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

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

EMPTY-DELETE-LIST

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

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

11.3 PARTIAL-ORDER PLANNING

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

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

LEAST COMMITMENT

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

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

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

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

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

PARTIAL-ORDER PLANNER

LINEARIZATION

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

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

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

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

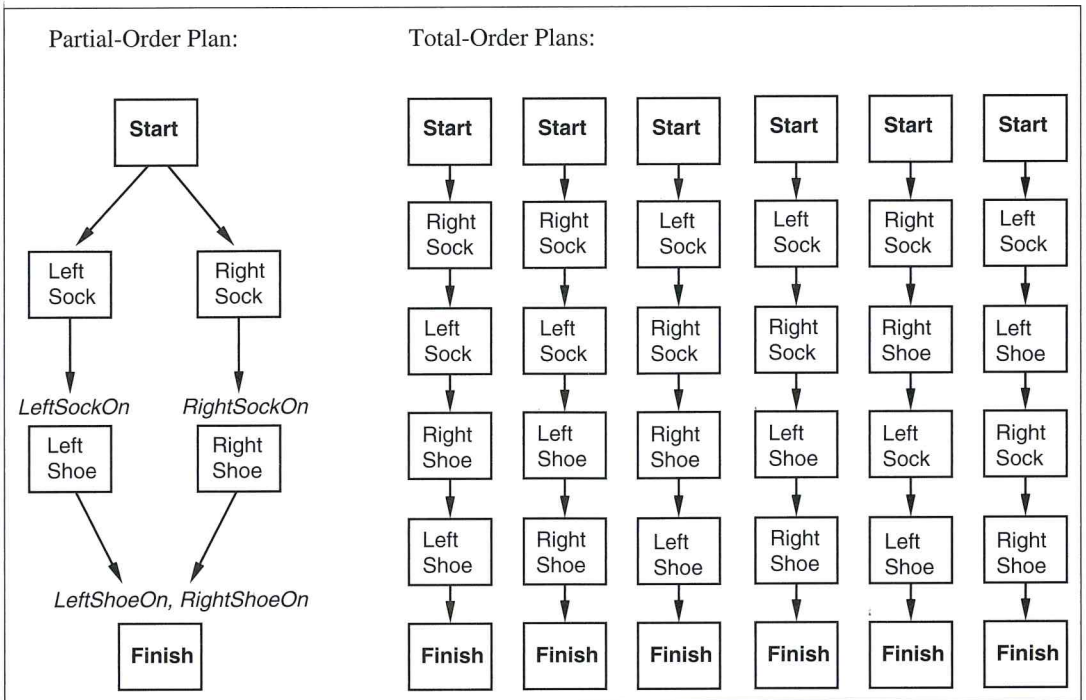

Figure 11.6 A partial-order plan for putting on shoes and socks, and the six corresponding linearizations into total-order plans.

ORDERING CONSTRAINTS

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

CAUSAL LINKS
ACHIEVES

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

$$RightSock \xrightarrow{RightSockOn} RightShoe$$

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

CONFLICTS

OPEN PRECONDITIONS

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

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

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

CONSISTENT PLAN

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

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

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

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

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

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

A partial-order planning example

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

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

Figure 11.7 The simple flat tire problem description.

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

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

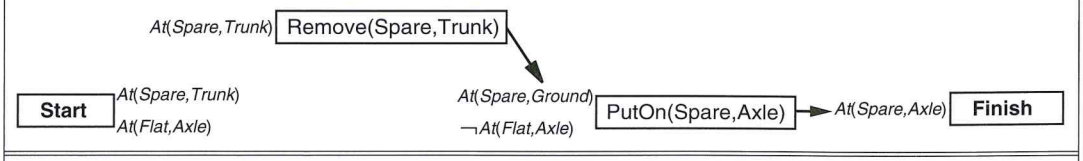

Figure 11.8 The incomplete partial-order plan for the tire problem, after choosing actions for the first two open preconditions. Boxes represent actions, with preconditions on the left and effects on the right. (Effects are omitted, except for that of the *Start* action.) Dark arrows represent causal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

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

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

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

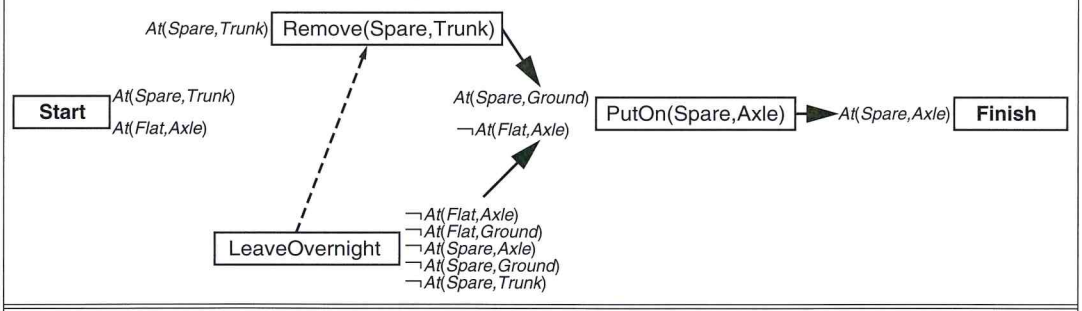

Figure 11.9 The plan after choosing $LeaveOvernight$ as the action for achieving $\neg At(Flat, Axle)$. To avoid a conflict with the causal link from $Remove(Spare, Trunk)$ that protects $At(Spare, Ground)$, $LeaveOvernight$ is constrained to occur before $Remove(Spare, Trunk)$, as shown by the dashed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

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

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

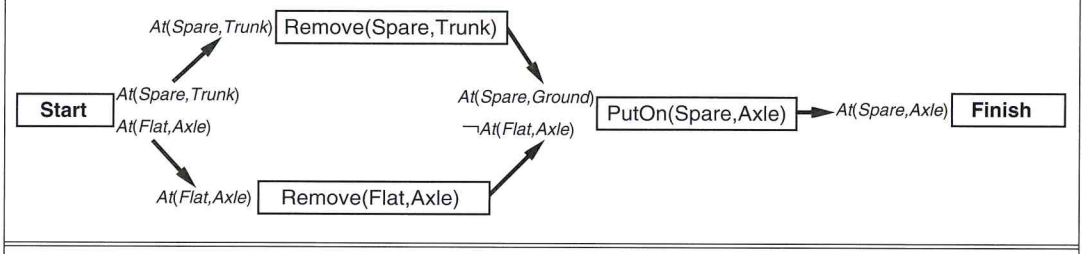

Figure 11.10 The final solution to the tire problem. Note that $Remove(Spare, Trunk)$ and $Remove(Flat, Axle)$ can be done in either order, as long as they are completed before the $PutOn(Spare, Axle)$ action.

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

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

Partial-order planning with unbound variables

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

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

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

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

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

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

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

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

INEQUALITY CONSTRAINTS

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

Heuristics for partial-order planning

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

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

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

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

11.4 PLANNING GRAPHS

PLANNING GRAPH

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

LEVELS

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

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

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