We can make this more formal. Assume a goal description \( g \) which contains a goal literal \( \phi \) and an action schema \( A \) that is standardized to produce \( A' \). If \( A' \) has an effect literal where \( \text{Unify}(\phi, \phi') = t \) and where we define \( \delta' = \text{SUB}(\delta, A') \) and if there is no effect in \( \delta' \) that is the negation of a literal in \( g \), then \( \delta' \) 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
problem involving tiles with a single schema \textit{Slide}:

\begin{verbatim}
Action(Slide(t, s1, s2)).
PRECOND: On(t, s1) \& Tile(t) \& Blank(s1) \& Adjacent(s1, s2),
EFFECT: On(t, s2) \& Blank(s1) \& On(t, s1) \& \neg Blank(s2).
\end{verbatim}

As we saw in Section 3.6, if we remove the preconditions $Blank(s_i)$ and $Adjacent(s_1, s_2)$ then any tile can move in one action to any space and we get the number-of-misplaced-tiles heuristic. If we remove $Blank(s_i)$ then we get the \texttt{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.

Another possibility is the \texttt{ignore delete lists} heuristic. Assume for a moment that all goals and preconditions contain only positive literals. 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 \textit{NP-hard} to find the optimal solution to this relaxed problem, but an approximate solution can be found in polynomial time by \texttt{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 \texttt{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 \texttt{hill-climbing} search will easily find a solution to these problems (although it may not be an optimal solution).

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 states or more, and relaxing the \texttt{actions} does nothing to reduce the number of states. Therefore, we now look at relaxations that decrease the number of states by forming a \texttt{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 \texttt{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 $50 \times 200^{10}$ 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 \texttt{At} \texttt{fluents} except for the ones involving one plane and one package at each of the 5 airports. Now there are only $5^6 \times 5^{10}$ 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 \texttt{Load} and \texttt{Unload} actions).

\footnote{Many problems are written with this convention. For problems that aren’t, replace every negative literal $\neg$ in a goal or precondition with a new positive literal,}