpreimg_{\tau_o}(S)$;
  output o;
  $S := wpreimg_{\tau_o}(S) \wedge D_j$;
end

Figure 3.7: Algorithm for deterministic planning (forward, in terms of formulae)

represents reachability by i steps or less. In the second version the transition relation $\mathcal{R}_1(A, A')$ encodes reachability by 0 or 1 steps, so we directly obtain reachability by i steps or less, without having to take union ($\vee$) with D_{i-1}.

Theorem 3.40 *Let a state s be in $D_i \backslash D_{i-1}$. Then there is a plan that reaches s from the initial state by i operator applications.*

Proof: □

3.7.3 A backward planning algorithm

The second algorithm computes the distances to the goal states. This computation proceeds by preimage computation starting from the goal states, so D_0 consists of the goal states, D_1 the states with distance 1 to the goal states, and so on. The algorithm is given in Figure 3.8.

We can express the same algorithm in terms of formulae in the propositional logic, see Figure 3.9.

Theorem 3.41 *Let a state s be in $D_i \backslash D_{i-1}$. Then there is a plan that reaches from s a goal state by i operator applications.*

Proof: □

3.8 Computational complexity

In this section we discuss the computational complexity of the main decision problems related to deterministic planning.

The plan existence problem of deterministic planning is PSPACE-complete. The result was proved by Bylander [1994]. He proved the hardness part by giving a simulation of deterministic

procedure planbwd(I,O,G)
D_0 := G;
i := 0;
while $I \notin D_i$ **and** ($i = 0$ **or** $D_{i-1} \neq D_i$) **do**
  $i := i + 1$;
  $D_i := D_{i-1} \cup \bigcup_{o \in O}$ *wpreimg*$_o(D_{i-1})$;
end
if $I \notin D_i$ **then** terminate; *(* There is no plan. *)*
$s := I$;
for $j := i - 1$ **to** 0 **do** *(* Output plan, first operator first. *)*
  choose $o \in O$ such that *app*$_o(s) \in D_j$;
  output o;
  $s :=$ *app*$_o(s)$;
end

Figure 3.8: Algorithm for deterministic planning (backward, in terms of states)

procedure planbwd(I,$\mathcal{R}_1(A, A')$,G)
D_0 := G;
i := 0;
while $I \not\models D_i$ **and** ($i = 0$ **or** $\not\models D_{i-1} \leftrightarrow D_i$) **do**
  $i := i + 1$;
  $D_i := \exists A'.(\mathcal{R}_1(A, A') \wedge (D_{i-1}[p'_1/p_1, \ldots, p'_n/p_n]))$;
end

Figure 3.9: Algorithm for deterministic planning (backward, in terms of formulae)

polynomial-space Turing machines, and the membership part by giving an algorithm that solves the problem in polynomial space. We later generalize his Turing machine simulation to alternating Turing machines to obtain an EXP-hardness proof for nondeterministic planning with full observability in Theorem 4.42.

Theorem 3.42 *The problem of testing the existence of a plan is PSPACE-hard.*

Proof: Let $\langle \Sigma, Q, \delta, q_0, g \rangle$ be any deterministic Turing machine with a polynomial space bound $p(x)$. Let σ be an input string of length n.

We construct a problem instance in deterministic planning with for simulating the Turing machine. The problem instance has a size that is polynomial in the size of the description of the Turing machine and the input string.

The set A of state variables in the problem instance consists of

1. $q \in Q$ for denoting the internal states of the TM,
2. s_i for every symbol $s \in \Sigma \cup \{|, \square\}$ and tape cell $i \in \{0, \ldots, p(n)\}$, and
3. h_i for the positions of the R/W head $i \in \{0, \ldots, p(n) + 1\}$.

The initial state of the problem instance represents the initial configuration of the TM. The initial state I is as follows.

1. $I(q_0) = 1$
2. $I(q) = 0$ for all $q \in Q \backslash \{q_0\}$.
3. $I(s_i) = 1$ if and only if ith input symbol is $s \in \Sigma$, for all $i \in \{1, \ldots, n\}$.
4. $I(s_i) = 0$ for all $s \in \Sigma$ and $i \in \{0, n + 1, n + 2, \ldots, p(n)\}$.
5. $I(\square_i) = 1$ for all $i \in \{n + 1, \ldots, p(n)\}$.
6. $I(\square_i) = 0$ for all $i \in \{0, \ldots, n\}$.
7. $I(|_0) = 1$
8. $I(|_i) = 0$ for all $n \in \{1, \ldots, p(n)\}$
9. $I(h_1) = 1$
10. $I(h_i) = 0$ for all $i \in \{0, 2, 3, 4, \ldots, p(n) + 1\}$

The goal is the following formula.

$$G = \bigvee \{q \in Q | g(q) = \text{accept}\}$$

To define the operators, we first define effects corresponding to all possible transitions.

For all $\langle s, q \rangle \in (\Sigma \cup \{|, \square\}) \times Q$, $i \in \{0, \ldots, p(n)\}$ and $\langle s', q', m \rangle \in (\Sigma \cup \{|\}) \times Q \times \{L, N, R\}$ define the effect $\tau_{s,q,i}(s', q', m)$ as $\alpha \wedge \kappa \wedge \theta$ where the effects α, κ and θ are defined as follows.

The effect α describes what happens to the tape symbol under the R/W head. If $s = s'$ then $\alpha = \top$ as nothing on the tape changes. Otherwise, $\alpha = \neg s_i \wedge s'_i$ to denote that the new symbol in the ith tape cell is s' and not s.

```
procedure reach(O,s,s',m)
if m = 0 then                                   (* Plans of length 0 and 1 *)
  if s = s' or there is o ∈ O such that s' = app_o(s) then return true
  else return false
else
  begin                                         (* Longer plans *)
    for all states s'' do                       (* Iteration over intermediate states *)
      if reach(O,s,s'',m − 1) and reach(O,s'',s',m − 1) then return true
    end
    return false;
  end
```

Figure 3.10: Algorithm for testing plan existence in polynomial space

The effect κ describes the change to the internal state of the TM. Again, either the state changes or does not, so $\kappa = \neg q \wedge q'$ if $q \neq q'$ and $\top$ otherwise. We define $\kappa = \neg q$ when $i = p(n)$ and $m = R$ so that when the space bound gets violated, no accepting state can be reached.

The effect θ describes the movement of the R/W head. Either there is movement to the left, no movement, or movement to the right. Hence

$$\theta \quad = \quad \begin{cases} \neg h_i \wedge h_{i-1} & \text{if } m = L \\ \top & \text{if } m = N \\ \neg h_i \wedge h_{i+1} & \text{if } m = R \end{cases}$$

By definition of TMs, movement at the left end of the tape is always to the right. Similarly, we have state variable for R/W head position $p(n) + 1$ and moving to that position is possible, but no transitions from that position are possible, as the space bound has been violated.

Now, these effects that represent possible transitions are used in the operators that simulate the Turing machine. Let $\langle s, q\rangle \in (\Sigma \cup \{|, \square\}) \times Q$, $i \in \{0, \ldots, p(n)\}$ and $\delta(s, q) = \{\langle s', q', m\rangle\}$. If $g(q) = \exists$, then define the operator

$$o_{s,q,i} = \langle h_i \wedge s_i \wedge q, \tau_{s,q,i}(s', q', m)\rangle.$$

We claim that the problem instance has a plan if and only if the Turing machine accepts without violating the space bound.

If the Turing machine violates the space bound, the state variable $h_{p(n)+1}$ becomes true and an accepting state cannot be reached because no further operator will be applicable.

So, because all deterministic Turing machines with a polynomial space bound can be in polynomial time translated to a planning problem, all decision problems in PSPACE are polynomial time many-one reducible to deterministic planning, and the plan existence problem is PSPACE-hard. □

Theorem 3.43 *The problem of testing the existence of a plan is in PSPACE.*

Proof: A recursive algorithm for testing m-step reachability between two states with $\log m$ memory consumption is given in Figure 3.10.

We show that when the algorithm is called with the number $n = |A|$ of state variables as the last argument, it consumes a polynomial amount of memory in n. The recursion depth is n. At the

recursive calls memory is needed for storing the intermediate states s'. The memory needed for this is polynomial in n. Hence at any point of time the space consumption is $\mathcal{O}(m^2)$.

A problem instance $\langle A, I, O, G\rangle$ with $n = |A|$ state variables has a plan if and only if reach(O,I,s',n) returns *true* for some s' such that $s' \models G$. Iteration over all states s' can be performed in polynomial space and testing $s' \models G$ can be performed in polynomial time in the size of G. Hence the whole memory consumption is polynomial. □

Part of the high complexity of planning is due to the fact that plans can be exponentially long. If a polynomial upper bound for plan length exists, testing the existence of plans is still intractable but much easier.

Theorem 3.44 *The problem of testing the existence of plans having a length bounded by some polynomial is NP-hard.*

Proof: We reduce the satisfiability problem of the classical propositional logic to the plan existence problem. The length of the plans, whenever they exist, is bounded by the number of propositional variables and hence is polynomial.

Let ϕ be a formula over the propositional variables in A. Let $N = \langle A, \{(a,0)|a \in A\}, O, \phi\rangle$ where $O = \{\langle \top, a\rangle | a \in A\}$ We show that the problem instance N has a plan if and only if the formula ϕ is satisfiable.

Assume $\phi \in SAT$, that is, there is a valuation $v : A \to \{0,1\}$ such that $v \models \phi$. Now take the operators $\{\langle \top, a\rangle | v \models a, a \in A\}$ in any order: these operators form a plan that reach the state v that satisfies ϕ.

Assume N has a plan $o_1, \ldots, o_m$. The valuation $v = \{(a,1)|(\top, a) \in \{o_1, \ldots, o_m\}\} \cup \{(a,0)|a \in A, (\top, a) \notin \{o_1, \ldots, o_m\}\}$ of A is the terminal state of the plan and satisfies ϕ. □

Theorem 3.45 *The problem of testing the existence of plan having a length bounded by some polynomial is in NP.*

Proof: Let $p(m)$ be a polynomial. We give a nondeterministic algorithm that runs in polynomial time and determines whether a plan of length $p(m)$ exists.

Let $N = \langle A, I, O, G\rangle$ be a problem instance.

1. Nondeterministically guess a sequence of $l \leq p(m)$ operators $o_1, \ldots, o_l$ from the set O. Because l is bounded by the polynomial $p(m)$, the time consumption $\mathcal{O}(p(m))$ is polynomial in the size of N.

2. Compute $s = app_{o_l}(app_{o_{l-1}}(\cdots app_{o_2}(app_{o_1}(I))\cdots))$. This takes polynomial time in the size of the operators and the number of state variables.

3. Test $s \models G$. This takes polynomial time in the size of the operators and the number of state variables.

This nondeterministic algorithm correctly determines whether a plan of length at most $p(m)$ exists and it runs in nondeterministic polynomial time. Hence the problem is in NP. □

These theorems show the NP-completeness of the plan existence problem for polynomial-length plans.

3.9 Literature

The idea of progression and regression in planning is old [Rosenschein, 1981]. Our definition of regression in Section 3.2.2 is related to the weakest precondition predicates for program synthesis [de Bakker and de Roever, 1972; Dijkstra, 1976]. Planning researchers have earlier used regression only for a very restricted type of operators without conditional effects.

There has recently been a lot of interest in using general-purpose search algorithms with progression and heuristics that estimate distances between states. Our distance estimation in Section 3.4 generalizes the additive heuristic by Bonet and Geffner [2001] by handling the truth-values symmetrically and by being applicable to a more type of operators with arbitrary preconditions and conditional effects. Other distance estimates with a flavor that is similar to Bonet and Geffner's exist [Haslum and Geffner, 2000; Hoffmann and Nebel, 2001].

Techniques for speeding up heuristic state-space planners include symmetry reduction [Starke, 1991; Emerson and Sistla, 1996] and partial-order reduction [Godefroid, 1991; Valmari, 1991; Alur *et al.*, 1997], both originally introduced outside planning in the context of reachability analysis and model-checking. Both of these techniques address the main problem in heuristic state-space search, high branching factor (number of applicable operators) and high number of states. Both techniques help in reducing the number of states to be traversed when searching for a plan.

The use of algorithms for the satisfiability problem of the classical propositional logic in planning was pioneered by Kautz and Selman, originally as a way of testing satisfiability algorithms, and later shown to be more efficient than other planning algorithms at that time [Kautz and Selman, 1992; 1996]. In addition to Kautz and Selman [1996], parallel plans were used by Blum and Furst in their Graphplan planner [Blum and Furst, 1997]. Parallelism in this context serves the same purpose as partial-order reduction [Godefroid, 1991; Valmari, 1991], namely to avoid considering all orderings of a number of independent actions and hence reduce the amount of search. The notion of parallel plans considered in this lecture is not the only possible one [Rintanen *et al.*, 2004].

The algorithm for invariant computation was originally presented for simple operators without conditional effects [Rintanen, 1998]. The computation parallels the construction of planning graphs in the Graphplan algorithm [Blum and Furst, 1997], and it would seem to us that the notion of planning graph emerged when Blum and Furst noticed that the intermediate stages of the invariant computation are useful for backward search algorithms: if a depth-bound of n is imposed on the search tree, then formulae obtained by m regression steps (suffixes of length m of possible plans) that do not satisfy clauses R_{n-m} cannot lead to a plan, and the search tree can be pruned.

Even though a lot of contemporary planning research uses Graphplan's planning graphs [Blum and Furst, 1997] for various purposes, we have not discussed them in more detail for several reasons. First, the graph character of planning graphs becomes inconvenient when preconditions are arbitrary formulae, not just conjunctions of state variables, and effects may be conditional. As a result, the basic construction steps of planning graphs become unintuitive. Second, even when the operators have the simple form, the practically and theoretically important properties of planning graphs are not graph-theoretic. We can equivalently and just as intuitively represent the contents of planning graphs as sequences of literals and 2-literal clauses, as we have done for instance in Section 3.6. So it seems that the graph representation does not provide advantages over more conventional logic based and set based representations.

The algorithms presented in this section cannot in general be ordered in terms of efficiency. The general-purpose search algorithms with distance heuristics are often very effective in solving

big problem instances with a suitable structure. Sometimes this entails better runtimes than in the SAT/CSP approach because of the high overheads with handling big formulae or constraint nets in the latter. Similarly, there are problems that are quickly solved by the SAT/CSP approach but on which the distance estimation fails and the heuristic search algorithms are not able to find plans quickly.

The main complexity result of the chapter, the PSPACE-completeness of the plan existence problem, is due to Bylander [1994]. Essentially the same result for other kinds of succinct representations of graphs had been established earlier by Lozano and Balcazar [1990].

Any computational problem just NP-hard – not to mention PSPACE-hard – is usually considered to be too difficult to be solved in any but the simplest cases. Because planning even in the deterministic case is PSPACE-hard, there has been interest in finding useful special cases in which it can be guaranteed that the worst-case complexity does not show up. Syntactic restrictions leading to polynomial time planning have been investigated by several researchers [Bylander, 1994; Bäckström and Nebel, 1995], but the restrictions are so strict that very few or no interesting problems can be represented.

The computational complexity of planning with schematic operators has also been analyzed. Schematic operators increase the conciseness of the representations of some problem instances exponentially, and lift the worst-case complexity accordingly. For example, deterministic planning with schematic operators is EXPSPACE-complete [Erol *et al.*, 1995]. If function symbols are allowed, encoding arbitrary Turing machines becomes possible, and the plan existence problem consequently becomes undecidable [Erol *et al.*, 1995].

3.10 Exercises

3.1 Show that regression for goals G that are sets (conjunctions) of state variables and operators with preconditions p that are sets (conjunctions) of state variables and effects that consist of an add list a (a set of state variables that become true) and a delete list d (a set of state variables that become false) can equivalently be defined as $(G \backslash a) \cup p$ when $d \cap G = \emptyset$.

3.2 Show that the problem in Lemma 3.9 is in NP and therefore NP-complete.

3.3 Satisfiability testing in the propositional logic is tractable in some special cases, like for sets of clauses with at most 2 literals in each, and for Horn clauses, that is sets of clauses with at most one positive literal in each clause.

Can you identify special cases in which existence of an n-step plan can be determined in polynomial time (in n and the size of the problem instance), because the corresponding formula transformed to CNF is a set of 2-literal clauses or a set of Horn clauses?

Chapter 4

Conditional planning

Now we relax the two assumptions that characterize deterministic planning, the determinism of the operators and the restriction to one initial state. Instead of an initial state, we will have a formula describing a set of initial states, and our definition of operators will be extended to cover nondeterministic actions.

These extensions to the planning problem mean that the notion of plans as sequences of operators is not sufficient, because the states that will be visited are not uniquely determined by the actions taken so far: different initial states may require different actions, and nondeterministic actions lead to several alternative successor states.

Plans will be mappings from the observations made so far to the next actions to be taken. There are several possibilities in representing such mapping. Our definition of plans has the form of programs consisting of operators, sequences of operators, and conditional that choose subplans based on observations.

What observations can be made has a strong effect on how planning can be done. There are two special cases we will discuss separately from the general conditional planning problem, those with no observations possible and with everything observable.

When there are no observations, the definition of plans reduces to sequences of actions like in deterministic planning, but executing the plans does not always generate the same sequence of states because of nondeterminism and multiple initial states.

For the fully observable case planning algorithms are much simpler than when observability is only partial. In this case plans can alternatively be defined as mappings from states to actions, and there is no need for the plans to have memory in the way program-like plans have, a form of program counter that keeps track which location of the plan is currently executed.

In this chapter we first discuss nondeterministic actions and transition systems, then define what conditional plans are, and then discuss algorithms for the three types of conditional planning, starting from the simplest case of planning with full observability, followed by planning without observability, and finally the general partially observable planning problem. The chapter is concluded by a discussion of the computational complexity of conditional planning.

4.1 Nondeterministic operators

There is often uncertainty about what the exacts effects of an action are. This is because not all aspects of the world can be exactly formalized, and part of the things that are not formalized may

affect the outcomes of the actions.

Consider for example a robot that plays basket ball. However well the robot is designed, there is still always small uncertainty about the exact physical properties of the ball and the hands the robot uses for throwing the ball. Therefore it is possible to predict the outcome of throwing the ball only up to a certain precision, and a ball thrown by the robot may still miss the basket. This would be a typical situation in which we would formalize an action as nondeterministic. It succeeds with a certain probability, and fails otherwise, and the exact conditions that lead to success or failure are outside the formalization of the action.

In other cases nondeterminism arises because formalizing all the things affecting the outcomes of an action does not bring further benefit. Consider a robot that makes and serves coffee for the members of the research lab. It might be well known that certain lab members never drink coffee, that certain lab members always drink coffee right after lunch, and so on. But it would often not be very relevant for the robot to know these things, as its task is simply to make and serve a cup of coffee whenever somebody requests it to do so. So for the coffee making robot we could just formalize the event that somebody requests coffee as a nondeterministic event, even though there are well known deeper regularities that govern these requests.

In this section we extend the definition of operators first given in Section 2.3 to cover nondeterminism and discuss two normal forms for nondeterministic operators, We then present a translation of nondeterministic operators into the propositional logic, and in the next sections we discuss several planning algorithms that can be efficiently implemented with binary decision diagrams that represent transition relations corresponding to nondeterministic actions.

Probabilities can often be associated with the alternative nondeterministic effects an operator may have, and we include the probabilities in our definition of nondeterministic operators. However, the algorithms discussed in this chapter ignore these probabilities, and they will be only needed later for the probabilistic variants of conditional planning in Chapter 5.

Definition 4.1 *Let A be a set of state variables. A* nondeterministic operator *is a pair $\langle c, e\rangle$ where c is a propositional formula over A describing the precondition, and e is a nondeterministic effect. Effects are recursively defined as follows.*

1. *a and $\neg a$ for state variables $a \in A$ are effects.*
2. *$e_1 \wedge \cdots \wedge e_n$ is an effect over A if $e_1, \ldots, e_n$ are effects over A (the special case with $n = 0$ is the empty conjunction $\top$.)*
3. *$c \rhd e$ is an effect over A if c is a formula over A and e is an effect over A.*
4. *$p_1 e_1 | \cdots | p_n e_n$ is an effect over A if $e_1, \ldots, e_n$ for $n \geq 2$ are effects over A, $p_i > 0$ for all $i \in \{1, \ldots, n\}$ and $\sum_{i=1}^{n} p_i = 1$.*

The definition extends Definition 2.7 by allowing nondeterministic choice as $p_1 e_1 | \cdots | p_n e_n$.

Next we give a formal semantics for the application of a nondeterministic operator. The definition of deterministic operator application (Definition 2.8) assigned a state to every state and operator. The new definition assigns a probability distribution over the set of successor states for a given state and operator.

Definition 4.2 (Nondeterministic operator application) *Let $\langle c, e\rangle$ be an operator over A. Let s be a state, that is an assignment of truth values to A. The operator is applicable in s if $s \models c$.*

Recursively assign each effect e a set $[e]_s$ of pairs $\langle p, l\rangle$ where p is a probability $0 < p \leq 1$ and l is a set of literals a and $\neg a$ for $a \in A$.

1. $[a]_s = \{\langle 1, \{a\}\rangle\}$ *and* $[\neg a]_s = \{\langle 1, \{\neg a\}\rangle\}$ *for* $a \in A$.
2. $[e_1 \wedge \cdots \wedge e_n]_s = \{\langle \Pi_{i=1}^n p_i, \bigcup_{i=1}^n f_i\rangle | \langle p_1, f_1\rangle \in [e_1]_s, \ldots, \langle p_n, f_n\rangle \in [e_n]_s\}$.
3. $[c' \rhd e']_s = [e']_s$ *if* $s \models c'$ *and* $[c' \rhd e']_s = \{\langle 1, \emptyset\rangle\}$ *otherwise.*
4. $[p_1 e_1 | \cdots | p_n e_n]_s = \{\langle p_1 \cdot p, e\rangle | \langle p, e\rangle \in [e_1]_s\} \cup \cdots \cup \{\langle p_n \cdot p, e\rangle | \langle p, e\rangle \in [e_n]_s\}$

Above in (4) the union of sets is defined so that for example $\{\langle 0.2, \{a\}\rangle\} \cup \{\langle 0.2, \{a\}\rangle\} = \{\langle 0.4, \{a\}\rangle\}$, that is, same sets of changes are combined by summing their probabilities.

The successor states of s under the operator are ones that are obtained from s by making the literals in f for $\langle p, f\rangle \in [e]_s$ true and retaining the truth-values of state variables not occurring in f. The probability of a successor state is the sum of the probabilities p for $\langle p, f\rangle \in [e]_s$ that lead to it.

Each $\langle p, l\rangle$ means that with probability p the literals that become true are those in l, and hence indicate the probabilities of the possible successor states of s. For any $[e]_s = \{\langle p_1, l_1\rangle, \ldots, \langle p_n, l_n\rangle\}$ the sum of probabilities is $\sum_{i=1}^n p_i = 1$.

In non-probabilistic variants of planning we also use a semantics that ignores the probabilities. The following definition gives those successor states that have a non-zero probability according to the preceding definition.

Definition 4.3 (Nondeterministic operator application II) *Let $\langle c, e\rangle$ be an operator over A. Let s be a state, that is an assignment of truth values to A. The operator is applicable in s if $s \models c$. Recursively assign each effect e a set $[e]_s$ of literals a and $\neg a$ for $a \in A$.*

1. $[a]_s = \{\{a\}\}$ *and* $[\neg a]_s = \{\{\neg a\}\}$ *for* $a \in A$.
2. $[e_1 \wedge \cdots \wedge e_n]_s = \{\bigcup_{i=1}^n f_i | f_1 \in [e_1]_s, \ldots, f_n \in [e_n]_s\}$.
3. $[c' \rhd e']_s = [e']_s$ *if* $s \models c'$ *and* $[c' \rhd e']_s = \{\emptyset\}$ *otherwise.*
4. $[p_1 e_1 | \cdots | p_n e_n]_s = [e_1]_s \cup \cdots \cup [e_n]_s$

The successor states under $\langle c, e\rangle$ are obtained from s by assigning the sets of literals in $[e]_s$ true.

4.1.1 Normal forms for nondeterministic operators

We can generalize the normal form defined in Section 2.3.2 to nondeterministic effects and operators. In the normal formal form the nondeterministic choices together with conjunctions are outside, and all atomic effects are as consequents of conditionals.

For showing that every nondeterministic effect can be transformed into normal form we have extended our set of equivalences on effects to cover nondeterministic choice. The whole set of equivalences is given in Table 4.1.

$$
\begin{aligned}
c \rhd (e_1 \wedge \cdots \wedge e_n) &\equiv (c \rhd e_1) \wedge \cdots \wedge (c \rhd e_n) && (4.1)\\
c \rhd (c' \rhd e) &\equiv (c \wedge c') \rhd e && (4.2)\\
c \rhd (p_1 e_1 | \cdots | p_n e_n) &\equiv p_1 (c \rhd e_1) | \cdots | p_n (c \rhd e_n) && (4.3)\\
(c_1 \rhd e) \wedge (c_2 \rhd e) &\equiv (c_1 \vee c_2) \rhd e && (4.4)\\
e \wedge (c \rhd e) &\equiv e && (4.5)\\
e &\equiv \top \rhd e && (4.6)\\
e \wedge (p_1 e_1 | \cdots | p_n e_n) &\equiv p_1 (e \wedge e_1) | \cdots | p_n (e \wedge e_n) && (4.7)\\
p_1 (p'_1 e'_1 | \cdots | p'_n e'_n) | p_2 e_2 | \cdots | p_n e_n &\equiv (p_1 p'_1) e'_1 | \cdots | (p_1 p'_n) e'_n | p_2 e_2 | \cdots | p_n e_n && (4.8)\\
p_1 (e' \wedge (c \rhd e_1)) | p_2 e_2 | \cdots | p_n e_n &\equiv (c \rhd (p_1 (e' \wedge e_1) | p_2 e_2 | \cdots | p_n e_n)) && (4.9)\\
&\quad \wedge (\neg c \rhd (p_1 e' | p_2 e_2 | \cdots | p_n e_n)) && (4.10)
\end{aligned}
$$

Table 4.1: Equivalences on effects

Definition 4.4 (Normal form for nondeterministic operators) *An effect is in* normal form *if it can be derived as follows.*

*A deterministic effect is in normal form if it is a conjunction (*0 *or more conjuncts) of effects $c \rhd p$ and $c \rhd \neg p$, with at most one occurrence of p and $\neg p$ for any state variable $p \in A$.*

A nondeterministic effect is in normal form if it is $p_1 e_1 | \cdots | p_n e_n$ for deterministic effects e_i that are in normal form, or it is a conjunction of nondeterministic effects in normal form.

A nondeterministic operator $\langle c, e \rangle$ is in normal form if its effect is in normal form.

Theorem 4.5 *For every operator there is an equivalent one in normal form. There is one that has a size that is polynomial in the size of the former.*

Proof: By using equivalences 4.1, 4.2 and 4.3 in Table 4.1 we can transform any effect so that all atomic effects l occur as consequents of conditional $c \rhd l$. By further using equivalence 4.7 we can transform the effect to normal form. □

Example 4.6 The effect

$$a \rhd (0.3b | 0.7(c \wedge f)) \wedge (0.2(d \wedge e) | 0.8(b \rhd e))$$

in normal form is

$$(0.3(a \rhd b) | 0.7((a \rhd c) \wedge (a \rhd f))) \wedge (0.2((\top \rhd d) \wedge (\top \rhd e)) | 0.8(b \rhd e)).$$

■

In certain cases, for example for defining regression for nondeterministic operators, it is best to restrict to operators in a slightly more restrictive normal form, in which nondeterminism may appear only at the topmost structure in the effect.

Definition 4.7 (Normal form II for nondeterministic operators) *An effect is in* normal form II *if it can be derived as follows.*

*A deterministic effect is in normal formal II if it is a conjunction (*0 *or more conjuncts) of effects* $c \rhd p$ *and* $c \rhd \neg p$*, with at most one occurrence of* p *and* $\neg p$ *for any state variable* $p \in A$*.*

A nondeterministic effect is in normal form II if it is of form $p_1 e_1 | \cdots | p_n e_n$ *where* e_i *are deterministic effects in normal form II.*

A nondeterministic operator $\langle c, e \rangle$ *is in normal form if its effect is in normal form.*

4.1.2 Translation of nondeterministic operators into propositional logic

In Section 3.5.2 we gave a translation of deterministic operators into the propositional logic. In this section we extend this translation to nondeterministic operators.

For expressing the translation we define for a given effect e a set *changes*(e) of state variables as follows. This is the set of state variables possibly changed by the effect, or in other words, the set of state variables occurring in the effect not in the antecedent c of a conditional $c \rhd e$.

$$\begin{aligned}
\textit{changes}(a) &= \{a\} \\
\textit{changes}(\neg a) &= \{a\} \\
\textit{changes}(c \rhd e) &= \textit{changes}(e) \\
\textit{changes}(e_1 \wedge \cdots \wedge e_n) &= \textit{changes}(e_1) \cup \cdots \cup \textit{changes}(e_n) \\
\textit{changes}(p_1 e_1 | \cdots | p_n e_n) &= \textit{changes}(e_1) \cup \cdots \cup \textit{changes}(e_n)
\end{aligned}$$

We make the following assumption to slightly simplify the translation.

Assumption 4.8 *Let* $a \in A$ *be a state variable. Let* $e_1 \wedge \cdots \wedge e_n$ *occur in the effect of an operator. If* $e_1, \ldots, e_n$ *are not all deterministic, then* a *or* $\neg a$ *may occur as an atomic effect in at most one of* $e_1, \ldots, e_n$*.*

This assumption rules out effects like $(0.5a|0.5b) \wedge (0.5\neg a|0.5c)$ that may make a simultaneously true and false. It also rules out effects like $(0.5(d \rhd a)|0.5b) \wedge (0.5(\neg d \rhd \neg a)|c)$ that are well-defined and could be translated into the propositional logic. However, the additional complexity to the translation outweights the benefit of allowing them.

We define the translation of effects satisfying Assumption 4.8 into propositional logic recursively. The problem in the translation that does not show up with deterministic operators is that for nondeterministic choices $p_1 e_1 | \cdots | p_n e_n$ the formula for each alternative e_i has to express for exactly the same set of state variables what changes take or do not take place. This becomes a bit tricky when we have a lot of nesting of nondeterministic choice and conjunctions.

Now we give the translation of an effect e (in normal form) restricted to state variables B. This means that only state variables in B may occur in e in atomic effects (but do not have to), and the formula does not say anything about the change of state variables not in B (but may of course refer to them in antecedents of conditionals.)

$$\begin{aligned}
\mathrm{PL}_B(e) &= \textstyle\bigwedge_{a \in B} (((a \wedge \neg \mathit{EPC}_{\neg a}(e)) \vee \mathit{EPC}_a(e)) \leftrightarrow a') \\
&\quad \text{when } e \text{ is deterministic} \\
\mathrm{PL}_B(p_1 e_1 | \cdots | p_n e_n) &= \mathrm{PL}_B(e_1) \vee \cdots \vee \mathrm{PL}_B(e_n) \\
\mathrm{PL}_B(e_1 \wedge \cdots \wedge e_n) &= \mathrm{PL}_{B \setminus (B_2 \cup \cdots \cup B_n)}(e_1) \wedge \mathrm{PL}_{B_2}(e_2) \wedge \cdots \wedge \mathrm{PL}_{B_n}(e_n) \\
&\quad \text{where } B_i = \textit{changes}(e_i) \text{ for all } i \in \{1, \ldots, n\}
\end{aligned}$$

The first part of the translation $\mathrm{PL}_B(e)$ for deterministic e is the translation of deterministic effects we presented in Section 3.5.2, but restricted to state variables in B. The other two cover all

nondeterministic effects in normal form. The idea of the translation of a conjunction $e_1 \wedge \cdots \wedge e_n$ of nondeterministic effects is that only the translation of the first effect e_1 indicates when state variables occurring in B do not change.

Additionally, we require that operators are not applied in states in which some state variable would be set simultaneously both true and false.

$$\begin{array}{rcl}
\mathrm{XPL}_B(e) & = & \bigwedge_{a\in B}(\neg(\mathit{EPC}_{\neg a}(e) \wedge \mathit{EPC}_a(e))) \\
 & & \text{when } e \text{ is deterministic} \\
\mathrm{XPL}_B(p_1e_1|\cdots|p_ne_n) & = & \mathrm{XPL}_B(e_1) \wedge \cdots \wedge \mathrm{XPL}_B(e_n) \\
\mathrm{XPL}_B(e_1 \wedge \cdots \wedge e_n) & = & \mathrm{XPL}_{B\setminus(B_2\cup\cdots\cup B_n)}(e_1) \wedge \mathrm{XPL}_{B_2}(e_2) \wedge \cdots \wedge \mathrm{XPL}_{B_n}(e_n) \\
 & & \text{where } B_i = \mathit{changes}(e_i) \text{ for all } i \in \{1,\ldots,n\}
\end{array}$$

The translation of an effect e in normal form into the propositional logic is $\mathrm{PL}_A(e) \wedge \mathrm{XPL}_A(e)$ where A is the set of all state variables.

Example 4.9 We translate the effect

$$e = (0.5A|0.5(C \rhd A)) \wedge (0.5B|0.5C)$$

into a propositional formula. The set of state variables is $A = \{A, B, C, D\}$.

$$\begin{array}{rcl}
\mathrm{PL}_{\{A,B,C,D\}}(e) & = & \mathrm{PL}_{\{A,D\}}(0.5A|0.5(C \rhd A)) \wedge \mathrm{PL}_{\{B,C\}}(0.5B|0.5C) \\
 & = & (\mathrm{PL}_{\{A,D\}}(A) \vee \mathrm{PL}_{\{A,D\}}(C \rhd A)) \wedge \\
 & & (\mathrm{PL}_{\{B,C\}}(B) \vee \mathrm{PL}_{\{B,C\}}(C)) \\
 & = & ((A' \wedge (D \leftrightarrow D')) \vee (((A \vee C) \leftrightarrow A') \wedge (D \leftrightarrow D'))) \wedge \\
 & & ((B' \wedge (C \leftrightarrow C')) \vee ((B \leftrightarrow B') \wedge C'))
\end{array}$$

■

4.1.3 Operations on nondeterministic transitions represented as formulae

In Section 3.7.1 we discussed the image and preimage computations of transition relations expressed as propositional formulae. In this section we consider also nondeterministic transition relations and want to compute the set of states from which reaching a state in a given set of states is certain, not just possible. The (weak) preimage operation in Section 3.7 does not do this. For example, the weak preimage of a with respect to the relation $\{\langle b, a\rangle, \langle b, c\rangle\}$ is $\{b\}$, although also c is a possible successor state of b.

The strong preimage of a set of states consists of those states from which only states inside the given set are reached. This is formally defined as follows.

$$\mathit{spreimg}_R(S) = \{s | s' \in S, \langle s, s'\rangle \in R, \mathit{img}_R(s) \subseteq S\}$$

Lemma 4.10 *Images, strong preimages and weak preimages of sets of states are related to each other as follows.*

1. $\mathit{spreimg}_o(S) \subseteq \mathit{wpreimg}_o(S)$
2. $\mathit{img}_o(\mathit{spreimg}_o(S)) \subseteq S$
3. $\mathit{wpreimg}_o(S) = \mathit{spreimg}_o(S)$ *when* o *is deterministic.*

Proof: □

Strong preimages can be computed by formula manipulation when sets of states and transition relations are represented as propositional formulae.

$$(\forall A'.(\mathcal{R}_o(A,A') \rightarrow (\phi[a'_1/a_1,\ldots,a'_n/a_n]))) \wedge (\exists A'.\mathcal{R}_o(A,A'))$$

Here $\forall a.\phi$ is *universal abstraction* which is defined analogously to existential abstraction as

$$\forall a.\phi = \phi[\top/a] \wedge \phi[\bot/a].$$

4.1.4 Regression for nondeterministic operators

Regression for deterministic operators was given as Definition 3.6. It is straightforward to generalize this definition for nondeterministic operators in the second (more restricted) normal form.

Definition 4.11 (Regression) *Let ϕ be a propositional formula describing a set of states. Let $\langle z, e\rangle$ be an operator in normal form II with $e = p_1e_1|\cdots|p_ne_n$.*

The regression *of ϕ with respect to $o = \langle z, e\rangle$ is defined as the formula $regr_o(\phi) = regr_{\langle z,e_1\rangle}(\phi) \wedge \cdots \wedge regr_{\langle z,e_n\rangle}(\phi)$ where $regr_{\langle z,e\rangle}(\phi)$ refers to regression of deterministic operators as given in Definition 3.6.*

It is presumably possible to define regression for nodeterministic operators in the first normal with no restriction on nesting of nondeterminism and conjunctions, but the definition is more complicated, and we do not discuss the topic further here.

Theorem 4.12 *Let $S' = \{s'|s' \models \phi\}$. Then $spreimg_o(S) = \{s|s \models regr_o(\phi)\}$.*

Proof: This is because a state in ϕ has to be reached no matter which effect e_i is chosen, so we take the intersection/conjunction of the states obtained by regression with $\langle z, e_1\rangle, \ldots, \langle z, e_n\rangle$. □

Example 4.13 Let $o = \langle A, (0.5B|0.5\neg C)\rangle$. Then

$$\begin{aligned} regr_o(B \leftrightarrow C) &= regr_{\langle A,B\rangle}(B \leftrightarrow C) \wedge regr_{\langle A,\neg C\rangle}(B \leftrightarrow C) \\ &= (A \wedge (\top \leftrightarrow C)) \wedge (A \wedge (B \leftrightarrow \bot)) \\ &\equiv (A \wedge C) \wedge (A \wedge \neg B) \\ &\equiv A \wedge C \wedge \neg B \end{aligned}$$

■

4.2 Problem definition

We state the conditional planning problem in the general form. Because the number of observations that are possible has a very strong effect on the type of solution techniques that are applicable, we will discuss algorithms for three classes of planning problems that are defined in terms of restrictions on the set B of observable state variables.

Definition 4.14 *A 5-tuple $\langle A, I, O, G, B\rangle$ consisting of a set A of state variables, a propositional formula I over A, a set O of operators over A, a propositional formula G over A, and a set $B \subseteq A$ of state variables is* a problem instance in nondeterministic planning.

The set B did not appear in the definition of deterministic planning. This is the set of *observable state variables*. The idea is that plans can make decisions about what operations to apply and how the execution proceeds based on the values of the observable state variables. Restrictions on observability and sensing emerge because of various restrictions on the sensors human beings and robots have: typically only a small part of the world can be observed.

The task in nondeterministic planning is the same as in deterministic planning (Section 3.1): to find a plan that starting from any state in I is guaranteed to reach a state in G.

However, because of nondeterminism and the possibility of more than one initial state, it is in general not possible to use the same sequence of operators for reaching the goals from all the initial states, and a more general notion of plans has to be used.

Nondeterministic planning problems under certain restrictions have very different properties than the problem in its full generality. In Chapter 3 we had the restriction to one initial state (I was defined as a valuation) and deterministic operators. We relax these two restrictions in this chapter, but still consider two special cases obtained by restrictions on the set B of observable state variables.

1. Full observability.

   This is the most direct extension of the deterministic planning problem of the previous chapter. The difference is that we have to use a more general notion of plans with branches (and with loops, if there is no upper bound on the number of actions that might be needed to reach the goals.)

2. No observability.

   Planning without observability can be considered more difficult than planning with full observability, although they are in many respects not directly comparable.

   The main difference to deterministic planning as discussed in Chapter 3 and to planning with full observability is that during plan execution it is not known what the actual current state is, and there are several possible current states. This complication means that planning takes place in *the belief space*: the role of individual states in deterministic planning is taken by sets of states, called *belief states*.

   Because no observations can be made, branching is not possible, and plans are still just sequences of actions, just like in deterministic planning with one initial state.

The type of observability we consider in this lecture is very restricted as only values of individual state variables can be observed (as opposed to arbitrary formulae) and observations are independent of what operators have been executed before. Hence we cannot for example directly express special sensing actions. However, extensions to the above definition like sensing actions can be relatively easily reduced to the basic definition but we will not discuss this topic further.

4.2.1 Conditional plans

Plans are directed graphs with nodes of degree 1 labeled with operators and edges from nodes of degree ≥ 2 labeled with formulae.

Definition 4.15 *Let $\langle A, I, O, G, B\rangle$ be a problem instance in nondeterministic planning. A* conditional plan *is a triple $\langle N, b, l\rangle$ where*

- *N is a finite set of nodes,*
- *$b \in N$ is the initial node,*
- *$l : N \to (O \times N) \cup 2^{\mathcal{L}\times N}$ is a function that assigns each node an operator and a successor node $\langle o, n\rangle \in O \times N$ or a set of conditions and successor nodes $\langle \phi, n\rangle$.*

  Here ϕ are formulae over B.

Plan execution begins from the initial node b, and the sequence of operators and states generated when executing a plan is determined as follows.

Let $n \in N$ be a node in the plan. If $l(n) = \langle o, n'\rangle$ then n is an operator node. If $l(n) = \emptyset$ then n is a terminal node. Otherwise n is a branch node and $l(n) = \{\langle \phi_1, n_1\rangle, \ldots, \langle \phi_m, n_m\rangle\}$ for some m.

Execution in an operator node with label $\langle o, n\rangle$ proceeds by applying operator o and making n the current plan node.

Execution in a branch node n with label $l(n) = \{\langle \phi_1, n_1\rangle, \ldots, \langle \phi_m, n_m\rangle\}$ proceeds by evaluating the formulae ϕ_i with respect to the valuation s of the observable state variables, and if $s \models \phi_i$, then making n_i the current plan node.[1]

Plan execution ends in a terminal node $n \in N, l(n) = \emptyset$.

The plans can of course be written in the same form as programs in conventional programming languages by using *case* statements for branching and *goto* statements for jumping to the successor nodes of a plan node.

Example 4.16 Consider the plan $\langle N, b, l\rangle$ for a problem instance with the operators $O = \{o_1, o_2, o_3\}$, where

$$\begin{aligned} N &= \{1, 2, 3, 4, 5\} \\ b &= 1 \\ l(1) &= \langle o_3, 2\rangle \\ l(2) &= \{\langle \phi_1, 1\rangle, \langle \phi_2, 3\rangle, \langle \phi_3, 4\rangle\} \\ l(3) &= \langle o_2, 4\rangle \\ l(4) &= \{\langle \phi_4, 1\rangle, \langle \phi_5, 5\rangle\} \\ l(5) &= \emptyset \end{aligned}$$

This could be visualized as the program.

1: o_3
2: CASE
   ϕ_1: GOTO 1
   ϕ_2: GOTO 3
   ϕ_3: GOTO 4
3: o_2
4: CASE
   ϕ_4: GOTO 1
   ϕ_5: GOTO 5
5:

[1] The result of plan execution is undefined if there are several formulae true in the state s.

Every plan $\langle N, b, l\rangle$ can be written as such a program. Nodes n with $l(n) = \emptyset$ corresponds to *goto*s to the program's last label after which nothing follows. ■

A plan is *acyclic* if it is a directed acyclic graph in the usual graph theoretic sense.

4.2.2 Execution graph

We define the satisfaction of plan objectives in terms of the transition system that is obtained when the original transition system is being controlled by a plan, that is, the plan chooses which of the transitions possible in a state is taken. For goal reachability, without unbounded looping it would be required that any maximal path from an initial state has finite length and ends in a goal state. With unbounded looping it would be required that from any state to which there is a path from an initial state that does not visit a goal state there is a path of length ≥ 0 to a goal state.

Definition 4.17 (Execution graph of a plan) *Let $\langle A, I, O, G, B\rangle$ be a problem instance and $\pi = \langle N, b, l\rangle$ be a plan. Then we define* the execution graph *of π as a pair $\langle M, E\rangle$ where*

1. *$M = S \times N$, where S is the set of Boolean valuations of A,*
2. *$E \subseteq M \times M$ has an edge from $\langle s, n\rangle$ to $\langle s', n'\rangle$ if and only if*
   - *(a) $n \in N$ is an operator node with $l(n) = \langle o, n'\rangle$ and $s' \in img_o(s)$, or*
   - *(b) $n \in N$ is a branch node with $\langle \phi, n'\rangle \in l(n)$ and $s' = s$ and $s \models \phi$.*

Definition 4.18 (Reachability goals RG) *A plan $\pi = \langle N, b, l\rangle$ solves a problem instance $\langle A, I, O, G, B\rangle$ under the* Reachability *(RG) criterion if its execution graph fulfills the following.*

For all states s such that $s \models I$, for every (s', n) to which there is a path from (s, b) that does not visit (s''', n'') for any s''' such that $s''' \models G$ and terminal node n'' there is also a path from (s', n) to some (s'', n') such that $s'' \models G$ and n' is a terminal node.

This plan objective with unbounded looping can be interpreted probabilistically. For every nondeterministic choice in an operator we have to assume that each of the alternatives has a non-zero probability. Then for goal reachability, a plan with unbounded looping is simply a plan that has no finite upper bound on the length of its executions, but that with probability 1 eventually reaches a goal state. A non-looping plan also reaches a goal state with probability 1, but there is a finite upper bound on the execution length.

Definition 4.19 (Maintenance goals MG) *A plan $\pi = \langle N, b, l\rangle$ solves a problem instance $\langle A, I, O, G, B\rangle$ under the* Maintenance *(MG) criterion if its execution graph fulfills the following.*

For all states s and s' and plan nodes $n \in N$ such that $s \models I$, if there is a path of length ≥ 0 from (s, b) to some (s', n), then $s' \models G$ and (s', n) has a successor.

We can also define a plan objective that combines the reachability and maintenance criteria: visit infinitely often one of the goal states. This is a proper generalization of both of the criteria because we can rather easily reduce both special cases to the general case. Algorithms for the general case generalize algorithms for both special cases.

4.3 Planning with full observability

When during plan execution the current state is always exactly known, plans can be found by the same kind of state space traversal algorithms already used for deterministic planning in Section 3.7.

The differences to algorithms for deterministic planning stem from nondeterminism. The main difference is that successor states are not uniquely determined by the current state and the action, and different action may be needed for each successor state. Further, nondeterminism may require loops. Consider tossing a die until it yields 6. Plan for this task involves tossing the die over and over, and there is no upper bound on the number of tosses that might be needed.[2] Hence we need plans with loops for representing the sequences of actions of unbounded length required for solving the problem.

Below in Section 4.3.1 we first discuss the simplest algorithm for planning with nondeterminism and full observability. The plans this algorithm produces are acyclic, and the algorithm does not find plans for problem instances that only have plans with loops. Then in Section 4.3.2 we present an algorithm that also produces plans with loops. The structure of the algorithm is more complicated. Efficient implementation of these algorithms requires the use of binary decision diagrams or similar representations of sets and transition relations, as discussed in Section 3.7. Like in the BDD-based algorithms for deterministic planning, these algorithm assign a distance to all the states, with a different meaning of distance in different algorithms, and based on the distances either synthesize a program-like plan, or a plan execution mechanism uses the distances directly for selecting the operators to execute. The algorithms in this section are best implemented by representing the formulae as BDDs.

Deterministic planning has both a forward and a backward algorithm that are similar to each other, as described in Section 3.7. However, for nondeterministic problems forward search does not seem to be a good way of doing planning. For backward distances, distance i of state s means that there is a plan for reaching the goals with at most i operators. But there does not seem to be a useful interpretation of the distances computed forwards from the initial states as images of nondeterministic operators. That a goal state or all goal states can be reached by applying some i nondeterministic operators does not say anything about the possible plans, because executing those i operators might also lead to states that are not goal states and from which goal states could be much more difficult to reach.

4.3.1 An algorithm for constructing acyclic plans

The algorithm for constructing acyclic plans is an extension of the algorithm for deterministic planning given in Section 3.7.3. In the first phase the algorithm computes distances of the states. In the second phase the algorithm constructs a plan based on the distances. The distance computation is almost identical to the algorithm for deterministic planning. The only difference is the use of strong preimages instead of the preimages.[3] The second phase is more complicated, and uses the distances for constructing a plan according to Definition 4.15.

The algorithm is given in Figure 4.1. We call the distances computed by the algorithm *strong*

[2] However, for every $p > 0$ there is a finite plan that reaches the goal with probability p or higher.

[3] The algorithm for deterministic planning could use the slightly more complicated strong preimage computation just as well, because strong and weak preimages coincide for deterministic operators. However, this would not have any advantage for deterministic planning.

procedure FOplan(I,O,G)
D_0 := G;
i := 0;
while $I \not\subseteq D_i$ **and** ($i = 0$ **or** $D_{i-1} \neq D_i$) **do**
  $i := i + 1$;
  $D_i := D_{i-1} \cup \bigcup_{o \in O} \mathit{spreimg}_o(D_{i-1})$;
end
$N := \emptyset$;
$l(j) := \emptyset$ for all j;
cnt := 1;
FOplanconstruct(0,I);

Figure 4.1: Algorithm for nondeterministic planning with full observability

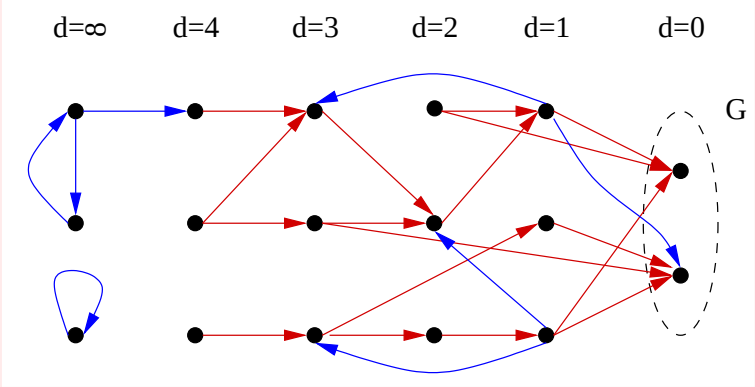


Figure 4.2: Goal distances in a nondeterministic transition system

distances because they are tight upper bounds on the number of operators needed for reaching the goals: if the distance of a state is i, then no more than i operators are needed, but it may be possible that a goal state is also reached with less than i operators if the relevant operators are nondeterministic and the right nondeterministic effects take place.

Example 4.20 We illustrate the distance computation by the diagram in Figure 4.2. The set of states with distance 0 is the set of goal states G. States with distance i are those for which at least one action always leads to states with distance $i-1$ or smaller. In this example the action depicted by the red arrow has this property for every state. States for which there is no finite upper bound on the number of actions for reaching a goal state have distance ∞. ■

Lemma 4.21 *Let a state s be in D_j. Then there is a plan that reaches a goal state from s by at most j operator applications.*

The distances alone could be directly used by a plan execution mechanism. The plan execution proceeds by observing the current state, looking up its distance j such that $s \in D_j \backslash D_{j-1}$, selecting an operator $o \in O$ so that $\mathit{img}_o(\{s\}) \subseteq D_{j-1}$, and executing the operator.

Similarly, a mapping from states to operators could be directly constructed. This kind of plan is called *memoryless* because the plan execution mechanism does not have to keep track of plan

```
procedure FOplanconstruct(n,S)
if S ⊆ G then return;                          (* Goal reached for all states. *)
for each o ∈ O
  S' := the maximal subset of S such that progress(o, S');
  if S' ≠ ∅ then                               (* Is operator o useful for some of the states? *)
    begin
      S := S\S';
      cnt := cnt+2;
      N := N ∪ {cnt-2,cnt-1};                  (* Create two new plan nodes. *)
      l(n) := l(n) ∪ {⟨S',cnt−2⟩};             (* First is reached from node n. *)
      l(cnt−2) := ⟨o,cnt−1⟩;                   (* Second is an operator node. *)
      FOplanconstruct(cnt−1,img_o(S'));        (* Continue from successors of S'. *)
    end
end
if S ≠ ∅ then                                  (* If something remains in S they must be goal states. *)
  begin
    cnt := cnt+1;
    l(n) := l(n) ∪ {⟨S,cnt−1⟩};                (* Create a terminal node for them. *)
  end
```

Figure 4.3: Algorithm for extracting an acyclic plan from goal distances

```
procedure progress(o, S)
for j := 1 to i do                             (* Does o take all states closer to goals? *)
  if img_o(S ∩ D_j) ⊈ D_{j−1} then return false;
end
return true;
```

Figure 4.4: Test whether successor states are closer to the goal states

nodes. It just chooses the next operator on the basis of the current state. This corresponds to a plan that consists of a loop in which each operator is selected for some subset of possible current states, a terminal node is selected for the goal states, and then the loop repeats again.

Memoryless plans are sufficienty powerful only for the simplest form of conditional planning in which the current state can be observed uniquely (full observability). Later we will see that when there are restrictions on which observations can be made it is necessary to have memory in the plan.

Figure 4.3 gives an algorithm for generating a plan according to Definition 4.15. The algorithm works forward starting from the set of initial states. Every operator is tried out, and for an operator that takes some of the states toward the goals a successor node is created, and the algorithm is recursively called for the states that are reached by applying the operator.

The function *progress* which is give in Figure 4.4 tests for a given operator and a set S of states that for every state $s \in S$ all the successor states are at least one step closer to the goals.

procedure prune(O,W,G);
i := 0;
W_0 := W;
repeat
  $i := i + 1$;
  k := 0;
  S_0 := G; *(* States from which G is reachable with 0 steps. *)*
  repeat
    $k := k + 1$; *(* States from which G is reachable with $\leq k$ steps. *)*
    $S_k := S_{k-1} \cup \bigcup_{o \in O}(\mathit{wpreimg}_o(S_{k-1}) \cap \mathit{spreimg}_o(W_{i-1}))$;
  until $S_k = S_{k-1}$; *(* States that stay within W_{i-1} and eventually reach G. *)*
  $W_i := W_{i-1} \cap S_k$;
until $W_i = W_{i-1}$; *(* States in W_i stay within W_i and eventually reach G. *)*
return W_i;

Figure 4.5: Algorithm for detecting a loop that eventually makes progress

4.3.2 An algorithm for constructing plans with loops

There are many nondeterministic planning problems that require plans with loops because there is no finite upper bound on the number of actions that might be needed for reaching the goals. These plan executions with an unbounded length cannot be handled in acyclic plans of a finite size. For unbounded execution lengths we have to allow loops (cycles) in the plans.

Example 4.22 ■

The problem is those states that do not have a finite strong distance as defined Section 4.3.1. Reaching the goals from these states is either impossible or there is no finite upper bound on the number of actions that might be needed. For the former states nothing can be done, but the latter states can be handled by plans with loops.

We present an algorithm based on a generalized notion of distances that does not require reachability by a finitely bounded number of actions. The algorithm is based on the procedure *prune* that identifies a set of states for which reaching a goal state eventually is guaranteed. The procedure *prune* given in Figure 4.5.

Lemma 4.23 (Procedure prune) *Let O be a set of operators and W and G sets of states. Then $W' = \mathit{prune}(O,W,G)$ is a set such that $W' \subseteq W$ and there is function $x : W' \rightarrow O$ such that*

1. *for every $s \in W'$ there is a sequence $s_0, s_1, \ldots, s_n$ with $n \geq 0$ such that $s = s_0$, $s_n \in G$ and $s_{i+1} \in \mathit{img}_{x(s_i)}(\{s_i\})$ for all $i \in \{0, \ldots, n-1\}$,*
2. *$\mathit{img}_{x(s)}(\{s\}) \subseteq W'$ for every $s \in W' \backslash G$, and*
3. *for no $s \in W \backslash W'$ there is a plan that guarantees reaching a state in G.*

Proof:
Let W_0 be the value of W when the procedure is called, and $W_1, W_2, \ldots$ the values of W at the end of the *repeat-until* loop on each iteration.

Induction hypothesis: If $i \geq 1$ then there is function $x : W_i \rightarrow O$ such that

1. for every $s \in W_i$ there is a sequence $s_0, s_1, \ldots, s_n$ with $n \geq 0$ such that $s = s_0$, $s_n \in G$ and $s_{j+1} \in \mathit{img}_{x(s_j)}(\{s_j\})$ for all $j \in \{0, \ldots, n-1\}$, and

2. $\mathit{img}_{x(s)}(\{s\}) \subseteq W_{i-1}$ for every $s \in W_i \backslash G$.

Base case $i = 0$: Trivial because nothing about W_i is claimed.
Inductive case $i \geq 1$:
For the inner *repeat-until* loop we prove inductively the following. Let $S_0 = G$ be the value of S before the loop, and $S_1, S_2, \ldots$ the values of S in the end of each iteration.
Induction hypothesis: If $i \geq 1$ then there is function $x : S_k \to O$ such that

1. for every $s \in S_k$ there is a sequence $s_0, s_1, \ldots, s_n$ with $n \in \{0, \ldots, k\}$ such that $s = s_0$, $s_n \in G$ and $s_{j+1} \in \mathit{img}_{x(s_j)}(\{s_j\})$ for all $j \in \{0, \ldots, n-1\}$, and

2. $\mathit{img}_{x(s)}(\{s\}) \subseteq W_{i-1}$ for every $s \in S_k \backslash G$.

Base case $k = 0$:

1. Because $S_0 = G$, for every $s \in S_0$ there is the sequence of states $s_0 = s$ such that the initial state is in S_0 and the final state is in G.

2. Because $S_0 = G$ there are no states in $S_0 \backslash G$.

Inductive case $k \geq 1$: Let s be a state in S_k. If $s \in S_{k-1}$ then we obtain the property by the induction hypothesis.
Otherwise $s \in S_k \backslash S_{k-1}$. Therefore by definition of S_k, $s \in \mathit{wpreimg}_o(S_{k-1}) \cap \mathit{spreimg}_o(W_{i-1})$ for some $o \in O$.

1. Because $s \in \mathit{wpreimg}_o(S_{k-1})$, there is a state $s' \in S_{k-1}$ such that $s' \in \mathit{img}_o(\{s\})$. By the induction hypothesis there is a sequence of states starting from s' that ends in a goal state. For s such a sequence is obtained from the sequence of s' by prefixing it with s. The corresponding operator is assigned to s by x.

2. Because $s \in \mathit{spreimg}_o(W_{i-1})$, by Lemma 4.10 $\mathit{img}_o(\{s\}) \subseteq W_{i-1}$.

This completes the inner induction. To establish the induction step of the outer induction consider the following. The inner repeat-until loops ends when $S_k = S_{k-1}$. This means that $S_z = S_k$ for all $z \geq k$. Hence the upper bound $n \leq k$ on the length of sequences $s_0, s_1, \ldots, s_n$ is infinite. The outer induction hypothesis is obtained from the inner induction hypothesis by removing the upper bound $n \leq k$ and replacing S_k by W_i. By definition $W_i = W_{i-1} \cap S_k$. **What happens here???**
kesken This finishes the outer induction proof. The claim of the lemma is obtained from the outer induction hypothesis by noticing that the outer loop exits when $W_i = W_{i-1}$ (it will exit after a finite number of steps because the cardinality of $W_0 = W$ is finite and it decreases on every iteration) and then we can replace both W_i and W_{i-1} by W' to obtain the claim of the lemma. □

Our first algorithm, given in Figure 4.6, is directly based on the procedure *prune* and identifying a set of states from which a goal state is reachable by some execution and no execution leads to a state outside the set.

procedure FOplanL2(I,O,G)
W_0 := G;
i := 0;
while $I \not\subseteq$ prune(O,W_i,G) **and** ($i = 0$ **or** $W_{i-1} \neq W_i$) **do**
  $i := i + 1$;
  $W_i := W_{i-1} \cup \bigcup_{o \in O}$ *wpreimg*$_o(W_{i-1})$;
end
S := G;
i := 0;
D_i := G;
L := prune(O,W_i,G);
repeat
  $S' := S$;
  $S := S \cup \bigcup_{o \in O}$(*wpreimg*$_o(S) \cap$ *spreimg*$_o(L \cup S)$);
  $i := i + 1$;
  $D_i := L \cap S$;
until $S = S'$

Figure 4.6: Algorithm for nondeterministic planning with full observability

procedure FOplanMAINTENANCE(I,O,G)
i := 0;
G_0 := G;
repeat
  $i := i + 1$; *(* Subset of G_{i-1} from which G_{i-1} can be always reached. *)*
  $G_i := \bigcup_{o \in O}$ (*spreimg*$_o(G_{i-1}) \cap G_{i-1}$);
until $G_i = G_{i-1}$;
return G_i;

Figure 4.7: Algorithm for nondeterministic planning with full observability and maintenance goals

4.3.3 An algorithm for constructing plans for maintenance goals

There are many important planning problems in which the objective is not to reach a goal state and then stop execution. When the objective is to keep the state of the system in any of a number of goal states indefinitely, we talk about *maintenance goals*.

Plans that satisfy a maintenance goal have only infinite executions.

Figure 4.7 gives an algorithm for finding plans for maintenance goals. The algorithm starts with the set G of all states that satisfy the property to be maintained. Then iteratively such states are removed from G for which the satisfaction of the property cannot be guaranteed in the next time point. More precisely, the sets G_i for $i \geq 0$ consist of all those states in which the goal objective can be maintained for the next i time points. For some i the sets G_i and G_{i-1} coincide, and then $G_j = G_i$ for all $j \geq i$. This means that starting from the states in G_i the goal objective can be maintained indefinitely.

Theorem 4.24 *Let I be a set of initial states, O a set of operator and G a set of goal states. Let G' be the set returned by the procedure FOplanMAINTENANCE in Figure 4.7.*

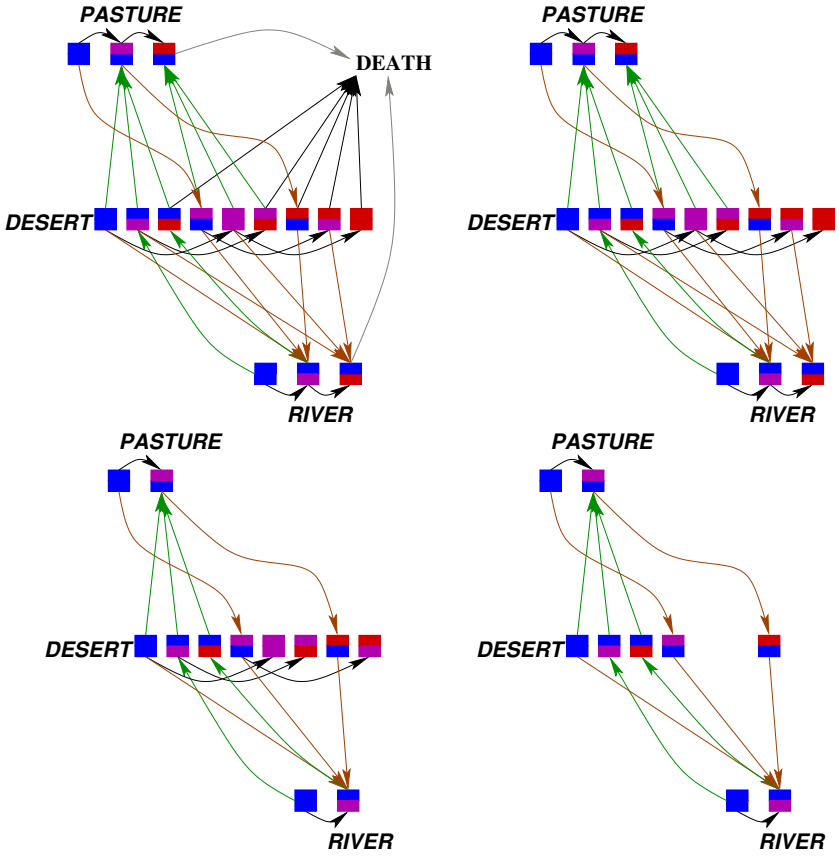


Figure 4.8: Example run of the algorithm for maintenance goals

Then for every state $s \in G'$ there is an operator $o \in O$ such that $img_o(\{s\}) \subseteq G'$. If $I \subseteq G'$ then the corresponding plan satisfies the maintenance criterion for $\langle I, O, G \rangle$.

Proof: □

Example 4.25 Consider the problem depicted in Figure 4.8. An animal may drink at a river and eat at a pasture. To get from the river to the pasture it must go through a desert. Its hunger and thirst increase after every time period. If either one reaches level 3 the animal dies. The hunger and thirst levels are indicated by different colors: the upper halves of the rectanges show thirst level and the lower halves the hunger level, and blue means no hunger or thirst and red means much hunger or thirst. The upper left diagram shows all the possible actions the animal can take. The objective of the animal is to stay alive. The three iterations of the algorithm for finding a plan that satisfies the goal of staying alive are depicted by the remaining three diagrams. The diagram on upper right depicts all the states that satisfy the goal. The diagram on lower left depicts all the states that satisfy the goal and after which the satisfaction of the goal can be guaranteed for at least one time period. The diagram on lower right depicts all the states that satisfy the goal and after which the satisfaction of the goal can be guaranteed for at least two time periods.

Further iterations of the algorithm do not eliminate further states, and hence the last diagram depicts all those states for which the satisfaction of the goal can be guaranteed indefinitely.

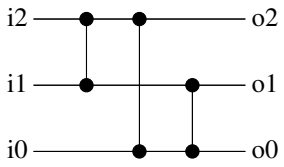


Figure 4.9: A sorting network with three inputs

Hence the only plan says that the animal has to go continuously back and forth between the pasture and the river. The only choice the animal has is in the beginning if in the initial state it is not at all hungry or thirsty. For instance, if it is in the desert initially, then it may freely choose whether to first go to the pasture or the river. ■

4.4 Planning with partial observability

4.4.1 Planning without observability by heuristic search

Planning under unobservability is similar to deterministic planning in the sense that the problem is to find a path from the initial state(s) to the goal states. For unobservable planning, however, the nodes in the graph do not correspond to individual states but to belief states, and the size of the belief space is exponentially higher than the size of the state space. Algorithms for deterministic planning have direct counterparts for unobservable planning, which is not the case for conditional planning with full or partial observability.

Example 4.26 A sorting network [Knuth, 1998, Section 5.3.4 in 2nd edition] consists of a sequence of gates acting on a number of input lines. Each gate combines a comparator and a swapper: if the first value is greater than the second, then swap them. The goal is to sort any given input sequence. The sorting network always has to perform the same operations irrespective of the input, and hence constructing a sorting network corresponds to planning without observability. Figure 4.9 depicts a sorting network with three inputs. An important property of sorting networks is that any network that sorts any sequence of zeros and ones will also sort any sequence of arbitrary numbers. Hence it suffices to consider Boolean 0-1 input values only.

Construction of sorting networks is essentially a planning problem without observability, because there are several initial states and a goal state has to be reached by using the same sequence of actions irrespective of the initial states.

For the 3-input sorting net the initial states are $000, 001, 010, 011, 100, 101, 110, 111$. and the goal states are $000, 001, 011, 111$ Now we can compute the images and strong preimages of the three sorting actions, sort12, sort02 and sort01 respectively starting from the initial or the goal states. These yield the following belief states at different stages of the sorting network.

$000, 001, 010, 011, 100, 101, 110, 111$	initially
$000, 001, 011, 100, 101, 111$	after sort12
$000, 001, 011, 101, 111$	after sort02
$000, 001, 011, 111$	after sort01

■

The most obvious approaches to planning with unobservability is to use regression, strong preimages or images, and to perform backward or forward search in the belief space. The dif-

ference to forward search with deterministic operators and one initial state is that belief states are used instead of states. The difference to backward search for deterministic planning is that regression for nondeterministic operators has to be used and testing whether (a subset of) the initial belief state has been reached involves the co-NP-hard inclusion test $\models I \rightarrow \mathit{regr}_o(\phi)$ for the belief states. With one initial state this is an easy polynomial time test $I \models \mathit{regr}_o(\phi)$ of whether $\mathit{regr}_o(\phi)$ is true in the initial state.

Deriving good heuristics for heuristic search in the belief space is more difficult than in deterministic planning. The main approaches have been to use distances in the state space as an estimate for distances in the belief space, and to use the cardinalities of belief spaces as a measure of progress.

Many problems cannot be solved by blindly taking actions that reduce the cardinality of the current belief state: the cardinality of the belief state may stay the same or increase during plan execution, and hence the decrease in cardinality is not characteristic to belief space planning in general, even though in many problems it is a useful measure of progress.

Similarly, distances in the state space ignore the most distinctive aspect of planning with partial observability: the same action must be used in two states if the states are not observationally distinguishable. A given (optimal) plan for an unobservable problem may increase the actual current state-space distance to the goal states (on a given execution) when the distance in the belief-space monotonically decreases, and vice versa. Hence, the state space distances may yield wildly misleading estimates of the distances in the corresponding belief space.

Heuristics based on state-space distances

The most obvious distance heuristics are based on the strong distances in the state space.

$$\begin{array}{rcl} D_0 & = & G \\ D_{i+1} & = & D_i \cup \bigcup_{o \in O} \mathit{spreimg}_o(D_i) \text{ for all } i \geq 1 \end{array}$$

A lower bound on plan length for belief state Z is j if $Z \subseteq D_j$ and $Z \not\subseteq D_{j-1}$.

Next we derive distance heuristics for the belief space based on state space distances. Strong distances yield an admissible distance heuristic for belief states.

Definition 4.27 (State space distance) *The* state space distance *of a belief state B is $d \geq 1$ when $B \subseteq D_d$ and $B \not\subseteq D_{d-1}$, and it is 0 when $B \subseteq D_0 = G$.*

Even though computing the exact distances for the operator based representation of state spaces is PSPACE-hard, the much higher complexity of planning problems with partial observability still often justifies it: this computation would in many cases be an inexpensive preprocessing step, preceding the much more expensive solution of the partially observable planning problem. Otherwise cheaper approximations can be used.

Heuristics based on belief state cardinality

The second heuristic that has been used in algorithms for partial observability is simply based on the cardinality of the belief states.

In forward search, prefer operators that maximally decrease the cardinality of the belief state.

In backward search, prefer operators that maximally increase the cardinality of the belief state.

These heuristics are not in general admissible, because there is no direct connection between the distance to a goal belief state and the cardinalities of the current belief state and a goal belief state. The belief state cardinality can decrease or increase arbitrarily much by one step.

4.4.2 Planning without observability by evaluation of QBF

In this section we extend the techniques from Section 3.5 to unobservable planning. Because of nondeterminism and several initial states, one plan may have several different executions. It turns out that propositional logic is not suitable for representing planning with unobservability, and the language of quantified Boolean formulae is needed instead. Intuitively, the reason for this is that we have to quantify over an exponential number of plan executions: we want to say "there is a plan so that for all executions...", and expressing this concisely in the propositional logic does not seem possible. We theoretically justify this in Section 4.5.4 by showing that testing the existence of plans for problems instances without observability even when restricting to plans with a polynomial length is complete for the complexity class Σ_2, and not contained in NP as the corresponding problem for deterministic planning. This strongly suggests, because of widely accepted complexity theoretic conjectures, that there is no efficient representation of the problem in the propositional logic.

In Section 4.1.2 we showed how nondeterministic operators can be translated into formulae in the propositional logic. The purpose of that translation was the use of the formulae in BDD-based planning algorithms for computing the images and preimages of sets of states. For the QBF representation of nondeterministic operators we have to have a possibility to universally quantify over all possible successor states an operator produces, and this cannot be easily expressed with the formulae derived in Section 4.1.2, so we give a new translation that uses quantified Boolean formulae (see Section 2.2.1.)

Translation of nondeterministic operators into propositional logic

For handling nondeterminism we need to universally quantify over all the nondeterministic choices, because for every choice the remaining operators in the plan must lead to a goal state. For an effect with n nondeterministic alternatives this can be achieved by using $m = \lceil \log_2 n \rceil$ universally quantified auxiliary variables. For every valuation of these variables one of the alternative effects is chosen.

We assign to every atomic effect a formula that is true if and only if that effect takes place. This is similar to and extends the functions $EPC_l(e)$ in Definition 3.3. The extension concerns nondeterminism: for literal l to become true, the auxiliary variables for nondeterminism have to have values corresponding to an effect making l true.

The condition for atomic effect l to take place when effect e is executed is $nEPC_l(e, \sigma, t)$. The sequence σ of integers is for deriving unique names for auxiliary variables in $nEPC_l(e, \sigma, t)$, and t is formula on the auxiliary variables for deciding when to execute e. The effect is assumed to be in normal form I.

$$
\begin{array}{lcl}
nEPC_l(e, \sigma, t) & = & EPC_l(e) \wedge t \text{ if } e \text{ is deterministic} \\
nEPC_l(p_1 e_1 | \cdots | p_n e_n, \sigma, t) & = & nEPC_l(e_1, \sigma; 1, c_1^n(\sigma) \wedge t) \vee \cdots \vee nEPC_l(e_n, \sigma; n, c_n^n(\sigma) \wedge t) \\
nEPC_l(e_1 \wedge \cdots \wedge e_n, \sigma, t) & = & nEPC_l(e_1, \sigma; 1, t) \vee \cdots \vee nEPC_l(e_n, \sigma; n, t)
\end{array}
$$

The function $c_i^n(\sigma)$ constructs a formula for selecting the ith effect from n alternatives. This formula is usually a conjunction of literals over the auxiliary propositions $A_\sigma^m = \{a_{\sigma,1}, \ldots, a_{\sigma,m}\}$

corresponding to one valuation of A^m_σ. Here $m = \lceil \log_2 n \rceil$. When n is not a power of 2, the last effect e_n corresponds to more than one valuation of A^m_σ. Define

$$d^m_i(\sigma) = \bigwedge(\{a_{\sigma,j} \in A^m_\sigma | j\text{th bit of } i-1 \text{ is } 1\}\{\neg a_{\sigma,j} | a_{\sigma,j} \in A^m_\sigma, j\text{th bit of } i-1 \text{ is } 0\}).$$

When $i \in \{1, \ldots, n-1\}$ (and in the special case $i = n = 2^m$), we define $c^n_i(\sigma)$ as $d^m_i(\sigma)$. When $i = n$ we define $c^n_i(\sigma)$ as

$$d^m_n(\sigma) \vee \cdots \vee d^m_{2^m}(\sigma)$$

Hence effects e_1 to e_{n-1} correspond to binary encodings of numbers 0 to $n-2$ and e_n covers all the remaining valuations of A^m_σ.

The following frame axioms express the conditions under which the state variable $a \in A$ may change from false to true and from true to false. We assume that the operators in $O = \{o_1, \ldots, o_n\}$ have a unique numbering $1, \ldots, n$.

$$(\neg a \wedge a') \rightarrow ((o_1 \wedge \mathit{nEPC}_a(e_1, 1, \top)) \vee \cdots \vee (o_n \wedge \mathit{nEPC}_a(e_n, n, \top)))$$
$$(a \wedge \neg a') \rightarrow ((o_1 \wedge \mathit{nEPC}_{\neg a}(e_1, 1, \top)) \vee \cdots \vee (o_n \wedge \mathit{nEPC}_{\neg a}(e_n, n, \top)))$$

For every operator $o = \langle z, e \rangle \in O$ we have formulae for describing values of state variables in the predecessor and in the successor states when the operator is applied. Let $A = \{a_1, \ldots, a_k\}$ be the state variables. The formulae describing the effects and preconditions of the operator $o_i \in O$ are the following.

$$\begin{array}{rcl}
(o_i \wedge \mathit{nEPC}_{a_1}(e_i, i, \top)) & \rightarrow & a'_1 \\
(o_i \wedge \mathit{nEPC}_{\neg a_1}(e_i, i, \top)) & \rightarrow & \neg a'_1 \\
\vdots & & \\
(o_i \wedge \mathit{nEPC}_{a_k}(e_i, i, \top)) & \rightarrow & a'_k \\
(o_i \wedge \mathit{nEPC}_{\neg a_k}(e_i, i, \top)) & \rightarrow & \neg a'_k \\
o_i & \rightarrow & z
\end{array}$$

Example 4.28 Consider the operators $o_1 = \langle A, (0.5B | 0.5(C \rhd D)) \rangle$ and $o_2 = \langle B, (0.5(D \rhd B) | 0.5C) \rangle$. The application of these operators is described by the following formula.

example missing ■

Two operators may be applied in parallel only if they do not interfere, so we have

$$\neg o_i \vee \neg o_j$$

for all operators i and j such that $i \neq j$ and the operators interfere.

The conjunction of all the above formulae is denoted by

$$\mathcal{R}_3(A, A')$$

When renaming the propositions for time point t, also the propositions o for operators $o \in O$ and the propositions $a \in A$ must be renamed, and for this we use then notation

$$\mathcal{R}^t_3(A^t, A^{t+1}).$$

Finding plans by evaluating QBF

In deterministic planning in propositional logic (Section 3.5) the problem is to find a sequence of operators so that a goal state is reached when the operators are applied starting in the initial state. When there is nondeterminism, the problem is to find a seqence of operators so that a goal state is reached for all possible executions of the sequence of operators. The number of executions of one sequence of operators may be higher than one because there may be several initial states and because the operators may be deterministic. Expressing the quantification over all possible executions of a sequence of operators cannot be concisely expressed in the propositional logic, and this is the reason why quantified Boolean formulae have to be used instead.

$$\begin{array}{l}\exists V_{plan}\\ \forall V_{exec}\\ \exists V_{rest}\\ I^0 \rightarrow (\mathcal{R}_3(A_0, A_1) \wedge \mathcal{R}_3(A_1, A_2) \wedge \cdots \wedge \mathcal{R}_3(A_{n-1}, A_n) \wedge G^n)\end{array}$$

Here $V_{exec} = A_0 \cup A^0 \cup \cdots \cup A^{t-1}$ where A is the set of auxiliary variables occurring in $nEPC_l(e, \epsilon, \top)$ for some $\langle c, e\rangle \in O$ and $l \in \{a, \neg a\}$ for some $a \in A$. The plan is expressed in terms of the variables o^i where $o \in O$ and $i \in \{0, \ldots, t-1\}$. The truth-values of the remaining variables $V_{rest} = A_1 \cup \cdots \cup A_t$ are determined by the operators and the execution chosen by propositions in V_{exec}.

There are algorithms for evaluating QBF that extend the Davis-Putnam procedure and that traversing and-or trees. And-nodes correspond to universally quantified propositions and or-nodes correspond to existentially quantified propositions. These algorithms return the valuation of the outermost existential propositions if the QBF has value *true*.

Finding plans for nondeterministic problems without observability may be more efficient than using standard search algorithms with regression or image/preimage computation with BDDs when the plans are short and there are many operators that can be applied in parallel. If long plans are required and there is little parallelism, the algorithms that traverse the belief space appear to be more efficient.

4.4.3 Algorithms for planning with partial observability

Planning with partial observability is much more complicated than its two special cases with full and no observability. Like planning without observability, the notion of belief states becomes very important. Like planning with full observability, formalization of plans as sequences of operators is insufficient. However, plans also cannot be formalized as mappings from states to operators because partial observability implies that the current state is not necessarily unambiguously known. Hence we will need the general definition of plans introduced in Section 4.2.1.

When executing operator o in belief state B the set of possible successor states is $img_o(B)$, and based on the observation that are made, this set is restricted to $B' = img_o(B) \cap C$ where C is the equivalence class of observationally indistinguishable states corresponding to the observation.

In planning with unobservability, a backward search algorithm starts from the goal belief state and uses regression or strong preimages for finding predecessor belief states until a belief state covering the initial belief state is found.

With partial observability, plans do not just contain operators but may also branch. With branching the sequence of operators may depend on the observations, and this makes it possible to reach

goals also when no fixed sequence of operators reaches the goals. Like strong preimages in backward search correspond to images, the question arises what does branching correspond to in backward search?

Assume that we have for belief states B_1 and B_2 respectively the plans π_1 and π_2 that reach the goals, and that these belief states are observationally distinguishable, that is, they are included in different observational classes. Now we can construct a plan π_{12} that starts with a branch node that makes an observation and continues with π_1 or with π_2, depending on which observation was made. If we are initially in any state in $B_1 \cup B_2$, the plan π_{12} always takes us to a goal state. We can continue extending π_{12} with operators. For example, if $B = \mathit{wpreimg}_o(B_1 \cup B_2)$, then the plan that first executes the operator o and then continues with π will lead to a goal state starting from any state in B.

Next we formalize these ideas and derive an algorithm that constructs branching plans in the backward direction starting from the goal states.

Let $\Pi = \langle C_1, \ldots, C_n \rangle$ be a partition of the state space to observational classes, each consisting of observationally indistinguishable states.

Sets of belief states generated by traversing the belief space backwards starting from the goal states contain many regularities induced by observability. For example, if we have plans for reaching the goals from three belief states B_1, B_2 and B_3, and these have non-empty intersections with the n observational classes, we may construct 3^n different branching plans for 3^n different sets of states. These 3^n sets have a concise representation in a factored form, simply as

$$\langle \{B_1 \cap C_1, B_2 \cap C_1, B_3 \cap C_1\}, \{B_1 \cap C_2, B_2 \cap C_2, B_3 \cap C_2\}, \ldots, \{B_1 \cap C_n, B_2 \cap C_n, B_3 \cap C_n\} \rangle$$

from which the sets can be obtained by taking the Cartesian product and then the union of the n components of each of the 3^n tuples. This motivates the following definitions.

Definition 4.29 (Factored belief space) *Let $\Pi = \langle C_1, \ldots, C_n \rangle$ be a partition of the set of all states. Then a factored belief space is $\langle G_1, \ldots, G_n \rangle$ where $s \subset s'$ for no $\{s, s'\} \subseteq G_i$ and $G_i \subseteq 2^{C_i}$ for all $i \in \{1, \ldots, n\}$.*

Intuitively, a factored belief space is a set of belief states, partitioned to subsets corresponding to the observational classes. This is just a technical definition that makes it easier to talk about the belief states corresponding to the same observational class. Notice the minimality condition: none of the belief states in a factored belief space may be a subset of another. We want to have the minimality condition because we use factored belief spaces as representations of those sets of states for which a plan exists. If a plan exists for some belief state B, then the same plan also works for any belief state B' such that $B' \subseteq B$.

The factored representation of a one-element set S of states is simply $\mathcal{F}(S) = \langle \{C_1 \cap S\}, \ldots, \{C_n \cap S\} \rangle$. When it is obvious from the context, we often write simply S instead of $\mathcal{F}(S)$.

When we have two sets of belief states in the factored form, we may combine them and keep the result in the factored form.

Definition 4.30 (Combination of factored belief spaces) *Let $G = \langle G_1, \ldots, G_n \rangle$ and $H = \langle H_1, \ldots, H_n \rangle$ be factored belief spaces. Define $G \oplus H$ as $\langle G_1 \Cup H_1, \ldots, G_n \Cup H_n \rangle$, where the operation $\Cup$ takes union of two sets of sets and eliminates sets that are not set-inclusion maximal. It is formally defined as $G \Cup H = \{R \in G \cup H | R \subset K \text{ for no } K \in G \cup H\}$.*

Important in this combination operation is that the minimality condition is preserved: any belief state that is a subset of another belief state is eliminated.

The combination operator has the following properties.

Lemma 4.31 (Belief spaces with $\oplus$ are commutative monoids) *The operator $\oplus$ is associative, commutative and its identity element is $\langle \emptyset, \ldots, \emptyset \rangle$.*

A factored belief space $G = \langle G_1, \ldots, G_n \rangle$ can be viewed as representing the set of sets of states flat$(G) = \{s_1 \cup \cdots \cup s_n | s_i \in G_i$ for all $i \in \{1, \ldots, n\}\}$, and its cardinality is $|G_1| \cdot |G_2| \cdot \ldots \cdot |G_n|$. The cardinality may be exponential on the size of the factored representation. Assuming that we have a plan for all belief states in G, we also have a plan for any sets in flat(G). This plan starts by a branch according to an observation C that is made, and then follows the plan for the respective belief state $B \cap C$.

Definition 4.32 (Inclusion relation on belief spaces) *A factored belief space G is included in factored belief space H if for all $S \in$ flat(G) there is $S' \in$ flat(H) such that $S \subseteq S'$. We write this $G \sqsubseteq H$.*

The definitions have the property that $S \in$ flat(G) if and only if $\mathcal{F}(S) \sqsubseteq G$.

We discuss the complexity of certain operations on belief spaces. The basic operations needed in the planning algorithms are testing the membership of a set of states in a factored belief space, and finding a set of states whose preimage with respect to an operator is not contained in the belief space. This last operation is needed in the backup steps of our planning algorithm: find a plan that covers belief states for which we did not have a plan earlier.

Theorem 4.33 *Testing $G \sqsubseteq H$ for factored belief spaces G and H is polynomial time.*

Proof: Testing $\langle G_1, \ldots, G_n \rangle \sqsubseteq \langle H_1, \ldots, H_n \rangle$ is simply by testing whether for all $i \in \{1, \ldots, n\}$ and all $s \in G_i$ there is $t \in H_i$ such that $s \subseteq t$. □

Example 4.34 Consider the blocks world with three blocks with the goal state in which all the blocks are on the table. There are three operators, each of which picks up one block (if there is nothing on top of it) and places it on the table. We can only observe which blocks are not below another block. This splits the state space to seven observational classes, corresponding to the valuations of the state variables clear-A, clear-B and clear-C in which at least one block is clear.

The plan construction steps are given in Figure 4.10. Starting from the top left, the first diagram depicts the goal belief state. The second diagram depicts the belief states obtained by computing the strong preimage of the goal belief state with respect to the move-A-onto-table action and splitting the set of states to belief states corresponding to the observational classes. The next two diagrams are similarly for strong preimages of move-B-onto-table and move-C-onto-table.

The fifth diagram depicts the computation of the strong preimage from the union of two existing belief states in which the block A is on the table and C is on B or B is on C. In the resulting belief state A is the topmost block in a stack containing all three blocks. The next two diagrams similarly construct belief states in which respectively B and C are the topmost blocks.

The last three diagrams depict the most interesting cases, constructing belief states that subsume two existing belief states in one observational class. The first diagram depicts the construction of the belief state consisting of both states in which A and B are clear and C is under either A or B.

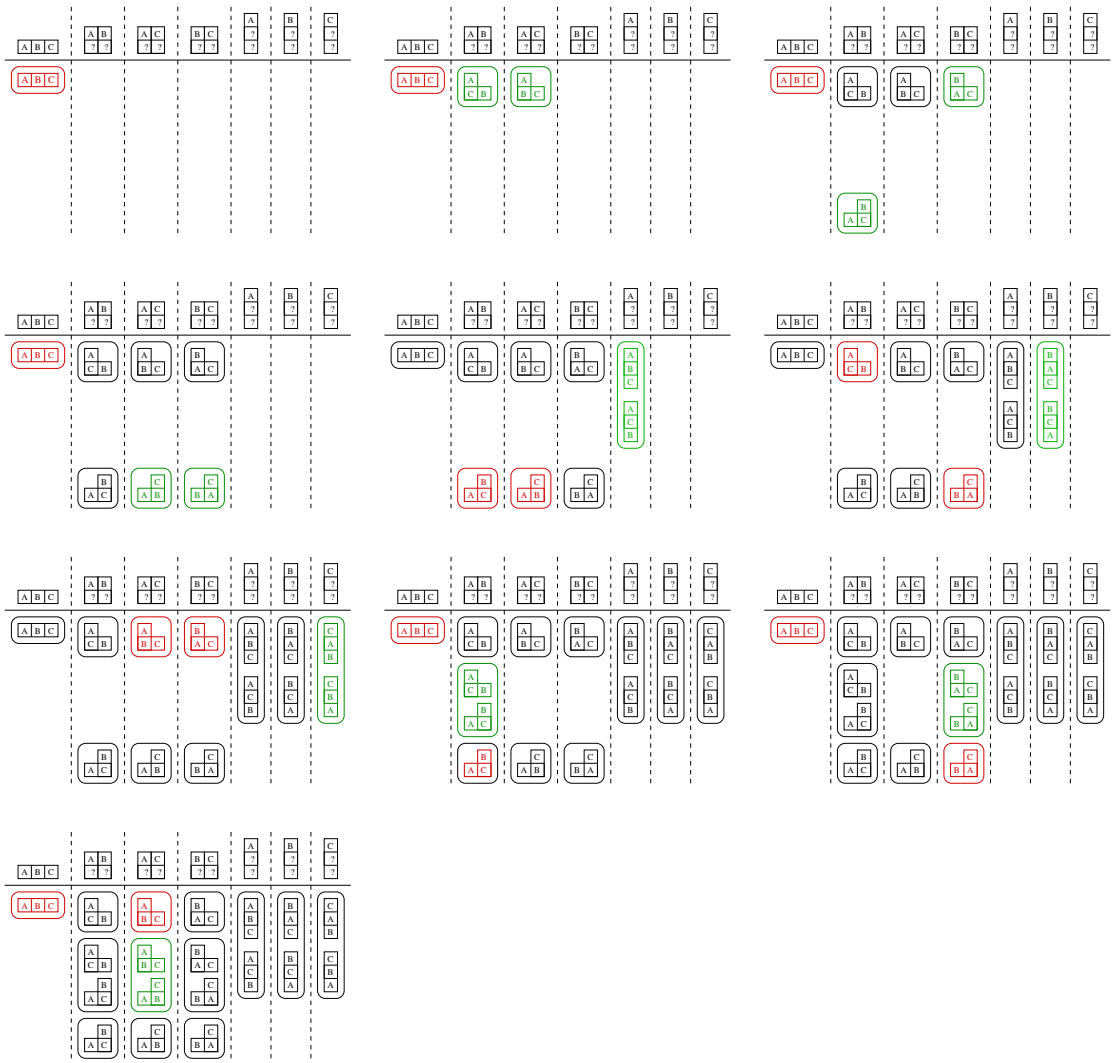


Figure 4.10: Solution of a simple blocks world problem

This belief state is obtained as the strong preimage of the union of two existing belief states, the one in which all blocks are on the table and the one in which A is on the table and B is on top of C. The action that moves A onto the table yields the belief state because if A is on C all blocks will be on the table and if A is already on the table nothing will happen. Construction of the belief states in which B and C are clear and A and C are clear is analogous and depicted in the last two diagrams.

The resulting plan reaches the goal state from any state in the blocks world. The plan in the program form is given in Figure 4.11 (order of construction is from the end to the beginning.)

■

The algorithm we give for extending factored belief spaces by computing the preimage of a combination of some of its belief states is based on exhaustive search and runs in worst-case exponential time. The algorithm is justified by the following theorem that shows that finding new belief states is NP-hard. The proof is a reduction from SAT: represent each clause as the set of

```
16:
  IF clear-A AND clear-B AND clear-C THEN GOTO end
  IF clear-A AND clear-C THEN GOTO 15
  IF clear-B AND clear-C THEN GOTO 13
  IF clear-A AND clear-B THEN GOTO 11
  IF clear-A THEN GOTO 5
  IF clear-B THEN GOTO 7
  IF clear-C THEN GOTO 9
15:
  move-C-onto-table
14:
  IF clear-A AND clear-B AND clear-C THEN GOTO end
  IF clear-A AND clear-C THEN GOTO 1
13:
  move-B-onto-table
12:
  IF clear-A AND clear-B AND clear-C THEN GOTO end
  IF clear-B AND clear-C THEN GOTO 3
11:
  move-A-onto-table
10:
  IF clear-A AND clear-B AND clear-C THEN GOTO end
  IF clear-A AND clear-B THEN GOTO 2
9:
  move-C-onto-table
8:
  IF clear-A AND clear-C THEN GOTO 1
  IF clear-B AND clear-C THEN GOTO 2
7:
  move-B-onto-table
6:
  IF clear-A AND clear-B THEN GOTO 1
  IF clear-B AND clear-C THEN GOTO 3
5:
  move-A-onto-table
4:
  IF clear-A AND clear-B THEN GOTO 2
  IF clear-A AND clear-C THEN GOTO 3
3:
  move-C-onto-table
  GOTO end
2:
  move-B-onto-table
  GOTO end
1:
  move-A-onto-table
end:
```

Figure 4.11: A plan for a partially observable blocks world problem

literals that are not in it, and then a satisfying assignment is a set of literals that is not included in any of the sets.

Theorem 4.35 *Testing whether $G = \langle G_1, \ldots, G_n \rangle$ contains a set S of states such that spreimg$_o(S)$ is not in G is NP-complete. This holds also for deterministic operators o.*

Proof: Membership in NP is trivial: nondeterministically choose $s_i \in G_i$ for every $i \in \{1, \ldots, n\}$, compute the preimage r of $s_1 \cup \cdots \cup s_n$ in deterministic polynomial time, and verify in polynomial time that the intersection $r \cap C_i$ of the preimage with one of the observational classes C_i is not in G_i.

Let $T = \{E_1, \ldots, E_m\}$ be a set of clauses over a set of propositional variables $A = \{a_1, \ldots, a_k\}$. We construct a factored belief space based on a state space in which the variables a and $\widehat{a}$ for $a \in A$ and all these variables with a replaced by z, are the states. The variables $\widehat{z}$ represent negative literals. Define

$$\begin{array}{rcl} E'_i & = & (A \backslash E_i) \cup \{\widehat{a} | a \in A, \neg a \notin E_i\} \text{ for } i \in \{1, \ldots, m\} \\ G & = & \langle \{E'_1, \ldots, E'_m\}, \{\{z_1\}, \{\widehat{z}_1\}\}, \ldots, \{\{z_k\}, \{\widehat{z}_k\}\} \rangle \end{array}$$

Let o map a_i to z_i and $\widehat{a}_i$ to $\widehat{z}_i$ for all $i \in \{1, \ldots, k\}$.

We claim that T is satisfiable if and only if flat(G) contains a belief state B such that *spreimg*$_o(B)$ is not in G.

Assume T is satisfiable, that is, there is M such that $M \models T$. Define $M' = \{z_i | a_i \in A, M \models a_i\} \cup \{\widehat{z}_i | a_i \in A, M \not\models a_i\}$. Clearly, M' is a belief state in G. Define $M'' = \{a_i \in A | M \models a_i\} \cup \{\widehat{a}_i | a_i \in A, M \not\models a_i\}$. Clearly, M'' is the preimage of M' with respect to o.

We show that M'' is not in G. Take any $i \in \{1, \ldots, m\}$. Because $M \models E_i$, there is $a_j \in A$ such that $a_j \in E_i$ and $M \models a_j$ (the case $\neg a \in E_i$ goes similarly.) Now $a_j \in M''$. By definition now $a_j \notin E'_j$. As this holds for all $i \in \{1, \ldots, m\}$, M'' is not a subset of any E_i, and hence it does not belong to G.

Assume there is belief state B in G such that the preimage of B with respect to o is not in G. Clearly, B is a subset of $A \cup \{\widehat{a} | a \in A\}$ with at most one of a_i or $\widehat{a}_i$ for any $i \in \{1, \ldots, k\}$. Define a propositional model M such that $M \models a$ if and only if $a \in B$. We show that $M \models T$. Take any clause E_i from T. As B is not in G, $B \not\subseteq E'_i$. Hence there is a_j or $\widehat{a}_j$ in $B \backslash E'_i$. Consider the case with a_j ($\widehat{a}_j$ goes similarly.) As $a_j \notin E'_i$, $a_j \in E_i$. By definition of M, $M \models a_j$ and hence $M \models E_i$. As this holds for all $i \in \{1, \ldots, m\}$, $M \models T$. This completes the proof. □

Example 4.36 The construction in the above proof can be illustrated by the following example. We use an operator that maps a variable x to the variable x_0. Let $T = \{A \vee B \vee C, \neg A \vee B, \neg C\}$. The corresponding factored belief space is

$$\begin{array}{ll} \langle & \{\{\widehat{A}, \widehat{B}, \widehat{C}\}, \{A, \widehat{B}, C, \widehat{C}\}, \{A, \widehat{A}, B, \widehat{B}, C\}\}, \\ & \{\{A_0\}, \{\widehat{A}_0\}\}, \\ & \{\{B_0\}, \{\widehat{B}_0\}\}, \\ & \{\{C_0\}, \{\widehat{C}_0\}\}\rangle. \end{array}$$

■

procedure findnew(o,A,F,H);
if $F = \langle\rangle$ **and** *spreimg*$_o(A) \not\subseteq S$ for no $S \in$ flat(H) **then return** A;
if $F = \langle\rangle$ **then return** $\emptyset$;
F is $\langle\{f_1, \ldots, f_m\}, F_2, \ldots, F_k\rangle$ for $k \geq 1$;
for i := 1 **to** m **do**
  S := findnew(o,$A \cup f_i$,$\langle F_2, \ldots, F_k\rangle$,$H$);
  if $S \neq \emptyset$ **then return** S;
end;
return $\emptyset$

Figure 4.12: An algorithm for finding new belief states

Next we give an algorithm for constructing conditional plans. The basic step in the algorithm is finding a belief state for which a plan can be shown to exist, based on a set of belief states with plans.

The procedure in Figure 4.12 performs this step: it finds a set S of states that is not contained in H and that is the strong preimage of a set S' of states in F with respect to an operator o. The procedure runs in exponential time on the size of F, and consumes space linear in the size of F. By Theorem 4.35 this is the best that can be expected (unless it turns out that $P = NP$).

Lemma 4.37 *The procedure call findnew(o,$\emptyset$,H, H') returns a set S' such that $S' =$ spreimg$_o(S)$ for some $S \in$ flat(H) and $S' \subseteq S''$ for no $S'' \in$ flat(H'), and if no such set exists it returns $\emptyset$.*

Proof: The procedure goes through the elements $\langle S_1, \ldots, S_n\rangle$ of $F_1 \times \cdots \times F_n$ and tests whether *spreimg*$_o(S_1 \cup \cdots \cup S_n)$ is in H. The sets $S_1 \cup \cdots \cup S_n$ are the elements of flat(F). The traversal through $F_1 \times \cdots \times F_n$ is by generating a search tree with elements of F_1 as children of the root node, elements of F_2 as children of every child of the root node, and testing whether the strong preimage is in it. □

The implementation of the procedure can be improved in many ways. The sets $f_1, \ldots, f_m$ can be ordered according to cardinality so that the bigger preimages are tried out first and a new belief state is found sooner. Also other kinds of heuristics could be applied here, for example ones that would try to produce belief states closer to the initial state for example according to the heuristics discussed in Section 4.4.1.

Define *altimg*$_o(S)$ as *img*$_o$(*wpreimg*$_o(S)$). This is the set of states that could have been reached when a state in S was reached instead. Now $S \subseteq$ *altimg*$_o(S)$, and for deterministic operators $S =$ *altimg*$_o(S)$.

Pruning techniques based o strong and weak preimages of f_i are the following.

1. Let o be deterministic. If *spreimg*$_o(f_i) \subseteq$ *spreimg*$_o(f_j)$ and $i > j$, or *spreimg*$_o(f_i) \subset$ *spreimg*$_o(f_j)$, then we can ignore f_i.

   If the strong preimage of f_i is smaller than that of f_j, the strong preimage that is found with f_i cannot be bigger than that with f_j, and hence using f_i is unnecessary.

2. Pruning techniques for nondeterministic operators are more complicated.

   If *wpreimg*$_o(f_i) \subseteq$ *wpreimg*$_o(f_j)$ and *altimg*$_o(f_i) \cap f_i \subseteq$ *altimg*$_o(f_j)$ and $i > j$, or *wpreimg*$_o(f_i) \subset$ *wpreimg*$_o(f_j)$ and *altimg*$_o(f_i) \cap f_i \subset$ *altimg*$_o(f_j)$ then f_i can be ignored.

```
procedure plan(I,O,G);
H := F(G);
progress := true;
while progress and I ⊈ S for all S ∈ flat(H) do
  progress := false;
  for each o ∈ O do
    S := findnew(o,∅,H,H);
    if S ≠ ∅ then
      begin
        H := H ⊕ F(spreimg_o(S));
        progress := true;
      end;
  end;
end;
if I ⊆ S for some S ∈ flat(H) then return true
else return false;
```

Figure 4.13: A backward search algorithm for partially observable planning

kesken

A more advanced version of this technique can be utilized during search. If sets included in $C_1, \ldots, C_k$ have already been chosen and their union is B, states $s \in f_i$ such that $\mathit{altimg}_o(\{s\}) \cap ((\bigcup_{i\{1,\ldots,k\}} C_i) \backslash B) \neq \emptyset$ do not help in finding a new (bigger) belief state.

kesken

Figure 4.13 shows an algorithm for finding plans for partially observable problems. The algorithm uses the subprocedure *findnew* for extending the belief space (this is the NP-hard subproblem from Theorem 4.35). The plans the algorithm produces are not guaranteed to be optimal because it does not produce all possible plans in a breadth-first manner.

We have not here described the book-keeping needed for outputting a plan, and the algorithm just returns *true* or *false* depending on whether a plan exists or not. Extending the algorithm with the necessary book-keeping is straightforward.

Lemma 4.38 *Assume $S \in \mathit{flat}(H)$. Then there is $S' \in \mathit{flat}(H \oplus G)$ so that $S \subseteq S'$.*

Lemma 4.39 *Let $S_1, \ldots, S_n$ be sets of states so that for every $i \in \{1, \ldots, n\}$ there is $S'_i \in \mathit{flat}(H)$ such that $S_i \subseteq S'_i$, and there is no observational class C such that for some $\{i, j\} \subseteq \{1, \ldots, n\}$ both $i \neq j$ and $S_i \cap C \neq \emptyset$ and $S_j \cap C \neq \emptyset$. Then there is $S' \in \mathit{flat}(H)$ such that $S_1 \cup \cdots \cup S_n \subseteq S'$.*

Theorem 4.40 *Whenever there exists a finite acyclic plan for a problem instance, the algorithm in Figure 4.13 returns* true.

Proof: So assume there is a plan for a problem instance $\langle A, I, O, G, B\rangle$. Label all nodes of the plan as follows. The root node N is labeled with I, that is, $l(N) = I$. When possible parent nodes of a node n are labeled, we can compute the label for n. Let $\langle o_1, n\rangle, \ldots, \langle o_m, n\rangle$ be the annotations of all operator nodes $n_1, \ldots, n_m$ in the plan with n as the child node, and let $\{\langle \phi_1, n\rangle, \ldots\}, \ldots, \{\langle \phi_k, n\rangle, \ldots\}$ the respective annotations of all branch nodes $n'_1, \ldots, n'_k$ with n

as one of the child nodes. Then the label of n is $\mathit{img}_{o_1}(l(n_1)) \cup \cdots \cup \mathit{img}_{o_m}(l(n_m)) \cup (l(n'_1) \cap \phi_1) \cup \cdots \cup (l(n'_k) \cap \phi_k)$. This labeling simply says what are the possible current states for every node of the plan when the plan is executed starting from some initial state.

We show that – *assuming that the algorithm does not terminate earlier after producing a superset of I* – the algorithm determines that for all node labels a plan for reaching G exists if plans exist for its child nodes.

Induction hypothesis: For each plan node n such that all paths to a terminal node have length i or less, its label $S = l(n)$ is a subset of some $S' \in \text{flat}(H)$, where H is the value of the program variable H after the *while* loop exits and H could not be extended further.

Base case $i = 0$: Terminal nodes of the plan are labeled with subsets of G. By Lemma 4.38, $G' \in \text{flat}(H)$ for some set G' such that $G \subseteq G'$ because G was in H initially.

Inductive case $i \geq 1$: Let n be a plan node. By the induction hypothesis for all child nodes n' of n, $l(n') \subseteq S$ for some $S \in \text{flat}(H)$.

If n is a branch node with child nodes $n_1, \ldots, n_k$ and respective conditions $\phi_1, \ldots, \phi_k$, then $l(n) \cap \phi_1$, …, $l(n) \cap \phi_k$ all occupy disjoint observational classes and superset of $l(n) \cap \phi_i$ for every $i \in \{1, \ldots, k\}$ is in $\text{flat}(H)$. Hence by Lemma 4.39 $l(n) \subseteq S$ for some $S \in \text{flat}(H)$.

If n is an operator node with operator o and child node n', then $\mathit{img}_o(l(n)) \subseteq l(n')$, and by the induction hypothesis $l(n') \subseteq S'$ for some $S' \in \text{flat}(H)$. We have to show that $l(n) \subseteq S''$ for some $S'' \in \text{flat}(H)$. Assume that there is no such S''. But now by Lemma 4.37 findnew(o,$\emptyset$,H,H) would return S''' such that $\mathit{spreimg}_o(S''') \subseteq S$ for no $S \in \text{flat}(H)$, and the *while* loop could not have exited with H, contrary to our assumption about H. □

Theorem 4.41 *Let $\Pi = \langle A, I, O, G, B \rangle$ be a problem instance. If procedure plan(I,O,G) in Figure 4.13 returns* true*, then Π has a solution plan.*

Proof: Let $H^0, H^1, \ldots$ be the sequence of factored belief spaces H produced by the algorithm. We show that for all $i \geq 0$, for every set of states in H^i there is a plan that reaches G.

Induction hypothesis: H^i contains only such sets $S \in \text{flat}(H^i)$ for which a plan reaching G exists.

Base case $i = 0$: Initially $H^0 = \mathcal{F}(G)$ and the only set in H^0 is G. The empty plan reaches G from G.

Inductive case $i \geq 1$: H^{i+1} is obtained as $H^i \oplus \mathcal{F}(\mathit{spreimg}_o(S))$ where $S =$findnew(o,$\emptyset$,H^i,H^i). By Lemma 4.37 $S \in \text{flat}(H^i)$ and $\mathit{spreimg}_o(S) \subseteq S'$ for no $S' \in \text{flat}(H^i)$. Because S is in H^i, there is a plan π for reaching G from S. The plan that executes o followed by π reaches G from $\mathit{spreimg}_o(S)$.

Let Z be any member of $\text{flat}(H^{i+1})$. We show that for Z there is a plan for reaching G. The plan for Z starts by a branch[4]. We show that for every possible observation, corresponding to one observational class, there is a plan that reaches G. Let C_j be the jth observational class. When observing C_j, the current state is in $Z_j = Z \cap C_j$. Now for Z_j there is $Z'_j \in H^{i+1}_j$ with $Z_j \subseteq Z'_j$, where H^{i+1}_j is the jth component of H^{i+1}. Now by induction hypothesis there is a plan for Z'_j if $Z'_j \in H^i_j$, and if $Z'_j \in H^{i+1}_j \backslash H^i_j$, then for branch corresponding to C_j we use the plan for $\mathit{spreimg}_o(S)$, as Z'_j must be $\mathit{spreimg}_o(S) \cap C_j$. □

[4]Some of the branches might not be needed, and if the intersection of Z with only one observational class is nonempty the plan could start with an operator node instead of a degenerate branch node.

4.5 Computational complexity

In this section we analyze the computational complexity of the main decision problems related to nondeterministic planning. The conditional planning problem is a generalization of the deterministic planning problem from Chapter 3, and therefore the plan existence problem is at least PSPACE-hard. In this section we discuss the computational complexity of each of the three planning problems, the fully observable, the unobservable, and the general partially observable planning problem, showing them respectively complete for the complexity classes EXP, EXPSPACE and 2-EXP.

4.5.1 Planning with full observability

We first show that the plan existence problem for nondeterministic planning with full observability is EXP-hard and then that the problem is in EXP.

The EXP-hardness proof in Theorem 4.42 is by simulating polynomial-space alternating Turing machines by nondeterministic planning problems with full observability and the using the fact that the complexity classes EXP and APSPACE are the same (see Section 2.4.) The most interesting thing in the proof is the representation of alternation. Theorem 3.42 already showed how deterministic Turing machines with a polynomial space bound are simulated, and the difference is that we now have nondeterminism, that is, a configuration of the TM may have several successor configurations, and that there are both $\forall$ and $\exists$ states.[5]

The $\forall$ states mean that all successor configurations must be accepting (terminal or non-terminal) configurations. The $\exists$ states mean that at least one successor configuration must be an accepting (terminal or non-terminal) configuration. Both of these requirements can be represented in the nondeterministic planning problem.

The transitions from a configuration with a $\forall$ state will correspond to one nondeterministic operator. That all successor configurations must be accepting (terminal or non-terminal) configurations corresponds to requirement in planning that from all successor states of a state a goal state must be reached.

Every transition from a configuration with $\exists$ state will correspond to a deterministic operator, that is, the transition may be chosen, as only one of the successor configurations needs to be accepting.

Theorem 4.42 *The problem of testing the existence of an acyclic plan for problem instances with full observability is EXP-hard.*

Proof: Let $\langle \Sigma, Q, \delta, q_0, g \rangle$ be any alternating Turing machine with a polynomial space bound $p(x)$. Let σ be an input string of length n.

We construct a problem instance in nondeterministic planning with full observability for simulating the Turing machine. The problem instance has a size that is polynomial in the size of the description of the Turing machine and the input string.

The set A of state variables in the problem instance consists of

1. $q \in Q$ for denoting the internal states of the TM,
2. s_i for every symbol $s \in \Sigma \cup \{|, \Box\}$ and tape cell $i \in \{0, \ldots, p(n)\}$, and

[5] Restricting the proof of Theorem 4.42 to $\exists$ states with nondeterministic transitions would yield a proof of the NPSPACE-hardness of deterministic planning, but this is not interesting as PSPACE=NPSPACE.

3. h_i for the positions of the R/W head $i \in \{0, \ldots, p(n)+1\}$.

The unique initial state of the problem instance represents the initial configuration of the TM. The corresponding formula is the conjunction of the following literals.

1. q_0
2. $\neg q$ for all $q \in Q \backslash \{q_0\}$.
3. s_i for all $s \in \Sigma$ and $i \in \{1, \ldots, n\}$ such that ith input symbol is s.
4. $\neg s_i$ for all $s \in \Sigma$ and $i \in \{1, \ldots, n\}$ such that ith input symbol is not s.
5. $\neg s_i$ for all $s \in \Sigma$ and $i \in \{0, n+1, n+2, \ldots, p(n)\}$.
6. $\Box_i$ for all $i \in \{n+1, \ldots, p(n)\}$.
7. $\neg \Box_i$ for all $i \in \{0, \ldots, n\}$.
8. $|_0$
9. $\neg |_i$ for all $n \in \{1, \ldots, p(n)\}$
10. h_1
11. $\neg h_i$ for all $i \in \{0, 2, 3, 4, \ldots, p(n)+1\}$

The goal is the following formula.

$$G = \bigvee \{q \in Q | g(q) = \text{accept}\}$$

Next we define the operators. All the transitions may be nondeterministic, and the important thing is whether the transition is for a $\forall$ state or an $\exists$ state.[6] For a given input symbol and a $\forall$ state, the transition corresponds to one nondeterministic operator, whereas for a given input symbol and an $\exists$ state the transitions corresponds to a set of deterministic operators.

To define the operators, we first define effects corresponding to all possible transitions.

For all $\langle s, q \rangle \in (\Sigma \cup \{|, \Box\}) \times Q$, $i \in \{0, \ldots, p(n)\}$ and $\langle s', q', m \rangle \in (\Sigma \cup \{|\}) \times Q \times \{L, N, R\}$ define the effect $\tau_{s,q,i}(s', q', m)$ as $\alpha \wedge \kappa \wedge \theta$ where the effects α, κ and θ are defined as follows.

The effect α describes what happens to the tape symbol under the R/W head. If $s = s'$ then $\alpha = \top$ as nothing on the tape changes. Otherwise, $\alpha = \neg s_i \wedge s'_i$ to denote that the new symbol in the ith tape cell is s' and not s.

The effect κ describes the change to the internal state of the TM. Again, either the state changes or does not, so $\kappa = \neg q \wedge q'$ if $q \neq q'$ and $\top$ otherwise. We define $\kappa = \neg q$ when $i = p(n)$ and $m = R$ so that when the space bound gets violated, no accepting state can be reached.

The effect θ describes the movement of the R/W head. Either there is movement to the left, no movement, or movement to the right. Hence

$$\theta \quad = \quad \begin{cases} \neg h_i \wedge h_{i-1} & \text{if } m = L \\ \top & \text{if } m = N \\ \neg h_i \wedge h_{i+1} & \text{if } m = R \end{cases}$$

[6] No operators are needed for accepting or rejecting states.

By definition of TMs, movement at the left end of the tape is always to the right. Similarly, we have state variable for R/W head position $p(n)+1$ and moving to that position is possible, but no transitions from that position are possible, as the space bound has been violated.

Now, these effects that represent possible transitions are used in the operators that simulate the ATM. Operators for existential states $q, g(q) = \exists$ and for universal states $q, g(q) = \forall$ differ. Let $\langle s, q\rangle \in (\Sigma \cup \{|, \square\}) \times Q$, $i \in \{0, \ldots, p(n)\}$ and $\delta(s, q) = \{\langle s_1, q_1, m_1\rangle, \ldots, \langle s_k, q_k, m_k\rangle\}$.

If $g(q) = \exists$, then define k deterministic operators

$$\begin{aligned}
o_{s,q,i,1} &= \langle h_i \wedge s_i \wedge q, \tau_{s,q,i}(s_1, q_1, m_1)\rangle \\
o_{s,q,i,2} &= \langle h_i \wedge s_i \wedge q, \tau_{s,q,i}(s_2, q_2, m_2)\rangle \\
&\vdots \\
o_{s,q,i,k} &= \langle h_i \wedge s_i \wedge q, \tau_{s,q,i}(s_k, q_k, m_k)\rangle
\end{aligned}$$

That is, the plan determines which transition is chosen.

If $g(q) = \forall$, then define one nondeterministic operator

$$\begin{aligned}
o_{s,q,i} = \langle h_i \wedge s_i \wedge q, \quad & (\tau_{s,q,i}(s_1, q_1, m_1)| \\
& \tau_{s,q,i}(s_2, q_2, m_2)| \\
& \vdots \\
& \tau_{s,q,i}(s_k, q_k, m_k))\rangle.
\end{aligned}$$

That is, the transition is chosen nondeterministically.

We claim that the problem instance has a plan if and only if the Turing machine accepts without violating the space bound.

If the Turing machine violates the space bound, the state variable $h_{p(n)+1}$ becomes true and an accepting state cannot be reached because no operator will be applicable.

Otherwise, we show inductively that from a computation tree of an accepting ATM we can extract a conditional plan that always reaches a goal state, and vice versa. For obtaining an correspondence between conditional plans and computation trees it is essential that the plans are acyclic.

kesken

So, because all alternating Turing machines with a polynomial space bound can be in polynomial time translated to a nondeterministic planning problem, all decision problems in APSPACE are polynomial time many-one reducible to nondeterministic planning, and the plan existence problem is APSPACE-hard and consequently EXP-hard. □

We can extend Theorem 4.42 to general plans with loops. The problem looping plans cause in the proofs of this theorem is that a Turing machine computation of infinite length is not accepting but the corresponding infinite length zero-probability plan execution is allowed to be a part of plan and would incorrectly count as an accepting Turing machine computation.

To eliminate infinite plan executions we have to modify the Turing machine simulation. This is by counting the length of the plan executions and failing when at least one state or belief state must have been visited more than once. This modification makes infinite loops ineffective, and any plan containing a loop can be translated to a finite non-looping plan by unfolding the loop. In the absence of loops the simulation of alternating Turing machines is faithful.

Theorem 4.43 *The plan existence problem for problem instances with full observability is EXP-hard.*

Proof: This is an easy extension of the proof of Theorem 4.42. If there are n state variables, an acyclic plan exists if and only if a plan with execution length at most 2^n exists, because visiting any state more than once is unnecessary. Plans that rely on loops can be invalidated by counting the number of actions taken and failing when this exceeds 2^n. This counting can be done by having $n + 1$ auxiliary state variables $c_0, \ldots, c_n$ that are initialized to false. Every operator $\langle p, e \rangle$ is extended to $\langle p, e \wedge t \rangle$ where t is an effect that increments the binary number encoded by $c_0, \ldots, c_n$ by one until the most significant bit c_n becomes one. The goal G is replaced by $G \wedge \neg c_n$.

Then a plan exists if and only if an acyclic plan exists if and only if the alternating Turing machine accepts. □

Theorem 4.44 *The problem of testing the existence of a plan for problem instances with full observability is in EXP.*

Proof: The algorithm in Section 4.3.2 runs in exponential time in the size of the problem instance. □

4.5.2 Planning without observability

The plan existence problem of conditional planning with unobservability is more complex than that of conditional planning with full observability.

To show the unobservable problem EXPSPACE-hard by a direct simulation of exponential space Turing machines, the first problem is how to encode the tape of the TM. With polynomial space, as in the PSPACE-hardness and APSPACE-hardness proofs of deterministic planning and conditional planning with full observability, it was possible to represent all the tape cells as the state variables of the planning problem. With an exponential space bound this is not possible any more, as we would need an exponential number of state variables, and the planning problem could not be constructed in polynomial time.

Hence we have to find a more clever way of encoding the working tape. It turns out that we can use the uncertainty about the initial state for this purpose. When an execution of the plan that simulates the Turing machine is started, we randomly choose one of the tape cells to be the *watched* tape cell. This is the only cell of the tape for which the current symbol is represented in the state variables. On all transitions the plan makes, if the watched tape cell changes, the change is reflected in the state variables.

That the plan corresponds to a simulation of the Turing machine it is tested whether the transition the plan makes when the current tape cell is the watched tape cell is the one that assumes the current symbol to be the one that is stored in the state variables. If it is not, the plan is not a valid plan. Because the watched tape cell could be any of the exponential number of tape cells, all the transitions the plan makes are guaranteed to correspond to the contents of the current tape cell of the Turing machine, so if the plan does not simulate the Turing machine, the plan is not guaranteed to reach the goal states.

The proof requires both several initial states and unobservability. Several initial states are needed for selecting the watched tape cell, and unobservability is needed so that the plan can-

not cheat: if the plan can determine what the current tape cell is, it could choose transitions that do not correspond to the Turing machine on all but the watched tape cell. Because of unobservability all the transitions have to correspond to the Turing machine.

Theorem 4.45 *The problem of testing the existence of a plan for problem instances with unobservability is EXPSPACE-hard.*

Proof: Proof is a special case of the proof of Theorem 4.48. We do not have $\forall$ states and restrict to deterministic Turing machines. Nondeterministic Turing machines could be simulated for a NEXPSPACE-hardness proof, but it is already known that EXPSPACE = NEXPSPACE, so this additional generality would not bring anything.

Let $\langle \Sigma, Q, \delta, q_0, g \rangle$ be any deterministic Turing machine with an exponential space bound $e(x)$. Let σ be an input string of length n. We denote the ith symbol of σ by σ_i.

The Turing machine may use space $e(n)$, and for encoding numbers from 0 to $e(n)+1$ corresponding to the tape cells we need $m = \lceil \log_2(e(n)+2) \rceil$ Boolean state variables.

We construct a problem instance in nondeterministic planning without observability for simulating the Turing machine. The problem instance has a size that is polynomial in the size of the description of the Turing machine and the input string.

We cannot have a state variable for every tape cell because the reduction from Turing machines to planning would not be polynomial time. It turns out that it is not necessary to encode the whole contents of the tape in the transition system of the planning problem, and that it suffices to keep track of only one tape cell (which we will call the *watched tape cell*) that is randomly chosen in the beginning of every execution of the plan.

The set A of state variables in the problem instance consists of

1. $q \in Q$ for denoting the internal states of the TM,
2. w_i for $i \in \{0, \ldots, m-1\}$ for the watched tape cell $i \in \{0, \ldots, e(n)\}$,
3. s for every symbol $s \in \Sigma \cup \{|, \Box\}$ for the contents of the watched tape cell,
4. h_i for $i \in \{0, \ldots, m-1\}$ for the position of the R/W head $i \in \{0, \ldots, e(n)+1\}$.

The uncertainty in the initial state is about which tape cell is the watched one. Otherwise the formula encodes the initial configuration of the TM, and it is the conjunction of the following formulae.

1. q_0
2. $\neg q$ for all $q \in Q \backslash \{q_0\}$.
3. Formulae for having the contents of the watched tape cell in state variables $\Sigma \cup \{|, \Box\}$.

$$\begin{array}{rcl} | & \leftrightarrow & (w = 0) \\ \Box & \leftrightarrow & (w > n) \\ s & \leftrightarrow & \bigvee_{i \in \{1,\ldots,n\}, \sigma_i = s} (w = i) \text{ for all } s \in \Sigma \end{array}$$

4. $h = 1$ for the initial position of the R/W head.

So the initial state formula allows any values for state variables w_i and the values of the state variables $s \in \Sigma$ are determined on the basis of the values of w_i. The expressions $w = i$, $w > i$ denote the obvious formulae for testing integer equality and inequality of the numbers encoded by $w_0, w_1, \ldots$. Later we will also use effects $h := h + 1$ and $h := h - 1$ that represent incrementing and decrementing the number encoded by $h_0, h_1, \ldots$.

The goal is the following formula.

$$G = \bigvee \{q \in Q | g(q) = \text{accept}\}$$

To define the operators, we first define effects corresponding to all possible transitions.

For all $\langle s, q\rangle \in (\Sigma \cup \{|, \square\}) \times Q$ and $\langle s', q', m\rangle \in (\Sigma \cup \{|\}) \times Q \times \{L, N, R\}$ define the effect $\tau_{s,q}(s', q', m)$ as $\alpha \wedge \kappa \wedge \theta$ where the effects α, κ and θ are defined as follows.

The effect α describes what happens to the tape symbol under the R/W head. If $s = s'$ then $\alpha = \top$ as nothing on the tape changes. Otherwise, $\alpha = ((h = w) \rhd (\neg s \wedge s'))$ to denote that the new symbol in the watched tape cell is s' and not s.

The effect κ describes the change to the internal state of the TM. Again, either the state changes or does not, so $\kappa = \neg q \wedge q'$ if $q \neq q'$ and $\top$ otherwise. If R/W head movement is to the right we define $\kappa = \neg q \wedge ((h < e(n)) \rhd q')$ if $q \neq q'$ and $(h = e(n)) \rhd \neg q$ otherwise. This prevents reaching an accepting state if the space bound is violated: no further operator applications are possible.

The effect θ describes the movement of the R/W head. Either there is movement to the left, no movement, or movement to the right. Hence

$$\theta \quad = \quad \begin{cases} h := h - 1 & \text{if } m = L \\ \top & \text{if } m = N \\ h := h + 1 & \text{if } m = R \end{cases}$$

By definition of TMs, movement at the left end of the tape is always to the right.

Now, these effects $\tau_{s,q}(s', q', m)$ which represent possible transitions are used in the operators that simulate the DTM. Let $\langle s, q\rangle \in (\Sigma \cup \{|, \square\}) \times Q$ and $\delta(s, q) = \{\langle s', q', m\rangle\}$.

If $g(q) = \exists$, then define the operator

$$o_{s,q} = \langle ((h \neq w) \vee s) \wedge q, \tau_{s,q}(s', q', m)\rangle$$

It is easy to verify that the planning problem simulates the DTM assuming that when operator $o_{s,q}$ is executed the current tape symbol is indeed s. So assume that some $o_{s,q}$ is the first operator that misrepresents the tape contents and that $h = c$ for some tape cell location c. Now there is an execution of the plan so that $w = c$. On this execution the precondition $o_{s,q}$ is not satisfied, and the plan is not executable. Hence a valid plan cannot contain operators that misrepresent the tape contents. □

Theorem 4.46 *The problem of testing the existence of a plan for problem instances with unobservability is in EXPSPACE.*

Proof: Proof is similar to the proof Theorem 3.43 but works at the level of belief states. □

The two theorems together yield the EXPSPACE-completeness of the plan existence problem for conditional planning without observability.

4.5.3 Planning with partial observability

We show that the plan existence problem of the general conditional planning problem with partial observability is 2-EXP-complete. The hardness proof is by a simulation of AEXPSPACE=2-EXP Turing machines. Membership in 2-EXP is obtained directly from the decision procedure discussed earlier: the procedure runs in polynomial time in the size of the enumerated belief space of doubly exponential size.

Showing that the plan existence problem for planning with partial observability is in 2-EXP is straightforward. The easiest way to see this is to view the partially observable planning problem as a nondeterministic fully observable planning problem with belief states viewed as states. An operator maps a belief state to another belief state nondeterministically: compute the image of a belief state with respect to an operator, and choose the subset of its states that correspond to one of the possible observations. Like pointed out in the proof of Theorem 4.44, the algorithms for fully observable planning run in polynomial time in the size of the state space. The state space with the belief states as the states has a doubly exponential size in the size of the problem instance, and hence the algorithm runs in doubly exponential time in the size of the problem instance. This gives us the membership in 2-EXP.

Theorem 4.47 *The plan existence problem for problem instances with partial observability is in 2-EXP.*

The hardness proof is an extension of both the EXP-hardness proof of Theorem 4.42 and of the EXPSPACE-hardness proof of Theorem 4.45. From the first proof we have the simulation of alternating Turing machines, and from the second proof the simulation of Turing machines with an exponentially long tape.

Theorem 4.48 *The problem of testing the existence of an acyclic plan for problem instances with partial observability is 2-EXP-hard.*

Proof: Let $\langle \Sigma, Q, \delta, q_0, g \rangle$ be any alternating Turing machine with an exponential space bound $e(x)$. Let σ be an input string of length n. We denote the ith symbol of σ by σ_i.

The Turing machine may use space $e(n)$, and for encoding numbers from 0 to $e(n)+1$ corresponding to the tape cells we need $m = \lceil \log_2(e(n)+2) \rceil$ Boolean state variables.

We construct a problem instance in nondeterministic planning with full observability for simulating the Turing machine. The problem instance has a size that is polynomial in the size of the description of the Turing machine and the input string.

We cannot have a state variable for every tape cell because the reduction from Turing machines to planning would not be polynomial time. It turns out that it is not necessary to encode the whole contents of the tape in the transition system of the planning problem, and that it suffices to keep track of only one tape cell (which we will call the *watched tape cell*) that is randomly chosen in the beginning of every execution of the plan.

The set A of state variables in the problem instance consists of

1. $q \in Q$ for denoting the internal states of the TM,
2. w_i for $i \in \{0, \ldots, m-1\}$ for the watched tape cell $i \in \{0, \ldots, e(n)\}$,
3. s for every symbol $s \in \Sigma \cup \{|, \Box\}$ for the contents of the watched tape cell,

4. s^* for every $s \in \Sigma \cup \{|\}$ for the symbol last written (important for nondeterministic transitions),

5. L, R and N for the last movement of the R/W head (important for nondeterministic transitions), and

6. h_i for $i \in \{0, \ldots, m-1\}$ for the position of the R/W head $i \in \{0, \ldots, e(n)+1\}$.

The observable state variables are L, N and R, $q \in Q$, and s^* for $s \in \Sigma$. These are needed by the plan to decide how to proceed execution after a nondeterministic transition with a $\forall$ state.

The uncertainty in the initial state is about which tape cell is the watched one. Otherwise the formula encodes the initial configuration of the TM, and it is the conjunction of the following formulae.

1. q_0

2. $\neg q$ for all $q \in Q \backslash \{q_0\}$.

3. $\neg s^*$ for all $s \in \Sigma \cup \{|\}$.

4. Formulae for having the contents of the watched tape cell in state variables $\Sigma \cup \{|, \Box\}$.

$$\begin{array}{rcl} | & \leftrightarrow & (w = 0) \\ \Box & \leftrightarrow & (w > n) \\ s & \leftrightarrow & \bigvee_{i \in \{1,\ldots,n\}, \sigma_i = s} (w = i) \text{ for all } s \in \Sigma \end{array}$$

5. $h = 1$ for the initial position of the R/W head.

So the initial state formula allows any values for state variables w_i and the values of the state variables $s \in \Sigma$ are determined on the basis of the values of w_i. The expressions $w = i$, $w > i$ denote the obvious formulae for testing integer equality and inequality of the numbers encoded by $w_0, w_1, \ldots$. Later we will also use effects $h := h + 1$ and $h := h - 1$ that represent incrementing and decrementing the number encoded by $h_0, h_1, \ldots$.

The goal is the following formula.

$$G = \bigvee \{q \in Q | g(q) = \text{accept}\}$$

Next we define the operators. All the transitions may be nondeterministic, and the important thing is whether the transition is for a $\forall$ state or an $\exists$ state. For a given input symbol and a $\forall$ state, the transition corresponds to one nondeterministic operator, whereas for a given input symbol and an $\exists$ state the transitions corresponds to a set of deterministic operators.

To define the operators, we first define effects corresponding to all possible transitions.

For all $\langle s, q \rangle \in (\Sigma \cup \{|, \Box\}) \times Q$ and $\langle s', q', m \rangle \in (\Sigma \cup \{|\}) \times Q \times \{L, N, R\}$ define the effect $\tau_{s,q}(s', q', m)$ as $\alpha \wedge \kappa \wedge \theta$ where the effects α, κ and θ are defined as follows.

The effect α describes what happens to the tape symbol under the R/W head. If $s = s'$ then $\alpha = \top$ as nothing on the tape changes. Otherwise, $\alpha = ((h = w) \rhd (\neg s \wedge s')) \wedge s'^* \wedge \neg s^*$ to denote that the new symbol in the watched tape cell is s' and not s, and to make it possible for the plan to detect which symbol was written to the tape by the possibly nondeterministic transition.

The effect κ describes the change to the internal state of the TM. Again, either the state changes or does not, so $\kappa = \neg q \wedge q'$ if $q \neq q'$ and $\top$ otherwise. If R/W head movement is to the right we

define $\kappa = \neg q \wedge ((h < e(n)) \rhd q')$ if $q \neq q'$ and $(h = e(n)) \rhd \neg q$ otherwise. This prevents reaching an accepting state if the space bound is violated: no further operator applications are possible.

The effect θ describes the movement of the R/W head. Either there is movement to the left, no movement, or movement to the right. Hence

$$\theta \quad = \quad \begin{cases} (h := h-1) \wedge L \wedge \neg N \wedge \neg R & \text{if } m = L \\ N \wedge \neg L \wedge \neg R & \text{if } m = N \\ (h := h+1) \wedge R \wedge \neg L \wedge \neg N & \text{if } m = R \end{cases}$$

By definition of TMs, movement at the left end of the tape is always to the right.

Now, these effects $\tau_{s,q}(s', q', m)$ which represent possible transitions are used in the operators that simulate the ATM. Operators for existential states $q, g(q) = \exists$ and for universal states $q, g(q) = \forall$ differ. Let $\langle s, q\rangle \in (\Sigma \cup \{|, \square\}) \times Q$ and $\delta(s, q) = \{\langle s_1, q_1, m_1\rangle, \ldots, \langle s_k, q_k, m_k\rangle\}$.

If $g(q) = \exists$, then define k deterministic operators

$$\begin{aligned} o_{s,q,1} &= \langle((h \neq w) \vee s) \wedge q, \tau_{s,q}(s_1, q_1, m_1)\rangle \\ o_{s,q,2} &= \langle((h \neq w) \vee s) \wedge q, \tau_{s,q}(s_2, q_2, m_2)\rangle \\ &\vdots \\ o_{s,q,k} &= \langle((h \neq w) \vee s) \wedge q, \tau_{s,q}(s_k, q_k, m_k)\rangle \end{aligned}$$

That is, the plan determines which transition is chosen.

If $g(q) = \forall$, then define one nondeterministic operator

$$\begin{aligned} o_{s,q} = \langle((h \neq w) \vee s) \wedge q, \quad & (\tau_{s,q}(s_1, q_1, m_1)| \\ & \tau_{s,q}(s_2, q_2, m_2)| \\ & \vdots \\ & \tau_{s,q}(s_k, q_k, m_k)\rangle). \end{aligned}$$

That is, the transition is chosen nondeterministically.

We claim that the problem instance has a plan if and only if the Turing machine accepts without violating the space bound. If the Turing machine violates the space bound, then $h > e(n)$ and an accepting state cannot be reached because no further operator will be applicable.

From an accepting computation tree of an ATM we can construct a plan, and vice versa. Accepting final configurations are mapped to terminal nodes of plans, $\exists$-configurations are mapped to operator nodes in which an operator corresponding to the transition to an accepting successor configuration is applied, and $\forall$-configurations are mapped to operator nodes corresponding to the matching nondeterministic operators followed by a branch node that selects the plan nodes corresponding to the successors of the $\forall$ configuration. The successors of $\forall$ and $\exists$ configurations are recursively mapped to plans.

Construction of computation trees from plans is similar, but involves small technicalities. A plan with DAG form can be turned into a tree by having several copies of the shared subplans. Branches not directly following the nondeterministic operator causing the uncertainty can be moved earlier so that every nondeterministic operator is directly followed by a branch that chooses a successor node for every possible new state, written symbol and last tape movement. With these transformations there is an exact match between plans and computation trees of the ATM, and mapping from plans to ATMs is straightforward like in the opposite direction.

Because alternating Turing machines with an exponential space bound are polynomial time reducible to the nondeterministic planning problem with partial observability, the plan existence problem is AEXPSPACE=2-EXP-hard. □

What remains to be done is the extension of the above theorem to the case with arbitrary (possibly cyclic) plans. For the fully observable case counting the execution length does not pose a problem because we only have to count an exponential number of execution steps, which can be represented by a polynomial number of state variables, but in the partially observable case we need to count a doubly exponential number of execution steps, as the number of belief states to be visited may be doubly exponential. A binary representation of these numbers requires an exponential number of bits, and we cannot use an exponential number of state variables for the purpose, because the reduction to planning would not be polynomial time. However, partial observability together with only a polynomial number of auxiliary state variables can be used to force the plans to count doubly exponentially far.

Theorem 4.49 *The plan existence problem for problem instances with partial observability is 2-EXP-hard.*

Proof: We extend the proof of Theorem 4.48 by a counting scheme that makes cyclic plans ineffective. We show how counting the execution length can be achieved within a problem instance obtained from the alternating Turing machine and the input string in polynomial time.

Instead of representing the exponential number of bits explicitly as state variables, we use a randomizing technique for forcing the plans to count the number of Turing machine transitions. The technique has resemblance to the idea in simulating exponentially long tapes in the proofs of Theorems 4.45 and 4.42.

For a problem instance with n state variables (representing the Turing machine configurations) executions that visit each belief state at most once may have length 2^{2^n}. Representing numbers from 0 to $2^{2^n} - 1$ requires 2^n binary digits. We introduce $n + 1$ new unobservable state variables $d_0, \ldots, d_n$ for representing the index of one of the digits and v_d for the value of that digit, and new state variables $c_0, \ldots, c_n$ through which the plan indicates changes in the counter of Turing machine transitions. There is a set of operators by means of which the plan sets the values of these variables before every transition of the Turing machine is made.

The idea of the construction is the following. Whenever the counter of TM transitions is incremented, one of the 2^n digits in the counter changes from 0 to 1 and all of the less significant digits change from 1 to 0. The plan is forced to communicate the index of the digit that changes from 0 to 1 by the state variables $c_0, \ldots, c_n$. The unobservable state variables $d_0, \ldots, d_n, v_d$ store the index and value of one of the digits (chosen randomly in the beginning of the plan execution), that we call *the watched digit*, and they are used for checking that the reporting of $c_0, \ldots, c_n$ by the plan is truthful. The test for truthful reporting is randomized, but this suffices to invalidate plans that incorrectly report the increments, as a valid plan has to reach the goals on every possible execution. The plan is invalid if reporting is false or when the count can exceed 2^{2^n}. For this reason a plan for the problem instance exists if and only if an acyclic plan exists if and only if the Turing machine accepts the input string.

Next we exactly define how the problem instances defined in the proof of Theorem 4.48 are extended with a counter to prevent unbounded looping.

The initial state description is extended with the conjunct $\neg d_v$ to signify that the watched digit

is initially 0 (all the digits in the counter implicitly represented in the belief state are 0.) The state variables $d_0, \ldots, d_n$ may have any values which means that the watched digit is chosen randomly. The state variables $d_v, d_0, \ldots, d_n$ are all unobservable so that the plan does not know the watched digit (may not depend on it).

There is also a failure flag f that is initially set to false by having $\neg f$ in the initial states formula.

The goal is extended by $\neg f \wedge ((d_0 \wedge \cdots \wedge d_n) \rightarrow \neg d_v)$ to prevent executions that lead to setting f true or that have length $2^{2^{n+1}-1}$ or more. The conjunct $(d_0 \wedge \cdots \wedge d_n) \rightarrow \neg d_v$ is false if the index of the watched digit is $2^{n+1} - 1$ and the digit is true, indicating an execution of length $\geq 2^{2^{n+1}-1}$.

Then we extend the operators simulating the Turing machine transitions, as well as introduce new operators for indicating which digit changes from 0 to 1.

The operators for indicating the changing digit are

$$\begin{array}{ll} \langle \top, c_i \rangle & \text{for all } i \in \{0, \ldots, n\} \\ \langle \top, \neg c_i \rangle & \text{for all } i \in \{0, \ldots, n\} \end{array}$$

The operators for Turing machine transitions are extended with the randomized test that the digit the plan claims to change from 0 to 1 is indeed the one: every operator $\langle p, e \rangle$ defined in the proof of Theorem 4.48 is replaced by $\langle p, e \wedge t \rangle$ where the test t is the conjunction of the following effects.

$$\begin{array}{rcl} ((c = d) \wedge d_v) & \rhd & f \\ (c = d) & \rhd & d_v \\ ((c > d) \wedge \neg d_v) & \rhd & f \\ (c > d) & \rhd & \neg d_v \end{array}$$

Here $c = d$ denotes $(c_0 \leftrightarrow d_0) \wedge \cdots \wedge (c_n \leftrightarrow d_n)$ and $c > d$ encodes the greater-than test for the binary numbers encoded by $c_0, \ldots, c_n$ and $d_0, \ldots, d_n$.

The above effects do the following.

1. When the plan claims that the watched digit changes from 0 to 1 and the value of d_v is 1, fail.
2. When the plan claims that the watched digit changes from 0 to 1, change d_v to 1.
3. When the plan claims that a more significant digit changes from 0 to 1 and the value of d_v is 0, fail.
4. When the plan claims that a more significant digit changes from 0 to 1, set the value of d_v to 0.

That these effects guarantee the invalidity of a plan that relies on unbounded looping is because the failure flag f will be set if the plan lies about the count, or the most significant bit with index $2^{n+1} - 1$ will be set if the count reaches $2^{2^{n+1}-1}$. Attempts of unfair counting are recognized and consequently f is set to true because of the following.

Assume that the binary digit at index i changes from 0 to 1 (and therefore all less significant digits change from 1 to 0) and the plan incorrectly claims that it is the digit j that changes, and this is the first time on that execution that the plan lies (hence the value of d_v is the true value of the watched digit.)

If $j > i$, then i could be the watched digit (and hence $c > d$), and for j to change from 0 to 1 the less significant bit i should be 1, but we would know that it is not because d_v is false. Consequently on this plan execution the failure flag f would be set.

If $j < i$, then j could be the watched digit (and hence $c = d$), and the value of d_v would indicate that the current value of digit j is 1, not 0. Consequently on this plan execution the failure flag f would be set.

So, if the plan does not correctly report the digit that changes from 0 to 1, then the plan is not valid. Hence any valid plan correctly counts the execution length which cannot exceed $2^{2^{n+1}-1}$. □

4.5.4 Polynomial size plans

We showed in Section 3.8 that the plan existence problem of deterministic planning is only NP-complete, in contrast to PSPACE-complete, when a restriction to plans of polynomial length is made. Here we investigate the same question for conditional plans.

Theorem 4.50 *The plan existence problem for conditional planning without observability restricted to polynomial length plans is in Σ_2^p.*

Proof: Let $p(n)$ be any polynomial. We give an NP^{NP} algorithm (Turing machine) that solves the problem. Let the problem instance $\langle A, I, O, G, \emptyset\rangle$ have size n.

First guess a sequence of operators $\sigma = o_0, o_1, \ldots, o_k$ for $k < p(n)$. This is nondeterministic polynomial time computation.

Then use an NP-oracle for testing that σ is a solution. The oracle is a nondeterministic polynomial-time Turing machine that accepts if a plan execution does not lead to a goal state or if the plan is not executable (operator precondition not satisfied). The oracle guesses an initial state and for each nondeterministic operator for each step which nondeterministic choices are made, and then in polynomial time tests whether the execution of the operator sequence leads to a goal state.

1. Guess valuation I' that satisfies I.
2. Guess the results of the nondeterministic choices for every operator in the plan: replace every $p_1e_1|\cdots|p_ne_n$ by a nondeterministically selected e_i.
3. Compute $s_j = \mathit{app}_{o_j}(\mathit{app}_{o_{j-1}}(\cdots \mathit{app}_{o_2}(\mathit{app}_{o_1}(I'))))$ for $j = 0, j = 1, j = 2, \ldots, j = k$.
4. If $s_j \not\models c_j$ for $o_j = \langle c_j, e_j\rangle$, accept.
5. If $s_k \not\models G$, accept.
6. Otherwise reject.

□

Theorem 4.51 *The plan existence problem for conditional planning without observability restricted to polynomial length plans is Σ_2^p-hard.*

Proof: Truth of QBF of the form $\exists x_1 \cdots x_n \forall y_1 \cdots y_m \phi$ is Σ_2^p-complete. We reduce this problem to the plan existence problem of unobservable planning with polynomial length plans.

- $A = \{x_1, \ldots, x_n, y_1, \ldots, y_m, s, g\}$

	deterministic context-independent	deterministic context-dependent	non-deterministic context-dependent
full observability	PSPACE	PSPACE	EXPTIME
no observability	PSPACE	EXPSPACE	EXPSPACE
partial observability	PSPACE	EXPSPACE	2-EXPTIME

Table 4.2: Computational complexity of plan existence problems

	deterministic context-independent	deterministic context-dependent	non-deterministic context-dependent
full observability	PSPACE	PSPACE	EXPTIME
no observability	PSPACE	PSPACE	EXPSPACE
partial observability	PSPACE	PSPACE	2-EXPTIME

Table 4.3: Computational complexity of plan existence problems with one initial state

- $I = \neg x_1 \wedge \cdots \wedge \neg x_n \wedge \neg g \wedge s$
- $O = \{\langle s, x_1\rangle, \langle s, x_2\rangle, \ldots, \langle s, x_n\rangle, \langle s, \neg s \wedge (\phi \rhd g)\rangle\}$
- $G = g$

Out claim is that there is a plan if and only if $\exists x_1 \cdots x_n \forall y_1 \cdots y_m \phi$ is true.

Assume the QBF is true, that is, there is a valuation x for $x_1, \ldots, x_n$ so that $x, y \models \phi$ for any valuation y of $y_1, \ldots, y_m$. Let $X = \{\langle s, x_i\rangle | i \in \{1, \ldots, n\}, x(x_i) = 1\}$. Now the operators X in any order followed by $\langle s, \neg s \wedge (\phi \rhd g)\rangle$ is a plan: whatever values $y_1, \ldots, y_m$ have, ϕ is true after executing the operators X, and hence the last operator makes $G = g$ true.

Assume there is a plan. The plan has one occurrence of $\langle s, \neg s \wedge (\phi \rhd g)\rangle$ and it must be the last operator. Define the valuation x of $x_1, \ldots, x_n$ as follows. Let $x(x_i) = 1$ iff $\langle s, x_i\rangle$ is one of the operators in the plan, for all $i \in \{1, \ldots, n\}$. Because g is reached, $x, y \models \phi$ for any valuation y of $y_1, \ldots, y_m$, and the QBF is therefore true. □

4.5.5 Summary of the results

The complexities of the plan existence problem under different restrictions on operators and observability are summarized in Tables 4.2 (with an arbitrary number of initial states) and 4.3 (with one initial state). The different columns list the complexities with different restrictions on the operators. In the previous sections we have considered the general problems with arbitrary operators containing conditional effects and nondeterministic choice. These results are summarized in the third column. The second column lists the complexities in the case without nondeterminism (choice |), and the first column without nondeterminism (choice |) and without conditional effects ($\rhd$). These results are not given in this lecture.

4.6 Literature

There is a difficult trade-off between the two extreme approaches, producing a conditional plan covering all situations that might be encountered, and planning only one action ahead. Schoppers

[1987] proposed *universal plans* as a solution to the high complexity of planning. Ginsberg [1989] attacked Schopper's idea. Schopper's proposal was to have memoryless plans that map any given observations to an action. He argued that plans have to be memoryless in order to be able to react to all the unforeseeable situations that might be encountered during plan execution. Ginsberg argued that plans that are able to react to all possible situations are necessarily much too big to be practical. It seems to us that Schopper's insistence on using plans without a memory is not realistic nor necessary, and that most of Ginsberg argumentation on impracticality of universal plans relies on the lack of any memory in the plan execution mechanism. Of course, we agree that a conditional plan that can be executed efficiently can be much bigger than a plan or a planner that has no restrictions on the amount of time consumed in deciding about the action to be taken. Plans without such restrictions could have as high expressivity as Turing machines, for example, and then a conditional plan does not have to be less succinct than the description of a general purpose planning algorithm.

There is some early work on conditional planning that mostly restricts to the fully observable case and is based on partial-order planning [Etzioni *et al.*, 1992; Peot and Smith, 1992; Pryor and Collins, 1996]. We have not discussed these algorithms because they have only been shown to solve very small problem instances.

A variants of the algorithm for constructing plans for nondeterministic planning with full observability in Section 4.3.1 was first presented by Cimatti et al. [2003]. The algorithms by Cimatti et al. construct mappings of states to actions whereas our presentation in Section 4.3 focuses on the computation of distances of states, and plans are synthesized afterwards on the basis of the distances. We believe that our algorithms are conceptually simpler. Cimatti et al. also presented an algorithm for finding *weak plans* that may reach the goals but are not guaranteed to. However, finding weak plans is polynomially equivalent to the deterministic planning problem of Chapter 3 by an easy reduction that replaces each nondeterministic operator by a set of deterministic operators.

The nondeterministic planning problem with unobservability is not very interesting because all robots and intelligent beings can sense their environment in at least some extent. However, there are problems (outside AI) that are equivalent to the unobservable planning problem. Finding homing/reset/synchronization sequences of circuits/automata is an example of such a problem [Pixley *et al.*, 1992]. There are extensions of the distance and cardinality based heuristics for planning without observability not discussed in this lecture [Rintanen, 2004].

Bertoli et al. have presented a forward search algorithm for finding conditional plans in the general partially observable case [Bertoli *et al.*, 2001].

The computational complexity of conditional planning was first investigated by Littman [1997] and Haslum and Jonsson [2000]. They presented proofs for the EXPTIME-completeness of planning with full observability and the EXPSPACE-completeness of planning without observability. The hardness parts of the proofs were reductions respectively from the existence problem of winning strategies for the game G_4 [Stockmeyer and Chandra, 1979] and from the universality problem of regular expressions with exponentiation [Hopcroft and Ullman, 1979]. In this chapter we gave more direct hardness proofs by direct simulation of alternating polynomial space (exponential time) and exponential space Turing machines.

Chapter 5

Probabilistic planning

Probabilistic planning is an extension of nondeterministic planning with exact information on the probabilities of nondeterministic events.

Exact probabilities are important because it is not just important to get things done, but to get them done efficiently, and for goals for which there is no guarantee that they are reached, it is important to reach them as likely as possible.

The introduction of probabilities complicates planning, both conceptually and computationally. Whereas in the non-probabilistic of conditional planning with partial observability it is sufficient to work in a finite discrete belief space, the introduction of probabilities makes the belief space continuous and thereby infinite. This means that there are no algorithms for doing planning, that is, there is no program that either delivers a plan (with a given property) or announces that no plans exist.

However, despite these difficulties one is forced to face, probabilities are important in many types of applications, and algorithms for probabilistic planning are therefore worth studying.

In this section we discuss a number of algorithms for probabilistic planning, starting from algorithms for the conditional planning problem with full observability. The use of probabilities allows to consider more general plan quality criteria than those that were considered in connection with non-probabilistic planning problems. A main difference is that there is no necessity to restrict to planning with the objective of reaching one of designated goal states. Instead, actions and states are associated with rewards/costs, and the objective is to maximize the rewards (or minimize costs) over the execution of a plan. This kind of problems naturally generalize to plan executions of infinite length.

5.1 Stochastic transition systems with rewards

In Section 2.1 we gave a basic definition of deterministic and nondeterministic transition systems. For expressing exact transition probabilities we need a new definition of transition systems.

A stochastic transition system consists of a finite set S of states. The actions do not just associate a set of possible successor states to each state, but a probability distribution on the set of possible successor states.

An action is a partial function from S to probability distributions over S. Partiality means that not all actions are applicable in all states. A probability distribution p is a function that maps S to real numbers $r \in [0, 1]$ so that $\sum_{s \in S} p(s) = 1.0$. The probability distribution indicates how likely

each state is as a successor state of a given state.

In many types of probabilistic planning problems considered in the literature the objective is not to reach one of a set of designated goal states. Instead, the objective is to act in a way that maximizes the *rewards* or minimizes the *costs*. Planning problems with a designated set of goal states can be expressed in terms of rewards, but not vice versa.

Definition 5.1 *A* stochastic transition system with rewards *is a 4-tuple* $\langle S, A, p, R, \rangle$ *where*

- S *is a finite set of states,*
- A *is a finite set of actions,*
- p *is a partial function that maps each state* $s \in S$ *and action* $a \in A$ *to a probability distribution on* S*, and*
- $R : S \times A \to \mathcal{R}$ *is a reward function which maps each state* $s \in S$ *and action* $a \in A$ *to real number.*

A major difference to the definition of Markov decision processes [Puterman, 1994] is that p is a partial function, that is, not all states are assigned a probability distribution. This is for having a match between the definition of operators in AI planning, where not all actions are applicable in all states. Below, we will denote the set of actions applicable in a state $s \in S$ by $A(S)$. We also require that $A(s)$ is non-empty for every $s \in S$.

Notice that we have not defined initial states or a probability distribution on possible initial states: the most important algorithms find plans that reach the goals from any initial state. Clearly, when the number of states is very high and the sets of initial states are small, more efficient planning could be obtained by taking information about the set of initial states into account.

Stochastic transition systems can be described in terms of state variables and operators just like the transition systems earlier discussed in this lecture. A nondeterministic operator $\langle c, e \rangle$, as given in Definition 4.1, assigns a probability distribution corresponding to e, as given in Definition 4.2, to any state s such that $s \models c$.

5.2 Problem definition

A given plan produces infinite sequences of rewards $r_1, r_2, \ldots$. Clearly, if the planning problem has several initial states or if the actions are nondeterministic this sequence of rewards is not unique. In either case, possible plans are assessed in terms of these rewards, and there are several possibilities how good plans are defined. Because the sequences are infinite, we in general cannot simply take their sum and compare them. Instead, several other possibilities have been considered.

1. Expected total rewards over a finite horizon.

   This is a natural alternative that allows using the normal arithmetic sum of the rewards. However, there is typically no natural bound on the horizon length.

2. Expected average rewards over an infinite horizon.

   This is probably the most natural way of assessing plans. However, there are several technical complications that make average rewards difficult to use.

3. Expected discounted rewards over an infinite horizon.

   This is the most often used criterion in connection with Markov decision processes. Discounting means multiplying the ith reward by λ^{i-1} and it means that early rewards are much more important than rewards obtained much later. The discount constant λ has a value strictly between 0.0 and 1.0. The sum of the geometrically discounted rewards is finite. Like with choosing the horizon length when evaluating plans with respect to their behavior within a finite horizon, it is often difficult to say why a certain discount constant λ is used.

For the latter two infinite horizon problems there always is an optimal plan that is a mapping from states to actions, and this is the type of plan used in most of this section.

In the first case with a bounded horizon the optimal plans cannot be represented as mappings from state to actions *if it really is the case that the length of the plan execution indeed equals the horizon length,* and instead mappings from states and time points to actions are needed. This is because for example at the last stage all rewards that are obtained are from the last action. The optimal plans are therefore time-dependent. However, nothing prevents using the first stage of the finite horizon plan as a normal plan, that is, as a mapping from states to actions.

We state the probabilistic conditional planning problem in the general form. Like with non-probabilistic conditional planning, observability restrictions are expressed in terms of a set of state variables that are observable.

Definition 5.2 *A 5-tuple $\langle A, I, O, B, R\rangle$ consisting of a set A of state variables, a probability distribution I over valuations of A, a set O of operators, a reward function R, and a set $B \subseteq A$ of state variables is* a problem instance in probabilistic nondeterministic planning.

I is a set $\{\langle\phi_1, p_1\rangle, \langle\phi_2, p_2\rangle, \ldots, \langle\phi_n, p_n\rangle\}$ that expresses a probability distribution over valuations of A. We require that $\phi_i \models \neg\phi_j$ for every $\{i, j\} \subseteq \{1, \ldots, n\}$.

$R(o)$ for every $o \in O$ is a set $\{\langle\phi_1, r_1\rangle, \langle\phi_2, r_2\rangle, \ldots, \langle\phi_m, r_m\rangle\}$ that expresses the rewards obtained when o is applied: if o is applied in s and $s \models \phi_k$, then reward is r_k. We require that $\phi_i \models \neg\phi_j$ for every $\{i, j\} \subseteq \{1, \ldots, m\}$.

Definition 5.3 *A* plan *for a problem instance is a function $\pi : S \rightarrow A$ that assigns each state an action.*

A plan is executed in the obvious way: when the current state is $s \in S$, then execute $\pi(s)$ to reach a new current state, and so on. Plan execution does not terminate.

5.3 Algorithms for finding finite horizon plans

Conceptually the simplest probabilistic planning is when plan executions are restricted to have a finite horizon of length N. We briefly describe this problem to illustrate the techniques that are used in connection with the infinite horizon planning problems.

The optimum values $v_i(s)$ that can be obtained in state $s \in S$ at time point $i \in \{1, \ldots, N\}$ fulfill the following equations.

$$v_N(s) \quad = \quad \max_{a \in A(s)} R(s, a)$$

$$v_i(s) = \max_{a \in A(s)} \left(R(s,a) + \sum_{s' \in S} p(s'|s,a) v_{i+1}(s') \right), \text{ for } i \in \{1, \ldots, N-1\}$$

The value at the last stage N is simply the best immediate reward that can be obtained, and values of states for the other stages are obtained in terms of the values of states for the later stages.

These equations also directly yield an algorithm for computing the optimal values and optimal plans: first compute v_N, then v_{N-1}, v_{N-2} and so on, until v_1 is obtained. The action to be taken in state $s \in S$ at time point i is $\pi(s,i)$ defined by

$$\begin{aligned} \pi(s,N) &= \arg\max_{a \in A(s)} R(s,a) \\ \pi(s,i) &= \arg\max_{a \in A(s)} \left(R(s,a) + \sum_{s' \in S} p(s'|s,a) v_{i+1}(s') \right), \text{ for } i \in \{1, \ldots, N-1\} \end{aligned}$$

5.4 Algorithms for finding plans under discounted rewards

The value $v(s)$ of a state $s \in S$ is the discounted sum of the expected rewards that can be obtained by choosing the best possible action in s and assuming that the best possible actions are also chosen in all the possible successor states. The following equations, one for each state $s \in S$, characterize the relations between the values of states of a stochastic transition system under an optimal plan and geometrically discounted rewards with discount constant λ.

$$v(s) = \max_{a \in A(s)} \left(R(s,a) + \sum_{s' \in S} \lambda p(s'|s,a) v(s') \right) \tag{5.1}$$

These equations are called the optimality equations or the Bellman equations, and they are the basis of the most important algorithms for finding optimal plans for probabilistic planning problems with full observability.

5.4.1 Evaluating the value of a given plan

Given a plan π its value under discounted rewards with discount constant λ satisfies the following equation for every $s \in S$.

$$v(s) = R(s, \pi(s)) + \sum_{s' \in S} \lambda p(s'|s, \pi(s)) v(s') \tag{5.2}$$

This yields a system of linear equation with $|S|$ equations and unknowns. The solution of these equations yields the value of the plan in each state.

5.4.2 Value iteration

The value iteration algorithm finds an approximation of the value of the optimal λ-discounted plan within a constant ϵ, and a plan with at least this value.

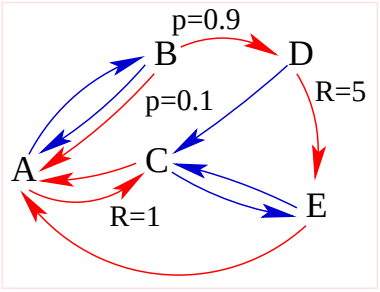


Figure 5.1: A stochastic transition system

1. $n := 0$
2. Assign (arbitrary) initial values to $v^0(s)$ for all $s \in S$.
3. For each $s \in S$, assign

$$v^{n+1}(s) := \max_{a \in A(s)} \left(R(s,a) + \sum_{s' \in S} \lambda p(s'|s,a) v^n(s') \right)$$

   If $|v^{n+1}(s) - v^n(s)| < \frac{\epsilon(1-\lambda)}{2\lambda}$ for all $s \in S$ then go to step 4.

   Otherwise, set $n := n + 1$ and go to step 3.

4. Assign

$$\pi(s) := \arg\max_{a \in A(s)} \left(R(s,a) + \sum_{s' \in S} \lambda p(s'|s,a) v^{n+1}(s') \right)$$

Theorem 5.4 *Let v_π be the value function of the plan produced by the value iteration algorithm, and let v^* be the value function of an optimal plan. Then $|v^*(s) - v_\pi(s)| \leq \epsilon$ for all $s \in S$.*

Notice that unlike in partially observable planning problems, under full observability there is never a trade-off between the values of two states: if the optimal value for state s_1 is r_1 and the optimal value for state s_2 is r_2, then there is one plan that achieves these both.

Example 5.5 Consider the stochastic transition system in Figure 5.1. Only one of the actions is nondeterministic and only in sate B, and all the other actions and states have zero reward except one of the actions in states A and D, with rewards 1 and 5, respectively. ■

5.4.3 Policy iteration

The second, also rather widely used algorithm for finding plans, is policy iteration[1]. It is slightly more complicated to implement than value iteration, but it typically converges after a smaller number of iterations, and it is guaranteed to produce an optimal plan.

The idea is to start with an arbitrary plan (assignment of actions to states), compute its value, and repeatedly choose for every state an action that is better than its old action.

[1] In connection with Markov decision processes the word *policy* is typically used instead of the word *plan*.

1. Assign $n := 0$.
2. Let π^0 be any mapping from states to actions.
3. Compute the values $v^n(s)$ of all $s \in S$ under π^n.
4. Let $\pi^{n+1}(s) = \arg\max_{a \in A(s)} \left(R(s,a) + \sum_{s' \in S} \lambda p(s'|s,a) v^n(s')\right)$.
5. Assign $n := n + 1$.
6. If $n = 1$ or $v^n \neq v^{n-1}$ then go to 3.

Theorem 5.6 *The policy iteration algorithm terminates after a finite number of steps and returns an optimal plan.*

Proof: Outline: There is only a finite number of different plans, and at each step the new plan assigns at least as high a value to each state as the old plan. □

It can be shown that the convergence rate of policy iteration is always at least as fast as that of value iteration [Puterman, 1994], that is, the number of iterations needed for finding an ϵ-optimal plan for policy iteration is never higher than the number of iterations needed by value iteration.

In practise policy iteration often finds an optimal plan after just a few iterations. However, the amount of computation in one round of policy iteration is substantially higher than in value iteration, and value iteration is often considered more practical.

5.4.4 Implementation of the algorithms with ADDs

Similar to the techniques in Section 3.7 for deterministic planning with binary decision diagrams, also probabilistic planning algorithms can be implemented with data structures that allow the representation of much bigger states spaces than what is possible by enumerative representations.

A main difference to the non-probabilistic case (Sections 3.7, 4.3 is that for probabilistic planning propositional formulae and binary decision diagrams are not suitable for representing the probabilities of nondeterministic operators nor the probabilities of the value functions needed in the value and policy iteration algorithms. However, instead of BDDs, we can use algebraic decision diagrams (Section 2.2.3).

In Section 4.1.2 we showed how the incidence matrices expressing the transition probabilities of nondeterministic operators can be represented as BDDs, when the exact probabilities can be ignored, and it is only necessary to know whether a certain nondeterministic event is possible or not.

Next we define a similar translation from nondeterministic operators to ADDs that does represent the exact probabilities.

Now we give the translation of an effect e restricted to state variables B. This means that only state variables in B may occur in e in atomic effects (but do not have to), and the formula does not say anything about the change of state variables not in B (but may of course refer to them in antecedents of conditionals.)

The last two cases, handling nondeterministic choice and conjunction of possibly nondeterministic effects is with ADD operations of multiplying an ADD with a constant, summing ADDs, and multiplying ADDs.

$$
\begin{array}{rcl}
\mathrm{PL}_B(e) & = & \bigwedge(\{((a \wedge \neg \mathit{EPC}_{\neg a}(e)) \vee \mathit{EPC}_a(e)) \leftrightarrow a' | a \in B\} \\
 & & \text{when } e \text{ is deterministic} \\
\mathrm{PL}_B(p_1 e_1 | \cdots | p_n e_n) & = & p_1 \cdot \mathrm{PL}_B(e_1) + \cdots + p_n \cdot \mathrm{PL}_B(e_n) \\
\mathrm{PL}_B(e_1 \wedge \cdots \wedge e_n) & = & \mathrm{PL}_{B \setminus (B_2 \cup \cdots \cup B_n)}(e_1) \cdot \mathrm{PL}_{B_2}(e_2) \cdot \ldots \cdot \mathrm{PL}_{B_n}(e_n) \\
 & & \text{where } B_i = \mathit{changes}(e_i) \text{ for all } i \in \{1, \ldots, n\}
\end{array}
$$

The first part of the translation $\mathrm{PL}_B(e)$ for deterministic e is the translation of deterministic effects we presented in Section 3.5.2, but restricted to state variables in B. The result of this translation is a normal propositional formula, which can be further transformed to a BDD and an ADD with only two terminal nodes 0 and 1. The other two cases cover all nondeterministic effects in normal form.

The translation of an effect e in normal form into an ADD is $\mathrm{PL}_A(e)$ where A is the set of all state variables. Translating an operators $\langle c, e \rangle$ to an ADD representing its incidence matrix is as $c \cdot \mathrm{PL}_A(e)$, where c is the ADD representing the precondition.

Example 5.7 Consider effect $(0.2\neg A | 0.8A) \wedge (0.5(B \rhd \neg B) | 0.5\top)$. The two conjunct translated to functions

AA'	f_A
00	0.2
01	0.8
10	0.2
11	0.8

BB'	f_B
00	1.0
01	0.0
10	0.5
11	0.5

Notice that the sum of the probabilities of the successor states is 1.0. These functions are below depicted in the same table. Notice that the third column, with the two functions componentwise multiplied, has the property that the sum of successor states of each state is 1.0.

$ABA'B'$	f_A	f_B	$f_A \cdot f_B$
0000	0.2	1.0	0.2
0001	0.2	0.0	0.0
0010	0.8	1.0	0.8
0011	0.8	0.0	0.0
0100	0.2	0.5	0.1
0101	0.2	0.5	0.1
0110	0.8	0.5	0.4
0111	0.8	0.5	0.4
1000	0.2	1.0	0.2
1001	0.2	0.0	0.0
1010	0.8	1.0	0.8
1011	0.8	0.0	0.0
1100	0.2	0.5	0.1
1101	0.2	0.5	0.1
1110	0.8	0.5	0.4
1111	0.8	0.5	0.4

■

We represent the rewards produced by operator $o = \langle c, e\rangle \in O$ in different states compactly as a list $R(o) = \{\langle\phi_1, r_1\rangle, \ldots, \langle\phi_n, r_n\rangle\}$ of pairs $\langle\phi, r\rangle$, meaning that when o is applied in a state satisfying ϕ the reward r is obtained. In any state only one of the formulae ϕ_i may be true, that is $\phi_i \models \neg\phi_j$ for all $\{i, j\} \subseteq \{1, \ldots, n\}$ such that $i \neq j$. If none of the formula is true in a given state, then the reward is zero. Hence R_o is simply a mapping from states to a real numbers.

The reward functions $R(o)$ can be easily translated to ADDs. First construct the BDDs for $\phi_1, \ldots, \phi_n$ and then multiply them with the respective rewards as

$$R_o = r_1 \cdot \phi_1 + \cdots + r_n \cdot \phi_n - \infty \cdot \neg c.$$

The summand $\infty \cdot \neg c$ handles the case in which the precondition of the operator is not satisfied: application yields immediate reward minus infinity. This prevent using the operator in any state.

Similarly, the probability distribution on possible initial states can be represented as $I = \{\langle\phi_1, p_1\rangle, \ldots, \langle\phi_n, p_n\rangle\}$ and translated to an ADD.

Now the value iteration algorithm can be rephrased in terms of ADD operations as follows.

1. Assign $n := 0$ and let v^n be an ADD that is constant 0.
2. $$v^{n+1} := \max_{\langle c,e\rangle = o \in O} \left(R_o + \lambda \cdot \exists A'.(T_o \cdot (v^n[A'/A])\right) \text{ for every } s \in S$$

   If all terminal nodes of ADD $|v^{n+1} - v^n|$ are $< \frac{\epsilon(1-\lambda)}{2\lambda}$ then stop.

   Otherwise, set $n := n + 1$ and repeat step 2.

5.5 Probabilistic planning with partial observability

5.5.1 Problem definition

5.5.2 Value iteration

Value of a plan in a state

Let $\langle C_1, \ldots, C_n\rangle$ be the partition of the state space S to the observational classes. Here $n \geq 1$.

The value of finite plans π for a state $s \in S$ is defined recursively as follows. Here $()$ is the empty plan.

$$\begin{aligned} v_{(),s} &= 0 \\ v_{(a,\pi_1,\ldots,\pi_n),s} &= \begin{cases} -\infty \text{ if action } a \text{ is not applicable in } s \\ R(s,a) + \lambda(\sum_{s' \in C_1} p(s'|s,a) v_{\pi_1,s'} + \cdots + \sum_{s' \in C_n} p(s'|s,a) v_{\pi_n,s'}) \end{cases} \end{aligned}$$

Given a belief state B and the values $v_{\pi,s_1}, \ldots, v_{\pi,s_m}$ of a plan π for all states $S \in \{s_1, \ldots, s_m\}$, the value of π for B is simply $\sum_{s \in S} v_{\pi,s} B(s)$.

Eliminating dominated plans

The test whether plan π is for at least one belief state strictly better than any other plan in $\Pi = \{\pi_1, \ldots, \pi_n\}$ can be performed by linear programming.

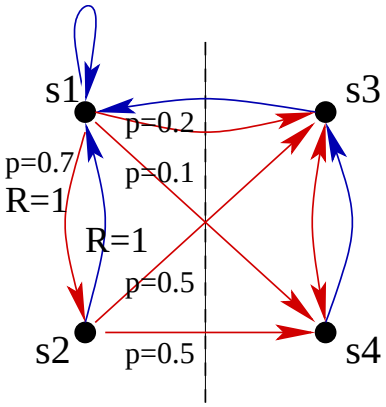


Figure 5.2: Stochastic transition system with two observational classes $\{s_1, s_2\}$ and $\{s_3, s_4\}$

The variables in the LP are d and p_s for every $s \in S$, and the expression to be maximized is the value of d. The constants $v_{\pi,s}$ are values of plans π in states $s \in S$.

$$\begin{array}{rcl} \sum_{s\in S} p_s v_{\pi,s} & \geq & \sum_{s\in S} p_s v_{\pi',s} + d \text{ for all } \pi' \in \Pi\backslash\{\pi\} \\ \sum_{s\in S} p_s & = & 1 \\ p_s & \geq & 0 \text{ for all } s \in S \end{array}$$

The number of equations in the LP is $|\Pi| + |S|$ and the number of unknowns is $|S| + 1$. If the maximum value of d is > 0, then there is a belief state in which the value of π is higher than the value of any other plan. This belief state is expressed by the values of the variables $p_s, s \in S$.

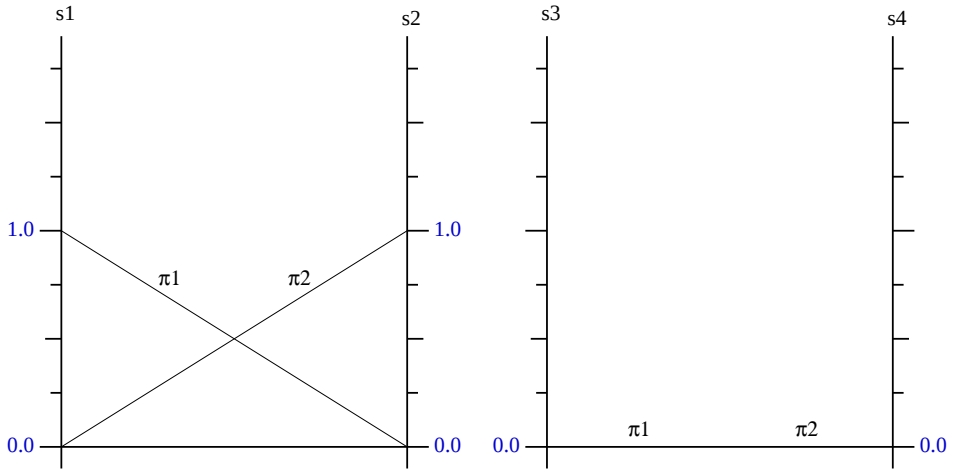
The main procedure of the algorithm

1. $i := 0$
2. $\Pi_0 := \{()\}$
3. $i := i + 1$
4. $\Pi_i := \{(a, \pi_1, \ldots, \pi_n) | a \in A, \{\pi_1, \ldots, \pi_n\} \subseteq \Pi_{i-1}\}$
5. Evaluate the values of plans in Π_i in all states.
6. As long as there is $\pi \in \Pi_i$ that is dominated by $\Pi_i\backslash\{\pi\}$, set $\Pi_i := \Pi_i\backslash\{\pi\}$.
7. If the difference between value functions represented by Π_i and Π_{i-1} is $> \epsilon$ for some belief state, go to 3.

Example 5.8 Consider the Now we run the value iteration algorithm for partially observable probabilistic planning problems. We use the discounting constant $\lambda = 0.5$.

Plans of depth 1 with the corresponding value vectors for all states $S = \{s_1, s_2, s_3, s_4\}$ are the following.

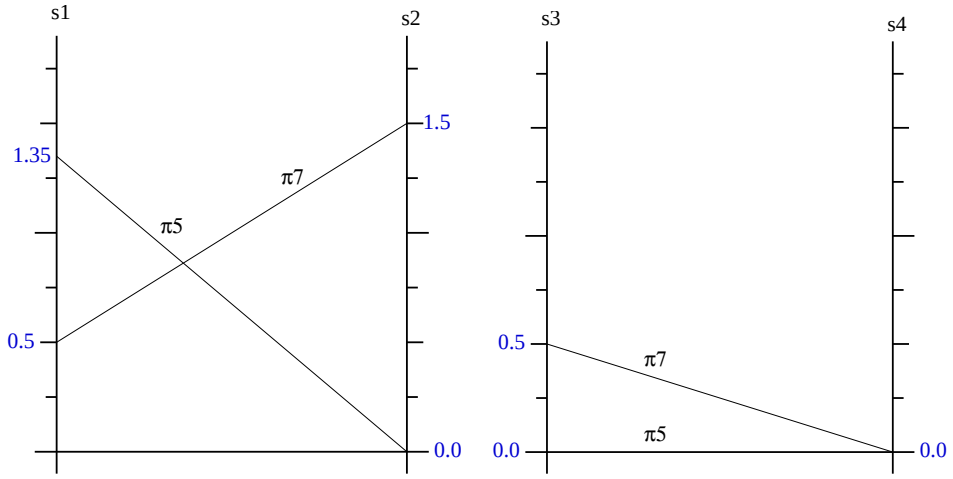
$$\begin{array}{rclrcl} \pi_1 & = & (\text{RED}, (), ()) & v(\pi_1) & = & \langle \mathbf{1.0}, \mathbf{0.0}, \mathbf{0.0}, \mathbf{0.0} \rangle \\ \pi_2 & = & (\text{BLUE}, (), ()) & v(\pi_2) & = & \langle \mathbf{0.0}, \mathbf{1.0}, \mathbf{0.0}, \mathbf{0.0} \rangle \end{array}$$



Plans of depth 2 and the corresponding value vectors are the following.

$$
\begin{array}{rclrcl}
\pi_3 & = & (\text{RED}, \pi_1, \pi_1) & v(\pi_3) & = & \langle 1.0, 0.0, 0.0, 0.0 \rangle \\
\pi_4 & = & (\text{RED}, \pi_1, \pi_2) & v(\pi_4) & = & \langle 1.0, 0.0, 0.0, 0.0 \rangle \\
\pi_5 & = & (\text{RED}, \pi_2, \pi_1) & v(\pi_5) & = & \langle \mathbf{1.35}, \mathbf{0.0}, \mathbf{0.0}, \mathbf{0.0} \rangle \\
\pi_6 & = & (\text{RED}, \pi_2, \pi_2) & v(\pi_6) & = & \langle 1.35, 0.0, 0.0, 0.0 \rangle \\
\pi_7 & = & (\text{BLUE}, \pi_1, \pi_1) & v(\pi_7) & = & \langle \mathbf{0.5}, \mathbf{1.5}, \mathbf{0.5}, \mathbf{0.0} \rangle \\
\pi_8 & = & (\text{BLUE}, \pi_1, \pi_2) & v(\pi_8) & = & \langle 0.5, 1.5, 0.5, 0.0 \rangle \\
\pi_9 & = & (\text{BLUE}, \pi_2, \pi_1) & v(\pi_9) & = & \langle 0.0, 1.0, 0.0, 0.0 \rangle \\
\pi_{10} & = & (\text{BLUE}, \pi_2, \pi_2) & v(\pi_{10}) & = & \langle 0.0, 1.0, 0.0, 0.0 \rangle
\end{array}
$$

The graphical representation of these vectors is as follows.

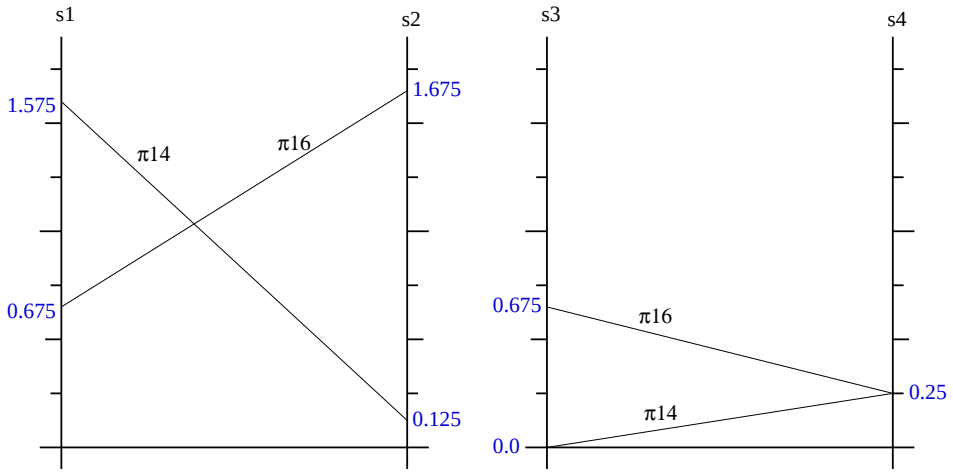


Because s_1 and s_2 are indistinguishable, but distinguishable from s_3 and s_4, the associated value functions can be depicted in two diagrams, each depicting probability distributions on a set of states that are indistinguishable from each other.

When enumerating plans of depth $i + 1$, it suffices to use as subplans only those plans of depth i that are the best plans for at least one belief state. Hence from the depth 2 plans we can ignore all but π_5 and π_7. Notice that in this example, we accidentally can recognize those plans that are

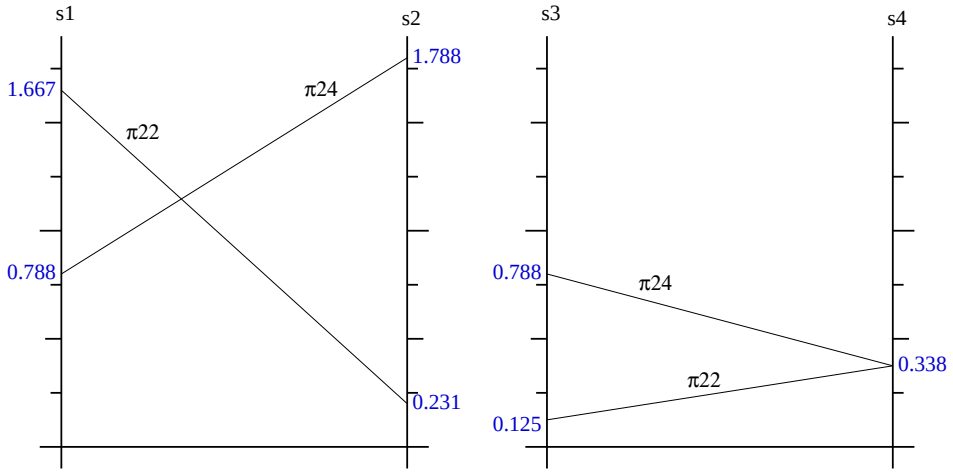
better anywhere from the fact that they are strictly worse on all states, and all the remaining plans are strictly better in at least one state. Plans of depth 3 and the corresponding value vectors are the following.

$$
\begin{array}{llll}
\pi_{11} = (\text{RED}, \pi_5, \pi_5) & v(\pi_{11}) = \langle 1.0, 0.0, 0.0, 0.0 \rangle \\
\pi_{12} = (\text{RED}, \pi_5, \pi_7) & v(\pi_{12}) = \langle 1.05, 0.125, 0.0, 0.25 \rangle \\
\pi_{13} = (\text{RED}, \pi_7, \pi_5) & v(\pi_{13}) = \langle 1.525, 0.0, 0.0, 0.0 \rangle \\
\pi_{14} = (\text{RED}, \pi_7, \pi_7) & v(\pi_{14}) = \langle \mathbf{1.575}, \mathbf{0.125}, \mathbf{0.0}, \mathbf{0.25} \rangle \\
\pi_{15} = (\text{BLUE}, \pi_5, \pi_5) & v(\pi_{15}) = \langle 0.675, 1.675, 0.675, 0.0 \rangle \\
\pi_{16} = (\text{BLUE}, \pi_5, \pi_7) & v(\pi_{16}) = \langle \mathbf{0.675}, \mathbf{1.675}, \mathbf{0.675}, \mathbf{0.25} \rangle \\
\pi_{17} = (\text{BLUE}, \pi_7, \pi_5) & v(\pi_{17}) = \langle 0.25, 1.25, 0.25, 0.0 \rangle \\
\pi_{18} = (\text{BLUE}, \pi_7, \pi_7) & v(\pi_{18}) = \langle 0.25, 1.25, 0.25, 0.25 \rangle
\end{array}
$$

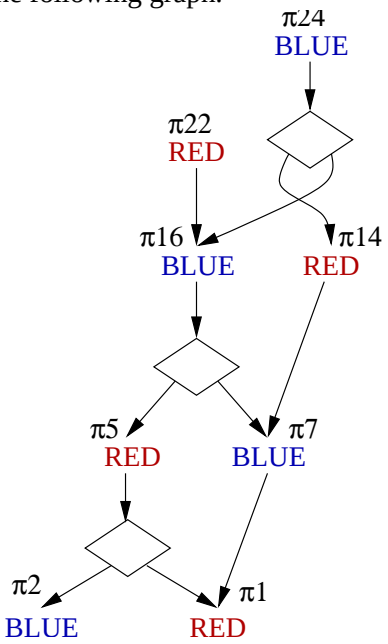


Plans of depth 4 and the corresponding value vectors are the following.

$$
\begin{array}{llll}
\pi_{19} = (\text{RED}, \pi_{14}, \pi_{14}) & v(\pi_{19}) = \langle 1.05625, 0.0625, 0.125, 0.0 \rangle \\
\pi_{20} = (\text{RED}, \pi_{14}, \pi_{16}) & v(\pi_{20}) = \langle 1.12375, 0.23125, 0.125, 0.3375 \rangle \\
\pi_{21} = (\text{RED}, \pi_{16}, \pi_{14}) & v(\pi_{21}) = \langle 1.59875, 0.0625, 0.125, 0.0 \rangle \\
\pi_{22} = (\text{RED}, \pi_{16}, \pi_{16}) & v(\pi_{22}) = \langle \mathbf{1.66625}, \mathbf{0.23125}, \mathbf{0.125}, \mathbf{0.3375} \rangle \\
\pi_{23} = (\text{BLUE}, \pi_{14}, \pi_{14}) & v(\pi_{23}) = \langle 0.7875, 1.7875, 0.7875, 0.0 \rangle \\
\pi_{24} = (\text{BLUE}, \pi_{14}, \pi_{16}) & v(\pi_{24}) = \langle \mathbf{0.7875}, \mathbf{1.7875}, \mathbf{0.7875}, \mathbf{0.3375} \rangle \\
\pi_{25} = (\text{BLUE}, \pi_{16}, \pi_{14}) & v(\pi_{25}) = \langle 0.3375, 1.3375, 0.3375, 0.0 \rangle \\
\pi_{26} = (\text{BLUE}, \pi_{16}, \pi_{16}) & v(\pi_{26}) = \langle 0.3375, 1.3375, 0.3375, 0.3375 \rangle
\end{array}
$$



The plan can be depicted as the following graph.



■

5.6 Literature

A comprehensive book on (fully observable) Markov decision processes has been written by Puterman [1994], and our presentation of the algorithms in Section 5.4 (5.4.2 and 5.4.3) follows that of Puterman. The book represents the traditional research on MDPs and uses exclusively enumerative representations of state spaces and transition probabilities. The book discusses all the main optimality criteria as well as algorithms for solving MDPs by iterative techniques and linear programming. There are also many other books on solving MDPs.

A planning system that implements the value iteration algorithm with ADDs is described by Hoey et al. [1999] and is shown to be capable of solving problems that could not be efficiently solved by conventional implementations of value iteration.

The best known algorithms for solving partially observable Markov decision processes were presented by Sondik and Smallwood in the early 1970's [Sondik, 1978; Smallwood and Sondik, 1973] and even today most of the work on POMDPs is based on those algorithms [Kaelbling *et al.*, 1998]. In this section we have presented the standard value iteration algorithm with the simplification that there is no sensing uncertainty, that is, for every state the same observation, dependent on the state, is always made.

The most general infinite-horizon planning problems and POMDP solution construction are undecidable [Madani *et al.*, 2003]. The complexity of probabilistic planning has been investigated for example by Mundhenk et al. [2000] and Littman [1997].

Bonet and Geffner [2000] and Hansen and Zilberstein [2001] have presented algorithms for probabilistic planning with Markov decision processes that use heuristic search.

5.7 Exercises

5.1 Prove that on each step of policy iteration the policy improves.

Bibliography

[Allen *et al.*, 1990] J. Allen, J. A. Hendler, and A. Tate, editors. *Readings in Planning*. Morgan Kaufmann Publishers, 1990.

[Alur *et al.*, 1997] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 340–351. Springer-Verlag, 1997.

[Bacchus and Kabanza, 2000] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1–2):123–191, 2000.

[Bäckström and Nebel, 1995] C. Bäckström and B. Nebel. Complexity results for SAS^+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

[Bahar *et al.*, 1997] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design: An International Journal*, 10(2/3):171–206, 1997.

[Balcázar *et al.*, 1988] J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity I*. Springer-Verlag, Berlin, 1988.

[Balcázar *et al.*, 1990] J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity II*. Springer-Verlag, Berlin, 1990.

[Bertoli *et al.*, 2001] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In B. Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 473–478. Morgan Kaufmann Publishers, 2001.

[Blum and Furst, 1997] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[Bonet and Geffner, 2000] B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In S. Chien, S. Kambhampati, and C. A. Knoblock, editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, pages 52–61. AAAI Press, 2000.

[Bonet and Geffner, 2001] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[Brooks, 1991] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[Bryant, 1992] R. E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[Burch *et al.*, 1994] J. R. Burch, E. M. Clarke, D. E. Long, K. L. MacMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[Bylander, 1994] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[Chandra *et al.*, 1981] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[Cimatti *et al.*, 2003] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1–2):35–84, 2003.

[Clarke *et al.*, 1994] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. Technical Report CS-94-204, Carnegie Mellon University, School of Computer Science, October 1994.

[Darwiche, 2001] A. Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):1–42, 2001.

[de Bakker and de Roever, 1972] J. W. de Bakker and W. P. de Roever. A calculus of recursive program schemes. In *Proceedings of the First International Colloquium on Automata, Languages and Programming*, pages 167–196. North-Holland, 1972.

[Dijkstra, 1976] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1976.

[Emerson and Sistla, 1996] E. A. Emerson and A. P. Sistla. Symmetry and model-checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131, 1996.

[Ernst *et al.*, 1969] G. Ernst, A. Newell, and H. Simon. *GPS: A Case Study in Generality and Problem Solving*. Academic Press, 1969.

[Erol *et al.*, 1995] K. Erol, D. S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1–2):75–88, 1995.

[Etzioni *et al.*, 1992] O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson. An approach to planning with incomplete information. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR '92)*, pages 115–125. Morgan Kaufmann Publishers, October 1992.

[Fikes and Nilsson, 1971] R. E. Fikes and N. J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(2-3):189–208, 1971.

[Fujita *et al.*, 1997] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. *Formal Methods in System Design: An International Journal*, 10(2/3):149–169, 1997.

[Ginsberg and Smith, 1988] M. L. Ginsberg and D. E. Smith. Reasoning about action I: A possible worlds approach. *Artificial Intelligence*, 35(2):165–195, 1988.

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

[Godefroid, 1991] P. Godefroid. Using partial orders to improve automatic verification methods. In E. M. Clarke, editor, *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV '90), Rutgers, New Jersey, 1990*, number 531 in Lecture Notes in Computer Science, pages 176–185. Springer-Verlag, 1991.

[Green, 1969] C. Green. Application of theorem-proving to problem solving. In D. E. Walker and L. M. Norton, editors, *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 219–239. William Kaufmann, 1969.

[Hansen and Zilberstein, 2001] E. A. Hansen and S. Zilberstein. LAO $*$: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 29(1-2):35–62, 2001.

[Hart *et al.*, 1968] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum-cost paths. *IEEE Transactions on System Sciences and Cybernetics*, SSC-4(2):100–107, 1968.

[Haslum and Geffner, 2000] P. Haslum and H. Geffner. Admissible heuristics for optimal planning. In S. Chien, S. Kambhampati, and C. A. Knoblock, editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, pages 140–149. AAAI Press, 2000.

[Haslum and Jonsson, 2000] P. Haslum and P. Jonsson. Some results on the complexity of planning with incomplete information. In S. Biundo and M. Fox, editors, *Recent Advances in AI Planning. Fifth European Conference on Planning (ECP'99)*, number 1809 in Lecture Notes in Artificial Intelligence, pages 308–318. Springer-Verlag, 2000.

[Hoey *et al.*, 1999] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. SPUDD: Stochastic planning using decision diagrams. In K. B. Laskey and H. Prade, editors, *Uncertainty in Artificial Intelligence, Proceedings of the Fifteenth Conference (UAI-99)*, pages 279–288. Morgan Kaufmann Publishers, 1999.

[Hoffmann and Nebel, 2001] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[Hopcroft and Ullman, 1979] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[Ichikawa and Hiraishi, 1988] A. Ichikawa and K. Hiraishi. Analysis and control of discrete-event systems represented as Petri nets. In P. Varaiya and B. Kurzhanski, editors, *Discrete Event*

Systems: Models and Applications, IIASA Conference, Soprpon Hungary, August 3-7, 1987, number 103 in Lecture Notes in Control and Information Sciences, pages 115–134. Springer-Verlag, 1988.

[Kaelbling *et al.*, 1998] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134, 1998.

[Kautz and Selman, 1992] H. Kautz and B. Selman. Planning as satisfiability. In B. Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363. John Wiley & Sons, 1992.

[Kautz and Selman, 1996] H. Kautz and B. Selman. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201. AAAI Press, August 1996.

[Kirkpatrick *et al.*, 1983] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[Knuth, 1998] D. E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Publishing Company, 1998.

[Korf, 1985] R. E. Korf. Depth-first iterative deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Kupferman and Vardi, 1999] O. Kupferman and M. Y. Vardi. Church's problem revisited. *The Bulletin of Symbolic Logic*, pages 245–263, 1999.

[Li and Wonham, 1993] Y. Li and W. M. Wonham. Control of vector discrete-event system I - the base model. *IEEE Transactions on Automatic Control*, 38(8):1214–1227, 1993.

[Littman, 1997] M. L. Littman. Probabilistic propositional planning: Representations and complexity. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97) and 9th Innovative Applications of Artificial Intelligence Conference (IAAI-97)*, pages 748–754, Menlo Park, July 1997. AAAI Press.

[Lozano and Balcázar, 1990] A. Lozano and J. L. Balcázar. The complexity of graph problems for succinctly represented graphs. In M. Nagl, editor, *Graph-Theoretic Concepts in Computer Science, 15th International Workshop, WG'89*, number 411 in Lecture Notes in Computer Science, pages 277–286, Castle Rolduc, The Netherlands, 1990. Springer-Verlag.

[Madani *et al.*, 2003] O. Madani, S. Hanks, and A. Condon. On the undecidability of probabilistic planning and related stochastic optimization problems. *Artificial Intelligence*, 147(1–2):5–34, 2003.

[McAllester and Rosenblitt, 1991] D. A. McAllester and D. Rosenblitt. Systematic nonlinear planning. In T. L. Dean and K. McKeown, editors, *Proceedings of the 9th National Conference on Artificial Intelligence*, volume 2, pages 634–639. AAAI Press / The MIT Press, 1991.

[Meyer and Stockmeyer, 1972] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential time. In *Proceedings of the 13th Annual Symposium on Switching and Automata Theory*, pages 125–129, Long Beach, California, 1972. IEEE Computer Society.

[Mundhenk *et al.*, 2000] M. Mundhenk, J. Goldsmith, C. Lusena, and E. Allender. Complexity of finite-horizon Markov decision process problems. *Journal of the ACM*, 47(4):681–720, 2000.

[Muscettola *et al.*, 1998] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams. Remote Agent: to boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[Papadimitriou and Yannakakis, 1986] C. H. Papadimitriou and M. Yannakakis. A note on succinct representations of graphs. *Information and Control*, 71:181–185, 1986.

[Papadimitriou, 1994] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.

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

[Peot and Smith, 1992] M. A. Peot and D. E. Smith. Conditional nonlinear planning. In J. Hendler, editor, *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 189–197, San Mateo, California, 1992. Morgan Kaufmann Publishers.

[Pixley *et al.*, 1992] C. Pixley, S.-W. Jeong, and G. D. Hachtel. Exact calculation of synchronization sequences based on binary decision diagrams. In *Proceedings of the 29th Design Automation Conference*, pages 620–623, 1992.

[Pryor and Collins, 1996] L. Pryor and G. Collins. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, 4:287–339, 1996.

[Puterman, 1994] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 1994.

[Ramadge and Wonham, 1987] P. Ramadge and W. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, January 1987.

[Rintanen *et al.*, 2004] J. Rintanen, K. Heljanko, and I. Niemelä. Parallel encodings of classical planning as satisfiability. In J. J. Alferes and J. Leite, editors, *Logics in Artificial Intelligence: 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004. Proceedings*, number 3229 in Lecture Notes in Computer Science, pages 307–319. Springer-Verlag, 2004.

[Rintanen, 1998] J. Rintanen. A planning algorithm not based on directional search. In A. G. Cohn, L. K. Schubert, and S. C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR '98)*, pages 617–624. Morgan Kaufmann Publishers, June 1998.

[Rintanen, 2004] J. Rintanen. Distance estimates for planning in the discrete belief space. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2004) and the Thirteenth Conference on Innovative Applications of Artificial Intelligence (IAAI-2004)*, pages 525–530. AAAI Press, 2004.

[Rosenschein, 1981] S. J. Rosenschein. Plan synthesis: A logical perspective. In P. J. Hayes, editor, *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, pages 331–337, Los Altos, California, August 1981. William Kaufmann.

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

[Sacerdoti, 1975] E. D. Sacerdoti. The nonlinear nature of plans. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence*, pages 206–214, 1975.

[Sandewall, 1994a] E. Sandewall. *Features and Fluents. The Representation of Knowledge about Dynamic Systems.*, volume I. Oxford University Press, 1994.

[Sandewall, 1994b] E. Sandewall. The range of applicability of nonmonotonic logics for the inertia problem. *Journal of Logic and Computation*, 4(5):581–615, 1994.

[Schoppers, 1987] M. J. Schoppers. Universal plans for real-time robots in unpredictable environments. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 1039–1046, Milano, 1987.

[Shoham, 1988] Y. Shoham. Chronological ignorance: Experiments in nonmonotonic temporal reasoning. *Artificial Intelligence*, 36(3):279–331, October 1988.

[Smallwood and Sondik, 1973] R. D. Smallwood and E. J. Sondik. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research*, 21:1071–1088, 1973.

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

[Starke, 1991] P. H. Starke. Reachability analysis of Petri nets using symmetries. *Journal of Mathematical Modelling and Simulation in Systems Analysis*, 8(4/5):293–303, 1991.

[Stein and Morgenstern, 1994] L. A. Stein and L. Morgenstern. Motivated action theory: a formal theory of causal reasoning. *Artificial Intelligence*, 71:1–42, 1994.

[Stockmeyer and Chandra, 1979] L. J. Stockmeyer and A. K. Chandra. Provably difficult combinatorial games. *SIAM Journal on Computing*, 8(2):151–174, 1979.

[Valmari, 1991] A. Valmari. Stubborn sets for reduced state space generation. In G. Rozenberg, editor, *Advances in Petri Nets 1990. 10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany*, number 483 in Lecture Notes in Computer Science, pages 491–515. Springer-Verlag, 1991.

[Vardi and Stockmeyer, 1985] M. Vardi and L. Stockmeyer. Improved upper and lower bounds for modal logics of programs. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 240–251. Association for Computing Machinery, 1985.

[Wonham, 1988] W. M. Wonham. A control theory for discrete-event systems. In M. Denham and A. Laub, editors, *Advanced Computing Concepts and Techniques in Control Engineering*, pages 129–169. Springer-Verlag, 1988.

Index