The goal state is defined by a *Finish* step describing the objects to be acquired and the final destination to be reached:

\[
\text{Op(ACTION: Finish, PRECOND: Have(Drill) \& Have(Milk) \& Have(Banana) \& At(Home))}
\]

The actions themselves are defined as follows:

\[
\begin{align*}
\text{Op(ACTION: Go(where), PRECOND: At(where), EFFECT: At(where) \& \neg At(where))} \\
\text{Op(ACTION: Buy(x), PRECOND: At(store) \& Sells(store, x), EFFECT: Have(x))}
\end{align*}
\]

Figure 11.6 shows a diagram of the initial plan for this problem. We will develop a solution to the problem step by step, showing at each point a figure illustrating the partial plan at that point in the development. As we go along, we will note some of the properties we require for the planning algorithm. After we finish the example, we will present the algorithm in detail.

![Figure 11.6: The initial plan for the shopping problem.](image)

The first thing to notice about Figure 11.6 is that there are many possible ways in which the initial plan can be elaborated. Some choices will work, and some will not. As we work out the solution to the problem, we will show some correct choices and some incorrect choices. For simplicity, we will start with some correct choices. In Figure 11.7 (top), we have selected three *Buy* actions to achieve three of the preconditions of the *Finish* action. In each case there is only one possible choice because the operator library offers no other way to achieve these conditions.

The bold arrows in the figure are causal links. For example, the leftmost causal link in the figure means that the step *Buy(Drill)* was added in order to achieve the *Finish* step's *Have(Drill)* precondition. The planner will make sure that this condition is maintained by *protecting* it: if a step might delete the *Have(Drill)* condition, then it will not be inserted between the *Buy(Drill)* step and the *Finish* step. Light arrows in the figure show ordering constraints. By definition, all actions are constrained to come after the *Start* action. Also, all causes are constrained to come before their effects, so you can think of each bold arrow as having a light arrow underneath it.

The second stage in Figure 11.7 shows the situation after the planner has chosen to achieve the *Sells* preconditions by linking them to the initial state. Again, the planner has no choice here because there is no other operator that achieves *Sells*. 
A Partial-Order Planning Example

Although it may not seem like we have done much yet, this is actually quite an improvement over what we could have done with the problem-solving approach. First, out of all the things that one can buy, and all the places that one can go, we were able to choose just the right Buy actions and just the right places, without having to waste time considering the others. Then, once we have chosen the actions, we need not decide how to order them; a partial-order planner can make that decision later.

In Figure 11.8, we extend the plan by choosing two Go actions to get us to the hardware store and supermarket, thus achieving the At preconditions of the Buy actions.

So far, everything has been easy. A planner could get this far without having to do any search. Now it gets harder. The two Go actions have unachieved preconditions that interact with each other, because the agent cannot be At two places at the same time. Each Go action has a precondition At(x), where x is the location that the agent was at before the Go action. Suppose
the planner tries to achieve the preconditions of \textit{Go(HWS)} and \textit{Go(SM)} by linking them to the \textit{At(Home)} condition in the initial state. This results in the plan shown in Figure 11.9.

Unfortunately, this will lead to a problem. The step \textit{Go(HWS)} adds the condition \textit{At(HWS)}, but it also deletes the condition \textit{At(Home)}. So if the agent goes to the hardware store, it can no longer go from home to the supermarket. (That is, unless it introduces another step to go back home from the hardware store—but the causal link means that the start step, not some other step...)

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure11.8}
\caption{A partial plan that achieves \textit{At} preconditions of the three \textit{Buy} actions.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure11.9}
\caption{A flawed plan that gets the agent to the hardware store and the supermarket.}
\end{figure}
Section 11.5. A Partial-Order Planning Example

achieves the \textit{At(Home)} precondition.) On the other hand, if the agent goes to the supermarket first, then it cannot go from home to the hardware store.

At this point, we have reached a dead end in the search for a solution, and must back up and try another choice. The interesting part is seeing how \textit{a planner could notice that this partial plan is a dead end without wasting a lot of time on it}. The key is that the causal links in a partial plan are \textbf{protected links}. A causal link is protected by ensuring that \textbf{threats}—that is, steps that might delete (or \textit{clobber}) the protected condition—are ordered to come before or after the protected link. Figure 11.10(a) shows a threat: The causal link \textit{S}_1 \rightarrow \textit{S}_2 \rightarrow \textit{S}_3 \rightarrow \textit{c} is threatened by the new step \textit{S}_3 because one effect of \textit{S}_3 is to delete \textit{c}. The way to resolve the threat is to add ordering constraints to make sure that \textit{S}_3 does not intervene between \textit{S}_1 and \textit{S}_2. If \textit{S}_3 is placed before \textit{S}_1 this is called \textit{demotion} (see Figure 11.10(b)), and if it is placed after \textit{S}_2, it is called \textit{promotion} (see Figure 11.10(c)).

\begin{figure}[h]
\begin{center}
\includegraphics[width=0.6\textwidth]{causal_links.png}
\end{center}
\caption{Protecting causal links. In (a), the step \textit{S}_3 threatens a condition \textit{c} that is established by \textit{S}_1 and protected by the causal link from \textit{S}_1 to \textit{S}_2. In (b), \textit{S}_3 has been demoted to come before \textit{S}_1, and in (c) it has been promoted to come after \textit{S}_2.}
\end{figure}

In Figure 11.9, there is no way to resolve the threat that each \textit{Go} step poses to the other. Whichever \textit{Go} step comes first will delete the \textit{At(Home)} condition on the other step. Whenever the planner is unable to resolve a threat by promotion or demotion, it gives up on the partial plan and backs up to try a different choice at some earlier point in the planning process.

Suppose the next choice is to try a different way to achieve the \textit{At(x)} precondition of the \textit{Go(SM)} step, this time by adding a causal link from \textit{Go(HWS)} to \textit{Go(SM)}. In other words, the plan is to go from home to the hardware store and then to the supermarket. This introduces another threat. Unless the plan is further refined, it will allow the agent to go from the hardware store to the supermarket without first buying the drill (which was why it went to the hardware store in the first place). However much this might resemble human behavior, we would prefer our planning agent to avoid such forgetfulness. Technically, the \textit{Go(SM)} step threatens the \textit{At(HWS)} precondition of the \textit{Buy(Drill)} step, which is protected by a causal link. The threat is resolved by constraining \textit{Go(SM)} to come after \textit{Buy(Drill)}. Figure 11.11 shows this.
Only the \( \text{At(Home)} \) precondition of the \( \text{Finish} \) step remains unachieved. Adding a \( \text{Go(Home)} \) step achieves it, but introduces an \( \text{At(x)} \) precondition that needs to be achieved.\(^6\)

Again, the protection of causal links will help the planner decide how to do this:

- If it tries to achieve \( \text{At(x)} \) by linking to \( \text{At(Home)} \) in the initial state, there will be no way to resolve the threats caused by \( \text{Go(HWS)} \) and \( \text{Go(SM)} \).
- If it tries to link \( \text{At(x)} \) to the \( \text{Go(HWS)} \) step, there will be no way to resolve the threat posed by the \( \text{Go(SM)} \) step, which is already constrained to come after \( \text{Go(HWS)} \).
- A link from \( \text{Go(SM)} \) to \( \text{At(x)} \) means that \( x \) is bound to \( \text{SM} \), so that now the \( \text{Go(Home)} \) step deletes the \( \text{At(SM)} \) condition. This results in threats to the \( \text{At(SM)} \) preconditions of \( \text{Buy(Milk)} \) and \( \text{Buy(Bananas)} \), but these can be resolved by ordering \( \text{Go(Home)} \) to come after these steps (Figure 11.11).

Figure 11.12 shows the complete solution plan, with the steps redrawn to reflect the ordering constraints on them. The result is an almost totally ordered plan; the only ambiguity is that \( \text{Buy(Milk)} \) and \( \text{Buy(Bananas)} \) can come in either order.

Let us take stock of what our partial-order planner has accomplished. It can take a problem that would require many thousands of search states for a problem-solving approach, and solve it with only a few search states. Moreover, the least commitment nature of the planner means it only needs to search at all in places where subplans interact with each other. Finally, the causal links allow the planner to recognize when to abandon a doomed plan without wasting a lot of time expanding irrelevant parts of the plan.

\(^6\) Notice that the \( \text{Go(Home)} \) step also has the effect \( \neg \text{At(x)} \), meaning that the step will delete an \( \text{At} \) condition for some location yet to be decided. This is a possible threat to protected conditions in the plan such as \( \text{At(SM)} \), but we will not worry about it for now. Possible threats are dealt with in Section 11.7.
11.6 A PARTIAL-ORDER PLANNING ALGORITHM

In this section, we develop a more formal algorithm for the planner sketched in the previous section. We call the algorithm POP, for Partial-Order Planner. The algorithm appears in Figure 11.13. (Notice that POP is written as a nondeterministic algorithm, using choose and fail rather than explicit loops. Nondeterministic algorithms are explained in Appendix B.)

POP starts with a minimal partial plan, and on each step extends the plan by achieving a precondition \( c \) of a step \( S \). It does this by choosing some operator—either from the existing steps of the plan or from the pool of operators—that achieves the precondition. It records the causal link for the newly achieved precondition, and then resolves any threats to causal links. The new step may threaten an existing causal link or an existing step may threaten the new causal link. If at any point the algorithm fails to find a relevant operator or resolve a threat, it backtracks to a previous choice point. An important subtlety is that the selection of a step and precondition in SELECT-SUBGOAL is not a candidate for backtracking. The reason is that every precondition needs to be considered eventually, and the handling of preconditions is commutative: handling \( c_1 \) and then \( c_2 \) leads to exactly the same set of possible plans as handling \( c_2 \) and then \( c_1 \). So we

Figure 11.12 A solution to the shopping problem.
function POP(initial, goal, operators) returns plan

plan ← MAKE-MINIMAL-PLAN(initial, goal)
loop do
  if SOLUTION?(plan) then return plan
  Sneed, c ← SELECT-SUBGOAL(plan)
  CHOOSE-OPERATOR(plan, operators, Sneed, c)
  RESOLVE-THREATS(plan)
end

function SELECT-SUBGOAL(plan) returns Sneed, c
pick a plan step Sneed from STEPS(plan)
with a precondition c that has not been achieved
return Sneed, c

procedure CHOOSE-OPERATOR(plan, operators, Sneed, c)
choose a step Sadd from operators or STEPS(plan) that has c as an effect
if there is no such step then fail
add the causal link Sadd \rightarrow Sneed to LINKS(plan)
add the ordering constraint Sadd < Sneed to ORDERINGS(plan)
if Sadd is a newly added step from operators then
  add Sadd to STEPS(plan)
  add Start < Sadd < Finish to ORDERINGS(plan)
end

procedure RESOLVE-THREATS(plan)
for each Sthreat that threatens a link Si \rightarrow Sj in LINKS(plan) do
  choose either
  Promotion: Add Sthreat < Si to ORDERINGS(plan)
  Demotion: Add Si < Sthreat to ORDERINGS(plan)
  if not CONSISTENT(plan) then fail
end

Figure 11.13 The partial-order planning algorithm, POP.

can just pick a precondition and move ahead without worrying about backtracking. The pick we make affects only the speed, and not the possibility, of finding a solution.

Notice that POP is a regression planner, because it starts with goals that need to be achieved and works backwards to find operators that will achieve them. Once it has achieved all the preconditions of all the steps, it is done; it has a solution. POP is sound and complete. Every plan it returns is in fact a solution, and if there is a solution, then it will be found (assuming a breadth-first or iterative deepening search strategy). At this point, we suggest that the reader return to the example of the previous section, and trace through the operation of POP in detail.
11.7 PLANNING WITH PARTIALLY INSTANTIATED OPERATORS

The version of POP in Figure 11.13 outlines the algorithm, but leaves some details unspecified. In particular, it does not deal with variable binding constraints. For the most part, all this entails is being diligent about keeping track of binding lists and unifying the right expressions at the right time. The implementation techniques of Chapter 10 are applicable here.

There is one substantive decision to make: in RESOLVE-THREATS, should an operator that has the effect, say, $\neg At(x)$ be considered a threat to the condition $At(Home)$? Currently we can distinguish between threats and non-threats, but this is a possible threat. There are three main approaches to dealing with possible threats:

- **Resolve now with an equality constraint:** Modify RESOLVE-THREATS so that it resolves all possible threats as soon as they are recognized. For example, when the planner chooses the operator that has the effect $\neg At(x)$, it would add a binding such as $x = HWS$ to make sure it does not threaten $At(Home)$.

- **Resolve now with an inequality constraint:** Extend the language of variable binding constraints to allow the constraint $x \neq Home$. This has the advantage of being a lower commitment—it does not require an arbitrary choice for the value of $x$—but it is a little more complicated to implement, because the unification routines we have used so far all deal with equalities, not inequalities.

- **Resolve later:** The third possibility is to ignore possible threats, and only deal with them when they become necessary threats. That is, RESOLVE-THREATS would not consider $\neg At(x)$, to be a threat to $At(Home)$. But if the constraint $x = Home$ were ever added to the plan, then the threat would be resolved (by promotion or demotion). This approach has the advantage of being low commitment, but has the disadvantage of making it harder to decide if a plan is a solution.

Figure 11.14 shows an implementation of the changes to CHOOSE-OPERATOR, along with the changes to RESOLVE-THREATS that are necessary for the third approach. It is certainly possible (and advisable) to do some bookkeeping so that RESOLVE-THREATS will not need to go through a triply nested loop on each call.

When partially instantiated operators appear in plans, the criterion for solutions needs to be refined somewhat. In our earlier definition (page 349), we were concerned mainly with the question of partial ordering; a solution was defined as a partial plan such that all linearizations are guaranteed to achieve the goal. With partially instantiated operators, we also need to ensure that all instantiations will achieve the goal. We therefore extend the definition of achievement for a step in a plan as follows:

A step $S_j$ achieves a precondition $c$ of the step $S_i$ if (1) $S_i \prec S_j$ and $S_j$ has an effect that necessarily unifies with $c$; and (2) there is no step $S_k$ such that $S_i \prec S_k \prec S_j$ in some linearization of the plan, and $S_k$ has an effect that possibly unifies with $\neg c$.

The POP algorithm can be seen as constructing a proof that each precondition of the goal step is achieved. CHOOSE-OPERATOR comes up with the $S_i$ that achieves (1), and RESOLVE-THREATS makes sure that (2) is satisfied by promoting or demoting possible threats. The tricky part is that
if we adopt the “resolve-later” approach, then there will be possible threats that are not resolved away. We therefore need some way of checking that these threats are all gone before we return the plan. It turns out that if the initial state contains no variables and if every operator mentions all its variables in its precondition, then any complete plan generated by POP is guaranteed to be fully instantiated. Otherwise we will need to change the function SOLUTION to check that there are no uninstantiated variables and choose bindings for them if there are. If this is done, then POP is guaranteed to be a sound planner in all cases.

It is harder to see that POP is complete—that is, finds a solution whenever one exists—but again it comes down to understanding how the algorithm mirrors the definition of achievement. The algorithm generates every possible plan that satisfies part (1), and then filters out those plans that do not satisfy part (2) or that are inconsistent. Thus, if there is a plan that is a solution, POP will find it. So if you accept the definition of solution (page 349), you should accept that POP is a sound and complete planner.