11.4 BASIC REPRESENTATIONS FOR PLANNING

The "classical" approach that most planners use today describes states and operators in a restricted language known as the STRIPS language,\(^2\) or in extensions thereof. The STRIPS language lends itself to efficient planning algorithms, while retaining much of the expressiveness of situation calculus representations.

Representations for states and goals

In the STRIPS language, states are represented by conjunctions of function-free ground literals, that is, predicates applied to constant symbols, possibly negated. For example, the initial state for the milk-and-bananas problem might be described as

\[
\text{At(Home)} \land \neg \text{Have(Milk)} \land \neg \text{Have(Bananas)} \land \neg \text{Have(Drill)} \land \ldots
\]

As we mentioned earlier, a state description does not need to be complete. An incomplete state description, such as might be obtained by an agent in an inaccessible environment, corresponds to a set of possible complete states for which the agent would like to obtain a successful plan. Many planning systems instead adopt the convention—analogous to the "negation as failure" convention used in logic programming—that if the state description does not mention a given positive literal then the literal can be assumed to be false.

Goals are also described by conjunctions of literals. For example, the shopping goal might be represented as

\[
\text{At(Home)} \land \text{Have(Milk)} \land \text{Have(Bananas)} \land \text{Have(Drill)}
\]

Goals can also contain variables. For example, the goal of being at a store that sells milk would be represented as

\[
\text{At}(x) \land \text{Sells}(x, \text{Milk})
\]

As with goals given to theorem provers, the variables are assumed to be existentially quantified. However, one must distinguish clearly between a goal given to a planner and a query given to a theorem prover. The former asks for a sequence of actions that makes the goal true if executed, and the latter asks whether the query sentence is true given the truth of the sentences in the knowledge base.

Although representations of initial states and goals are used as inputs to planning systems, it is quite common for the planning process itself to maintain only implicit representations of states. Because most actions change only a small part of the state representation, it is more efficient to keep track of the changes. We will see how this is done shortly.

\(^2\) Named after a pioneering planning program known as the Stanford Research Institute Problem Solver. There are two unfortunate things about the name STRIPS. First, the organization no longer uses the name "Stanford" and is now known as SRI International. Second, the program is what we now call a planner, not a problem solver, but when it was developed in 1976, the distinction had not been articulated. Although the STRIPS planner has long since been superseded, the STRIPS language for describing actions has been invaluable, and many "STRIPS-like" variants have been developed.
Representations for actions

Our STRIPS operators consist of three components:

- **The action description** is what an agent actually returns to the environment in order to do something. Within the planner it serves only as a name for a possible action.
- **The precondition** is a conjunction of atoms (positive literals) that says what must be true before the operator can be applied.
- **The effect** of an operator is a conjunction of literals (positive or negative) that describes how the situation changes when the operator is applied.3

Here is an example of the syntax we will use for forming a STRIPS operator for going from one place to another:

\[
\text{Op}(\text{ACTION: Go}(\text{there}), \text{PRECOND: At}(\text{here}) \land \text{Path}(\text{here, there}), \text{EFFECT: At}(\text{there}) \land \neg \text{At}(\text{here}))
\]

(We will also use a graphical notation to describe operators, as shown in Figure 11.3.) Notice that there are no explicit situation variables. Everything in the precondition implicitly refers to the situation immediately before the action, and everything in the effect implicitly refers to the situation that is the result of the action.

![Diagrammatic notation for the operator Go(there). The preconditions appear above the action, and the effects below.](image)

An operator with variables is known as an **operator schema**, because it does not correspond to a single executable action but rather to a family of actions, one for each different instantiation of the variables. Usually, only fully instantiated operators can be executed; our planning algorithms will ensure that each variable has a value by the time the planner is done. As with state descriptions, the language of preconditions and effects is quite restricted. The precondition must be a conjunction of positive literals, and the effect must be a conjunction of positive and/or negative literals. All variables are assumed universally quantified, and there can be no additional quantifiers. In Chapter 12, we will relax these restrictions.

We say that an operator \( o \) is **applicable** in a state \( s \) if there is some way to instantiate the variables in \( o \) so that every one of the preconditions of \( o \) is true in \( s \), that is, if \( \text{Precond}(o) \subseteq s \). In the resulting state, all the positive literals in \( \text{Effect}(o) \) hold, as do all the literals that held in \( s \).

---

3 The original version of STRIPS divided the effects into an **add list** and a **delete list**.
except for those that are negative literals in *Effect(o)*. For example, if the initial situation includes the literals

\[ \text{At(Home)}, \text{Path(Home, Supermarket)} \ldots \]

then the action *Go(Supermarket)* is applicable, and the resulting situation contains the literals

\[ \neg \text{At(Home)}, \text{At(Supermarket)}, \text{Path(Home, Supermarket)} \ldots \]

**Situation Space and Plan Space**

In Figure 11.2, we showed a search space of *situations* in the world (in this case the shopping world). A path through this space from the initial state to the goal state constitutes a plan for the shopping problem. If we wanted, we could take a problem described in the STRIPS language and solve it by starting at the initial state and applying operators one at a time until we reached a state that includes all the literals in the goal. We could use any of the search methods of Part II. An algorithm that did this would clearly be considered a problem solver, but we could also consider it a planner. We would call it a *situation space* planner because it searches through the space of possible situations, and a *progression* planner because it searches forward from the initial situation to the goal situation. The main problem with this approach is the high branching factor and thus the huge size of the search space.

One way to try to cut the branching factor is to search backwards, from the goal state to the initial state; such a search is called *regression* planning. This approach is *possible* because the operators contain enough information to regress from a partial description of a result state to a partial description of the state before an operator is applied. We cannot get complete descriptions of states this way, but we don’t need to. The approach is *desirable* because in typical problems the goal state has only a few conjunctions, each of which has only a few appropriate operators, whereas the initial state usually has many applicable operators. (An operator is appropriate to a goal if the goal is an effect of the operator.) Unfortunately, searching backwards is complicated somewhat by the fact that we often have to achieve a conjunction of goals, not just one. The original STRIPS algorithm was a situation-space regression planner that was incomplete (it could not always find a plan when one existed) because it had an inadequate way of handling the complication of conjunctive goals. Fixing this incompleteness makes the planner very inefficient.

In summary, the nodes in the search tree of a situation-space planner correspond to situations, and the path through the search tree is the plan that will be ultimately returned by the planner. Each branch point adds another step to either the beginning (regression) or end (progression) of the plan.

An alternative is to search through the space of *plans* rather than the space of *situations*. That is, we start with a simple, incomplete plan, which we call a *partial plan*. Then we consider ways of expanding the partial plan until we come up with a complete plan that solves the problem. The operators in this search are operators on plans: adding a step, imposing an ordering that puts one step before another, instantiating a previously unbound variable, and so on. The solution is the final plan, and the path taken to reach it is irrelevant.

Operations on plans come in two categories. *Refinement operators* take a partial plan and add constraints to it. One way of looking at a partial plan is as a representation for a set
of complete, fully constrained plans. Refinement operators eliminate some plans from this set, but they never add new plans to it. Anything that is not a refinement operator is a **modification operator**. Some planners work by constructing potentially incorrect plans, and then “debugging” them using modification operators. In this chapter, we use only refinement operators.

**Representations for plans**

If we are going to search through a space of plans, we need to be able to represent them. We can settle on a good representation for plans by considering partial plans for a simple problem: putting on a pair of shoes. The goal is the conjunction of $\text{RightShoeOn} \land \text{LeftShoeOn}$, the initial state has no literals at all, and the four operators are

\[
\begin{align*}
\text{Op(} & \text{ACTION: RightShoe, PRECOND: RightSockOn, EFFECT: RightShoeOn)} \\
\text{Op(} & \text{ACTION: RightSock, PRECOND: RightSockOn, EFFECT: RightShoeOn)} \\
\text{Op(} & \text{ACTION: LeftShoe, PRECOND: LeftSockOn, EFFECT: LeftShoeOn)} \\
\text{Op(} & \text{ACTION: LeftSock, PRECOND: LeftSockOn, EFFECT: LeftShoeOn)}
\end{align*}
\]

A partial plan for this problem consists of the two steps RightShoe and LeftShoe. But which step should come first? Many planners use the principle of **least commitment**, which says that one should only make choices about things that you currently care about, leaving the other choices to be worked out later. This is a good idea for programs that search, because if you make a choice about something you don’t care about now, you are likely to make the wrong choice and have to backtrack later. A least commitment planner could leave the ordering of the two steps unspecified. When a third step, RightSock, is added to the plan, we want to make sure that putting on the right sock comes before putting on the right shoe, but we do not care where they come with respect to the left shoe. A planner that can represent plans in which some steps are ordered (before or after) with respect to each other and other steps are unordered is called a **partial order** planner. The alternative is a **total order** planner, in which plans consist of a simple list of steps. A totally ordered plan that is derived from a plan $P$ by adding ordering constraints is called a **linearization** of $P$.

The socks-and-shoes example does not show it, but planners also have to commit to bindings for variables in operators. For example, suppose one of your goals is $\text{Have(Milk)}$, and you have the action $\text{Buy(item, store)}$. A sensible commitment is to choose this action with the variable $\text{item}$ bound to $\text{Milk}$. However, there is no good reason to pick a binding for $\text{store}$, so the principle of least commitment says to leave it unbound and make the choice later. Perhaps another goal will be to buy an item that is only available in one specialty store. If that store also carries milk, then we can bind the variable $\text{store}$ to the specialty store at that time. By delaying the commitment to a particular store, we allow the planner to make a good choice later. This strategy can also help prune out bad plans. Suppose that for some reason the branch of the search space that includes the partially instantiated action $\text{Buy(Milk, store)}$ leads to a failure for some reason unrelated to the choice of store (perhaps the agent has no money). If we had committed to a particular store, then the search algorithm would force us to backtrack and consider another store. But if we have not committed, then there is no choice to backtrack over and we can discard this whole branch of the search tree without having to enumerate any of the stores. Plans in which every variable is bound to a constant are called **fully instantiated plans**.
In this chapter, we will use a representation for plans that allows for deferred commitments about ordering and variable binding. A plan is formally defined as a data structure consisting of the following four components:

- A set of plan steps. Each step is one of the operators for the problem.
- A set of step ordering constraints. Each ordering constraint is of the form $S_i \prec S_j$, which is read as “$S_i$ before $S_j$” and means that step $S_i$ must occur sometime before step $S_j$ (but not necessarily immediately before).\(^4\)
- A set of variable binding constraints. Each variable constraint is of the form $v = x$, where $v$ is a variable in some step, and $x$ is either a constant or another variable.
- A set of causal links.\(^5\) A causal link is written as $S_i \rightarrow S_j$ and read as “$S_i$ achieves $c$ for $S_j$.” Causal links serve to record the purpose(s) of steps in the plan: here a purpose of $S_i$ is to achieve the precondition $c$ of $S_j$.

The initial plan, before any refinements have taken place, simply describes the unsolved problem. It consists of two steps, called Start and Finish, with the ordering constraint Start $\prec$ Finish. Both Start and Finish have null actions associated with them, so when it is time to execute the plan, they are ignored. The Start step has no preconditions, and its effect is to add all the propositions that are true in the initial state. The Finish step has the goal state as its precondition, and no effects. By defining a problem this way, our planners can start with the initial plan and manipulate it until they come up with a plan that is a solution. The shoes-and-socks problem is defined by the four operators given earlier and an initial plan that we write as follows:

\[
\begin{align*}
\text{Plan}(\text{STEPS:} & \{ S_1: \text{Op(ACTION:Start)}, \\
& S_2: \text{Op(ACTION:Finish,} \\
& \text{PRECOND:RightShoeOn} \land \text{LeftShoeOn})\}, \\
& \text{ORDERINGS:} \{ S_1 < S_2 \}, \\
& \text{BINDINGS:} \{ \}, \\
& \text{LINKS:} \{ \})
\end{align*}
\]

As with individual operators, we will use a graphical notation to describe plans (Figure 11.4(a)). The initial plan for the shoes-and-socks problem is shown in Figure 11.4(b). Later in the chapter we will see how this notation is extended to deal with more complex plans.

Figure 11.5 shows a partial-order plan that is a solution to the shoes-and-socks problem, and six linearizations of the plan. This example shows that the partial-order plan representation is powerful because it allows a planner to ignore ordering choices that have no effect on the correctness of the plan. As the number of steps grows, the number of possible ordering choices grows exponentially. For example, if we added a hat and a coat to the problem, which interact neither with each other nor with the shoes and socks, then there would still be one partial plan that represents all the solutions, but there would be 180 linearizations of that partial plan. (Exercise 11.1 asks you to derive this number.)

---

\(^4\) We use the notation $A \prec B \prec C$ to mean $(A \prec B) \land (B \prec C)$.

\(^5\) Some authors call causal links protection intervals.
Figure 11.4  (a) Problems are defined by partial plans containing only Start and Finish steps. The initial state is entered as the effects of the Start step, and the goal state is the precondition of the Finish step. Ordering constraints are shown as arrows between boxes. (b) The initial plan for the shoes-and-socks problem.

Figure 11.5  A partial-order plan for putting on shoes and socks (including preconditions on steps), and the six possible linearizations of the plan.
Section 11.5. A Partial-Order Planning Example 349

Solutions

A solution is a plan that an agent can execute, and that guarantees achievement of the goal. If we wanted to make it really easy to check that a plan is a solution, we could insist that only fully instantiated, totally ordered plans can be solutions. But this is unsatisfactory for three reasons. First, for problems like the one in Figure 11.5, it is more natural for the planner to return a partial-order plan than to arbitrarily choose one of the many linearizations of it. Second, some agents are capable of performing actions in parallel, so it makes sense to allow solutions with parallel actions. Lastly, when creating plans that may later be combined with other plans to solve larger problems, it pays to retain the flexibility afforded by the partial ordering of actions. Therefore, we allow partially ordered plans as solutions using a simple definition: a solution is a complete, consistent plan. We need to define these terms.

A complete plan is one in which every precondition of every step is achieved by some other step. A step achieves a condition if the condition is one of the effects of the step, and if no other step can possibly cancel out the condition. More formally, a step $S_i$ achieves a precondition $c$ of the step $S_j$ if (1) $S_i < S_j$ and $c \in \text{Effects}(S_i)$; and (2) there is no step $S_k$ such that $(\neg c) \in \text{Effects}(S_k)$, where $S_i < S_k < S_j$ in some linearization of the plan.

A consistent plan is one in which there are no contradictions in the ordering or binding constraints. A contradiction occurs when both $S_i < S_j$ and $S_j < S_i$ hold or both $v = A$ and $v = B$ hold (for two different constants $A$ and $B$). Both $<$ and $=$ are transitive, so, for example, a plan with $S_1 < S_2, S_2 < S_3,$ and $S_3 < S_1$ is inconsistent.

The partial plan in Figure 11.5 is a solution because all the preconditions are achieved. From the preceding definitions, it is easy to see that any linearization of a solution is also a solution. Hence the agent can execute the steps in any order consistent with the constraints, and still be assured of achieving the goal.

11.5 A Partial-Order Planning Example

In this section, we sketch the outline of a partial-order regression planner that searches through plan space. The planner starts with an initial plan representing the start and finish steps, and on each iteration adds one more step. If this leads to an inconsistent plan, it backtracks and tries another branch of the search space. To keep the search focused, the planner only considers adding steps that serve to achieve a precondition that has not yet been achieved. The causal links are used to keep track of this.

We illustrate the planner by returning to the problem of getting some milk, a banana, and a drill, and bringing them back home. We will make some simplifying assumptions. First, the Go action can be used to travel between any two locations. Second, the description of the Buy action ignores the question of money (see Exercise 11.2). The initial state is defined by the following operator, where HWS means hardware store and SM means supermarket:

\[ Op(\text{ACTION}: \text{Start}, \text{EFFECT}: \text{At(Home)} \land \text{Sells(HWS, Drill)} \land \text{Sells(SM, Milk, Sells(SM, Banana)} \)