11 PLANNING

In which we see how an agent can take advantage of the structure of a problem to construct complex plans of action.

The task of coming up with a sequence of actions that will achieve a goal is called **planning**. We have seen two examples of planning agents so far: the search-based problem-solving agent of Chapter 3 and the logical planning agent of Chapter 10. This chapter is concerned primarily with *scaling up* to complex planning problems that defeat the approaches we have seen so far.

Section 11.1 develops an expressive yet carefully constrained language for representing planning problems, including actions and states. The language is closely related to the propositional and first-order representations of actions in Chapters 7 and 10. Section 11.2 shows how forward and backward search algorithms can take advantage of this representation, primarily through accurate heuristics that can be derived automatically from the structure of the representation. (This is analogous to the way in which effective heuristics were constructed for constraint satisfaction problems in Chapter 5.) Sections 11.3 through 11.5 describe planning algorithms that go beyond forward and backward search, taking advantage of the representation of the problem. In particular, we explore approaches that are not constrained to consider only totally ordered sequences of actions.

For this chapter, we consider only environments that are fully observable, deterministic, finite, static (change happens only when the agent acts), and discrete (in time, action, objects, and effects). These are called **classical planning** environments. In contrast, nonclassical planning is for partially observable or stochastic environments and involves a different set of algorithms and agent designs, outlined in Chapters 12 and 17.

CLASSICAL PLANNING

11.1 THE PLANNING PROBLEM

Let us consider what can happen when an ordinary problem-solving agent using standard search algorithms—depth-first, A*, and so on—comes up against large, real-world problems. That will help us design better planning agents.

The most obvious difficulty is that the problem-solving agent can be overwhelmed by irrelevant actions. Consider the task of buying a copy of *AI: A Modern Approach* from an online bookseller. Suppose there is one buying action for each 10-digit ISBN number, for a total of 10 billion actions. The search algorithm would have to examine the outcome states of all 10 billion actions to find one that satisfies the goal, which is to own a copy of ISBN 0137903952. A sensible planning agent, on the other hand, should be able to work back from an explicit goal description such as $Have(ISBN0137903952)$ and generate the action $Buy(ISBN0137903952)$ directly. To do this, the agent simply needs the general knowledge that $Buy(x)$ results in $Have(x)$. Given this knowledge and the goal, the planner can decide in a single unification step that $Buy(ISBN0137903952)$ is the right action.

The next difficulty is finding a good **heuristic function**. Suppose the agent's goal is to buy four different books online. Then there will be 10^{40} plans of just four steps, so searching without an accurate heuristic is out of the question. It is obvious to a human that a good heuristic estimate for the cost of a state is the number of books that remain to be bought; unfortunately, this insight is not obvious to a problem-solving agent, because it sees the goal test only as a black box that returns true or false for each state. Therefore, the problem-solving agent lacks autonomy; it requires a human to supply a heuristic function for each new problem. On the other hand, if a planning agent has access to an explicit representation of the goal as a conjunction of subgoals, then it can use a single *domain-independent* heuristic: the number of unsatisfied conjuncts. For the book-buying problem, the goal would be $Have(A) \wedge Have(B) \wedge Have(C) \wedge Have(D)$, and a state containing $Have(A) \wedge Have(C)$ would have cost 2. Thus, the agent automatically gets the right heuristic for this problem, and for many others. We shall see later in the chapter how to construct more sophisticated heuristics that examine the available actions as well as the structure of the goal.

PROBLEM DECOMPOSITION

Finally, the problem solver might be inefficient because it cannot take advantage of **problem decomposition**. Consider the problem of delivering a set of overnight packages to their respective destinations, which are scattered across Australia. It makes sense to find out the nearest airport for each destination and divide the overall problem into several subproblems, one for each airport. Within the set of packages routed through a given airport, whether further decomposition is possible depends on the destination city. We saw in Chapter 5 that the ability to do this kind of decomposition contributes to the efficiency of constraint satisfaction problem solvers. The same holds true for planners: in the worst case, it can take $O(n!)$ time to find the best plan to deliver n packages, but only $O((n/k)! \times k)$ time if the problem can be decomposed into k equal parts.

As we noted in Chapter 5, perfectly decomposable problems are delicious but rare.[1] The design of many planning systems—particularly the partial-order planners described in Section 11.3—is based on the assumption that most real-world problems are **nearly decomposable**. That is, the planner can work on subgoals independently, but might need to do some additional work to combine the resulting subplans. For some problems, this assump-

NEARLY DECOMPOSABLE

[1] Notice that even the delivery of a package is not perfectly decomposable. There may be cases in which it is better to assign packages to a more distant airport if that renders a flight to the nearest airport unnecessary. Nevertheless, most delivery companies prefer the computational and organizational simplicity of sticking with decomposed solutions.

tion breaks down because working on one subgoal is likely to undo another subgoal. These interactions among subgoals are what makes puzzles (like the 8-puzzle) puzzling.

The language of planning problems

The preceding discussion suggests that the representation of planning problems—states, actions, and goals—should make it possible for planning algorithms to take advantage of the logical structure of the problem. The key is to find a language that is expressive enough to describe a wide variety of problems, but restrictive enough to allow efficient algorithms to operate over it. In this section, we first outline the basic representation language of classical planners, known as the STRIPS language.[2] Later, we point out some of the many possible variations in STRIPS-like languages.

Representation of states. Planners decompose the world into logical conditions and represent a state as a conjunction of positive literals. We will consider propositional literals; for example, $Poor \wedge Unknown$ might represent the state of a hapless agent. We will also use first-order literals; for example, $At(Plane_1, Melbourne) \wedge At(Plane_2, Sydney)$ might represent a state in the package delivery problem. Literals in first-order state descriptions must be **ground** and **function-free**. Literals such as $At(x, y)$ or $At(Father(Fred), Sydney)$ are not allowed. The **closed-world assumption** is used, meaning that any conditions that are not mentioned in a state are assumed false.

Representation of goals. A goal is a partially specified state, represented as a conjunction of positive ground literals, such as $Rich \wedge Famous$ or $At(P_2, Tahiti)$. A propositional state s **satisfies** a goal g if s contains all the atoms in g (and possibly others). For example, the state $Rich \wedge Famous \wedge Miserable$ satisfies the goal $Rich \wedge Famous$.

GOAL SATISFACTION

Representation of actions. An action is specified in terms of the preconditions that must hold before it can be executed and the effects that ensue when it is executed. For example, an action for flying a plane from one location to another is:

$$\begin{array}{l} Action(Fly(p, from, to), \\ \quad \text{PRECOND:} At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to) \\ \quad \text{EFFECT:} \neg At(p, from) \wedge At(p, to)) \end{array}$$

ACTION SCHEMA

This is more properly called an **action schema**, meaning that it represents a number of different actions that can be derived by instantiating the variables p, $from$, and to to different constants. In general, an action schema consists of three parts:

- The action name and parameter list—for example, $Fly(p, from, to)$—serves to identify the action.
- PRECONDITION The **precondition** is a conjunction of function-free positive literals stating what must be true in a state before the action can be executed. Any variables in the precondition must also appear in the action's parameter list.
- EFFECT The **effect** is a conjunction of function-free literals describing how the state changes when the action is executed. A positive literal P in the effect is asserted to be true in

[2] STRIPS stands for STanford Research Institute Problem Solver.

the state resulting from the action, whereas a negative literal $\neg P$ is asserted to be false. Variables in the effect must also appear in the action's parameter list.

ADD LIST
DELETE LIST

To improve readability, some planning systems divide the effect into the **add list** for positive literals and the **delete list** for negative literals.

Having defined the syntax for representations of planning problems, we can now define the semantics. The most straightforward way to do this is to describe how actions affect states. (An alternative method is to specify a direct translation into successor-state axioms, whose semantics comes from first-order logic; see Exercise 11.3.) First, we say that an action

APPLICABLE

is **applicable** in any state that satisfies the precondition; otherwise, the action has no effect. For a first-order action schema, establishing applicability will involve a substitution θ for the variables in the precondition. For example, suppose the current state is described by

$$At(P_1, JFK) \wedge At(P_2, SFO) \wedge Plane(P_1) \wedge Plane(P_2) \\ \wedge Airport(JFK) \wedge Airport(SFO) \ .$$

This state satisfies the precondition

$$At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$$

with substitution $\{p/P_1, from/JFK, to/SFO\}$ (among others—see Exercise 11.2). Thus, the concrete action $Fly(P_1, JFK, SFO)$ is applicable.

RESULT

Starting in state s, the **result** of executing an applicable action a is a state s' that is the same as s except that any positive literal P in the effect of a is added to s' and any negative literal $\neg P$ is removed from s'. Thus, after $Fly(P_1, JFK, SFO)$, the current state becomes

$$At(P_1, SFO) \wedge At(P_2, SFO) \wedge Plane(P_1) \wedge Plane(P_2) \\ \wedge Airport(JFK) \wedge Airport(SFO) \ .$$

Note that if a positive effect is already in s it is not added twice, and if a negative effect is not in s, then that part of the effect is ignored. This definition embodies the so-called STRIPS

STRIPS ASSUMPTION

assumption: that every literal not mentioned in the effect remains unchanged. In this way, STRIPS avoids the representational **frame problem** described in Chapter 10.

SOLUTION

Finally, we can define the **solution** for a planning problem. In its simplest form, this is just an action sequence that, when executed in the initial state, results in a state that satisfies the goal. Later in the chapter, we will allow solutions to be partially ordered sets of actions, provided that every action sequence that respects the partial order is a solution.

Expressiveness and extensions

The various restrictions imposed by the STRIPS representation were chosen in the hope of making planning algorithms simpler and more efficient, without making it too difficult to describe real problems. One of the most important restrictions is that literals be *function-free*. With this restriction, we can be sure that any action schema for a given problem can be propositionalized—that is, turned into a finite collection of purely propositional action representations with no variables. (See Chapter 9 for more on this topic.) For example, in the air cargo domain for a problem with 10 planes and five airports, we could translate the $Fly(p, from, to)$ schema into $10 \times 5 \times 5 = 250$ purely propositional actions. The planners

STRIPS **Language**	ADL **Language**
Only positive literals in states: $Poor \wedge Unknown$	Positive and negative literals in states: $\neg Rich \wedge \neg Famous$
Closed World Assumption: Unmentioned literals are false.	Open World Assumption: Unmentioned literals are unknown.
Effect $P \wedge \neg Q$ means add P and delete Q.	Effect $P \wedge \neg Q$ means add P and $\neg Q$ and delete $\neg P$ and Q.
Only ground literals in goals: $Rich \wedge Famous$	Quantified variables in goals: $\exists x At(P_1, x) \wedge At(P_2, x)$ is the goal of having P_1 and P_2 in the same place.
Goals are conjunctions: $Rich \wedge Famous$	Goals allow conjunction and disjunction: $\neg Poor \wedge (Famous \vee Smart)$
Effects are conjunctions.	Conditional effects allowed: **when** P: E means E is an effect only if P is satisfied.
No support for equality.	Equality predicate ($x = y$) is built in.
No support for types.	Variables can have types, as in ($p : Plane$).

Figure 11.1 Comparison of STRIPS and ADL languages for representing planning problems. In both cases, goals behave as the preconditions of an action with no parameters.

in Sections 11.4 and 11.5 work directly with propositionalized descriptions. If we allow function symbols, then infinitely many states and actions can be constructed.

In recent years, it has become clear that STRIPS is insufficiently expressive for some real domains. As a result, many language variants have been developed. Figure 11.1 briefly describes one important one, the Action Description Language or **ADL**, by comparing it with the basic STRIPS language. In ADL, the *Fly* action could be written as

ADL

$$\begin{array}{l} Action(Fly(p : Plane, from : Airport, to : Airport), \\ \quad \text{PRECOND:} At(p, from) \wedge (from \neq to) \\ \quad \text{EFFECT:} \neg At(p, from) \wedge At(p, to)) \ . \end{array}$$

The notation $p : Plane$ in the parameter list is an abbreviation for $Plane(p)$ in the precondition; this adds no expressive power, but can be easier to read. (It also cuts down on the number of possible propositional actions that can be constructed.) The precondition $(from \neq to)$ expresses the fact that a flight cannot be made from an airport to itself. This could not be expressed succinctly in STRIPS.

The various planning formalisms used in AI have been systematized within a standard syntax called the the Planning Domain Definition Language, or PDDL. This language allows researchers to exchange becnchmark problems and compare results. PDDL includes sublanguages for STRIPS, ADL, and the hierarchical task networks we will see in Chapter 12.

$Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK)$
$\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2)$
$\quad \land Airport(JFK) \land Airport(SFO))$
$Goal(At(C_1, JFK) \land At(C_2, SFO))$
$Action(Load(c, p, a),$
$\quad$ PRECOND: $At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
$\quad$ EFFECT: $\neg At(c, a) \land In(c, p))$
$Action(Unload(c, p, a),$
$\quad$ PRECOND: $In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
$\quad$ EFFECT: $At(c, a) \land \neg In(c, p))$
$Action(Fly(p, from, to),$
$\quad$ PRECOND: $At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)$
$\quad$ EFFECT: $\neg At(p, from) \land At(p, to))$

Figure 11.2 A STRIPS problem involving transportation of air cargo between airports.

The STRIPS and ADL notations are adequate for many real domains. The subsections that follow show some simple examples. There are still some significant restrictions, however. The most obvious is that they cannot represent in a natural way the **ramifications** of actions. For example, if there are people, packages, or dust motes in the airplane, then they too change location when the plane flies. We can represent these changes as the direct effects of flying, whereas it seems more natural to represent the location of the plane's contents as a logical consequence of the location of the plane. We will see more examples of such **state constraints** in Section 11.5. Classical planning systems do not even attempt to address the **qualification** problem: the problem of unrepresented circumstances that could cause an action to fail. We will see how to address qualifications in Chapter 12.

STATE CONSTRAINTS

Example: Air cargo transport

Figure 11.2 shows an air cargo transport problem involving loading and unloading cargo onto and off of planes and flying it from place to place. The problem can be defined with three actions: $Load$, $Unload$, and Fly. The actions affect two predicates: $In(c, p)$ means that cargo c is inside plane p, and $At(x, a)$ means that object x (either plane or cargo) is at airport a. Note that cargo is not At anywhere when it is In a plane, so At really means "available for use at a given location." It takes some experience with action definitions to handle such details consistently. The following plan is a solution to the problem:

$$[Load(C_1, P_1, SFO), Fly(P_1, SFO, JFK),$$
$$Load(C_2, P_2, JFK), Fly(P_2, JFK, SFO)] .$$

Our representation is pure STRIPS. In particular, it allows a plane to fly to and from the same airport. Inequality literals in ADL could prevent this.

Example: The spare tire problem

Consider the problem of changing a flat tire. More precisely, the goal is to have a good spare tire properly mounted onto the car's axle, where the initial state has a flat tire on the axle and a good spare tire in the trunk. To keep it simple, our version of the problem is a very abstract one, with no sticky lug nuts or other complications. There are just four actions: removing the spare from the trunk, removing the flat tire from the axle, putting the spare on the axle, and leaving the car unattended overnight. We assume that the car is in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear.

The ADL description of the problem is shown in Figure 11.3. Notice that it is purely propositional. It goes beyond STRIPS in that it uses a negated precondition, $\neg At(Flat, Axle)$, for the $PutOn(Spare, Axle)$ action. This could be avoided by using $Clear(Axle)$ instead, as we will see in the next example.

$Init(At(Flat, Axle) \ \wedge \ At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(Spare, Trunk),$
  PRECOND: $At(Spare, Trunk)$
  EFFECT: $\neg \ At(Spare, Trunk) \ \wedge \ At(Spare, Ground))$
$Action(Remove(Flat, Axle),$
  PRECOND: $At(Flat, Axle)$
  EFFECT: $\neg \ At(Flat, Axle) \ \wedge \ At(Flat, Ground))$
$Action(PutOn(Spare, \ Axle),$
  PRECOND: $At(Spare, Ground) \ \wedge \ \neg \ At(Flat, Axle)$
  EFFECT: $\neg \ At(Spare, Ground) \ \wedge \ At(Spare, Axle))$
$Action(LeaveOvernight,$
  PRECOND:
  EFFECT: $\neg \ At(Spare, Ground) \ \wedge \ \neg \ At(Spare, Axle) \ \wedge \ \neg \ At(Spare, Trunk)$
  $\wedge \neg \ At(Flat, Ground) \ \wedge \ \neg \ At(Flat, Axle))$

Figure 11.3 The simple spare tire problem.

Example: The blocks world

BLOCKS WORLD

One of the most famous planning domains is known as the **blocks world**. This domain consists of a set of cube-shaped blocks sitting on a table.[3] The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it. The goal will always be to build one or more stacks of blocks, specified in terms of what blocks are on top of what other blocks. For example, a goal might be to get block A on B and block C on D.

[3] The blocks world used in planning research is much simpler than SHRDLU's version, shown on page 20.

We will use $On(b, x)$ to indicate that block b is on x, where x is either another block or the table. The action for moving block b from the top of x to the top of y will be $Move(b, x, y)$. Now, one of the preconditions on moving b is that no other block be on it. In first-order logic, this would be $\neg\exists\, x\ On(x, b)$ or, alternatively, $\forall\, x\ \neg On(x, b)$. These could be stated as preconditions in ADL. We can stay within the STRIPS language, however, by introducing a new predicate, $Clear(x)$, that is true when nothing is on x.

The action $Move$ moves a block b from x to y if both b and y are clear. After the move is made, x is clear but y is not. A formal description of $Move$ in STRIPS is

$Action(Move(b, x, y),$
PRECOND:$On(b, x) \wedge Clear(b) \wedge Clear(y),$
EFFECT:$On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y))$.

Unfortunately, this action does not maintain $Clear$ properly when x or y is the table. When $x = Table$, this action has the effect $Clear(Table)$, but the table should not become clear, and when $y = Table$, it has the precondition $Clear(Table)$, but the table does not have to be clear to move a block onto it. To fix this, we do two things. First, we introduce another action to move a block b from x to the table:

$Action(MoveToTable(b, x),$
PRECOND:$On(b, x) \wedge Clear(b)),$
EFFECT:$On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))$.

Second, we take the interpretation of $Clear(b)$ to be "there is a clear space on b to hold a block." Under this interpretation, $Clear(Table)$ will always be true. The only problem is that nothing prevents the planner from using $Move(b, x, Table)$ instead of $MoveToTable(b, x)$. We could live with this problem—it will lead to a larger-than-necessary search space, but will not lead to incorrect answers—or we could introduce the predicate $Block$ and add $Block(b) \wedge Block(y)$ to the precondition of $Move$.

Finally, there is the problem of spurious actions such as $Move(B, C, C)$, which should be a no-op, but which has contradictory effects. It is common to ignore such problems, because they seldom cause incorrect plans to be produced. The correct approach is add inequality preconditions as shown in Figure 11.4.

11.2 PLANNING WITH STATE-SPACE SEARCH

Now we turn our attention to planning algorithms. The most straightforward approach is to use state-space search. Because the descriptions of actions in a planning problem specify both preconditions and effects, it is possible to search in either direction: either forward from the initial state or backward from the goal, as shown in Figure 11.5. We can also use the explicit action and goal representations to derive effective heuristics automatically.

Forward state-space search

Planning with forward state-space search is similar to the problem-solving approach of Chapter 3. It is sometimes called **progression** planning, because it moves in the forward direction.

PROGRESSION

$Init(On(A, Table) \wedge On(B, Table) \wedge On(C, Table)$
$\quad \wedge Block(A) \wedge Block(B) \wedge Block(C)$
$\quad \wedge Clear(A) \wedge Clear(B) \wedge Clear(C))$
$Goal(On(A, B) \wedge On(B, C))$
$Action(Move(b, x, y),$
PRECOND: $On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge$
$\quad (b \neq x) \wedge (b \neq y) \wedge (x \neq y),$
EFFECT: $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y))$
$Action(MoveToTable(b, x),$
PRECOND: $On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x),$
EFFECT: $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))$

Figure 11.4 A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[Move(B, Table, C), Move(A, Table, B)]$.

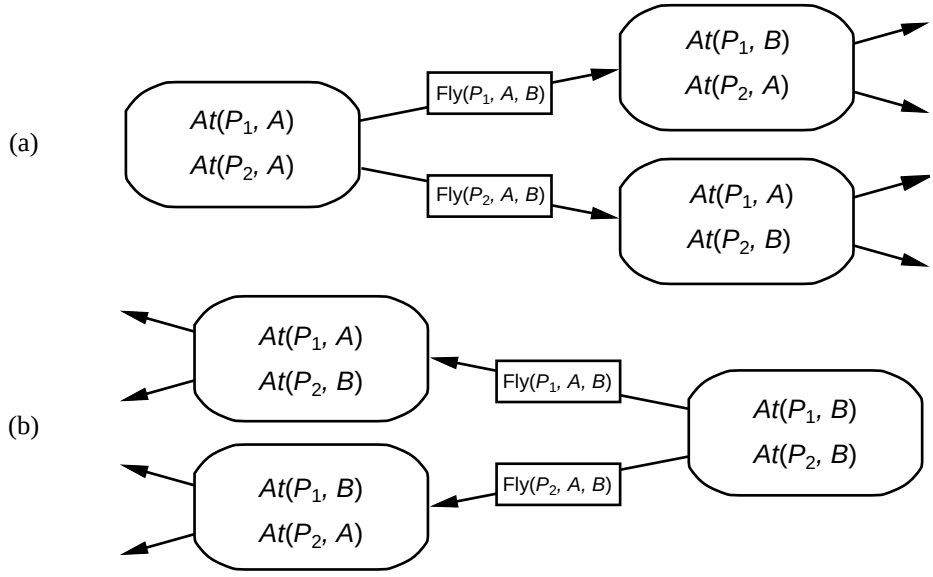

Figure 11.5 Two approaches to searching for a plan. (a) Forward (progression) state-space search, starting in the initial state and using the problem's actions to search forward for the goal state. (b) Backward (regression) state-space search: a belief-state search (see page 84) starting at the goal state(s) and using the inverse of the actions to search backward for the initial state.

We start in the problem's initial state, considering sequences of actions until we find a sequence that reaches a goal state. The formulation of planning problems as state-space search problems is as follows:

- The **initial state** of the search is the initial state from the planning problem. In general, each state will be a set of positive ground literals; literals not appearing are false.

- The **actions** that are applicable to a state are all those whose preconditions are satisfied. The successor state resulting from an action is generated by adding the positive effect literals and deleting the negative effect literals. (In the first-order case, we must apply the unifier from the preconditions to the effect literals.) Note that a single successor function works for all planning problems—a consequence of using an explicit action representation.
- The **goal test** checks whether the state satisfies the goal of the planning problem.
- The **step cost** of each action is typically 1. Although it would be easy to allow different costs for different actions, this is seldom done by STRIPS planners.

Recall that, in the absence of function symbols, the state space of a planning problem is finite. Therefore, any graph search algorithm that is complete—for example, A*—will be a complete planning algorithm.

From the earliest days of planning research (around 1961) until recently (around 1998) it was assumed that forward state-space search was too inefficient to be practical. It is not hard to come up with reasons why—just refer back to the start of Section 11.1. First, forward search does not address the irrelevant action problem—all applicable actions are considered from each state. Second, the approach quickly bogs down without a good heuristic. Consider an air cargo problem with 10 airports, where each airport has 5 planes and 20 pieces of cargo. The goal is to move all the cargo at airport A to airport B. There is a simple solution to the problem: load the 20 pieces of cargo into one of the planes at A, fly the plane to B, and unload the cargo. But finding the solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded), or loaded into any plane at its airport (if it is unloaded). On average, let's say there are about 1000 possible actions, so the search tree up to the depth of the obvious solution has about 1000^{41} nodes. It is clear that a very accurate heuristic will be needed to make this kind of search efficient. We will discuss some possible heuristics after looking at backward search.

Backward state-space search

Backward state-space search was described briefly as part of bidirectional search in Chapter 3. We noted there that backward search can be difficult to implement when the goal states are described by a set of constraints rather than being listed explicitly. In particular, it is not always obvious how to generate a description of the possible **predecessors** of the set of goal states. We will see that the STRIPS representation makes this quite easy because sets of states can be described by the literals that must be true in those states.

RELEVANCE

The main advantage of backward search is that it allows us to consider only **relevant** actions. An action is relevant to a conjunctive goal if it achieves one of the conjuncts of the goal. For example, the goal in our 10-airport air cargo problem is to have 20 pieces of cargo at airport B, or more precisely,

$$At(C_1, B) \wedge At(C_2, B) \wedge \ldots \wedge At(C_{20}, B) \ .$$

Now consider the conjunct $At(C_1, B)$. Working backwards, we can seek actions that have this as an effect. There is only one: $Unload(C_1, p, B)$, where plane p is unspecified.

Notice that there are many *irrelevant* actions that can also lead to a goal state. For example, we can fly an empty plane from JFK to SFO; this action reaches a goal state from a predecessor state in which the plane is at JFK and all the goal conjuncts are satisfied. A backward search that allows irrelevant actions will still be complete, but it will be much less efficient. If a solution exists, it will be found by a backward search that allows only relevant actions. The restriction to relevant actions means that backward search often has a much lower branching factor than forward search. For example, our air cargo problem has about 1000 actions leading forward from the initial state, but only 20 actions working backward from the goal.

REGRESSION

Searching backwards is sometimes called **regression** planning. The principal question in regression planning is this: what are the states from which applying a given action leads to the goal? Computing the description of these states is called **regressing** the goal through the action. To see how to do it, consider the air cargo example. We have the goal

$$At(C_1, B) \wedge At(C_2, B) \wedge \ldots \wedge At(C_{20}, B)$$

and the relevant action $Unload(C_1, p, B)$, which achieves the first conjunct. The action will work only if its preconditions are satisfied. Therefore, any predecessor state must include these preconditions: $In(C_1, p) \wedge At(p, B)$. Moreover, the subgoal $At(C_1, B)$ should not be true in the predecessor state.[4] Thus, the predecessor description is

$$In(C_1, p) \wedge At(p, B) \wedge At(C_2, B) \wedge \ldots \wedge At(C_{20}, B) \ .$$

CONSISTENCY

In addition to insisting that actions achieve some desired literal, we must insist that the actions *not undo* any desired literals. An action that satisfies this restriction is called **consistent**. For example, the action $Load(C_2, p)$ would not be consistent with the current goal, because it would negate the literal $At(C_2, B)$.

Given definitions of relevance and consistency, we can describe the general process of constructing predecessors for backward search. Given a goal description G, let A be an action that is relevant and consistent. The corresponding predecessor is as follows:

- Any positive effects of A that appear in G are deleted.
- Each precondition literal of A is added, unless it already appears.

Any of the standard search algorithms can be used to carry out the search. Termination occurs when a predecessor description is generated that is satisfied by the initial state of the planning problem. In the first-order case, satisfaction might require a substitution for variables in the predecessor description. For example, the predecessor description in the preceding paragraph is satisfied by the initial state

$$In(C_1, P_{12}) \wedge At(P_{12}, B) \wedge At(C_2, B) \wedge \ldots \wedge At(C_{20}, B)$$

with substitution $\{p/P_{12}\}$. The substitution must be applied to the actions leading from the state to the goal, producing the solution $[Unload(C_1, P_{12}, B)]$.

[4] If the subgoal were true in the predecessor state, the action would still lead to a goal state. On the other hand, such actions are irrelevant because they do not *make* the goal true.

Heuristics for state-space search

It turns out that neither forward nor backward search is efficient without a good heuristic function. Recall from Chapter 4 that a heuristic function estimates the distance from a state to the goal; in STRIPS planning, the cost of each action is 1, so the distance is the number of actions. The basic idea is to look at the effects of the actions and at the goals that must be achieved and to guess how many actions are needed to achieve all the goals. Finding the exact number is NP hard, but it is possible to find reasonable estimates most of the time without too much computation. We might also be able to derive an **admissible** heuristic—one that does not overestimate. This could be used with A* search to find optimal solutions.

There are two approaches that can be tried. The first is to derive a **relaxed problem** from the given problem specification, as described in Chapter 4. The optimal solution cost for the relaxed problem—which we hope is very easy to solve—gives an admissible heuristic for the original problem. The second approach is to pretend that a pure divide-and-conquer algorithm will work. This is called the **subgoal independence** assumption: the cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving each subgoal *independently*. The subgoal independence assumption can be optimistic or pessimistic. It is optimistic when there are negative interactions between the subplans for each subgoal—for example, when an action in one subplan deletes a goal achieved by another subplan. It is pessimistic, and therefore inadmissible, when subplans contain redundant actions—for instance, two actions that could be replaced by a single action in the merged plan.

SUBGOAL INDEPENDENCE

Let us consider how to derive relaxed planning problems. Since explicit representations of preconditions and effects are available, the process will work by modifying those representations. (Compare this approach with search problems, where the successor function is a black box.) The simplest idea is to relax the problem by *removing all preconditions* from the actions. Then every action will always be applicable, and any literal can be achieved in one step (if there is an applicable action—if not, the goal is impossible). This almost implies that the number of steps required to solve a conjunction of goals is the number of unsatisfied goals—almost but not quite, because (1) there may be two actions, each of which deletes the goal literal achieved by the other, and (2) some action may achieve multiple goals. If we combine our relaxed problem with the subgoal independence assumption, both of these issues are assumed away and the resulting heuristic is exactly the number of unsatisfied goals.

In many cases, a more accurate heuristic is obtained by considering at least the positive interactions arising from actions that achieve multiple goals. First, we relax the problem further by *removing negative effects* (see Exercise 11.6). Then, we count the minimum number of actions required such that the union of those actions' positive effects satisfies the goal. For example, consider

$$Goal(A \wedge B \wedge C)$$
$$Action(X, \text{EFFECT:}A \wedge P)$$
$$Action(Y, \text{EFFECT:}B \wedge C \wedge Q)$$
$$Action(Z, \text{EFFECT:}B \wedge P \wedge Q)\ .$$

The minimal set cover of the goal $\{A, B, C\}$ is given by the actions $\{X, Y\}$, so the set cover heuristic returns a cost of 2. This improves on the subgoal independence assumption, which

gives a heuristic value of 3. There is one minor irritation: the set cover problem is NP-hard. A simple greedy set-covering algorithm is guaranteed to return a value that is within a factor of $\log n$ of the true minimum value, where n is the number of literals in the goal, and usually works much better than this in practice. Unfortunately, the greedy algorithm loses the guarantee of admissibility for the heuristic.

It is also possible to generate relaxed problems by removing negative effects without removing preconditions. That is, if an action has the effect $A \wedge \neg B$ in the original problem, it will have the effect A in the relaxed problem. This means that we need not worry about negative interactions between subplans, because no action can delete the literals achieved by another action. The solution cost of the resulting relaxed problem gives what is called the

EMPTY-DELETE-LIST **empty-delete-list** heuristic. The heuristic is quite accurate, but computing it involves actually running a (simple) planning algorithm. In practice, the search in the relaxed problem is often fast enough that the cost is worthwhile.

The heuristics described here can be used in either the progression or the regression direction. At the time of writing, progression planners using the empty-delete-list heuristic hold the lead. That is likely to change as new heuristics and new search techniques are explored. Since planning is exponentially hard,[5] no algorithm will be efficient for all problems, but many practical problems can be solved with the heuristic methods in this chapter—far more than could be solved just a few years ago.

11.3 PARTIAL-ORDER PLANNING

Forward and backward state-space search are particular forms of *totally ordered* plan search. They explore only strictly linear sequences of actions directly connected to the start or goal. This means that they cannot take advantage of problem decomposition. Rather than work on each subproblem separately, they must always make decisions about how to sequence actions from all the subproblems. We would prefer an approach that works on several subgoals independently, solves them with several subplans, and then combines the subplans.

Such an approach also has the advantage of flexibility in the order in which it *constructs* the plan. That is, the planner can work on "obvious" or "important" decisions first, rather than being forced to work on steps in chronological order. For example, a planning agent that is in Berkeley and wishes to be in Monte Carlo might first try to find a flight from San Francisco to Paris; given information about the departure and arrival times, it can then work on ways to get to and from the airports.

LEAST COMMITMENT The general strategy of delaying a choice during search is called a **least commitment** strategy. There is no formal definition of least commitment, and clearly some degree of commitment is necessary, lest the search would make no progress. Despite the informality, least commitment is a useful concept for analyzing when decisions should be made in any search problem.

[5] Technically, STRIPS-style planning is PSPACE-complete unless actions have only positive preconditions and only one effect literal (Bylander, 1994).

Our first concrete example will be much simpler than planning a vacation. Consider the simple problem of putting on a pair of shoes. We can describe this as a formal planning problem as follows:

$$
\begin{aligned}
&Goal(RightShoeOn \wedge LeftShoeOn)\\
&Init()\\
&Action(RightShoe, \text{PRECOND:}RightSockOn, \text{EFFECT:}RightShoeOn)\\
&Action(RightSock, \text{EFFECT:}RightSockOn)\\
&Action(LeftShoe, \text{PRECOND:}LeftSockOn, \text{EFFECT:}LeftShoeOn)\\
&Action(LeftSock, \text{EFFECT:}LeftSockOn)\ .
\end{aligned}
$$

A planner should be able to come up with the two-action sequence $RightSock$ followed by $RightShoe$ to achieve the first conjunct of the goal and the sequence $LeftSock$ followed by $LeftShoe$ for the second conjunct. Then the two sequences can be combined to yield the final plan. In doing this, the planner will be manipulating the two subsequences independently, without committing to whether an action in one sequence is before or after an action in the other. Any planning algorithm that can place two actions into a plan without specifying which comes first is called a **partial-order planner**. Figure 11.6 shows the partial-order plan that is the solution to the shoes and socks problem. Note that the solution is represented as a *graph* of actions, not a sequence. Note also the "dummy" actions called $Start$ and $Finish$, which mark the beginning and end of the plan. Calling them actions symplifies things, because now every step of a plan is an action. The partial-order solution corresponds to six possible total-order plans; each of these is called a **linearization** of the partial-order plan.

PARTIAL-ORDER PLANNER

LINEARIZATION

Partial-order planning can be implemented as a search in the space of partial-order plans. (From now on, we will just call them "plans.") That is, we start with an empty plan. Then we consider ways of refining the plan until we come up with a complete plan that solves the problem. The actions in this search are not actions in the world, but actions on plans: adding a step to the plan, imposing an ordering that puts one action before another, and so on.

We will define the POP algorithm for partial-order planning. It is traditional to write out the POP algorithm as a stand-alone program, but we will instead formulate partial-order planning as an instance of a search problem. This allows us to focus on the plan refinement steps that can be applied, rather than worrying about how the algorithm explores the space. In fact, a wide variety of uninformed or heuristic search methods can be applied once the search problem is formulated.

Remember that the states of our search problem will be (mostly unfinished) plans. To avoid confusion with the states of the world, we will talk about plans rather than states. Each plan has the following four components, where the first two define the steps of the plan and the last two serve a bookkeeping function to determine how plans can be extended:

- A set of **actions** that make up the steps of the plan. These are taken from the set of actions in the planning problem. The "empty" plan contains just the $Start$ and $Finish$ actions. $Start$ has no preconditions and has as its effect all the literals in the initial state of the planning problem. $Finish$ has no effects and has as its preconditions the goal literals of the planning problem.

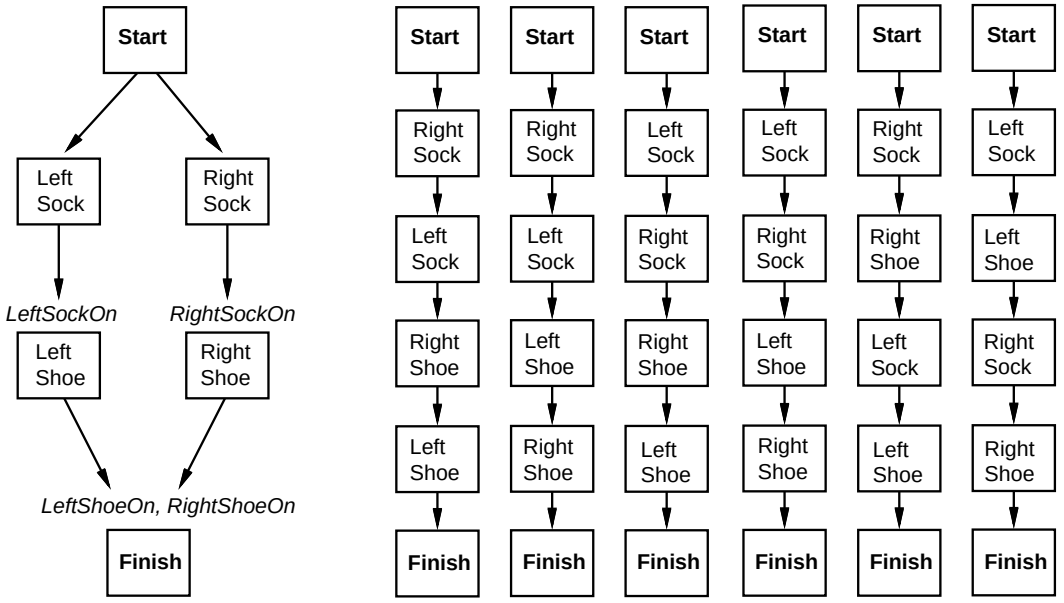

Figure 11.6 A partial-order plan for putting on shoes and socks, and the six corresponding linearizations into total-order plans.

ORDERING CONSTRAINTS

- A set of **ordering constraints**. Each ordering constraint is of the form $A \prec B$, which is read as "A before B" and means that action A must be executed sometime before action B, but not necessarily immediately before. The ordering constraints must describe a proper partial order. Any cycle—such as $A \prec B$ and $B \prec A$—represents a contradiction, so an ordering constraint cannot be added to the plan if it creates a cycle.

CAUSAL LINKS

ACHIEVES

- A set of **causal links**. A causal link between two actions A and B in the plan is written as $A \xrightarrow{p} B$ and is read as "A **achieves** p for B." For example, the causal link

$$RightSock \xrightarrow{RightSockOn} RightShoe$$

asserts that $RightSockOn$ is an effect of the $RightSock$ action and a precondition of $RightShoe$. It also asserts that $RightSockOn$ must remain true from the time of action $RightSock$ to the time of action $RightShoe$. In other words, the plan may not be extended by adding a new action C that **conflicts** with the causal link. An action C conflicts with $A \xrightarrow{p} B$ if C has the effect $\neg p$ and if C could (according to the ordering constraints) come after A and before B. Some authors call causal links **protection intervals**, because the link $A \xrightarrow{p} B$ protects p from being negated over the interval from A to B.

CONFLICTS

OPEN PRECONDITIONS

- A set of **open preconditions**. A precondition is open if it is not achieved by some action in the plan. Planners will work to reduce the set of open preconditions to the empty set, without introducing a contradiction.

For example, the final plan in Figure 11.6 has the following components (not shown are the ordering constraints that put every other action after $Start$ and before $Finish$):

Actions:$\{RightSock, RightShoe, LeftSock, LeftShoe, Start, Finish\}$
Orderings:$\{RightSock \prec RightShoe, LeftSock \prec LeftShoe\}$
Links:$\{RightSock \xrightarrow{RightSockOn} RightShoe, LeftSock \xrightarrow{LeftSockOn} LeftShoe,$
$RightShoe \xrightarrow{RightShoeOn} Finish, LeftShoe \xrightarrow{LeftShoeOn} Finish\}$
Open Preconditions:$\{\}$.

CONSISTENT PLAN

We define a **consistent plan** as a plan in which there are no cycles in the ordering constraints and no conflicts with the causal links. A consistent plan with no open preconditions is a **solution**. A moment's thought should convince the reader of the following fact: *every linearization of a partial-order solution is a total-order solution whose execution from the initial state will reach a goal state.* This means that we can extend the notion of "executing a plan" from total-order to partial-order plans. A partial-order plan is executed by repeatedly choosing *any* of the possible next actions. We will see in Chapter 12 that the flexibility available to the agent as it executes the plan can be very useful when the world fails to cooperate. The flexible ordering also makes it easier to combine smaller plans into larger ones, because each of the small plans can reorder its actions to avoid conflict with the other plans.

Now we are ready to formulate the search problem that POP solves. We will begin with a formulation suitable for propositional planning problems, leaving the first-order complications for later. As usual, the definition includes the initial state, actions, and goal test.

- The initial plan contains $Start$ and $Finish$, the ordering constraint $Start \prec Finish$, and no causal links and has all the preconditions in $Finish$ as open preconditions.
- The successor function arbitrarily picks one open precondition p on an action B and generates a successor plan for every possible consistent way of choosing an action A that achieves p. Consistency is enforced as follows:
  1. The causal link $A \xrightarrow{p} B$ and the ordering constraint $A \prec B$ are added to the plan. Action A may be an existing action in the plan or a new one. If it is new, add it to the plan and also add $Start \prec A$ and $A \prec Finish$.
  2. We resolve conflicts between the new causal link and all existing actions and between the action A (if it is new) and all existing causal links. A conflict between $A \xrightarrow{p} B$ and C is resolved by making C occur at some time outside the protection interval, either by adding $B \prec C$ or $C \prec A$. We add successor states for either or both if they result in consistent plans.
- The goal test checks whether a plan is a solution to the original planning problem. Because only consistent plans are generated, the goal test just needs to check that there are no open preconditions.

Remember that the actions considered by the search algorithms under this formulation are plan refinement steps rather than the real actions from the domain itself. The path cost is therefore irrelevant, strictly speaking, because the only thing that matters is the total cost of the real actions in the plan to which the path leads. Nonetheless, it *is* possible to specify a path cost function that reflects the real plan costs: we charge 1 for each real action added to

the plan and 0 for all other refinement steps. In this way, $g(n)$, where n is a plan, will be equal to the number of real actions in the plan. A heuristic estimate $h(n)$ can also be used.

At first glance, one might think that the successor function should include successors for *every* open p, not just for one of them. This would be redundant and inefficient, however, for the same reason that constraint satisfaction algorithms don't include successors for every possible variable: the order in which we consider open preconditions (like the order in which we consider CSP variables) is commutative. (See page 141.) Thus, we can choose an arbitrary ordering and still have a complete algorithm. Choosing the right ordering can lead to a faster search, but all orderings end up with the same set of candidate solutions.

A partial-order planning example

Now let's look at how POP solves the spare tire problem from Section 11.1. The problem description is repeated in Figure 11.7.

$Init(At(Flat, Axle) \wedge At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(Spare, Trunk),$
  PRECOND: $At(Spare, Trunk)$
  EFFECT: $\neg At(Spare, Trunk) \wedge At(Spare, Ground))$
$Action(Remove(Flat, Axle),$
  PRECOND: $At(Flat, Axle)$
  EFFECT: $\neg At(Flat, Axle) \wedge At(Flat, Ground))$
$Action(PutOn(Spare, Axle),$
  PRECOND: $At(Spare, Ground) \wedge \neg At(Flat, Axle)$
  EFFECT: $\neg At(Spare, Ground) \wedge At(Spare, Axle))$
$Action(LeaveOvernight,$
  PRECOND:
  EFFECT: $\neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk)$
  $\wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle))$

Figure 11.7 The simple flat tire problem description.

The search for a solution begins with the initial plan, containing a $Start$ action with the effect $At(Spare, Trunk) \wedge At(Flat, Axle)$ and a $Finish$ action with the sole precondition $At(Spare, Axle)$. Then we generate successors by picking an open precondition to work on (irrevocably) and choosing among the possible actions to achieve it. For now, we will not worry about a heuristic function to help with these decisions; we will make seemingly arbitrary choices. The sequence of events is as follows:

1. Pick the only open precondition, $At(Spare, Axle)$ of $Finish$. Choose the only applicable action, $PutOn(Spare, Axle)$.
2. Pick the $At(Spare, Ground)$ precondition of $PutOn(Spare, Axle)$. Choose the only applicable action, $Remove(Spare, Trunk)$ to achieve it. The resulting plan is shown in Figure 11.8.

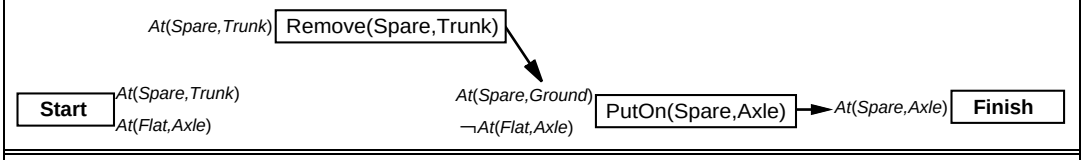

Figure 11.8 The incomplete partial-order plan for the tire problem, after choosing actions for the first two open preconditions. Boxes represent actions, with preconditions on the left and effects on the right. (Effects are omitted, except for that of the $Start$ action.) Dark arrows represent causal links protecting the proposition at the head of the arrow.

3. Pick the $\neg At(Flat, Axle)$ precondition of $PutOn(Spare, Axle)$. Just to be contrary, choose the $LeaveOvernight$ action rather than the $Remove(Flat, Axle)$ action. Notice that $LeaveOvernight$ also has the effect $\neg At(Spare, Ground)$, which means it conflicts with the causal link

$$Remove(Spare, Trunk) \overset{At(Spare,Ground)}{\longrightarrow} PutOn(Spare, Axle) .$$

To resolve the conflict we add an ordering constraint putting $LeaveOvernight$ before $Remove(Spare, Trunk)$. The resulting plan is shown in Figure 11.9. (Why does this resolve the conflict, and why is there no other way to resolve it?)

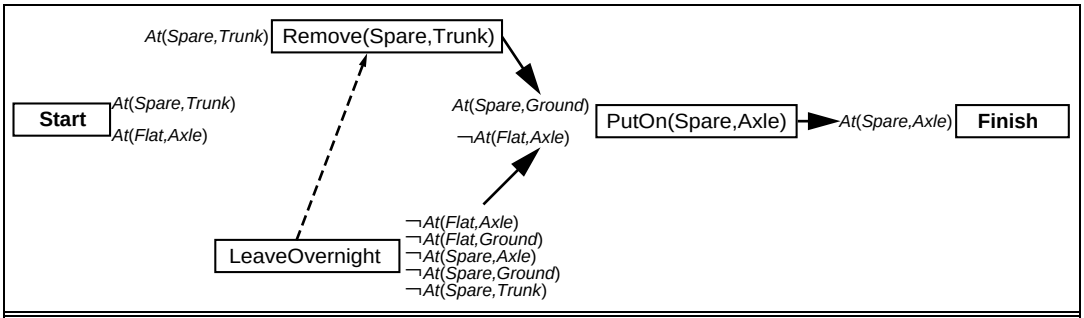

Figure 11.9 The plan after choosing $LeaveOvernight$ as the action for achieving $\neg At(Flat, Axle)$. To avoid a conflict with the causal link from $Remove(Spare, Trunk)$ that protects $At(Spare, Ground)$, $LeaveOvernight$ is constrained to occur before $Remove(Spare, Trunk)$, as shown by the dashed arrow.

4. The only remaining open precondition at this point is the $At(Spare, Trunk)$ precondition of the action $Remove(Spare, Trunk)$. The only action that can achieve it is the existing $Start$ action, but the causal link from $Start$ to $Remove(Spare, Trunk)$ is in conflict with the $\neg At(Spare, Trunk)$ effect of $LeaveOvernight$. This time there is no way to resolve the conflict with $LeaveOvernight$: we cannot order it before $Start$ (because nothing can come before $Start$), and we cannot order it after $Remove(Spare, Trunk)$ (because there is already a constraint ordering it before $Remove(Spare, Trunk)$). So we are forced to back up, remove the $Remove(Spare, Trunk)$ action and the last two causal links, and return to the state in Figure 11.8. In essence, the planner has proved that $LeaveOvernight$ doesn't work as a way to change a tire.

5. Consider again the $\neg At(Flat, Axle)$ precondition of $PutOn(Spare, Axle)$. This time, we choose $Remove(Flat, Axle)$.
6. Once again, pick the $At(Spare, Tire)$ precondition of $Remove(Spare, Trunk)$ and choose $Start$ to achieve it. This time there are no conflicts.
7. Pick the $At(Flat, Axle)$ precondition of $Remove(Flat, Axle)$, and choose $Start$ to achieve it. This gives us a complete, consistent plan—in other words a solution—as shown in Figure 11.10.

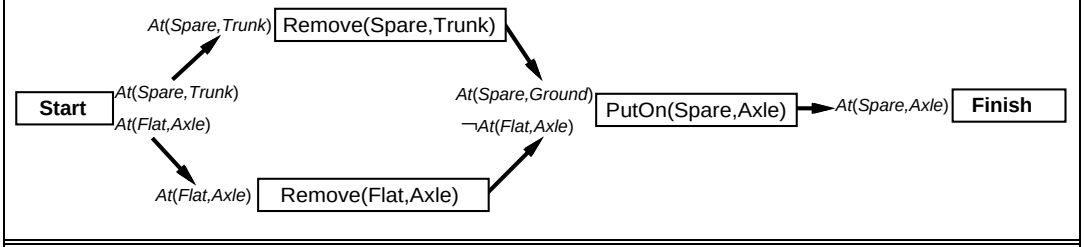

Figure 11.10 The final solution to the tire problem. Note that $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$ can be done in either order, as long as they are completed before the $PutOn(Spare, Axle)$ action.

Although this example is very simple, it illustrates some of the strengths of partial-order planning. First, the causal links lead to early pruning of portions of the search space that, because of irresolvable conflicts, contain no solutions. Second, the solution in Figure 11.10 is a partial-order plan. In this case the advantage is small, because there are only two possible linearizations; nonetheless, an agent might welcome the flexibility—for example, if the tire has to be changed in the middle of heavy traffic.

The example also points to some possible improvements that could be made. For example, there is duplication of effort: $Start$ is linked to $Remove(Spare, Trunk)$ before the conflict causes a backtrack and is then unlinked by backtracking even though it is not involved in the conflict. It is then relinked as the search continues. This is typical of chronological backtracking and might be mitigated by dependency-directed backtracking.

Partial-order planning with unbound variables

In this section, we consider the complications that can arise when POP is used with first-order action representations that include variables. Suppose we have a blocks world problem (Figure 11.4) with the open precondition $On(A, B)$ and the action

$$Action(Move(b, x, y),$$
$$\text{PRECOND:}On(b, x) \wedge Clear(b) \wedge Clear(y),$$
$$\text{EFFECT:}On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)) .$$

This action achieves $On(A, B)$ because the effect $On(b, y)$ unifies with $On(A, B)$ with the substitution $\{b/A, y/B\}$. We then apply this substitution to the action, yielding

$$Action(Move(A, x, B),$$
$$\text{PRECOND:}On(A, x) \wedge Clear(A) \wedge Clear(B),$$
$$\text{EFFECT:}On(A, B) \wedge Clear(x) \wedge \neg On(A, x) \wedge \neg Clear(B)) \ .$$

This leaves the variable x unbound. That is, the action says to move block A from *somewhere*, without yet saying whence. This is another example of the least commitment principle: we can delay making the choice until some other step in the plan makes it for us. For example, suppose we have $On(A, D)$ in the initial state. Then the $Start$ action can be used to achieve $On(A, x)$, binding x to D. This strategy of waiting for more information before choosing x is often more efficient than trying every possible value of x and backtracking for each one that fails.

The presence of variables in preconditions and actions complicates the process of detecting and resolving conflicts. For example, when $Move(A, x, B)$ is added to the plan, we will need a causal link

$$Move(A, x, B) \stackrel{On(A,B)}{\longrightarrow} Finish \ .$$

If there is another action M_2 with effect $\neg On(A, z)$, then M_2 conflicts only if z is B. To accommodate this possibility, we extend the representation of plans to include a set of **inequality constraints** of the form $z \neq X$ where z is a variable and X is either another variable or a constant symbol. In this case, we would resolve the conflict by adding $z \neq B$, which means that future extensions to the plan can instantiate z to any value except B. Anytime we apply a substitution to a plan, we must check that the inequalities do not contradict the substitution. For example, a substitution that includes x/y conflicts with the inequality constraint $x \neq y$. Such conflicts cannot be resolved, so the planner must backtrack.

INEQUALITY CONSTRAINTS

A more extensive example of POP planning with variables in the blocks world is given in Section 12.6.

Heuristics for partial-order planning

Compared with total-order planning, partial-order planning has a clear advantage in being able to decompose problems into subproblems. It also has a disadvantage in that it does not represent states directly, so it is harder to estimate how far a partial-order plan is from achieving a goal. At present, there is less understanding of how to compute accurate heuristics for partial-order planning than for total-order planning.

The most obvious heuristic is to count the number of distinct open preconditions. This can be improved by subtracting the number of open preconditions that match literals in the $Start$ state. As in the total-order case, this overestimates the cost when there are actions that achieve multiple goals and underestimates the cost when there are negative interactions between plan steps. The next section presents an approach that allows us to get much more accurate heuristics from a relaxed problem.

The heuristic function is used to choose which plan to refine. Given this choice, the algorithm generates successors based on the selection of a single open precondition to work

on. As in the case of variable selection on constraint satisfaction algorithms, this selection has a large impact on efficiency. The **most-constrained-variable** heuristic from CSPs can be adapted for planning algorithms and seems to work well. The idea is to select the open condition that can be satisfied in the *fewest* number of ways. There are two special cases of this heuristic. First, if an open condition cannot be achieved by any action, the heuristic will select it; this is a good idea because early detection of impossibility can save a great deal of work. Second, if an open condition can be achieved in only one way, then it should be selected because the decision is unavoidable and could provide additional constraints on other choices still to be made. Although full computation of the number of ways to satisfy each open condition is expensive and not always worthwhile, experiments show that handling the two special cases provides very substantial speedups.

11.4 PLANNING GRAPHS

PLANNING GRAPH

All of the heuristics we have suggested for total-order and partial-order planning can suffer from inaccuracies. This section shows how a special data structure called a **planning graph** can be used to give better heuristic estimates. These heuristics can be applied to any of the search techniques we have seen so far. Alternatively, we can extract a solution directly from the planning graph, using a specialized algorithm such as the one called GRAPHPLAN.

LEVELS

A planning graph consists of a sequence of **levels** that correspond to time steps in the plan, where level 0 is the initial state. Each level contains a set of literals and a set of actions. Roughly speaking, the literals are all those that *could* be true at that time step, depending on the actions executed at preceding time steps. Also roughly speaking, the actions are all those actions that *could* have their preconditions satisfied at that time step, depending on which of the literals actually hold. We say "roughly speaking" because the planning graph records only a restricted subset of the possible negative interactions among actions; therefore, it might be optimistic about the minimum number of time steps required for a literal to become true. Nonetheless, this number of steps in the planning graph provides a good estimate of how difficult it is to achieve a given literal from the initial state. More importantly, the planning graph is defined in such a way that it can be constructed very efficiently.

Planning graphs work only for propositional planning problems—ones with no variables. As we mentioned in Section 11.1, both STRIPS and ADL representations can be propositionalized. For problems with large numbers of objects, this could result in a very substantial blowup in the number of action schemata. Despite this, planning graphs have proved to be effective tools for solving hard planning problems.

We will illustrate planning graphs with a simple example. (More complex examples lead to graphs that won't fit on the page.) Figure 11.11 shows a problem, and Figure 11.12 shows its planning graph. We start with state level S_0, which represents the problem's initial state. We follow that with action level A_0, in which we place all the actions whose preconditions are satisfied in the previous level. Each action is connected to its preconditions in S_0 and its effects in S_1, in this case introducing new literals into S_1 that were not in S_0.

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg\, Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg\, Have(Cake)$
EFFECT: $Have(Cake)$

Figure 11.11 The "have cake and eat cake too" problem.

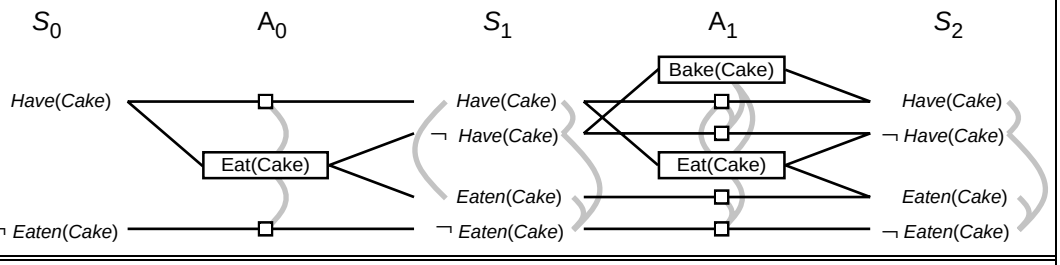

Figure 11.12 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions) and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines.

The planning graph needs a way to represent inaction as well as action. That is, it needs the equivalent of the frame axioms in situation calculus that allow a literal to remain true from one situation to the next if no action alters it. In a planning graph this is done with a set of **persistence actions**. For every positive and negative literal C, we add to the problem a persistence action with precondition C and effect C. Figure 11.12 shows one "real" action, $Eat(Cake)$ in A_0, along with two persistence actions drawn as small square boxes.

PERSISTENCE ACTIONS

Level A_0 contains all the actions that *could* occur in state S_0, but just as importantly it records conflicts between actions that would prevent them from occurring together. The gray lines in Figure 11.12 indicate **mutual exclusion** (or **mutex**) links. For example, $Eat(Cake)$ is mutually exclusive with the persistence of either $Have(Cake)$ or $\neg Eaten(Cake)$. We shall see shortly how mutex links are computed.

MUTUAL EXCLUSION
MUTEX

Level S_1 contains all the literals that could result from picking any subset of the actions in A_0. It also contains mutex links (gray lines) indicating literals that could not appear together, regardless of the choice of actions. For example, $Have(Cake)$ and $Eaten(Cake)$ are mutex: depending on the choice of actions in A_0, one or the other, but not both, could be the result. In other words, S_1 represents multiple states, just as regression state-space search does, and the mutex links are constraints that define the set of possible states.

We continue in this way, alternating between state level S_i and action level A_i until we reach a level where two consecutive levels are identical. At this point, we say that the graph

LEVELED OFF

has **leveled off**. Every subsequent level will be identical, so further expansion is unnecessary.

What we end up with is a structure where every A_i level contains all the actions that are applicable in S_i, along with constraints saying which pairs of actions cannot both be executed. Every S_i level contains all the literals that could result from any possible choice of actions in A_{i-1}, along with constraints saying which pairs of literals are not possible. It is important to note that the process of constructing the planning graph does *not* require choosing among actions, which would entail combinatorial search. Instead, it just records the impossibility of certain choices using mutex links. The complexity of constructing the planning graph is a low-order polynomial in the number of actions and literals, whereas the state space is exponential in the number of literals.

We now define mutex links for both actions and literals. A mutex relation holds between two *actions* at a given level if any of the following three conditions holds:

- *Inconsistent effects:* one action negates an effect of the other. For example $Eat(Cake)$ and the persistence of $Have(Cake)$ have inconsistent effects because they disagree on the effect $Have(Cake)$.
- *Interference:* one of the effects of one action is the negation of a precondition of the other. For example $Eat(Cake)$ interferes with the persistence of $Have(Cake)$ by negating its precondition.
- *Competing needs:* one of the preconditions of one action is mutually exclusive with a precondition of the other. For example, $Bake(Cake)$ and $Eat(Cake)$ are mutex because they compete on the value of the $Have(Cake)$ precondition.

A mutex relation holds between two *literals* at the same level if one is the negation of the other or if each possible pair of actions that could achieve the two literals is mutually exclusive. This condition is called *inconsistent support*. For example, $Have(Cake)$ and $Eaten(Cake)$ are mutex in S_1 because the only way of achieving $Have(Cake)$, the persistence action, is mutex with the only way of achieving $Eaten(Cake)$, namely $Eat(Cake)$. In S_2 the two literals are not mutex because there are new ways of achieving them, such as $Bake(Cake)$ and the persistence of $Eaten(Cake)$, that are not mutex.

Planning graphs for heuristic estimation

A planning graph, once constructed, is a rich source of information about the problem. For example, *a literal that does not appear in the final level of the graph cannot be achieved by any plan.* This observation can be used in backward search as follows: any state containing an unachievable literal has a cost $h(n) = \infty$. Similarly, in partial-order planning, any plan with an unachievable open condition has $h(n) = \infty$.

This idea can be made more general. We can estimate the cost of achieving any goal literal as the level at which it first appears in the planning graph. We will call this the **level cost** of the goal. In Figure 11.12, $Have(Cake)$ has level cost 0 and $Eaten(Cake)$ has level cost 1. It is easy to show (Exercise 11.9) that these estimates are admissible for the individual goals. The estimate might not be very good, however, because planning graphs allow several actions at each level whereas the heuristic counts just the level and not the number of actions. For this reason, it is common to use a **serial planning graph** for computing heuristics. A

LEVEL COST

SERIAL PLANNING GRAPH

serial graph insists that only one action can actually occur at any given time step; this is done by adding mutex links between every pair of actions except persistence actions. Level costs extracted from serial graphs are often quite reasonable estimates of actual costs.

MAX-LEVEL
LEVEL SUM
SET-LEVEL

To estimate the cost of a conjunction of goals, there are three simple approaches. The **max-level** heuristic simply takes the maximum level cost of any of the goals; this is admissible, but not necessarily very accurate. The **level sum** heuristic, following the subgoal independence assumption, returns the sum of the level costs of the goals; this is inadmissible but works very well in practice for problems that are largely decomposable. It is much more accurate than the number-of-unsatisfied-goals heuristic from Section 11.2. For our problem, the heuristic estimate for the conjunctive goal $Have(Cake) \wedge Eaten(Cake)$ will be $0+1=1$, whereas the correct answer is 2. Moreover, if we eliminated the $Bake(Cake)$ action, the estimate would still be 1, but the conjunctive goal would be impossible. Finally, the **set-level** heuristic finds the level at which all the literals in the conjunctive goal appear in the planning graph without any pair of them being mutually exclusive. This heuristic gives the correct values of 2 for our original problem and infinity for the problem without $Bake(Cake)$. It dominates the max-level heuristic and works extremely well on tasks in which there is a good deal of interaction among subplans.

As a tool for generating accurate heuristics, we can view the planning graph as a relaxed problem that is efficiently soluble. To understand the nature of the relaxed problem, we need to understand exactly what it means for a literal g to appear at level S_i in the planning graph. Ideally, we would like it to be a guarantee that there exists a plan with i action levels that achieves g, and also that if g does not appear that there is no such plan. Unfortunately, making that guarantee is as difficult as solving the original planning problem. So the planning graph makes the second half of the guarantee (if g does not appear, there is no plan), but if g does appear, then all the planning graph promises is that there is a plan that *possibly* achieves g and has no "obvious" flaws. An obvious flaw is defined as a flaw that can be detected by considering two actions or two literals at a time—in other words, by looking at the mutex relations. There could be more subtle flaws involving three, four, or more actions, but experience has shown that it is not worth the computational effort to keep track of these possible flaws. This is similar to the lesson learned from constraint satisfaction problems that it is often worthwhile to compute 2-consistency before searching for a solution, but less often worthwhile to compute 3-consistency or higher. (See Section 5.2.)

The GRAPHPLAN algorithm

This subsection shows how to extract a plan directly from the planning graph, rather than just using the graph to provide a heuristic. The GRAPHPLAN algorithm (Figure 11.13) has two main steps, which alternate within a loop. First, it checks whether all the goal literals are present in the current level with no mutex links between any pair of them. If this is the case, then a solution *might* exist within the current graph, so the algorithm tries to extract that solution. Otherwise, it expands the graph by adding the actions for the current level and the state literals for the next level. The process continues until either a solution is found or it is learned that no solution exists.

```
function GRAPHPLAN(problem) returns solution or failure

  graph ← INITIAL-PLANNING-GRAPH(problem)
  goals ← GOALS[problem]
  loop do
    if goals all non-mutex in last level of graph then do
      solution ← EXTRACT-SOLUTION(graph, goals, LENGTH(graph))
      if solution ≠ failure then return solution
      else if NO-SOLUTION-POSSIBLE(graph) then return failure
    graph ← EXPAND-GRAPH(graph, problem)
```

Figure 11.13 The GRAPHPLAN algorithm. GRAPHPLAN alternates between a solution extraction step and a graph expansion step. EXTRACT-SOLUTION looks for whether a plan can be found, starting at the end and searching backwards. EXPAND-GRAPH adds the actions for the current level and the state literals for the next level.

Let us now trace the operation of GRAPHPLAN on the spare tire problem from Section 11.1. The entire graph is shown in Figure 11.14. The first line of GRAPHPLAN initializes the planning graph to a one-level (S_0) graph consisting of the five literals from the initial state. The goal literal $At(Spare, Axle)$ is not present in S_0, so we need not call EXTRACT-SOLUTION—we are certain that there is no solution yet. Instead, EXPAND-GRAPH adds the three actions whose preconditions exist at level S_0 (i.e., all the actions except $PutOn(Spare, Axle)$), along with persistence actions for all the literals in S_0. The effects of the actions are added at level S_1. EXPAND-GRAPH then looks for mutex relations and adds them to the graph.

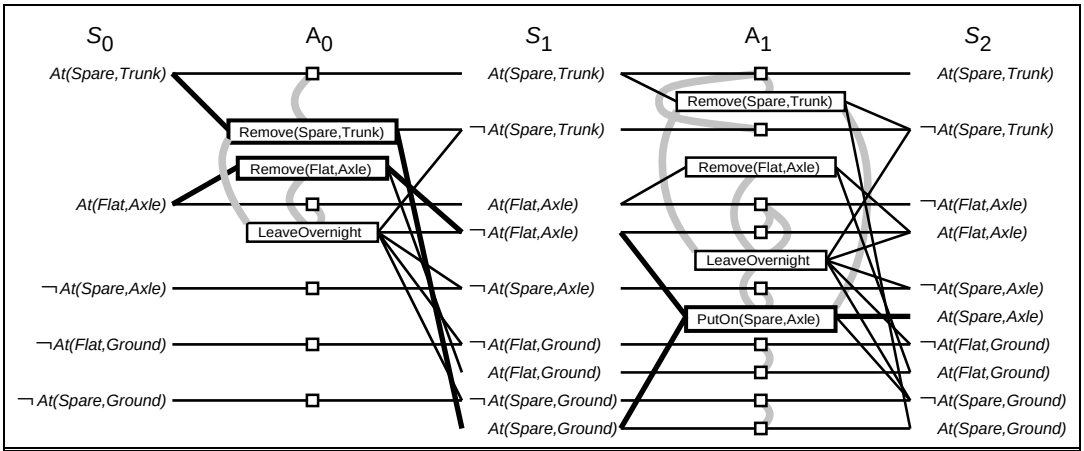

Figure 11.14 The planning graph for the spare tire problem after expansion to level S_2. Mutex links are shown as gray lines. Only some representative mutexes are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.

$At(Spare, Axle)$ is still not present in S_1, so again we do not call EXTRACT-SOLUTION. The call to EXPAND-GRAPH gives us the planning graph shown in Figure 11.14. Now that we have the full complement of actions, it is worthwhile to look at some of the examples of mutex relations and their causes:

- *Inconsistent effects:* $Remove(Spare, Trunk)$ is mutex with $LeaveOvernight$ because one has the effect $At(Spare, Ground)$ and the other has its negation.
- *Interference:* $Remove(Flat, Axle)$ is mutex with $LeaveOvernight$ because one has the precondition $At(Flat, Axle)$ and the other has its negation as an effect.
- *Competing needs:* $PutOn(Spare, Axle)$ is mutex with $Remove(Flat, Axle)$ because one has $At(Flat, Axle)$ as a precondition and the other has its negation.
- *Inconsistent support:* $At(Spare, Axle)$ is mutex with $At(Flat, Axle)$ in S_2 because the only way of achieving $At(Spare, Axle)$ is by $PutOn(Spare, Axle)$, and that is mutex with the persistence action that is the only way of achieving $At(Flat, Axle)$. Thus, the mutex relations detect the immediate conflict that arises from trying to put two objects in the same place at the same time.

This time, when we go back to the start of the loop, all the literals from the goal are present in S_2, and none of them is mutex with any other. That means that a solution might exist, and EXTRACT-SOLUTION will try to find it. In essence, EXTRACT-SOLUTION solves a Boolean CSP whose variables are the actions at each level, and the values for each variable are *in* or *out* of the plan. We can use standard CSP algorithms for this, or we can define EXTRACT-SOLUTION as a search problem, where each state in the search contains a pointer to a level in the planning graph and a set of unsatisfied goals. We define this search problem as follows:

- The initial state is the last level of the planning graph, S_n, along with the set of goals from the planning problem.
- The actions available in a state at level S_i are to select any conflict-free subset of the actions in A_{i-1} whose effects cover the goals in the state. The resulting state has level S_{i-1} and has as its set of goals the preconditions for the selected set of actions. By "conflict-free," we mean a set of actions such that no two of them are mutex, and no two of their preconditions are mutex.
- The goal is to reach a state at level S_0 such that all the goals are satisfied.
- The cost of each action is 1.

For this particular problem, we start at S_2 with the goal $At(Spare, Axle)$. The only choice we have for achieving the goal set is $PutOn(Spare, Axle)$. That brings us to a search state at S_1 with goals $At(Spare, Ground)$ and $\neg At(Flat, Axle)$. The former can be achieved only by $Remove(Spare, Trunk)$, and the latter by either $Remove(Flat, Axle)$ or $LeaveOvernight$. But $LeaveOvernight$ is mutex with $Remove(Spare, Trunk)$, so the only solution is to choose $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$. That brings us to a search state at S_0 with the goals $At(Spare, Trunk)$ and $At(Flat, Axle)$. Both of these are present in the state, so we have a solution: the actions $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$ in level A_0, followed by $PutOn(Spare, Axle)$ in A_1.

We know that planning is PSPACE-complete and that constructing the planning graph takes polynomial time, so it must be the case that solution extraction is intractable in the worst case. Therefore, we will need some heuristic guidance for choosing among actions during the backward search. One approach that works well in practice is a greedy algorithm based on the level cost of the literals. For any set of goals, we proceed in the following order:

1. Pick first the literal with the highest level cost.
2. To achieve that literal, choose the action with the easiest preconditions first. That is, choose an action such that the sum (or maximum) of the level costs of its preconditions is smallest.

Termination of GRAPHPLAN

So far, we have skated over the question of termination. If a problem has no solution, can we be sure that GRAPHPLAN will not loop forever, extending the planning graph at each iteration? The answer is yes, but the full proof is beyond the scope of this book. Here, we outline just the main ideas, particularly the ones that shed light on planning graphs in general.

The first step is to notice that certain properties of planning graphs are monotonically increasing or decreasing. "X increases monotonically" means that the set of Xs at level $i + 1$ is a superset (not necessarily proper) of the set at level i. The properties are as follows:

- *Literals increase monotonically:* Once a literal appears at a given level, it will appear at all subsequent levels. This is because of the persistence actions; once a literal shows up, persistence actions cause it to stay forever.
- *Actions increase monotonically:* Once an action appears at a given level, it will appear at all subsequent levels. This is a consequence of literals' increasing; if the preconditions of an action appear at one level, they will appear at subsequent levels, and thus so will the action.
- *Mutexes decrease monotonically:* If two actions are mutex at a given level A_i, then they will also be mutex for all *previous* levels at which they both appear. The same holds for mutexes between literals. It might not always appear that way in the figures, because the figures have a simplification: they display neither literals that cannot hold at level S_i nor actions that cannot be executed at level A_i. We can see that "mutexes decrease monotonically" is true if you consider that these invisible literals and actions are mutex with everything.

  The proof is a little complex, but can be handled by cases: if actions A and B are mutex at level A_i, it must be because of one of the three types of mutex. The first two, inconsistent effects and interference, are properties of the actions themselves, so if the actions are mutex at A_i, they will be mutex at every level. The third case, competing needs, depends on conditions at level S_i: that level must contain a precondition of A that is mutex with a precondition of B. Now, these two preconditions can be mutex if they are negations of each other (in which case they would be mutex in every level) or if all actions for achieving one are mutex with all actions for achieving the other. But we already know that the available actions are increasing monotonically, so by induction, the mutexes must be decreasing.

Because the actions and literals increase and the mutexes decrease, and because there are only a finite number of actions and literals, every planning graph will eventually level off—all subsequent levels will be identical. Once a graph has leveled off, if it is missing one of the goals of the problem, or if two of the goals are mutex, then the problem can never be solved, and we can stop the GRAPHPLAN algorithm and return failure. If the graph levels off with all goals present and nonmutex, but EXTRACT-SOLUTION fails to find a solution, then we might have to extend the graph again a finite number of times, but eventually we can stop. This aspect of termination is more complex and is not covered here.

11.5 PLANNING WITH PROPOSITIONAL LOGIC

We saw in Chapter 10 that planning can be done by proving a theorem in situation calculus. That theorem says that, given the initial state and the successor-state axioms that describe the effects of actions, the goal will be true in a situation that results from a certain action sequence. As early as 1969, this approach was thought to be too inefficient for finding interesting plans. Recent developments in efficient reasoning algorithms for propositional logic (see Chapter 7) have generated renewed interest in planning as logical reasoning.

The approach we take in this section is based on testing the **satisfiability** of a logical sentence rather than on proving a theorem. We will be finding models of propositional sentences that look like this:

initial state $\wedge$ *all possible action descriptions* $\wedge$ *goal* .

The sentence will contain proposition symbols corresponding to every possible action occurrence; a model that satisfies the sentence will assign $true$ to the actions that are part of a correct plan and $false$ to the others. An assignment that corresponds to an incorrect plan will not be a model, because it will be inconsistent with the assertion that the goal is true. If the planning problem is unsolvable, then the sentence will be unsatisfiable.

Describing planning problems in propositional logic

The process we will follow to translate STRIPS problems into propositional logic is a textbook example (so to speak) of the knowledge representation cycle: We will begin with what seems to be a reasonable set of axioms, we will find that these axioms allow for spurious unintended models, and we will write more axioms.

Let us begin with a very simple air transport problem. In the initial state (time 0), plane P_1 is at SFO and plane P_2 is at JFK. The goal is to have P_1 at JFK and P_2 at SFO; that is, the planes are to change places. First, we will need distinct proposition symbols for assertions about each time step. We will use superscripts to denote the time step, as in Chapter 7. Thus, the initial state will be written as

$$At(P_1, SFO)^0 \wedge At(P_2, JFK)^0 \ .$$

(Remember that $At(P_1, SFO)^0$ is an atomic symbol.) Because propositional logic has no closed-world assumption, we must also specify the propositions that are *not* true in the initial

state. If some propositions are unknown in the initial state, then they can be left unspecified (the **open world assumption**). In this example we specify:

$$\neg At(P_1, JFK)^0 \wedge \neg At(P_2, SFO)^0 \ .$$

The goal itself must be associated with a particular time step. Since we do not know *a priori* how many steps it takes to achieve the goal, we can try asserting that the goal is true in the initial state, time $T = 0$. That is, we assert $At(P_1, JFK)^0 \wedge At(P_2, SFO)^0$. If that fails, we try again with $T = 1$, and so on until we reach the minimum feasible plan length. For each value of T, the knowledge base will include only sentences covering the time steps from 0 up to T. To ensure termination, an arbitrary upper limit, $T_{\max}$, is imposed. This algorithm is shown in Figure 11.15. An alternative approach that avoids multiple solution attempts is discussed in Exercise 11.17.

```
function SATPLAN(problem, T_max) returns solution or failure
  inputs: problem, a planning problem
          T_max, an upper limit for plan length

  for T = 0 to T_max do
    cnf, mapping ← TRANSLATE-TO-SAT(problem, T)
    assignment ← SAT-SOLVER(cnf)
    if assignment is not null then
      return EXTRACT-SOLUTION(assignment, mapping)
  return failure
```

Figure 11.15 The SATPLAN algorithm. The planning problem is translated into a CNF sentence in which the goal is asserted to hold at a fixed time step T and axioms are included for each time step up to T. (Details of the translation are given in the text.) If the satisfiability algorithm finds a model, then a plan is extracted by looking at those proposition symbols that refer to actions and are assigned *true* in the model. If no model exists, then the process is repeated with the goal moved one step later.

The next issue is how to encode action descriptions in propositional logic. The most straightforward approach is to have one proposition symbol for each action occurrence; for example, $Fly(P_1, SFO, JFK)^0$ is true if plane P_1 flies from SFO to JFK at time 0. As in Chapter 7, we write propositional versions of the successor-state axioms developed for the situation calculus in Chapter 10. For example, we have

$$\begin{aligned} At(P_1, JFK)^1 \ \Leftrightarrow \ & (At(P_1, JFK)^0 \wedge \neg(Fly(P_1, JFK, SFO)^0 \wedge At(P_1, JFK)^0)) \\ & \vee (Fly(P_1, SFO, JFK)^0 \wedge At(P_1, SFO)^0) \ . \end{aligned} \tag{11.1}$$

That is, plane P_1 will be at JFK at time 1 if it was at JFK at time 0 and didn't fly away, or it was at SFO at time 0 and flew to JFK. We need one such axiom for each plane, airport, and time step. Moreover, each additional airport adds another way to travel to or from a given airport and hence adds more disjuncts to the right-hand side of each axiom.

With these axioms in place, we can run the satisfiability algorithm to find a plan. There ought to be a plan that achieves the goal at time $T = 1$, namely, the plan in which the two

planes swap places. Now, suppose the KB is

$$\textit{initial state} \wedge \textit{successor-state axioms} \wedge \textit{goal}^1 \ , \tag{11.2}$$

which asserts that the goal is true at time $T = 1$. You can check that the assignment in which

$$Fly(P_1, SFO, JFK)^0 \text{ and } Fly(P_2, JFK, SFO)^0$$

are true and all other action symbols are false is a model of the KB. So far, so good. Are there other possible models that the satisfiability algorithm might return? Indeed, yes. Are all these other models satisfactory plans? Alas, no. Consider the rather silly plan specified by the action symbols

$$Fly(P_1, SFO, JFK)^0 \text{ and } Fly(P_1, JFK, SFO)^0 \text{ and } Fly(P_2, JFK, SFO)^0 \ .$$

This plan is silly because plane P_1 starts at SFO, so the action $Fly(P_1, JFK, SFO)^0$ is infeasible. Nonetheless, the plan *is* a model of the sentence in Equation (11.2)! That is, it is consistent with everything we have said so far about the problem. To understand why, we need to look more carefully at what the successor-state axioms (such as Equation (11.1)) say about actions whose preconditions are not satisfied. The axioms *do* predict correctly that nothing will happen when such an action is executed (see Exercise 11.15), but they do *not* say that the action cannot be executed! To avoid generating plans with illegal actions, we must add **precondition axioms** stating that an action occurrence requires the preconditions to be satisfied.[6] For example, we need

PRECONDITION AXIOMS

$$Fly(P_1, JFK, SFO)^0 \ \Rightarrow \ At(P_1, JFK)^0 \ .$$

Because $At(P_1, JFK)^0$ is stated to be false in the initial state, this axiom ensures that every model also has $Fly(P_1, JFK, SFO)^0$ set to false. With the addition of precondition axioms, there is exactly one model that satisfies all of the axioms when the goal is to be achieved at time 1, namely the model in which plane P_1 flies to JFK and plane P_2 flies to SFO. Notice that this solution has two parallel actions, just as with GRAPHPLAN or POP.

More surprises emerge when we add a third airport, LAX. Now, each plane has two actions that are legal in each state. When we run the satisfiability algorithm, we find that a model with $Fly(P_1, SFO, JFK)^0$ and $Fly(P_2, JFK, SFO)^0$ and $Fly(P_2, JFK, LAX)^0$ satisfies all the axioms. That is, the successor-state and precondition axioms allow a plane to fly to two destinations at once! The preconditions for the two flights by P_2 are satisfied in the initial state; the successor-state axioms say that P_2 will be at SFO *and* LAX at time 1; so the goal is satisfied. Clearly, we must add more axioms to eliminate these spurious solutions. One approach is to add **action exclusion axioms** that prevent simultaneous actions. For example, we can insist on complete exclusion by adding all possible axioms of the form

ACTION EXCLUSION AXIOMS

$$\neg(Fly(P_2, JFK, SFO)^0 \wedge Fly(P_2, JFK, LAX)^0) \ .$$

These axioms ensure that no two actions can occur at the same time. They eliminate all spurious plans, but also force every plan to be totally ordered. This loses the flexibility of partially ordered plans; also, by increasing the number of time steps in the plan, computation time may be lengthened.

[6] Notice that the addition of precondition axioms means that we need not include preconditions for actions in the successor-state axioms.

Instead of complete exclusion, we can require only partial exclusion—that is, rule out simultaneous actions only if they interfere with each other. The conditions are the same as those for mutex actions: two actions cannot occur simultaneously if one negates a precondition or effect of the other. For example, $Fly(P_2, JFK, SFO)^0$ and $Fly(P_2, JFK, LAX)^0$ cannot both occur, because each negates the precondition of the other; on the other hand, $Fly(P_1, SFO, JFK)^0$ and $Fly(P_2, JFK, SFO)^0$ can occur together because the two planes do not interfere. Partial exclusion eliminates spurious plans without forcing a total ordering.

Exclusion axioms sometimes seem a rather blunt instrument. Instead of saying that a plane cannot fly to two airports at the same time, we might simply insist that no object can be in two places at once:

$$\forall\, p, x, y, t \;\; x \neq y \;\Rightarrow\; \neg(At(p, x)^t \wedge At(p, y)^t) \;.$$

STATE CONSTRAINTS

This fact, combined with the successor-state axioms, *implies* that a plane cannot fly to two airports at the same time. Facts such as this are called **state constraints**. In propositional logic, of course, we have to write out all the ground instances of each state constraint. For the airport problem, the state constraint suffices to rule out all spurious plans. State constraints are often much more compact than action exclusion axioms, but they are not always easy to derive from the original STRIPS description of a problem.

To summarize, planning as satisfiability involves finding models for a sentence containing the initial state, the goal, the successor-state axioms, the precondition axioms, and either the action exclusion axioms or the state constraints. It can be shown that this collection of axioms is sufficient, in the sense that there are no longer any spurious "solutions." Any model satisfying the propositional sentence will be a valid plan for the original problem—that is, every linearization of the plan is a legal sequence of actions that reaches the goal.

Complexity of propositional encodings

The principal drawback of the propositional approach is the sheer size of the propositional knowledge base that is generated from the original planning problem. For example, the action schema $Fly(p, a_1, a_2)$ becomes $T \times |Planes| \times |Airports|^2$ different proposition symbols. In general, the total number of action symbols is bounded by $T \times |Act| \times |O|^P$, where $|Act|$ is the number of action schemata, $|O|$ is the number of objects in the domain, and P is the maximum arity (number of arguments) of any action schema. The number of clauses is larger still. For example, with 10 time steps, 12 planes, and 30 airports, the complete action exclusion axiom has 583 million clauses.

Because the number of action symbols is exponential in the arity of the action schema, one answer might be to try to reduce the arity. We can do this by borrowing an idea from semantic networks (Chapter 10). Semantic networks use only binary predicates; predicates with more arguments are reduced to a set of binary predicates that describe each argument separately. Applying this idea to an action symbol such as $Fly(P_1, SFO, JFK)^0$, we obtain three new symbols:

$Fly_1(P_1)^0$: plane P_1 flew at time 0
$Fly_2(SFO)^0$: the origin of the flight was SFO
$Fly_3(JFK)^0$: the destination of the flight was JFK .

SYMBOL SPLITTING

This process, called **symbol splitting**, eliminates the need for an exponential number of symbols. Now we only need $T \times |Act| \times P \times |O|$.

Symbol splitting by itself can reduce the number of symbols, but does not automatically reduce the number of axioms in the KB. That is, if each action symbol in each clause were simply replaced by a conjunction of three symbols, then the total size of the KB would remain roughly the same. Symbol splitting actually does reduce the size of the KB because some of the split symbols will be irrelevant to certain axioms and can be omitted. For example, consider the successor-state axiom in Equation (11.1), modified to include LAX and to omit action preconditions (which will be covered by separate precondition axioms):

$$At(P_1, JFK)^1 \Leftrightarrow (At(P_1, JFK)^0 \wedge \neg Fly(P_1, JFK, SFO)^0 \wedge \neg Fly(P_1, JFK, LAX)^0) \\ \vee Fly(P_1, SFO, JFK)^0 \vee Fly(P_1, LAX, JFK)^0 .$$

The first condition says that P_1 will be at JFK if it was there at time 0 and didn't fly from JFK to any other city, no matter which one; the second says it will be there if it flew to JFK from another city, no matter which one. Using the split symbols, we can simply omit the argument whose value does not matter:

$$At(P_1, JFK)^1 \; Leftrightarrow \; (At(P_1, JFK)^0 \wedge \neg(Fly_1(P_1)^0 \wedge Fly_2(JFK)^0)) \\ \vee (Fly_1(P_1)^0 \wedge Fly_3(JFK)^0) .$$

Notice that SFO and LAX are no longer mentioned in the axiom. More generally, the split action symbols now allow the size of each successor-state axiom to be independent of the number of airports. Similar reductions occur with the precondition axioms and action exclusion axioms (see Exercise 11.16). For the case described earlier with 10 time steps, 12 planes, and 30 airports, the complete action exclusion axiom is reduced from 583 million clauses to 9,360 clauses.

There is one drawback: the split-symbol representation does not allow for parallel actions. Consider the two parallel actions $Fly(P_1, SFO, JFK)^0$ and $Fly(P_2, JFK, SFO)^0$. Converting to the split representation, we have

$$Fly_1(P_1)^0 \wedge Fly_2(SFO)^0 \wedge Fly_3(JFK)^0 \wedge \\ Fly_1(P_2)^0 \wedge Fly_2(JFK)^0 \wedge Fly_3(SFO)^0 .$$

It is no longer possible to determine what happened! We know that P_1 and P_2 flew, but we cannot identify the origin and destination of each flight. This means that a complete action exclusion axiom must be used, with the drawbacks noted previously.

Planners based on satisfiability can handle large planning problems—for example, finding optimal 30-step solutions to blocks-world planning problems with dozens of blocks. The size of the propositional encoding and the cost of solution are highly problem-dependent, but in most cases the memory required to store the propositional axioms is the bottleneck. One interesting finding from this work has been that backtracking algorithms such as DPLL are often better at solving planning problems than local search algorithms such as WALKSAT. This is because the majority of the propositional axioms are Horn clauses, which are handled efficiently by the unit propagation technique. This observation has led to the development of hybrid algorithms combining some random search with backtracking and unit propagation.

11.6 ANALYSIS OF PLANNING APPROACHES

Planning is an area of great current interest within AI. One reason for this is that it combines the two major areas of AI we have covered so far: *search* and *logic*. That is, a planner can be seen either as a program that searches for a solution or as one that (constructively) proves the existence of a solution. The cross-fertilization of ideas from the two areas has led to both improvements in performance amounting to several orders of magnitude in the last decade and an increased use of planners in industrial applications. Unfortunately, we do not yet have a clear understanding of which techniques work best on which kinds of problems. Quite possibly, new techniques will emerge that dominate existing methods.

Planning is foremost an exercise in controlling combinatorial explosion. If there are p primitive propositions in a domain, then there are 2^p states. For complex domains, p can grow quite large. Consider that objects in the domain have properties ($Location$, $Color$, etc.) and relations (At, On, $Between$, etc.). With d objects in a domain with ternary relations, we get 2^{d^3} states. We might conclude that, in the worst case, planning is hopeless.

Against such pessimism, the divide-and-conquer approach can be a powerful weapon. In the best case—full decomposability of the problem—divide-and-conquer offers an exponential speedup. Decomposability is destroyed, however, by negative interactions between actions. Partial-order planners deal with this with causal links, a powerful representational approach, but unfortunately each conflict must be resolved with a choice (put the conflicting action before or after the link), and the choices can multiply exponentially. GRAPHPLAN avoids these choices during the graph construction phase, using mutex links to record conflicts without actually making a choice as to how to resolve them. SATPLAN represents a similar range of mutex relations, but does so by using the general CNF form rather than a specific data structure. How well this works depends on the SAT solver used.

SERIALIZABLE SUBGOALS

Sometimes it is possible to solve a problem efficiently by recognizing that negative interactions can be ruled out. We say that a problem has **serializable subgoals** if there exists an order of subgoals such that the planner can achieve them in that order, without having to undo any of the previously achieved subgoals. For example, in the blocks world, if the goal is to build a tower (e.g., A on B, which in turn is on C, which in turn is on the $Table$), then the subgoals are serializable bottom to top: if we first achieve C on $Table$, we will never have to undo it while we are achieving the other subgoals. A planner that uses the bottom-to-top trick can solve any problem in the blocks world domain without backtracking (although it might not always find the shortest plan).

As a more complex example, for the Remote Agent planner which commanded NASA's Deep Space One spacecraft, it was determined that the propositions involved in commanding a spacecraft are serializable. This is perhaps not too surprising, because a spacecraft is designed by its engineers to be as easy as possible to control (subject to other constraints). Taking advantage of the serialized ordering of goals, the Remote Agent planner was able to eliminate most of the search. This meant that it was fast enough to control the spacecraft in real time, something previously considered impossible.

There is more than one way to control combinatorial explosions. We saw in Chapter 5 that there are many techniques for controlling backtracking in constraint satisfaction problems (CSPs), such as dependency-directed backtracking. All of these techniques can be applied to planning. For example, extracting a solution from a planning graph can be formulated as a Boolean CSP whose variables state whether a given action should occur at a given time. The CSP can be solved using any of the algorithms in Chapter 5, such as min-conflicts. A closely related method, used in the BLACKBOX system, is to convert the planning graph into a CNF expression and then extract a plan by using a SAT solver. This approach seems to work better than SATPLAN, presumably because the planning graph has already eliminated many of the impossible states and actions from the problem. It also works better than GRAPHPLAN, presumably because a satisfiability search such as WALKSAT has much greater flexibility than the strict backtracking search that GRAPHPLAN uses.

There is no doubt that planners such as GRAPHPLAN, SATPLAN, and BLACKBOX have moved the field of planning forward, both by raising the level of performance of planning systems and by clarifying the representational and combinatorial issues involved. These methods are, however, inherently propositional and thus are limited in the domains they can express. (For example, logistics problems with a few dozen objects and locations can require gigabytes of storage for the corresponding CNF expressions.) It seems likely that first-order representations and algorithms will be required if further progress is to occur, although structures such as planning graphs will continue to be useful as a source of heuristics.

11.7 SUMMARY

In this chapter, we defined the problem of planning in deterministic, fully observable environments. We described the principal representations used for planning problems and several algorithmic approaches for solving them. The points to remember are:

- Planning systems are problem-solving algorithms that operate on explicit propositional (or first-order) representations of states and actions. These representations make possible the derivation of effective heuristics and the development of powerful and flexible algorithms for solving problems.
- The STRIPS language describes actions in terms of their preconditions and effects and describes the initial and goal states as conjunctions of positive literals. The ADL language relaxes some of these constraints, allowing disjunction, negation, and quantifiers.
- State-space search can operate in the forward direction (**progression**) or the backward direction (**regression**). Effective heuristics can be derived by making a subgoal independence assumption and by various relaxations of the planning problem.
- Partial-order planning (POP) algorithms explore the space of plans without committing to a totally ordered sequence of actions. They work back from the goal, adding actions to the plan to achieve each subgoal. They are particularly effective on problems amenable to a divide-and-conquer approach.

- A **planning graph** can be constructed incrementally, starting from the initial state. Each layer contains a superset of all the literals or actions that could occur at that time step and encodes mutual exclusion, or **mutex**, relations among literals or actions that cannot co-occur. Planning graphs yield useful heuristics for state-space and partial-order planners and can be used directly in the GRAPHPLAN algorithm.
- The GRAPHPLAN algorithm processes the planning graph, using a backward search to extract a plan. It allows for some partial ordering among actions.
- The SATPLAN algorithm translates a planning problem into propositional axioms and applies a satisfiability algorithm to find a model that corresponds to a valid plan. Several different propositional representations have been developed, with varying degrees of compactness and efficiency.
- Each of the major approaches to planning has its adherents, and there is as yet no consensus on which is best. Competition and cross-fertilization among the approaches have resulted in significant gains in efficiency for planning systems.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

AI planning arose from investigations into state-space search, theorem proving, and control theory and from the practical needs of robotics, scheduling, and other domains. STRIPS (Fikes and Nilsson, 1971), the first major planning system, illustrates the interaction of these influences. STRIPS was designed as the planning component of the software for the Shakey robot project at SRI. Its overall control structure was modeled on that of GPS, the General Problem Solver (Newell and Simon, 1961), a state-space search system that used means–ends analysis. STRIPS used a version of the QA3 theorem proving system (Green, 1969b) as a subroutine for establishing the truth of preconditions for actions. Lifschitz (1986) offers precise definitions and an analysis of the STRIPS language. Bylander (1992) shows simple STRIPS planning to be PSPACE-complete. Fikes and Nilsson (1993) give a historical retrospective on the STRIPS project and a survey of its relationship to more recent planning efforts.

The action representation used by STRIPS has been far more influential than its algorithmic approach. Almost all planning systems since then have used one variant or another of the STRIPS language. Unfortunately, the proliferation of variants has made comparisons needlessly difficult. With time came a better understanding of the limitations and tradeoffs among formalisms. The Action Description Language, or ADL, (Pednault, 1986) relaxed some of the restrictions in the STRIPS language and made it possible to encode more realistic problems. Nebel (2000) explores schemes for compiling ADL into STRIPS. The Problem Domain Description Language or PDDL (Ghallab *et al.*, 1998) was introduced as a computer-parsable, standardized syntax for representing STRIPS, ADL, and other languages. PDDL has been used as the standard language for the planning competitions at the AIPS conference, beginning in 1998.

Planners in the early 1970s generally worked with totally ordered action sequences. Problem decomposition was achieved by computing a subplan for each subgoal and then

LINEAR PLANNING

stringing the subplans together in some order. This approach, called **linear planning** by Sacerdoti (1975), was soon discovered to be incomplete. It cannot solve some very simple problems, such as the Sussman anomaly (see Exercise 11.11), found by Allen Brown during experimentation with the HACKER system (Sussman, 1975). A complete planner must allow for **interleaving** of actions from different subplans within a single sequence. The notion of serializable subgoals (Korf, 1987) corresponds exactly to the set of problems for which noninterleaved planners are complete.

INTERLEAVING

One solution to the interleaving problem was goal regression planning, a technique in which steps in a totally ordered plan are reordered so as to avoid conflict between subgoals. This was introduced by Waldinger (1975) and also used by Warren's (1974) WARPLAN. WARPLAN is also notable in that it was the first planner to be written in a logic programming language (Prolog) and is one of the best examples of the remarkable economy that can sometimes be gained by using logic programming: WARPLAN is only 100 lines of code, a small fraction of the size of comparable planners of the time. INTERPLAN (Tate, 1975a, 1975b) also allowed arbitrary interleaving of plan steps to overcome the Sussman anomaly and related problems.

The ideas underlying partial-order planning include the detection of conflicts (Tate, 1975a) and the protection of achieved conditions from interference (Sussman, 1975). The construction of partially ordered plans (then called **task networks**) was pioneered by the NOAH planner (Sacerdoti, 1975, 1977) and by Tate's (1975b, 1977) NONLIN system.[7]

Partial-order planning dominated the next 20 years of research, yet for much of that time, the field was not widely understood. TWEAK (Chapman, 1987) was a logical reconstruction and simplification of planning work of this time; his formulation was clear enough to allow proofs of completeness and intractability (NP-hardness and undecidability) of various formulations of the planning problem. Chapman's work led to what was arguably the first simple and readable description of a complete partial-order planner (McAllester and Rosenblitt, 1991). An implementation of McAllester and Rosenblitt's algorithm called SNLP (Soderland and Weld, 1991) was widely distributed and allowed many researchers to understand and experiment with partial-order planning for the first time. The POP algorithm described in this chapter is based on SNLP.

Weld's group also developed UCPOP (Penberthy and Weld, 1992), the first planner for problems expressed in ADL. UCPOP incorporated the number-of-unsatisfied-goals heuristic. It ran somewhat faster than SNLP, but was seldom able to find plans with more than a dozen or so steps. Although improved heuristics were developed for UCPOP (Joslin and Pollack, 1994; Gerevini and Schubert, 1996), partial-order planning fell into disrepute in the 1990s as faster methods emerged. Nguyen and Kambhampati (2001) suggest that a rehabilitation is merited: with accurate heuristics derived from a planning graph, their REPOP planner scales up much better than GRAPHPLAN and is competitive with the fastest state-space planners.

Avrim Blum and Merrick Furst (1995, 1997) revitalized the field of planning with their GRAPHPLAN system, which was orders of magnitude faster than the partial-order planners of

[7] Some confusion exists over terminology. Many authors use the term **nonlinear** to mean partially ordered. This is slightly different from Sacerdoti's original usage referring to interleaved plans.

the time. Other graph planning systems, such as IPP (Koehler *et al.*, 1997), STAN (Fox and Long, 1998) and SGP (Weld *et al.*, 1998), soon followed. A data structure closely resembling the planning graph had been developed slightly earlier by Ghallab and Laruelle (1994), whose IXTET partial-order planner used it to derive accurate heuristics to guide searches. Nguyen *et al.* (2001) give a very thorough analysis of heuristics derived from planning graphs. Our discussion of planning graphs is based partly on this work and on lecture notes by Subbarao Kambhampati. As mentioned in the chapter, a planning graph can be used in many different ways to guide the search for a solution. The winner of the 2002 AIPS planning competition, LPG (Gerevini and Serina, 2002), searched planning graphs using a local search technique inspired by WALKSAT.

Planning as satisfiability and the SATPLAN algorithm were proposed by Kautz and Selman (1992), who were inspired by the surprising success of greedy local search for satisfiability problems. (See Chapter 7.) Kautz *et al.* (1996) also investigated various forms of propositional representations for STRIPS axioms, finding that the most compact forms did not necessarily lead to the fastest solution times. A systematic analysis was carried out by Ernst *et al.* (1997), who also developed an automatic "compiler" for generating propositional representations from PDDL problems. The BLACKBOX planner, which combines ideas from GRAPHPLAN and SATPLAN, was developed by Kautz and Selman (1998).

The resurgence of interest in state-space planning was pioneered by Drew McDermott's UNPOP program (1996), which was the first to suggest a distance heuristic based on a relaxed problem with delete lists ignored. The name UNPOP was a reaction to the overwhelming concentration on partial-order planning at the time; McDermott suspected that other approaches were not getting the attention they deserved. Bonet and Geffner's Heuristic Search Planner (HSP) and its later derivatives (Bonet and Geffner, 1999) were the first to make state-space search practical for large planning problems. The most successful state-space searcher to date is Hoffmann's (2000) FASTFORWARD or FF, winner of the AIPS 2000 planning competition. FF uses a simplified planning graph heuristic with a very fast search algorithm that combines forward and local search in a novel way.

BINARY DECISION DIAGRAMS

Most recently, there has been interest in the representation of plans as **binary decision diagrams**, a compact description of finite automata widely studied in the hardware verification community (Clarke and Grumberg, 1987; McMillan, 1993). There are techniques for proving properties of binary decision diagrams, including the property of being a solution to a planning problem. Cimatti *et al.* (1998) present a planner based on this approach. Other representations have also been used; for example, Vossen *et al.* (2001) survey the use of integer programming for planning.

The jury is still out, but there are now some interesting comparisons of the various approaches to planning. Helmert (2001) analyzes several classes of planning problems, and shows that constraint-based approaches, such as GRAPHPLAN and SATPLAN are best for NP-hard domains, while search-based approaches do better in domains where feasible solutions can be found without backtracking. GRAPHPLAN and SATPLAN have trouble in domains with many objects, because that means they must create many actions. In some cases the problem can be delayed or avoided by generating the propositionalized actions dynamically, only as needed, rather than instantiating them all before the search begins.

Weld (1994, 1999) provides two excellent surveys of modern planning algorithms. It is interesting to see the change in the five years between the two surveys: the first concentrates on partial-order planning, and the second introduces GRAPHPLAN and SATPLAN. *Readings in Planning* (Allen *et al.*, 1990) is a comprehensive anthology of many of the best earlier articles in the field, including several good surveys. Yang (1997) provides a book-length overview of partial-order planning techniques.

Planning research has been central to AI since its inception, and papers on planning are a staple of mainstream AI journals and conferences. There are also specialized conferences such as the International Conference on AI Planning Systems (AIPS), the International Workshop on Planning and Scheduling for Space, and the European Conference on Planning.

EXERCISES

11.1 Describe the differences and similarities between problem solving and planning.

11.2 Given the axioms from Figure 11.2, what are all the applicable concrete instances of $Fly(p, from, to)$ in the state described by

$$At(P_1, JFK) \wedge At(P_2, SFO) \wedge Plane(P_1) \wedge Plane(P_2) \\ \wedge Airport(JFK) \wedge Airport(SFO) \; ?$$

11.3 Let us consider how we might translate a set of STRIPS schemata into the successor-state axioms of situation calculus. (See Chapter 10.)

- Consider the schema for $Fly(p, from, to)$. Write a logical definition for the predicate $FlyPrecond(p, from, to, s)$, which is true if the preconditions for $Fly(p, from, to)$ are satisfied in situation s.
- Next, assuming that $Fly(p, from, to)$ is the only action schema available to the agent, write down a successor-state axiom for $At(p, x, s)$ that captures the same information as the action schema.
- Now suppose there is an additional method of travel: $Teleport(p, from, to)$. It has the additional precondition $\neg\, Warped(p)$ and the additional effect $Warped(p)$. Explain how the situation calculus knowledge base must be modified.
- Finally, develop a general and precisely specified procedure for carrying out the translation from a set of STRIPS schemata to a set of successor-state axioms.

11.4 The monkey-and-bananas problem is faced by a monkey in a laboratory with some bananas hanging out of reach from the ceiling. A box is available that will enable the monkey to reach the bananas if he climbs on it. Initially, the monkey is at A, the bananas at B, and the box at C. The monkey and box have height Low, but if the monkey climbs onto the box he will have height $High$, the same as the bananas. The actions available to the monkey include Go from one place to another, $Push$ an object from one place to another, $ClimbUp$ onto or

$ClimbDown$ from an object, and $Grasp$ or $Ungrasp$ an object. Grasping results in holding the object if the monkey and object are in the same place at the same height.

a. Write down the initial state description.

b. Write down STRIPS-style definitions of the six actions.

c. Suppose the monkey wants to fool the scientists, who are off to tea, by grabbing the bananas, but leaving the box in its original place. Write this as a general goal (i.e., not assuming that the box is necessarily at C) in the language of situation calculus. Can this goal be solved by a STRIPS-style system?

d. Your axiom for pushing is probably incorrect, because if the object is too heavy, its position will remain the same when the $Push$ operator is applied. Is this an example of the ramification problem or the qualification problem? Fix your problem description to account for heavy objects.

11.5 Explain why the process for generating predecessors in backward search does not need to add the literals that are negative effects of the action.

11.6 Explain why dropping negative effects from every action schema in a STRIPS problem results in a relaxed problem.

11.7 Examine the definition of **bidirectional search** in Chapter 3.

a. Would bidirectional state-space search be a good idea for planning?

b. What about bidirectional search in the space of partial-order plans?

c. Devise a version of partial-order planning in which an action can be added to a plan if its preconditions can be achieved by the effects of actions already in the plan. Explain how to deal with conflicts and ordering constraints. Is the algorithm essentially identical to forward state-space search?

d. Consider a partial-order planner that combines the method in part (c) with the standard method of adding actions to achieve open conditions. Would the resulting algorithm be the same as part (b)?

11.8 Construct levels 0, 1, and 2 of the planning graph for the problem in Figure 11.2.

11.9 Prove the following assertions about planning graphs:

- A literal that does not appear in the final level of the graph cannot be achieved.
- The level cost of a literal in a serial graph is no greater than the actual cost of an optimal plan for achieving it.

11.10 We contrasted forward and backward state-space search planners with partial-order planners, saying that the latter is a plan-space searcher. Explain how forward and backward state-space search can also be considered plan-space searchers, and say what the plan refinement operators are.

SUSSMAN ANOMALY

11.11 Figure 11.16 shows a blocks-world problem known as the **Sussman anomaly**. The problem was considered anomalous because the noninterleaved planners of the early 1970s could not solve it. Write a definition of the problem in STRIPS notation and solve it, either by hand or with a planning program. A noninterleaved planner is a planner that, when given two subgoals G_1 and G_2, produces either a plan for G_1 concatenated with a plan for G_2, or vice-versa. Explain why a noninterleaved planner cannot solve this problem.

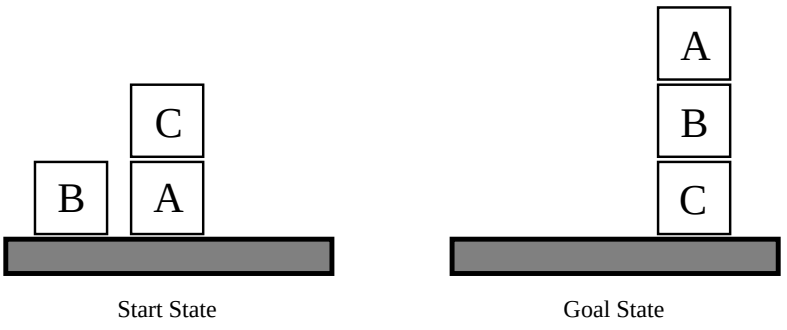

Figure 11.16 The "Sussman anomaly" blocks-world planning problem.

11.12 Consider the problem of putting on one's shoes and socks, as defined in Section 11.3. Apply GRAPHPLAN to this problem and show the solution obtained. Now add actions for putting on a coat and a hat. Show the partial order plan that is a solution, and show that there are 180 different linearizations of the partial-order plan. What is the minimum number of different planning graph solutions needed to represent all 180 linearizations?

11.13 The original STRIPS program was designed to control Shakey the robot. Figure 11.17 shows a version of Shakey's world consisting of four rooms lined up along a corridor, where each room has a door and a light switch.

The actions in Shakey's world include moving from place to place, pushing movable objects (such as boxes), climbing onto and down from rigid objects (such as boxes), and turning light switches on and off. The robot itself was never dexterous enough to climb on a box or toggle a switch, but the STRIPS planner was capable of finding and printing out plans that were beyond the robot's abilities. Shakey's six actions are the following:

- $Go(x, y)$, which requires that Shakey be at x and that x and y are locations in the same room. By convention a door between two rooms is in both of them.
- Push a box b from location x to location y within the same room: $Push(b, x, y)$. We will need the predicate Box and constants for the boxes.
- Climb onto a box: $ClimbUp(b)$; climb down from a box: $ClimbDown(b)$. We will need the predicate On and the constant $Floor$.
- Turn a light switch on: $TurnOn(s)$; turn it off: $TurnOff(s)$. To turn a light on or off, Shakey must be on top of a box at the light switch's location.

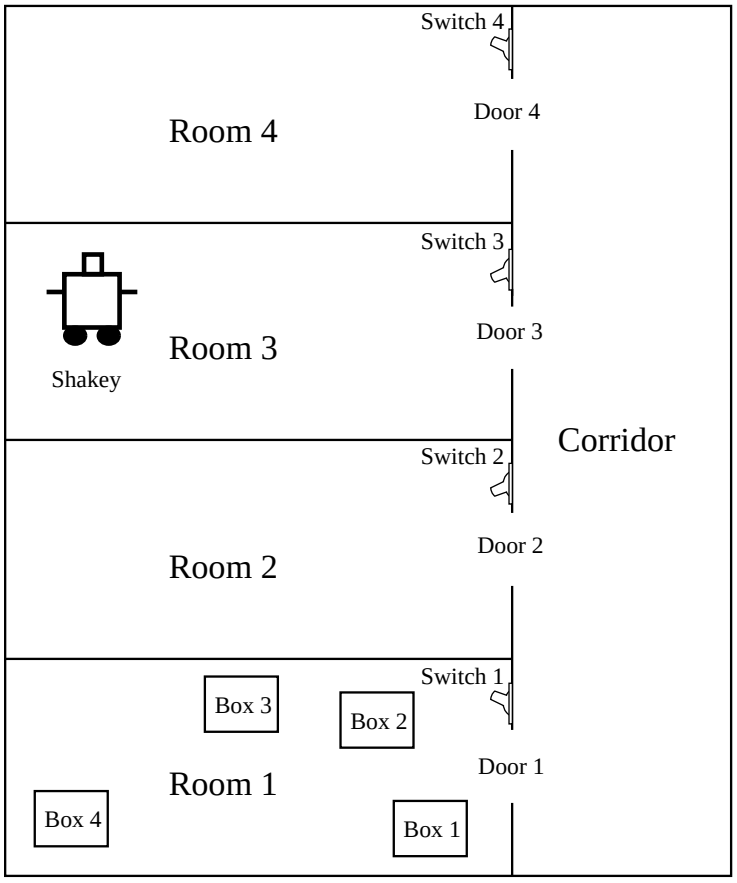

Figure 11.17 Shakey's world. Shakey can move between landmarks within a room, can pass through the door between rooms, can climb climbable objects and push pushable objects, and can flip light switches.

Describe Shakey's six actions and the initial state from Figure 11.17 in STRIPS notation. Construct a plan for Shakey to get Box_2 into $Room_2$.

11.14 We saw that planning graphs can handle only propositional actions. What if we want to use planning graphs for a problem with variables in the goal, such as $At(P_1, x) \wedge At(P_2, x)$, where x ranges over a finite domain of locations? How could you encode such a problem to work with planning graphs? (Hint: remember the $Finish$ action from POP planning. What preconditions should it have?)

11.15 Up to now we have assumed that actions are only executed in the appropriate situations. Let us see what propositional successor-state axioms such as Equation (11.1) have to say about actions whose preconditions are not satisfied.

a. Show that the axioms predict that nothing will happen when an action is executed in a state where its preconditions are not satisifed.

b. Consider a plan p that contains the actions required to achieve a goal but also includes illegal actions. Is it the case that

$$\textit{initial state} \wedge \textit{successor-state axioms} \wedge p \models \textit{goal} \ ?$$

c. With first-order successor-state axioms in situation calculus (as in Chapter 10), is it possible to prove that a plan containing illegal actions will achieve the goal?

11.16 Giving examples from the airport domain, explain how symbol-splitting reduces the size of the precondition axioms and the action exclusion axioms. Derive a general formula for the size of each axiom set in terms of the number of time steps, the number of action schemata, their arities, and the number of objects.

11.17 In the SATPLAN algorithm in Figure 11.15, each call to the satisfiability algorithm asserts a goal g^T, where T ranges from 0 to $T_{\max}$. Suppose instead that the satisfiability algorithm is called only once, with the goal $g^0 \vee g^1 \vee \cdots \vee g^{T_{\max}}$.

a. Will this always return a plan if one exists with length less than or equal to $T_{\max}$?

b. Does this approach introduce any new spurious "solutions"?

c. Discuss how one might modify a satisfiability algorithm such as WALKSAT so that it finds short solutions (if they exist) when given a disjunctive goal of this form.