

# 10 CLASSICAL PLANNING

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

We have defined AI as the study of rational action, which means that **planning**—devising a plan of action to achieve one’s goals—is a critical part of AI. We have seen two examples of planning agents so far: the search-based problem-solving agent of Chapter 3 and the hybrid logical agent of Chapter 7. In this chapter we introduce a representation for planning problems that scales up to problems that could not be handled by those earlier approaches.

Section 10.1 develops an expressive yet carefully constrained language for representing planning problems. Section 10.2 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 domain-independent heuristics were constructed for constraint satisfaction problems in Chapter 6.) Section 10.3 shows how a data structure called the planning graph can make the search for a plan more efficient. We then describe a few of the other approaches to planning, and conclude by comparing the various approaches.

This chapter covers fully observable, deterministic, static environments with single agents. Chapters 11 and 17 cover partially observable, stochastic, dynamic environments with multiple agents.

## 10.1 DEFINITION OF CLASSICAL PLANNING

---

The problem-solving agent of Chapter 3 can find sequences of actions that result in a goal state. But it deals with atomic representations of states and thus needs good domain-specific heuristics to perform well. The hybrid propositional logical agent of Chapter 7 can find plans without domain-specific heuristics because it uses domain-independent heuristics based on the logical structure of the problem. But it relies on ground (variable-free) propositional inference, which means that it may be swamped when there are many actions and states. For example, in the wumpus world, the simple action of moving a step forward had to be repeated for all four agent orientations,  $T$  time steps, and  $n^2$  current locations.

PDDL

In response to this, planning researchers have settled on a **factored representation**—one in which a state of the world is represented by a collection of variables. We use a language called **PDDL**, the Planning Domain Definition Language, that allows us to express all  $4Tn^2$  actions with one action schema. There have been several versions of PDDL; we select a simple version and alter its syntax to be consistent with the rest of the book.<sup>1</sup> We now show how PDDL describes the four things we need to define a search problem: the initial state, the actions that are available in a state, the result of applying an action, and the goal test.

SET SEMANTICS

Each **state** is represented as a conjunction of fluents that are ground, functionless atoms. For example,  $Poor \wedge Unknown$  might represent the state of a hapless agent, and a state in a package delivery problem might be  $At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$ . **Database semantics** is used: the closed-world assumption means that any fluents that are not mentioned are false, and the unique names assumption means that  $Truck_1$  and  $Truck_2$  are distinct. The following fluents are *not* allowed in a state:  $At(x, y)$  (because it is non-ground),  $\neg Poor$  (because it is a negation), and  $At(Father(Fred), Sydney)$  (because it uses a function symbol). The representation of states is carefully designed so that a state can be treated either as a conjunction of fluents, which can be manipulated by logical inference, or as a *set* of fluents, which can be manipulated with set operations. The **set semantics** is sometimes easier to deal with.

ACTION SCHEMA

**Actions** are described by a set of action schemas that implicitly define the  $ACTIONS(s)$  and  $RESULT(s, a)$  functions needed to do a problem-solving search. We saw in Chapter 7 that any system for action description needs to solve the frame problem—to say what changes and what stays the same as the result of the action. Classical planning concentrates on problems where most actions leave most things unchanged. Think of a world consisting of a bunch of objects on a flat surface. The action of nudging an object causes that object to change its location by a vector  $\Delta$ . A concise description of the action should mention only  $\Delta$ ; it shouldn't have to mention all the objects that stay in place. PDDL does that by specifying the result of an action in terms of what changes; everything that stays the same is left unmentioned.

A set of ground (variable-free) actions can be represented by a single **action schema**. The schema is a **lifted** representation—it lifts the level of reasoning from propositional logic to a restricted subset of first-order logic. For example, here is an action schema for flying a plane from one location to another:

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

PRECONDITION

EFFECT

The schema consists of the action name, a list of all the variables used in the schema, a **precondition** and an **effect**. Although we haven't said yet how the action schema converts into logical sentences, think of the variables as being universally quantified. We are free to choose whatever values we want to instantiate the variables. For example, here is one ground

<sup>1</sup> PDDL was derived from the original STRIPS planning language (Fikes and Nilsson, 1971), which is slightly more restricted than PDDL: STRIPS preconditions and goals cannot contain negative literals.

action that results from substituting values for all the variables:

$Action(Fly(P_1, SFO, JFK),$   
 $PRECOND: At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK)$   
 $EFFECT: \neg At(P_1, SFO) \wedge At(P_1, JFK))$

The precondition and effect of an action are each conjunctions of literals (positive or negated atomic sentences). The precondition defines the states in which the action can be executed, and the effect defines the result of executing the action. An action  $a$  can be executed in state  $s$  if  $s$  entails the precondition of  $a$ . Entailment can also be expressed with the set semantics:  $s \models q$  iff every positive literal in  $q$  is in  $s$  and every negated literal in  $q$  is not. In formal notation we say

$$(a \in ACTIONS(s)) \Leftrightarrow s \models PRECOND(a),$$

where any variables in  $a$  are universally quantified. For example,

$$\forall p, from, to \ (Fly(p, from, to) \in ACTIONS(s)) \Leftrightarrow$$

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

APPLICABLE

We say that action  $a$  is **applicable** in state  $s$  if the preconditions are satisfied by  $s$ . When an action schema  $a$  contains variables, it may have multiple applicable instantiations. For example, with the initial state defined in Figure 10.1, the  $Fly$  action can be instantiated as  $Fly(P_1, SFO, JFK)$  or as  $Fly(P_2, JFK, SFO)$ , both of which are applicable in the initial state. If an action  $a$  has  $v$  variables, then, in a domain with  $k$  unique names of objects, it takes  $O(v^k)$  time in the worst case to find the applicable ground actions.

PROPOSITIONALIZE

Sometimes we want to **propositionalize** a PDDL problem—replace each action schema with a set of ground actions and then use a propositional solver such as SATPLAN to find a solution. However, this is impractical when  $v$  and  $k$  are large.

DELETE LIST

ADD LIST

The **result** of executing action  $a$  in state  $s$  is defined as a state  $s'$  which is represented by the set of fluents formed by starting with  $s$ , removing the fluents that appear as negative literals in the action's effects (what we call the **delete list** or  $DEL(a)$ ), and adding the fluents that are positive literals in the action's effects (what we call the **add list** or  $ADD(a)$ ):

$$RESULT(s, a) = (s - DEL(a)) \cup ADD(a). \quad (10.1)$$

For example, with the action  $Fly(P_1, SFO, JFK)$ , we would remove  $At(P_1, SFO)$  and add  $At(P_1, JFK)$ . It is a requirement of action schemas that any variable in the effect must also appear in the precondition. That way, when the precondition is matched against the state  $s$ , all the variables will be bound, and  $RESULT(s, a)$  will therefore have only ground atoms. In other words, ground states are closed under the  $RESULT$  operation.

Also note that the fluents do not explicitly refer to time, as they did in Chapter 7. There we needed superscripts for time, and successor-state axioms of the form

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t).$$

In PDDL the times and states are implicit in the action schemas: the precondition always refers to time  $t$  and the effect to time  $t + 1$ .

A set of action schemas serves as a definition of a planning *domain*. A specific *problem* within the domain is defined with the addition of an initial state and a goal. The **initial**

```

Init(At(C1, SFO) ∧ At(C2, JFK) ∧ At(P1, SFO) ∧ At(P2, JFK)
    ∧ Cargo(C1) ∧ Cargo(C2) ∧ Plane(P1) ∧ Plane(P2)
    ∧ Airport(JFK) ∧ Airport(SFO))
Goal(At(C1, JFK) ∧ At(C2, SFO))
Action(Load(c, p, a),
    PRECOND: At(c, a) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
    EFFECT: ¬ At(c, a) ∧ In(c, p))
Action(Unload(c, p, a),
    PRECOND: In(c, p) ∧ At(p, a) ∧ Cargo(c) ∧ Plane(p) ∧ Airport(a)
    EFFECT: At(c, a) ∧ ¬ In(c, p))
Action(Fly(p, from, to),
    PRECOND: At(p, from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
    EFFECT: ¬ At(p, from) ∧ At(p, to))

```

**Figure 10.1** A PDDL description of an air cargo transportation planning problem.

INITIAL STATE

GOAL

**state** is a conjunction of ground atoms. (As with all states, the closed-world assumption is used, which means that any atoms that are not mentioned are false.) The **goal** is just like a precondition: a conjunction of literals (positive or negative) that may contain variables, such as *At*(*p*, *SFO*) ∧ *Plane*(*p*). Any variables are treated as existentially quantified, so this goal is to have *any* plane at SFO. The problem is solved when we can find a sequence of actions that end in a state *s* that entails the goal. For example, the state *Rich* ∧ *Famous* ∧ *Miserable* entails the goal *Rich* ∧ *Famous*, and the state *Plane*(*Plane*<sub>1</sub>) ∧ *At*(*Plane*<sub>1</sub>, *SFO*) entails the goal *At*(*p*, *SFO*) ∧ *Plane*(*p*).

Now we have defined planning as a search problem: we have an initial state, an ACTIONS function, a RESULT function, and a goal test. We'll look at some example problems before investigating efficient search algorithms.

### 10.1.1 Example: Air cargo transport

Figure 10.1 shows an air cargo transport problem involving loading and unloading cargo 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 some care must be taken to make sure the *At* predicates are maintained properly. When a plane flies from one airport to another, all the cargo inside the plane goes with it. In first-order logic it would be easy to quantify over all objects that are inside the plane. But basic PDDL does not have a universal quantifier, so we need a different solution. The approach we use is to say that a piece of cargo ceases to be *At* anywhere when it is *In* a plane; the cargo only becomes *At* the new airport when it is unloaded. So *At* really means “available for use at a given location.” The following plan is a solution to the problem:

```

[Load(C1, P1, SFO), Fly(P1, SFO, JFK), Unload(C1, P1, JFK),
 Load(C2, P2, JFK), Fly(P2, JFK, SFO), Unload(C2, P2, SFO)] .

```

Finally, there is the problem of spurious actions such as  $Fly(P_1, JFK, JFK)$ , which should be a no-op, but which has contradictory effects (according to the definition, the effect would include  $At(P_1, JFK) \wedge \neg At(P_1, JFK)$ ). It is common to ignore such problems, because they seldom cause incorrect plans to be produced. The correct approach is to add inequality preconditions saying that the *from* and *to* airports must be different; see another example of this in Figure 10.3.

### 10.1.2 Example: The spare tire problem

Consider the problem of changing a flat tire (Figure 10.2). 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 an 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 parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tires disappear. A solution to the problem is  $[Remove(Flat, Axle), Remove(Spare, Trunk), PutOn(Spare, Axle)]$ .

```

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

```

**Figure 10.2** The simple spare tire problem.

### 10.1.3 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.<sup>2</sup> 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

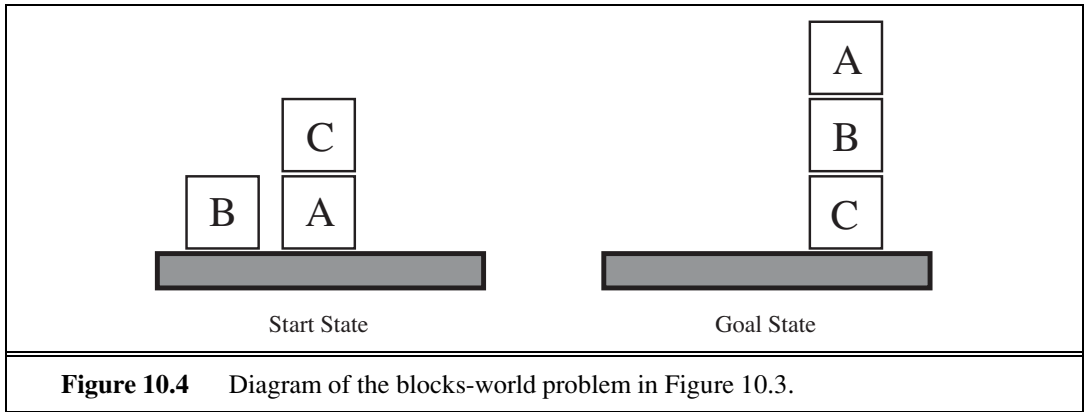
<sup>2</sup> The blocks world used in planning research is much simpler than SHRDLU's version, shown on page 20.

```

Init( $On(A, Table) \wedge On(B, Table) \wedge On(C, A)$ 
 $\wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C)$ )
Goal( $On(A, B) \wedge On(B, C)$ )
Action( $Move(b, x, y)$ ,
  PRECOND:  $On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge$ 
 $(b \neq x) \wedge (b \neq y) \wedge (x \neq y)$ ,
  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 10.3** A planning problem in the blocks world: building a three-block tower. One solution is the sequence  $[MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)]$ .



**Figure 10.4** Diagram of the blocks-world problem in Figure 10.3.

of what other blocks. For example, a goal might be to get block  $A$  on  $B$  and block  $B$  on  $C$  (see Figure 10.4).

We 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,  $\forall x \neg On(x, b)$ . Basic PDDL does not allow quantifiers, so instead we introduce a predicate  $Clear(x)$  that is true when nothing is on  $x$ . (The complete problem description is in Figure 10.3.)

The action  $Move$  moves a block  $b$  from  $x$  to  $y$  if both  $b$  and  $y$  are clear. After the move is made,  $b$  is still clear but  $y$  is not. A first attempt at the  $Move$  schema 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 does not maintain  $Clear$  properly when  $x$  or  $y$  is the table. When  $x$  is the *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

for us 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:

$$\begin{aligned} & \text{Action}(\text{MoveToTable}(b, x), \\ & \quad \text{PRECOND: } On(b, x) \wedge \text{Clear}(b), \\ & \quad \text{EFFECT: } On(b, \text{Table}) \wedge \text{Clear}(x) \wedge \neg On(b, x)) . \end{aligned}$$

Second, we take the interpretation of  $\text{Clear}(x)$  to be “there is a clear space on  $x$  to hold a block.” Under this interpretation,  $\text{Clear}(\text{Table})$  will always be true. The only problem is that nothing prevents the planner from using  $\text{Move}(b, x, \text{Table})$  instead of  $\text{MoveToTable}(b, x)$ . 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  $\text{Block}$  and add  $\text{Block}(b) \wedge \text{Block}(y)$  to the precondition of  $\text{Move}$ .

### 10.1.4 The complexity of classical planning

In this subsection we consider the theoretical complexity of planning and distinguish two decision problems. **PlanSAT** is the question of whether there exists any plan that solves a planning problem. **Bounded PlanSAT** asks whether there is a solution of length  $k$  or less; this can be used to find an optimal plan.

The first result is that both decision problems are decidable for classical planning. The proof follows from the fact that the number of states is finite. But if we add function symbols to the language, then the number of states becomes infinite, and PlanSAT becomes only semidecidable: an algorithm exists that will terminate with the correct answer for any solvable problem, but may not terminate on unsolvable problems. The Bounded PlanSAT problem remains decidable even in the presence of function symbols. For proofs of the assertions in this section, see Ghallab *et al.* (2004).

Both PlanSAT and Bounded PlanSAT are in the complexity class PSPACE, a class that is larger (and hence more difficult) than NP and refers to problems that can be solved by a deterministic Turing machine with a polynomial amount of space. Even if we make some rather severe restrictions, the problems remain quite difficult. For example, if we disallow negative effects, both problems are still NP-hard. However, if we also disallow negative preconditions, PlanSAT reduces to the class P.

These worst-case results may seem discouraging. We can take solace in the fact that agents are usually not asked to find plans for arbitrary worst-case problem instances, but rather are asked for plans in specific domains (such as blocks-world problems with  $n$  blocks), which can be much easier than the theoretical worst case. For many domains (including the blocks world and the air cargo world), Bounded PlanSAT is NP-complete while PlanSAT is in P; in other words, optimal planning is usually hard, but sub-optimal planning is sometimes easy. To do well on easier-than-worst-case problems, we will need good search heuristics. That’s the true advantage of the classical planning formalism: it has facilitated the development of very accurate domain-independent heuristics, whereas systems based on successor-state axioms in first-order logic have had less success in coming up with good heuristics.

PlanSAT

Bounded PlanSAT

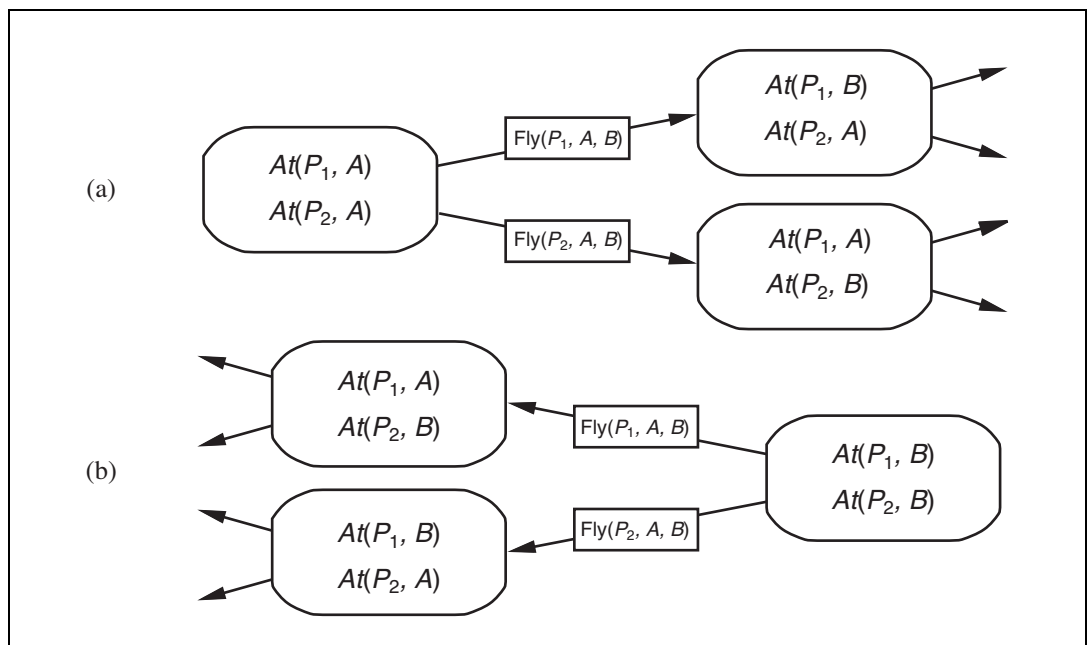
## 10.2 ALGORITHMS FOR PLANNING AS STATE-SPACE SEARCH

Now we turn our attention to planning algorithms. We saw how the description of a planning problem defines a search problem: we can search from the initial state through the space of states, looking for a goal. One of the nice advantages of the declarative representation of action schemas is that we can also search backward from the goal, looking for the initial state. Figure 10.5 compares forward and backward searches.

### 10.2.1 Forward (progression) state-space search

Now that we have shown how a planning problem maps into a search problem, we can solve planning problems with any of the heuristic search algorithms from Chapter 3 or a local search algorithm from Chapter 4 (provided we keep track of the actions used to reach the goal). From the earliest days of planning research (around 1961) until 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.

First, forward search is prone to exploring irrelevant actions. Consider the noble task of buying a copy of *AI: A Modern Approach* from an online bookseller. Suppose there is an



**Figure 10.5** Two approaches to searching for a plan. (a) Forward (progression) search through the space of states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through sets of relevant states, starting at the set of states representing the goal and using the inverse of the actions to search backward for the initial state.



action schema  $Buy(isbn)$  with effect  $Own(isbn)$ . ISBNs are 10 digits, so this action schema represents 10 billion ground actions. An uninformed forward-search algorithm would have to start enumerating these 10 billion actions to find one that leads to the goal.

Second, planning problems often have large state spaces. 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. 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). So in any state there is a minimum of 450 actions (when all the packages are at airports with no planes) and a maximum of 10,450 (when all packages and planes are at the same airport). On average, let's say there are about 2000 possible actions per state, so the search graph up to the depth of the obvious solution has about  $2000^{41}$  nodes.

Clearly, even this relatively small problem instance is hopeless without an accurate heuristic. Although many real-world applications of planning have relied on domain-specific heuristics, it turns out (as we see in Section 10.2.3) that strong domain-independent heuristics can be derived automatically; that is what makes forward search feasible.

### 10.2.2 Backward (regression) relevant-states search

RELEVANT-STATES

In regression search we start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial state. It is called **relevant-states** search because we only consider actions that are relevant to the goal (or current state). As in belief-state search (Section 4.4), there is a *set* of relevant states to consider at each step, not just a single state.

We start with the goal, which is a conjunction of literals forming a description of a set of states—for example, the goal  $\neg Poor \wedge Famous$  describes those states in which *Poor* is false, *Famous* is true, and any other fluent can have any value. If there are  $n$  ground fluents in a domain, then there are  $2^n$  ground states (each fluent can be true or false), but  $3^n$  descriptions of sets of goal states (each fluent can be positive, negative, or not mentioned).

In general, backward search works only when we know how to regress from a state description to the predecessor state description. For example, it is hard to search backwards for a solution to the  $n$ -queens problem because there is no easy way to describe the states that are one move away from the goal. Happily, the PDDL representation was designed to make it easy to regress actions—if a domain can be expressed in PDDL, then we can do regression search on it. Given a ground goal description  $g$  and a ground action  $a$ , the regression from  $g$  over  $a$  gives us a state description  $g'$  defined by

$$g' = (g - \text{ADD}(a)) \cup \text{Precond}(a) .$$

That is, the effects that were added by the action need not have been true before, and also the preconditions must have held before, or else the action could not have been executed. Note that  $\text{DEL}(a)$  does not appear in the formula; that's because while we know the fluents in  $\text{DEL}(a)$  are no longer true after the action, we don't know whether or not they were true before, so there's nothing to be said about them.

To get the full advantage of backward search, we need to deal with partially uninstantiated actions and states, not just ground ones. For example, suppose the goal is to deliver a specific piece of cargo to SFO:  $At(C_2, SFO)$ . That suggests the action  $Unload(C_2, p', SFO)$ :

$Action(Unload(C_2, p', SFO),$   
     PRECOND:  $In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO)$   
     EFFECT:  $At(C_2, SFO) \wedge \neg In(C_2, p')$  .

(Note that we have **standardized** variable names (changing  $p$  to  $p'$  in this case) so that there will be no confusion between variable names if we happen to use the same action schema twice in a plan. The same approach was used in Chapter 9 for first-order logical inference.) This represents unloading the package from an *unspecified* plane at SFO; any plane will do, but we need not say which one now. We can take advantage of the power of first-order representations: a single description summarizes the possibility of using *any* of the planes by implicitly quantifying over  $p'$ . The regressed state description is

$$g' = In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO) .$$

The final issue is deciding which actions are candidates to regress over. In the forward direction we chose actions that were **applicable**—those actions that could be the next step in the plan. In backward search we want actions that are **relevant**—those actions that could be the *last* step in a plan leading up to the current goal state.

RELEVANCE

For an action to be relevant to a goal it obviously must contribute to the goal: at least one of the action's effects (either positive or negative) must unify with an element of the goal. What is less obvious is that the action must not have any effect (positive or negative) that negates an element of the goal. Now, if the goal is  $A \wedge B \wedge C$  and an action has the effect  $A \wedge B \wedge \neg C$  then there is a colloquial sense in which that action is very relevant to the goal—it gets us two-thirds of the way there. But it is not relevant in the technical sense defined here, because this action could not be the *final* step of a solution—we would always need at least one more step to achieve  $C$ .

Given the goal  $At(C_2, SFO)$ , several instantiations of  $Unload$  are relevant: we could choose any specific plane to unload from, or we could leave the plane unspecified by using the action  $Unload(C_2, p', SFO)$ . We can reduce the branching factor without ruling out any solutions by always using the action formed by substituting the most general unifier into the (standardized) action schema.

As another example, consider the goal  $Own(0136042597)$ , given an initial state with 10 billion ISBNs, and the single action schema

$$A = Action(Buy(i), PRECOND: ISBN(i), EFFECT: Own(i)) .$$

As we mentioned before, forward search without a heuristic would have to start enumerating the 10 billion ground  $Buy$  actions. But with backward search, we would unify the goal  $Own(0136042597)$  with the (standardized) effect  $Own(i')$ , yielding the substitution  $\theta = \{i'/0136042597\}$ . Then we would regress over the action  $Subst(\theta, A')$  to yield the predecessor state description  $ISBN(0136042597)$ . This is part of, and thus entailed by, the initial state, so we are done.

We can make this more formal. Assume a goal description  $g$  which contains a goal literal  $g_i$  and an action schema  $A$  that is standardized to produce  $A'$ . If  $A'$  has an effect literal  $e'_j$  where  $\text{Unify}(g_i, e'_j) = \theta$  and where we define  $a' = \text{SUBST}(\theta, A')$  and if there is no effect in  $a'$  that is the negation of a literal in  $g$ , then  $a'$  is a relevant action towards  $g$ .

Backward search keeps the branching factor lower than forward search, for most problem domains. However, the fact that backward search uses state sets rather than individual states makes it harder to come up with good heuristics. That is the main reason why the majority of current systems favor forward search.

### 10.2.3 Heuristics for planning

Neither forward nor backward search is efficient without a good heuristic function. Recall from Chapter 3 that a heuristic function  $h(s)$  estimates the distance from a state  $s$  to the goal and that if we can derive an **admissible** heuristic for this distance—one that does not overestimate—then we can use  $A^*$  search to find optimal solutions. An admissible heuristic can be derived by defining a **relaxed problem** that is easier to solve. The exact cost of a solution to this easier problem then becomes the heuristic for the original problem.

By definition, there is no way to analyze an atomic state, and thus it requires some ingenuity by a human analyst to define good domain-specific heuristics for search problems with atomic states. Planning uses a factored representation for states and action schemas. That makes it possible to define good domain-independent heuristics and for programs to automatically apply a good domain-independent heuristic for a given problem.

Think of a search problem as a graph where the nodes are states and the edges are actions. The problem is to find a path connecting the initial state to a goal state. There are two ways we can relax this problem to make it easier: by adding more edges to the graph, making it strictly easier to find a path, or by grouping multiple nodes together, forming an abstraction of the state space that has fewer states, and thus is easier to search.

We look first at heuristics that add edges to the graph. For example, the **ignore preconditions heuristic** drops all preconditions from actions. Every action becomes applicable in every state, and any single goal fluent can be achieved in one step (if there is an applicable action—if not, the problem is impossible). This almost implies that the number of steps required to solve the relaxed problem is the number of unsatisfied goals—almost but not quite, because (1) some action may achieve multiple goals and (2) some actions may undo the effects of others. For many problems an accurate heuristic is obtained by considering (1) and ignoring (2). First, we relax the actions by removing all preconditions and all effects except those that are literals in the goal. Then, we count the minimum number of actions required such that the union of those actions' effects satisfies the goal. This is an instance of the **set-cover problem**. There is one minor irritation: the set-cover problem is NP-hard. Fortunately a simple greedy algorithm is guaranteed to return a set covering whose size is within a factor of  $\log n$  of the true minimum covering, where  $n$  is the number of literals in the goal. Unfortunately, the greedy algorithm loses the guarantee of admissibility.

It is also possible to ignore only *selected* preconditions of actions. Consider the sliding-block puzzle (8-puzzle or 15-puzzle) from Section 3.2. We could encode this as a planning

IGNORE  
PRECONDITIONS  
HEURISTIC

SET-COVER  
PROBLEM

problem involving tiles with a single schema *Slide*:

$Action(Slide(t, s_1, s_2),$

$PRECOND: On(t, s_1) \wedge Tile(t) \wedge Blank(s_2) \wedge Adjacent(s_1, s_2)$

$EFFECT: On(t, s_2) \wedge Blank(s_1) \wedge \neg On(t, s_1) \wedge \neg Blank(s_2))$

As we saw in Section 3.6, if we remove the preconditions  $Blank(s_2) \wedge Adjacent(s_1, s_2)$  then any tile can move in one action to any space and we get the number-of-misplaced-tiles heuristic. If we remove  $Blank(s_2)$  then we get the Manhattan-distance heuristic. It is easy to see how these heuristics could be derived automatically from the action schema description. The ease of manipulating the schemas is the great advantage of the factored representation of planning problems, as compared with the atomic representation of search problems.

IGNORE DELETE  
LISTS

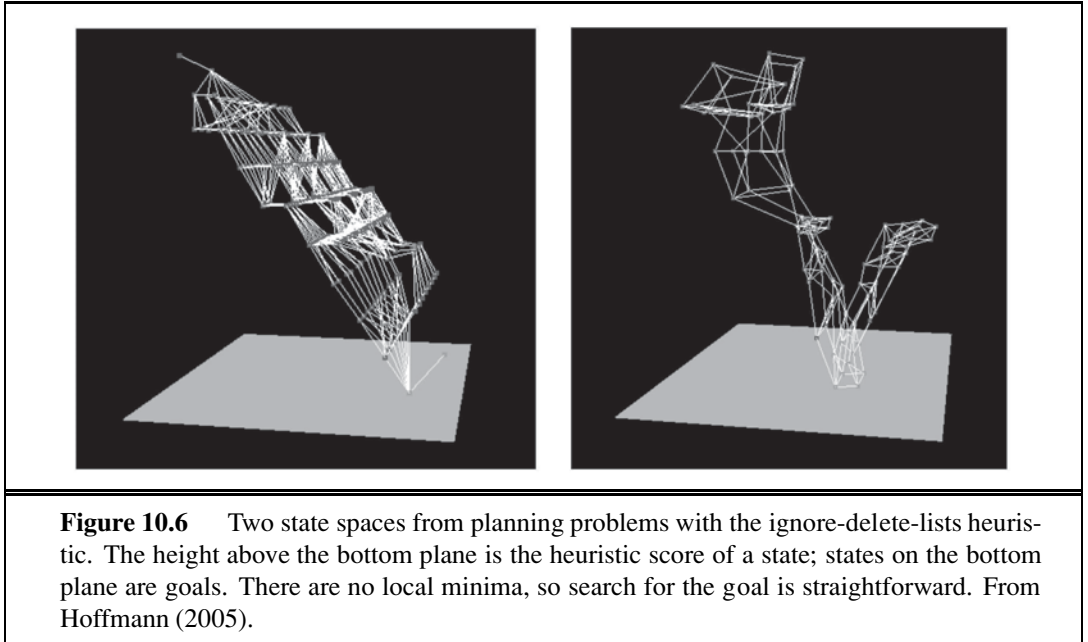
Another possibility is the **ignore delete lists** heuristic. Assume for a moment that all goals and preconditions contain only positive literals<sup>3</sup> We want to create a relaxed version of the original problem that will be easier to solve, and where the length of the solution will serve as a good heuristic. We can do that by removing the delete lists from all actions (i.e., removing all negative literals from effects). That makes it possible to make monotonic progress towards the goal—no action will ever undo progress made by another action. It turns out it is still NP-hard to find the optimal solution to this relaxed problem, but an approximate solution can be found in polynomial time by hill-climbing. Figure 10.6 diagrams part of the state space for two planning problems using the ignore-delete-lists heuristic. The dots represent states and the edges actions, and the height of each dot above the bottom plane represents the heuristic value. States on the bottom plane are solutions. In both these problems, there is a wide path to the goal. There are no dead ends, so no need for backtracking; a simple hillclimbing search will easily find a solution to these problems (although it may not be an optimal solution).

STATE ABSTRACTION

The relaxed problems leave us with a simplified—but still expensive—planning problem just to calculate the value of the heuristic function. Many planning problems have  $10^{100}$  states or more, and relaxing the *actions* does nothing to reduce the number of states. Therefore, we now look at relaxations that decrease the number of states by forming a **state abstraction**—a many-to-one mapping from states in the ground representation of the problem to the abstract representation.

The easiest form of state abstraction is to ignore some fluents. For example, consider an air cargo problem with 10 airports, 50 planes, and 200 pieces of cargo. Each plane can be at one of 10 airports and each package can be either in one of the planes or unloaded at one of the airports. So there are  $50^{10} \times 200^{50+10} \approx 10^{155}$  states. Now consider a particular problem in that domain in which it happens that all the packages are at just 5 of the airports, and all packages at a given airport have the same destination. Then a useful abstraction of the problem is to drop all the *At* fluents except for the ones involving one plane and one package at each of the 5 airports. Now there are only  $5^{10} \times 5^{5+10} \approx 10^{17}$  states. A solution in this abstract state space will be shorter than a solution in the original space (and thus will be an admissible heuristic), and the abstract solution is easy to extend to a solution to the original problem (by adding additional *Load* and *Unload* actions).

<sup>3</sup> Many problems are written with this convention. For problems that aren't, replace every negative literal  $\neg P$  in a goal or precondition with a new positive literal,  $P'$ .



DECOMPOSITION  
SUBGOAL  
INDEPENDENCE

A key idea in defining heuristics is **decomposition**: dividing a problem into parts, solving each part independently, and then combining the parts. The **subgoal independence** assumption is that 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.

Suppose the goal is a set of fluents  $G$ , which we divide into disjoint subsets  $G_1, \dots, G_n$ . We then find plans  $P_1, \dots, P_n$  that solve the respective subgoals. What is an estimate of the cost of the plan for achieving all of  $G$ ? We can think of each  $\text{Cost}(P_i)$  as a heuristic estimate, and we know that if we combine estimates by taking their maximum value, we always get an admissible heuristic. So  $\max_i \text{COST}(P_i)$  is admissible, and sometimes it is exactly correct: it could be that  $P_1$  serendipitously achieves all the  $G_i$ . But in most cases, in practice the estimate is too low. Could we sum the costs instead? For many problems that is a reasonable estimate, but it is not admissible. The best case is when we can determine that  $G_i$  and  $G_j$  are **independent**. If the effects of  $P_i$  leave all the preconditions and goals of  $P_j$  unchanged, then the estimate  $\text{COST}(P_i) + \text{COST}(P_j)$  is admissible, and more accurate than the max estimate. We show in Section 10.3.1 that planning graphs can help provide better heuristic estimates.

It is clear that there is great potential for cutting down the search space by forming abstractions. The trick is choosing the right abstractions and using them in a way that makes the total cost—defining an abstraction, doing an abstract search, and mapping the abstraction back to the original problem—less than the cost of solving the original problem. The tech-

niques of **pattern databases** from Section 3.6.3 can be useful, because the cost of creating the pattern database can be amortized over multiple problem instances.

An example of a system that makes use of effective heuristics is FF, or FASTFORWARD (Hoffmann, 2005), a forward state-space searcher that uses the ignore-delete-lists heuristic, estimating the heuristic with the help of a planning graph (see Section 10.3). FF then uses hill-climbing search (modified to keep track of the plan) with the heuristic to find a solution. When it hits a plateau or local maximum—when no action leads to a state with better heuristic score—then FF uses iterative deepening search until it finds a state that is better, or it gives up and restarts hill-climbing.

## 10.3 PLANNING GRAPHS

### PLANNING GRAPH

All of the heuristics we have suggested 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 search for a solution over the space formed by the planning graph, using an algorithm called GRAPHPLAN.

A planning problem asks if we can reach a goal state from the initial state. Suppose we are given a tree of all possible actions from the initial state to successor states, and their successors, and so on. If we indexed this tree appropriately, we could answer the planning question “can we reach state  $G$  from state  $S_0$ ” immediately, just by looking it up. Of course, the tree is of exponential size, so this approach is impractical. A planning graph is polynomial-size approximation to this tree that can be constructed quickly. The planning graph can’t answer definitively whether  $G$  is reachable from  $S_0$ , but it can *estimate* how many steps it takes to reach  $G$ . The estimate is always correct when it reports the goal is not reachable, and it never overestimates the number of steps, so it is an admissible heuristic.

### LEVEL

A planning graph is a directed graph organized into **levels**: first a level  $S_0$  for the initial state, consisting of nodes representing each fluent that holds in  $S_0$ ; then a level  $A_0$  consisting of nodes for each ground action that might be applicable in  $S_0$ ; then alternating levels  $S_i$  followed by  $A_i$ ; until we reach a termination condition (to be discussed later).

Roughly speaking,  $S_i$  contains all the literals that *could* hold at time  $i$ , depending on the actions executed at preceding time steps. If it is possible that either  $P$  or  $\neg P$  could hold, then both will be represented in  $S_i$ . Also roughly speaking,  $A_i$  contains all the actions that *could* have their preconditions satisfied at time  $i$ . We say “roughly speaking” because the planning graph records only a restricted subset of the possible negative interactions among actions; therefore, a literal might show up at level  $S_j$  when actually it could not be true until a later level, if at all. (A literal will never show up too late.) Despite the possible error, the level  $j$  at which a literal first appears is a good estimate of how difficult it is to achieve the literal from the initial state.

Planning graphs work only for propositional planning problems—ones with no variables. As we mentioned on page 368, it is straightforward to propositionalize a set of ac-



depending on the choice of actions in  $A_0$ , either, but not both, could be the result. In other words,  $S_1$  represents a belief state: a set of possible states. The members of this set are all subsets of the literals such that there is no mutex link between any members of the subset.

We continue in this way, alternating between state level  $S_i$  and action level  $A_i$  until we reach a point where two consecutive levels are identical. At this point, we say that the graph has **leveled off**. The graph in Figure 10.8 levels off at  $S_2$ .

LEVELED OFF

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 that two actions cannot both be executed at the same level. 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.

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.

A planning graph is polynomial in the size of the planning problem. For a planning problem with  $l$  literals and  $a$  actions, each  $S_i$  has no more than  $l$  nodes and  $l^2$  mutex links, and each  $A_i$  has no more than  $a + l$  nodes (including the no-ops),  $(a + l)^2$  mutex links, and  $2(al + l)$  precondition and effect links. Thus, an entire graph with  $n$  levels has a size of  $O(n(a + l)^2)$ . The time to build the graph has the same complexity.

### 10.3.1 Planning graphs for heuristic estimation

A planning graph, once constructed, is a rich source of information about the problem. First, if any goal literal fails to appear in the final level of the graph, then the problem is unsolvable. Second, we can estimate the cost of achieving any goal literal  $g_i$  from state  $s$  as the level at which  $g_i$  first appears in the planning graph constructed from initial state  $s$ . We call this the



LEVEL COST

**level cost** of  $g_i$ . In Figure 10.8, *Have*(*Cake*) has level cost 0 and *Eaten*(*Cake*) has level cost 1. It is easy to show (Exercise 10.10) that these estimates are admissible for the individual goals. The estimate might not always be accurate, 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 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 nonpersistence actions. Level costs extracted from serial graphs are often quite reasonable estimates of actual costs.

SERIAL PLANNING GRAPH

MAX-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 accurate.

LEVEL SUM

The **level sum** heuristic, following the subgoal independence assumption, returns the sum of the level costs of the goals; this can be inadmissible but works well in practice for problems that are largely decomposable. It is much more accurate than the number-of-unsatisfied-goals heuristic from Section 10.2. For our problem, the level-sum heuristic estimate for the conjunctive goal *Have*(*Cake*)  $\wedge$  *Eaten*(*Cake*) will be  $0 + 1 = 1$ , whereas the correct answer is 2, achieved by the plan [*Eat*(*Cake*), *Bake*(*Cake*)]. That doesn't seem so bad. A more serious error is that if *Bake*(*Cake*) were not in the set of actions, then the estimate would still be 1, when in fact the conjunctive goal would be impossible.

SET-LEVEL

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 is admissible, it dominates the max-level heuristic, and it works extremely well on tasks in which there is a good deal of interaction among subplans. It is not perfect, of course; for example, it ignores interactions among three or more literals.

As a tool for generating accurate heuristics, we can view the planning graph as a relaxed problem that is efficiently solvable. 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, 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 a 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 page 211.)

One example of an unsolvable problem that cannot be recognized as such by a planning graph is the blocks-world problem where the goal is to get block  $A$  on  $B$ ,  $B$  on  $C$ , and  $C$  on  $A$ . This is an impossible goal; a tower with the bottom on top of the top. But a planning graph

cannot detect the impossibility, because any two of the three subgoals are achievable. There are no mutexes between any pair of literals, only between the three as a whole. To detect that this problem is impossible, we would have to search over the planning graph.

### 10.3.2 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 10.9) repeatedly adds a level to a planning graph with EXPAND-GRAPH. Once all the goals show up as non-mutex in the graph, GRAPHPLAN calls EXTRACT-SOLUTION to search for a plan that solves the problem. If that fails, it expands another level and tries again, terminating with failure when there is no reason to go on.

```

function GRAPHPLAN(problem) returns solution or failure
  graph  $\leftarrow$  INITIAL-PLANNING-GRAPH(problem)
  goals  $\leftarrow$  CONJUNCTS(problem.GOAL)
  nogoods  $\leftarrow$  an empty hash table
  for tl = 0 to  $\infty$  do
    if goals all non-mutex in  $S_t$  of graph then
      solution  $\leftarrow$  EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
      if solution  $\neq$  failure then return solution
    if graph and nogoods have both leveled off then return failure
    graph  $\leftarrow$  EXPAND-GRAPH(graph, problem)

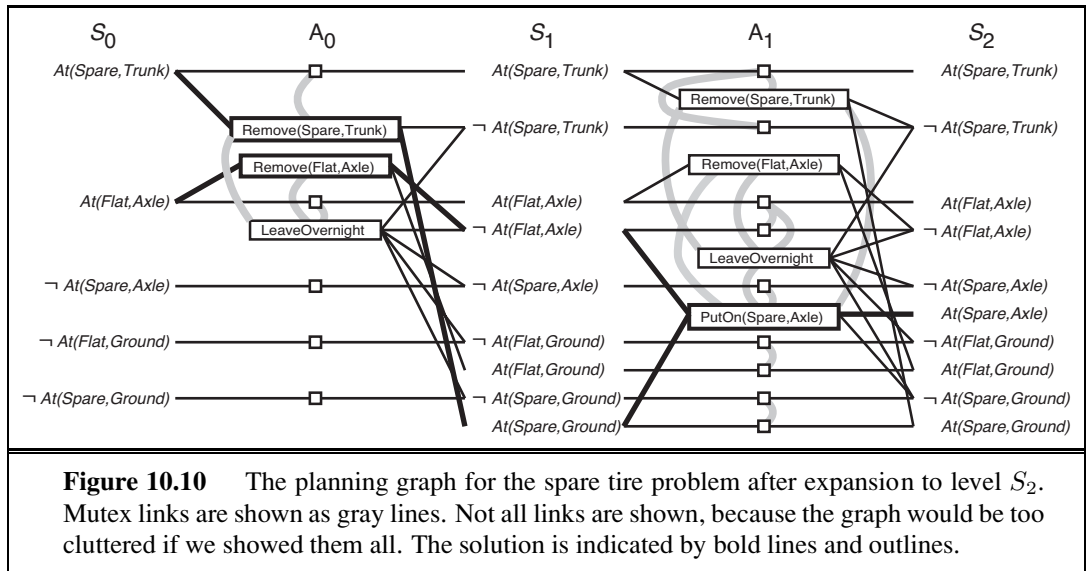
```

**Figure 10.9** The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.

Let us now trace the operation of GRAPHPLAN on the spare tire problem from page 370. The graph is shown in Figure 10.10. The first line of GRAPHPLAN initializes the planning graph to a one-level ( $S_0$ ) graph representing the initial state. The positive fluents from the problem description's initial state are shown, as are the relevant negative fluents. Not shown are the unchanging positive literals (such as  $Tire(Spare)$ ) and the irrelevant negative literals. The goal  $At(Spare, Axle)$  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 into  $A_0$  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.

$At(Spare, Axle)$  is still not present in  $S_1$ , so again we do not call EXTRACT-SOLUTION. We call EXPAND-GRAPH again, adding  $A_1$  and  $S_1$  and giving us the planning graph shown in Figure 10.10. 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. We can formulate EXTRACT-SOLUTION as a Boolean constraint satisfaction problem (CSP) where the variables are the actions at each level, the values for each variable are *in* or *out* of the plan, and the constraints are the mutexes and the need to satisfy each goal and precondition.

Alternatively, we can define EXTRACT-SOLUTION as a backward 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  $A_1$ .

In the case where EXTRACT-SOLUTION fails to find a solution for a set of goals at a given level, we record the  $(level, goals)$  pair as a **no-good**, just as we did in constraint learning for CSPs (page 220). Whenever EXTRACT-SOLUTION is called again with the same level and goals, we can find the recorded no-good and immediately return failure rather than searching again. We see shortly that no-goods are also used in the termination test.

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, prefer actions with easier preconditions. That is, choose an action such that the sum (or maximum) of the level costs of its preconditions is smallest.

### 10.3.3 Termination of GRAPHPLAN

So far, we have skated over the question of termination. Here we show that GRAPHPLAN will in fact terminate and return failure when there is no solution.

The first thing to understand is why we can't stop expanding the graph as soon as it has leveled off. Consider an air cargo domain with one plane and  $n$  pieces of cargo at airport  $A$ , all of which have airport  $B$  as their destination. In this version of the problem, only one piece of cargo can fit in the plane at a time. The graph will level off at level 4, reflecting the fact that for any single piece of cargo, we can load it, fly it, and unload it at the destination in three steps. But that does not mean that a solution can be extracted from the graph at level 4; in fact a solution will require  $4n - 1$  steps: for each piece of cargo we load, fly, and unload, and for all but the last piece we need to fly back to airport  $A$  to get the next piece.

How long do we have to keep expanding after the graph has leveled off? If the function EXTRACT-SOLUTION fails to find a solution, then there must have been at least one set of goals that were not achievable and were marked as a no-good. So if it is possible that there might be fewer no-goods in the next level, then we should continue. As soon as the graph itself and the no-goods have both leveled off, with no solution found, we can terminate with failure because there is no possibility of a subsequent change that could add a solution.

Now all we have to do is prove that the graph and the no-goods will always level off. The key to this proof is 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 the monotonic increase of literals; 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 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.

- *No-goods decrease monotonically*: If a set of goals is not achievable at a given level, then they are not achievable in any *previous* level. The proof is by contradiction: if they were achievable at some previous level, then we could just add persistence actions to make them achievable at a subsequent level.

Because the actions and literals increase monotonically and because there are only a finite number of actions and literals, there must come a level that has the same number of actions and literals as the previous level. Because mutexes and no-goods decrease, and because there can never be fewer than zero mutexes or no-goods, there must come a level that has the same number of mutexes and no-goods as the previous level. Once a graph has reached this state, then if one of the goals is missing or is mutex with another goal, then we can stop the GRAPHPLAN algorithm and return failure. That concludes a sketch of the proof; for more details see Ghallab *et al.* (2004).

| Year | Track       | Winning Systems (approaches)   |
|------|-------------|--|
| 2008 | Optimal     | GAMER (model checking, bidirectional search)                             |
| 2008 | Satisficing | LAMA (fast downward search with FF heuristic)                            |
| 2006 | Optimal     | SATPLAN, MAXPLAN (Boolean satisfiability)                                |
| 2006 | Satisficing | SGPLAN (forward search; partitions into independent subproblems)         |
| 2004 | Optimal     | SATPLAN (Boolean satisfiability)   |
| 2004 | Satisficing | FAST DIAGONALLY DOWNWARD (forward search with causal graph)              |
| 2002 | Automated   | LPG (local search, planning graphs converted to CSPs)                    |
| 2002 | Hand-coded  | TLPLAN (temporal action logic with control rules for forward search)     |
| 2000 | Automated   | FF (forward search)  |
| 2000 | Hand-coded  | TALPLANNER (temporal action logic with control rules for forward search) |
| 1998 | Automated   | IPP (planning graphs); HSP (forward search)                              |

**Figure 10.11** Some of the top-performing systems in the International Planning Competition. Each year there are various tracks: “Optimal” means the planners must produce the shortest possible plan, while “Satisficing” means nonoptimal solutions are accepted. “Hand-coded” means domain-specific heuristics are allowed; “Automated” means they are not.

## 10.4 OTHER CLASSICAL PLANNING APPROACHES

Currently the most popular and effective approaches to fully automated planning are:

- Translating to a Boolean satisfiability (SAT) problem
- Forward state-space search with carefully crafted heuristics (Section 10.2)
- Search using a planning graph (Section 10.3)

These three approaches are not the only ones tried in the 40-year history of automated planning. Figure 10.11 shows some of the top systems in the International Planning Competitions, which have been held every even year since 1998. In this section we first describe the translation to a satisfiability problem and then describe three other influential approaches: planning as first-order logical deduction; as constraint satisfaction; and as plan refinement.

### 10.4.1 Classical planning as Boolean satisfiability

In Section 7.7.4 we saw how SATPLAN solves planning problems that are expressed in propositional logic. Here we show how to translate a PDDL description into a form that can be processed by SATPLAN. The translation is a series of straightforward steps:

- Propositionalize the actions: replace each action schema with a set of ground actions formed by substituting constants for each of the variables. These ground actions are not part of the translation, but will be used in subsequent steps.
- Define the initial state: assert  $F^0$  for every fluent  $F$  in the problem’s initial state, and  $\neg F$  for every fluent not mentioned in the initial state.
- Propositionalize the goal: for every variable in the goal, replace the literals that contain the variable with a disjunction over constants. For example, the goal of having block  $A$

on another block,  $On(A, x) \wedge Block(x)$  in a world with objects  $A, B$  and  $C$ , would be replaced by the goal

$$(On(A, A) \wedge Block(A)) \vee (On(A, B) \wedge Block(B)) \vee (On(A, C) \wedge Block(C)).$$

- Add successor-state axioms: For each fluent  $F$ , add an axiom of the form

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t),$$

where  $ActionCausesF$  is a disjunction of all the ground actions that have  $F$  in their add list, and  $ActionCausesNotF$  is a disjunction of all the ground actions that have  $F$  in their delete list.

- Add precondition axioms: For each ground action  $A$ , add the axiom  $A^t \Rightarrow PRE(A)^t$ , that is, if an action is taken at time  $t$ , then the preconditions must have been true.
- Add action exclusion axioms: say that every action is distinct from every other action.

The resulting translation is in the form that we can hand to SATPLAN to find a solution.

### 10.4.2 Planning as first-order logical deduction: Situation calculus

PDDL is a language that carefully balances the expressiveness of the language with the complexity of the algorithms that operate on it. But some problems remain difficult to express in PDDL. For example, we can't express the goal "move all the cargo from  $A$  to  $B$  regardless of how many pieces of cargo there are" in PDDL, but we can do it in first-order logic, using a universal quantifier. Likewise, first-order logic can concisely express global constraints such as "no more than four robots can be in the same place at the same time." PDDL can only say this with repetitious preconditions on every possible action that involves a move.

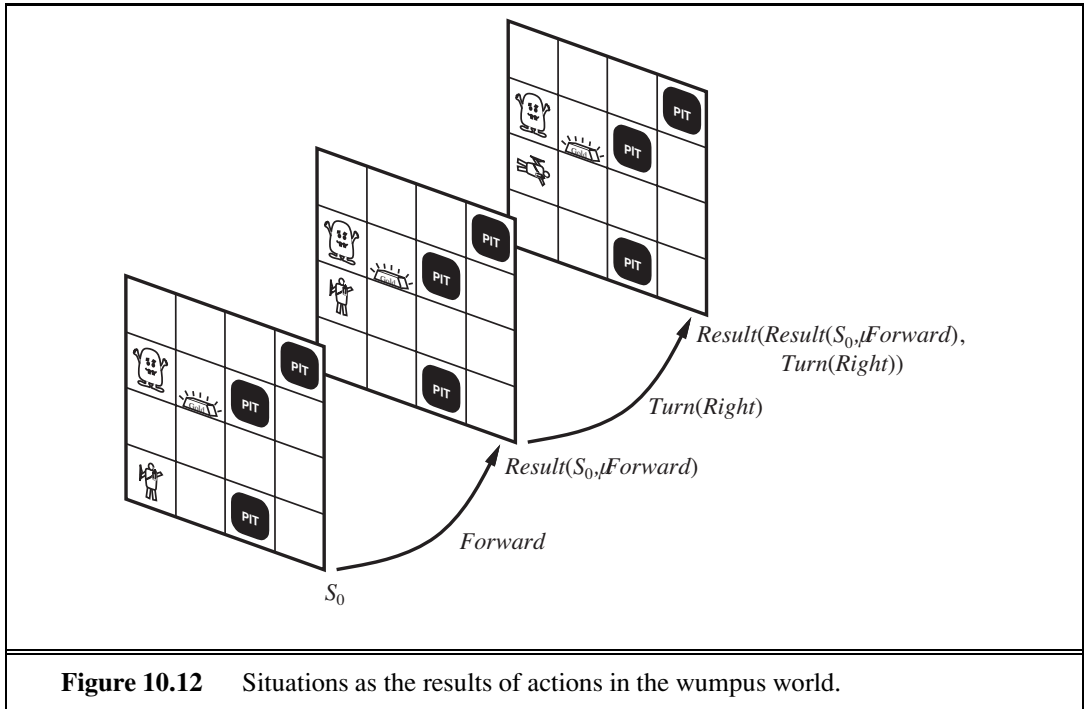
The propositional logic representation of planning problems also has limitations, such as the fact that the notion of time is tied directly to fluents. For example,  $South^2$  means "the agent is facing south at time 2." With that representation, there is no way to say "the agent would be facing south at time 2 if it executed a right turn at time 1; otherwise it would be facing east." First-order logic lets us get around this limitation by replacing the notion of linear time with a notion of branching *situations*, using a representation called **situation calculus** that works like this:

- The initial state is called a **situation**. If  $s$  is a situation and  $a$  is an action, then  $RESULT(s, a)$  is also a situation. There are no other situations. Thus, a situation corresponds to a sequence, or history, of actions. You can also think of a situation as the result of applying the actions, but note that two situations are the same only if their start and actions are the same:  $(RESULT(s, a) = RESULT(s', a')) \Leftrightarrow (s = s' \wedge a = a')$ . Some examples of actions and situations are shown in Figure 10.12.
- A function or relation that can vary from one situation to the next is a **fluent**. By convention, the situation  $s$  is always the last argument to the fluent, for example  $At(x, l, s)$  is a relational fluent that is true when object  $x$  is at location  $l$  in situation  $s$ , and  $Location$  is a functional fluent such that  $Location(x, s) = l$  holds in the same situations as  $At(x, l, s)$ .
- Each action's preconditions are described with a **possibility axiom** that says when the action can be taken. It has the form  $\Phi(s) \Rightarrow Poss(a, s)$  where  $\Phi(s)$  is some formula

SITUATION  
CALCULUS

SITUATION

POSSIBILITY AXIOM



**Figure 10.12** Situations as the results of actions in the wumpus world.

involving  $s$  that describes the preconditions. An example from the wumpus world says that it is possible to shoot if the agent is alive and has an arrow:

$$Alive(Agent, s) \wedge Have(Agent, Arrow, s) \Rightarrow Poss(Shoot, s)$$

- Each fluent is described with a **successor-state axiom** that says what happens to the fluent, depending on what action is taken. This is similar to the approach we took for propositional logic. The axiom has the form

$$\begin{aligned} \text{Action is possible} &\Rightarrow \\ &(\text{Fluent is true in result state} \Leftrightarrow \text{Action's effect made it true} \\ &\quad \vee \text{It was true before and action left it alone}) . \end{aligned}$$

For example, the axiom for the relational fluent *Holding* says that the agent is holding some gold  $g$  after executing a possible action if and only if the action was a *Grab* of  $g$  or if the agent was already holding  $g$  and the action was not releasing it:

$$\begin{aligned} Poss(a, s) &\Rightarrow \\ &(\text{Holding}(Agent, g, \text{Result}(a, s)) \Leftrightarrow \\ &\quad a = \text{Grab}(g) \vee (\text{Holding}(Agent, g, s) \wedge a \neq \text{Release}(g))) . \end{aligned}$$

UNIQUE ACTION  
AXIOMS

- We need **unique action axioms** so that the agent can deduce that, for example,  $a \neq \text{Release}(g)$ . For each distinct pair of action names  $A_i$  and  $A_j$  we have an axiom that says the actions are different:

$$A_i(x, \dots) \neq A_j(y, \dots)$$



and for each action name  $A_i$  we have an axiom that says two uses of that action name are equal if and only if all their arguments are equal:

$$A_i(x_1, \dots, x_n) = A_i(y_1, \dots, y_n) \Leftrightarrow x_1 = y_1 \wedge \dots \wedge x_n = y_n.$$

- A solution is a situation (and hence a sequence of actions) that satisfies the goal.

Work in situation calculus has done a lot to define the formal semantics of planning and to open up new areas of investigation. But so far there have not been any practical large-scale planning programs based on logical deduction over the situation calculus. This is in part because of the difficulty of doing efficient inference in FOL, but is mainly because the field has not yet developed effective heuristics for planning with situation calculus.

### 10.4.3 Planning as constraint satisfaction

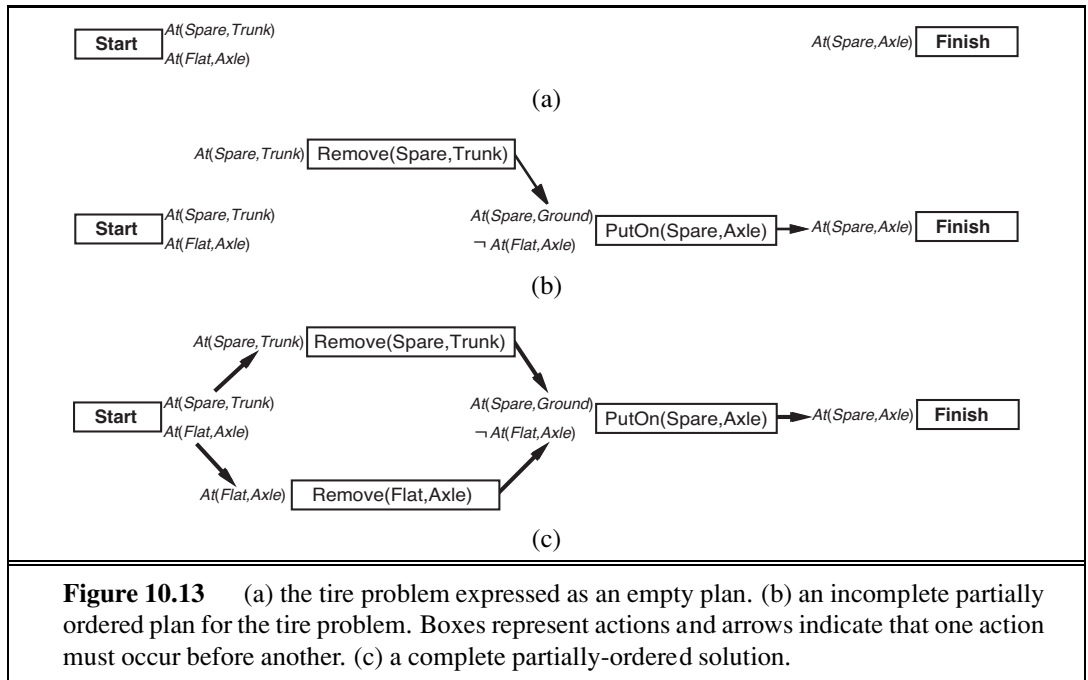
We have seen that constraint satisfaction has a lot in common with Boolean satisfiability, and we have seen that CSP techniques are effective for scheduling problems, so it is not surprising that it is possible to encode a bounded planning problem (i.e., the problem of finding a plan of length  $k$ ) as a constraint satisfaction problem (CSP). The encoding is similar to the encoding to a SAT problem (Section 10.4.1), with one important simplification: at each time step we need only a single variable,  $Action^t$ , whose domain is the set of possible actions. We no longer need one variable for every action, and we don't need the action exclusion axioms. It is also possible to encode a planning graph into a CSP. This is the approach taken by GP-CSP (Do and Kambhampati, 2003).

### 10.4.4 Planning as refinement of partially ordered plans

All the approaches we have seen so far construct *totally ordered* plans consisting of a strictly linear sequences of actions. This representation ignores the fact that many subproblems are independent. A solution to an air cargo problem consists of a totally ordered sequence of actions, yet if 30 packages are being loaded onto one plane in one airport and 50 packages are being loaded onto another at another airport, it seems pointless to come up with a strict linear ordering of 80 load actions; the two subsets of actions should be thought of independently.

An alternative is to represent plans as *partially ordered* structures: a plan is a set of actions and a set of constraints of the form  $Before(a_i, a_j)$  saying that one action occurs before another. In the bottom of Figure 10.13, we see a partially ordered plan that is a solution to the spare tire problem. Actions are boxes and ordering constraints are arrows. Note that  $Remove(Spare, Trunk)$  and  $Remove(Flat, Axle)$  can be done in either order as long as they are both completed before the  $PutOn(Spare, Axle)$  action.

Partially ordered plans are created by a *search through the space of plans* rather than through the state space. We start with the empty plan consisting of just the initial state and the goal, with no actions in between, as in the top of Figure 10.13. The search procedure then looks for a **flaw** in the plan, and makes an addition to the plan to correct the flaw (or if no correction can be made, the search backtracks and tries something else). A flaw is anything that keeps the partial plan from being a solution. For example, one flaw in the empty plan is that no action achieves  $At(Spare, Axle)$ . One way to correct the flaw is to insert into the plan



the action  $PutOn(Spare, Axle)$ . Of course that introduces some new flaws: the preconditions of the new action are not achieved. The search keeps adding to the plan (backtracking if necessary) until all flaws are resolved, as in the bottom of Figure 10.13. At every step, we make the **least commitment** possible to fix the flaw. For example, in adding the action  $Remove(Spare, Trunk)$  we need to commit to having it occur before  $PutOn(Spare, Axle)$ , but we make no other commitment that places it before or after other actions. If there were a variable in the action schema that could be left unbound, we would do so.

LEAST COMMITMENT

In the 1980s and 90s, partial-order planning was seen as the best way to handle planning problems with independent subproblems—after all, it was the only approach that explicitly represents independent branches of a plan. On the other hand, it has the disadvantage of not having an explicit representation of states in the state-transition model. That makes some computations cumbersome. By 2000, forward-search planners had developed excellent heuristics that allowed them to efficiently discover the independent subproblems that partial-order planning was designed for. As a result, partial-order planners are not competitive on fully automated classical planning problems.

However, partial-order planning remains an important part of the field. For some specific tasks, such as operations scheduling, partial-order planning with domain specific heuristics is the technology of choice. Many of these systems use libraries of high-level plans, as described in Section 11.2. Partial-order planning is also often used in domains where it is important for humans to understand the plans. Operational plans for spacecraft and Mars rovers are generated by partial-order planners and are then checked by human operators before being uploaded to the vehicles for execution. The plan refinement approach makes it easier for the humans to understand what the planning algorithms are doing and verify that they are correct.

## 10.5 ANALYSIS OF PLANNING APPROACHES

Planning combines the two major areas of AI we have covered so far: *search* and *logic*. 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 both to improvements in performance amounting to several orders of magnitude in the last decade and to 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  $n$  propositions in a domain, then there are  $2^n$  states. As we have seen, planning is PSPACE-hard. Against such pessimism, the identification of independent subproblems can be a powerful weapon. In the best case—full decomposability of the problem—we get an exponential speedup. Decomposability is destroyed, however, by negative interactions between actions. GRAPHPLAN records mutexes to point out where the difficult interactions are. SATPLAN represents a similar range of mutex relations, but does so by using the general CNF form rather than a specific data structure. Forward search addresses the problem heuristically by trying to find patterns (subsets of propositions) that cover the independent subproblems. Since this approach is heuristic, it can work even when the subproblems are not completely independent.

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*, as in Figure 10.4 on page 371), 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 without backtracking (although it might not always find the shortest plan).

As a more complex example, for the Remote Agent planner that 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.

Planners such as GRAPHPLAN, SATPLAN, and FF have moved the field of planning forward, by raising the level of performance of planning systems, by clarifying the representational and combinatorial issues involved, and by the development of useful heuristics. However, there is a question of how far these techniques will scale. It seems likely that further progress on larger problems cannot rely only on factored and propositional representations, and will require some kind of synthesis of first-order and hierarchical representations with the efficient heuristics currently in use.

SERIALIZABLE  
SUBGOAL

---

## 10.6 SUMMARY

---

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

- Planning systems are problem-solving algorithms that operate on explicit propositional or relational 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.
- PDDL, the Planning Domain Definition Language, describes the initial and goal states as conjunctions of literals, and actions in terms of their preconditions and effects.
- State-space search can operate in the forward direction (**progression**) or the backward direction (**regression**). Effective heuristics can be derived by subgoal independence assumptions and by various relaxations of the planning problem.
- 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 (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.
- Other approaches include first-order deduction over situation calculus axioms; encoding a planning problem as a Boolean satisfiability problem or as a constraint satisfaction problem; and explicitly searching through the space of partially ordered plans.
- 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. Bylander (1992) shows simple STRIPS planning to be PSPACE-complete. Fikes and Nilsson (1993) give a historical retrospective on the STRIPS project and its relationship to more recent planning efforts.

The representation language used by STRIPS has been far more influential than its algorithmic approach; what we call the “classical” language is close to what STRIPS used.

The Action Description Language, or ADL (Pednault, 1986), relaxed some of the STRIPS restrictions 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 planning problems and has been used as the standard language for the International Planning Competition since 1998. There have been several extensions; the most recent version, PDDL 3.0, includes plan constraints and preferences (Gerevini and Long, 2005).

LINEAR PLANNING

Planners in the early 1970s generally considered totally ordered action sequences. Problem decomposition was achieved by computing a subplan for each subgoal and then 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 10.7), 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 with logic programming: WARPLAN is only 100 lines of code, a small fraction of the size of comparable planners of the time.

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.

Partial-order planning dominated the next 20 years of research, yet the first clear formal exposition was TWEAK (Chapman, 1987), a planner that was simple enough to allow proofs of completeness and intractability (NP-hardness and undecidability) of various planning problems. Chapman's work led to a straightforward description of a complete partial-order planner (McAllester and Rosenblitt, 1991), then to the widely distributed implementations SNLP (Soderland and Weld, 1991) and UCPOP (Penberthy and Weld, 1992). Partial-order planning fell out of favor in the late 1990s as faster methods emerged. Nguyen and Kambhampati (2001) suggest that a reconsideration is merited: with accurate heuristics derived from a planning graph, their REPOP planner scales up much better than GRAPHPLAN in parallelizable domains and is competitive with the fastest state-space planners.

The resurgence of interest in state-space planning was pioneered by Drew McDermott's UNPOP program (1996), which was the first to suggest the ignore-delete-list heuristic. 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; Haslum *et al.*, 2005; Haslum, 2006) were the first to make

state-space search practical for large planning problems. HSP searches in the forward direction while HSPR (Bonet and Geffner, 1999) searches backward. The most successful state-space searcher to date is FF (Hoffmann, 2001; Hoffmann and Nebel, 2001; Hoffmann, 2005), winner of the AIPS 2000 planning competition. FASTDOWNWARD (Helmert, 2006) is a forward state-space search planner that preprocesses the action schemas into an alternative representation which makes some of the constraints more explicit. FASTDOWNWARD (Helmert and Richter, 2004; Helmert, 2006) won the 2004 planning competition, and LAMA (Richter and Westphal, 2008), a planner based on FASTDOWNWARD with improved heuristics, won the 2008 competition.

Bylander (1994) and Ghallab *et al.* (2004) discuss the computational complexity of several variants of the planning problem. Helmert (2003) proves complexity bounds for many of the standard benchmark problems, and Hoffmann (2005) analyzes the search space of the ignore-delete-list heuristic. Heuristics for the set-covering problem are discussed by Caprara *et al.* (1995) for scheduling operations of the Italian railway. Edelkamp (2009) and Haslum *et al.* (2007) describe how to construct pattern databases for planning heuristics. As we mentioned in Chapter 3, Felner *et al.* (2004) show encouraging results using pattern databases for sliding blocks puzzles, which can be thought of as a planning domain, but Hoffmann *et al.* (2006) show some limitations of abstraction for classical planning problems.

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 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) thoroughly analyze heuristics derived from planning graphs. Our discussion of planning graphs is based partly on this work and on lecture notes and articles by Subbarao Kambhampati (Bryce and Kambhampati, 2007). 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, 2003), searched planning graphs using a local search technique inspired by WALKSAT.

The situation calculus approach to planning was introduced by John McCarthy (1963). The version we show here was proposed by Ray Reiter (1991, 2001).

Kautz *et al.* (1996) investigated various ways to propositionalize action schemas, 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). CPLAN, a planner based on constraint satisfaction, was described by van Beek and Chen (1999).

Most recently, there has been interest in the representation of plans as **binary decision diagrams**, compact data structures for Boolean expressions 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.

*Readings in Planning* (Allen *et al.*, 1990) is a comprehensive anthology of early work in the field. Weld (1994, 1999) provides two excellent surveys of planning algorithms of the 1990s. 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. *Automated Planning* (Ghallab *et al.*, 2004) is an excellent textbook on all aspects of planning. LaValle's text *Planning Algorithms* (2006) covers both classical and stochastic planning, with extensive coverage of robot motion planning.

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, the International Workshop on Planning and Scheduling for Space, and the European Conference on Planning.

---

## EXERCISES

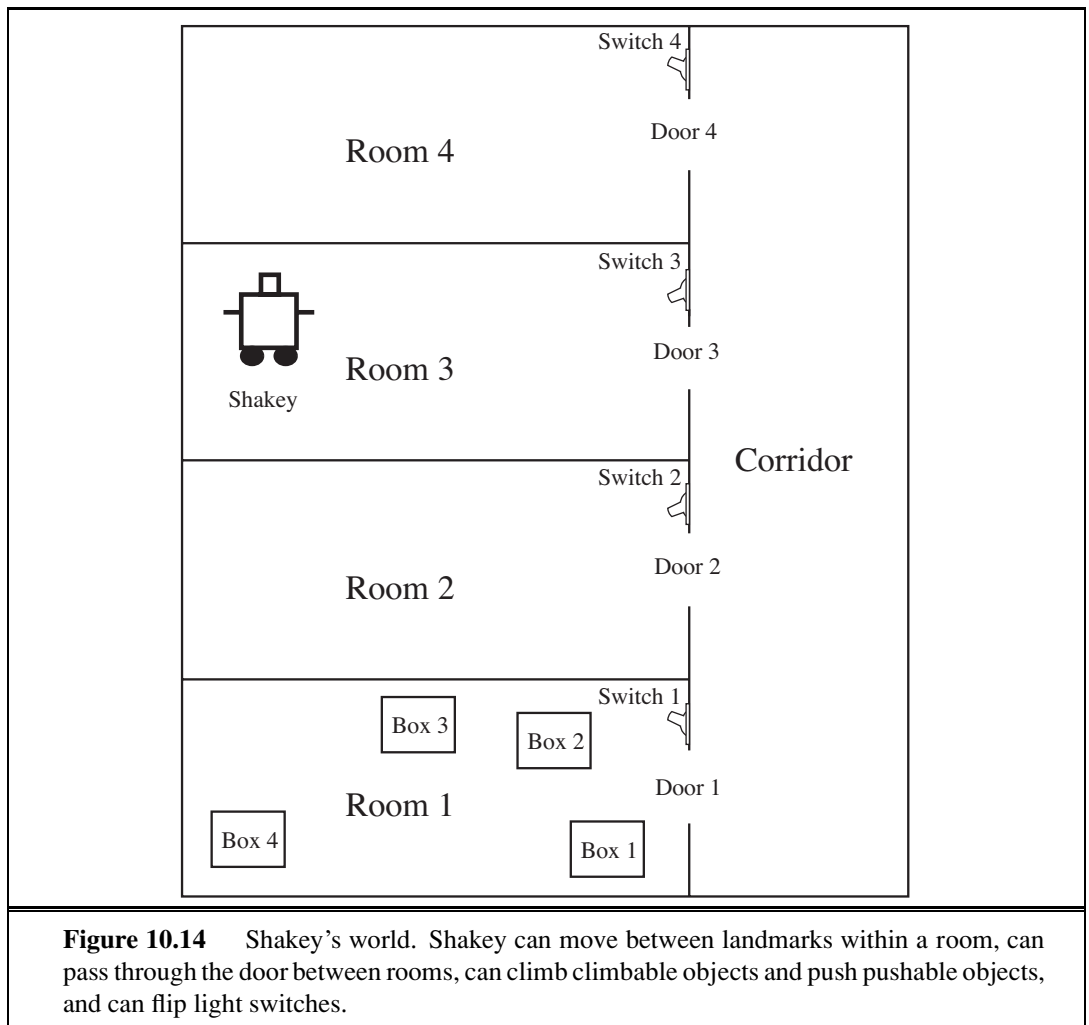
**10.1** Describe the differences and similarities between problem solving and planning.

**10.2** Given the action schemas and initial state from Figure 10.1, what are all the applicable concrete instances of  $Fly(p, from, to)$  in the state described by

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

**10.3** 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. The result of a *Grasp* is that the monkey holds 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 the six action schemas.
- 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 classical planning system?
- d. Your schema for pushing is probably incorrect, because if the object is too heavy, its position will remain the same when the *Push* schema is applied. Fix your action schema to account for heavy objects.

**10.4** The original STRIPS planner was designed to control Shakey the robot. Figure 10.14 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 could not climb on a box or toggle a switch, but the planner was capable of finding and printing out plans that were beyond the robot's abilities. Shakey's six actions are the following:

- $Go(x, y, r)$ , which requires that Shakey be *At*  $x$  and that  $x$  and  $y$  are locations *In* the same room  $r$ . 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:  $Push(b, x, y, r)$ . You will need the predicate *Box* and constants for the boxes.
- Climb onto a box from position  $x$ :  $ClimbUp(x, b)$ ; climb down from a box to position  $x$ :  $ClimbDown(b, x)$ . We will need the predicate *On* and the constant *Floor*.
- Turn a light switch on or off:  $TurnOn(s, b)$ ;  $TurnOff(s, b)$ . To turn a light on or off, Shakey must be on top of a box at the light switch's location.

Write PDDL sentences for Shakey's six actions and the initial state from Figure 10.14. Construct a plan for Shakey to get  $Box_2$  into  $Room_2$ .

**10.5** A finite Turing machine has a finite one-dimensional tape of cells, each cell containing one of a finite number of symbols. One cell has a read and write head above it. There is a finite set of states the machine can be in, one of which is the accept state. At each time step, depending on the symbol on the cell under the head and the machine's current state, there are a set of actions we can choose from. Each action involves writing a symbol to the cell under the head, transitioning the machine to a state, and optionally moving the head left or right. The mapping that determines which actions are allowed is the Turing machine's program. Your goal is to control the machine into the accept state.

Represent the Turing machine acceptance problem as a planning problem. If you can do this, it demonstrates that determining whether a planning problem has a solution is at least as hard as the Turing acceptance problem, which is PSPACE-hard.

**10.6** Explain why dropping negative effects from every action schema in a planning problem results in a relaxed problem.

**10.7** Figure 10.4 (page 371) shows a blocks-world problem that is 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 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.

SUSSMAN ANOMALY

**10.8** Prove that backward search with PDDL problems is complete.

**10.9** Construct levels 0, 1, and 2 of the planning graph for the problem in Figure 10.1.

**10.10** Prove the following assertions about planning graphs:

- a. A literal that does not appear in the final level of the graph cannot be achieved.

- b. The level cost of a literal in a serial graph is no greater than the actual cost of an optimal plan for achieving it.

**10.11** The set-level heuristic (see page 382) uses a planning graph to estimate the cost of achieving a conjunctive goal from the current state. What relaxed problem is the set-level heuristic the solution to?

**10.12** 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?

**10.13** We contrasted forward and backward state-space searchers 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.

**10.14** Up to now we have assumed that the plans we create always make sure that an action's preconditions are satisfied. Let us now investigate what propositional successor-state axioms such as  $HaveArrow^{t+1} \Leftrightarrow (HaveArrow^t \wedge \neg Shoot^t)$  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 satisfied.
- b. Consider a plan  $p$  that contains the actions required to achieve a goal but also includes illegal actions. Is it the case that

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

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

**10.15** Consider how to translate a set of action schemas into the successor-state axioms of situation calculus.

- a. Consider the schema for  $Fly(p, from, to)$ . Write a logical definition for the predicate  $Poss(Fly(p, from, to), s)$ , which is true if the preconditions for  $Fly(p, from, to)$  are satisfied in situation  $s$ .
- b. 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.

- c. 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.
- d. Finally, develop a general and precisely specified procedure for carrying out the translation from a set of action schemas to a set of successor-state axioms.

**10.16** In the SATPLAN algorithm in Figure 7.22 (page 272), 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  $g^0 \vee g^1 \vee \dots \vee g^{T_{\max}}$ .

- 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.