2 Deterministic Representation and Acting

This chapter is about representing state-transition systems and using them in acting. Section 2.2 gives formal definitions of state-transition systems and planning problems, and a simple acting algorithm. Section 2.3 describes state-variable representations of state-transition-systems, and Section 2.6 describes several acting procedures that use this representation. Section 2.4 describes classical representation, an alternative to state-variable representation that is often used in the planning literature.

2.1 Motivating Example

Most of the examples in this chapter will involve *Dock-Worker Robots (DWR)* domains. These are highly simplified "toy" versions of harbor and warehouse systems like the one in Figure 2.1. Depending on the example, the objects may include ships, cranes, loading docks, piles of containers, robot vehicles, roads, and delivery gates.

Example 2.1. Figure 2.2 shows a simple DWR domain that is a running example in this chapter. There are two robots, three loading docks, three containers, and three piles (stacks of containers). The domain has *states*, each of which is a configuration

Figure 2.1. A container terminal in Barcelona.

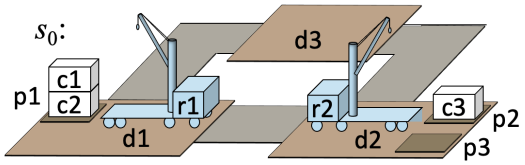


Figure 2.2. Running example: a simple DWR domain.

of the objects (as in the figure), and *actions* that cause transitions from one state to another. □

The next section is rather abstract, but we will return to this example in Section 2.3.

2.2 State-Transition Systems

Definition 2.2.[1] A *state-transition system*, or *classical planning domain*, is a tuple

$$\Sigma = (S, A, \gamma, \text{cost}) \quad \text{or} \quad \Sigma = (S, A, \gamma), \tag{2.1}$$

where

- S and A are the finite sets of *states* and *actions*,
- $\gamma : S \times A \to S$, the *state-transition function*, is a *partial function* (that is, its domain is a subset of $S \times A$) telling what state will be produced if the actor executes action a in state s. The set of *applicable* actions in s is[2]

$$\mathit{Applicable}(s) = \{a \in A \mid \gamma(s, a) \text{ is defined}\}. \tag{2.2}$$

- $\text{cost} : A \to [0, \infty)$, the *cost function*,[3] is a partial function having the same domain as γ. It may represent monetary cost, time, or some other numeric quantity that one might want to minimize. In the second form of Equation 2.1, in which the cost function is not given explicitly, $\text{cost}(a) = 1$ for every $a \in A$. □

2.2.1 Plans

In order to act purposefully, an actor will need some notion of what actions it needs to perform. In a deterministic state-transition model, this will be a *plan*: a finite sequence of actions

$$\pi = \langle a_1, \ldots, a_n \rangle. \tag{2.3}$$

The *length* of π is $|\pi| = n$, and the *cost* of π is the sum of the action costs:

$$\text{cost}(\pi) = \sum_{i=1}^{n} \text{cost}(a_i). \tag{2.4}$$

[1] Definitions, examples, and theorems are all numbered in the same numerical sequence. Algorithms and equations, however, are in separate sequences.

[2] Section 8.1 will change Equation 2.2 to model actions that have multiple possible outcomes.

[3] This definition prevents the cost from depending on s. See Remark 2.6 for a discussion of this restriction and some cases in which it can be lifted.

As a special case, $\langle\rangle$ is the *empty* plan, which contains no actions. Its length and cost are both 0.

A *subplan* π' of π is a (contiguous) subsequence[4] $\langle a_i, \ldots, a_j\rangle$ of π. As special cases, the subplans $\langle a_1, \ldots, a_i\rangle$ and $\langle a_j, \ldots, a_n\rangle$ are a *prefix* and *suffix* of π, respectively.[5]

Here is some notation for concatenation of plans and actions. If $\pi = \langle a_1, \ldots, a_n\rangle$ and $\pi' = \langle a'_1, \ldots, a'_{n'}\rangle$ are plans and a is an action, then

$$\begin{aligned} \pi \cdot a &= \langle a_1, \ldots, a_n, a\rangle; \\ a \cdot \pi &= \langle a, a_1, \ldots, a_n\rangle; \\ \pi \cdot \pi' &= \langle a_1, \ldots, a_n, a'_1, \ldots, a'_{n'}\rangle; \\ \pi \cdot \langle\rangle &= \langle\rangle \cdot \pi = \pi. \end{aligned} \tag{2.5}$$

The state-transition function γ can easily be extended to include plans, by letting $\gamma(s, \pi)$ be the state produced by starting at s and applying the actions in π in the order that they are given, if all of them are applicable. More specifically:

- The empty plan $\langle\rangle$ is applicable in every state s, and $\gamma(s, \langle\rangle) = s$.
- If $\pi = a \cdot \pi'$, where a is applicable in s and π' is applicable in $\gamma(s, a)$, then π is applicable in s and

$$\gamma(s, \pi) = \gamma(\gamma(s, a), \pi'). \tag{2.6}$$

It immediately follows that if a plan $\pi = \langle a_1, a_2, \ldots, a_n\rangle$ is applicable in a state s_0, then it produces a sequence of states $\langle s_0, s_1, \ldots, s_n\rangle$ such that

$$s_1 = \gamma(s_0, a_1), \quad s_2 = \gamma(s_1, a_2), \quad \ldots, \quad s_n = \gamma(s_{i-n}, a_n). \tag{2.7}$$

In this case, the *transitive closure* of π on s_0 is the path

$$\widehat{\gamma}(s_0, \pi) = \begin{cases} \langle s_0, s_1, \ldots, s_n\rangle, & \text{if } \pi = \langle a_1, a_2, \ldots, a_n\rangle, \\ \langle s_0\rangle, & \text{if } \pi = \langle\rangle. \end{cases} \tag{2.8}$$

From the usual definitions of the length and cost of a path, we get

$$|\widehat{\gamma}(s_0, \pi)| = |\pi|; \tag{2.9}$$

$$\text{cost}(\widehat{\gamma}(s_0, \pi)) = \text{cost}(\pi). \tag{2.10}$$

2.2.2 Planning Problems

A *classical planning problem* is a triple

$$P = (\Sigma, s_0, S_g), \tag{2.11}$$

[4] We use "subsequence" to mean a *contiguous* subsequence. This is consistent with our previous books [409, 410], but differs from the terminology in some other subfields of computer science [35, 256].

[5] This terminology is common in the AI planning literature, but it differs from ordinary English usage, in which a prefix or suffix would be something added to π, not part of π itself.

```
Run-Plan(Σ, π)
    while True do
1       s ← observe current state
        if π = ⟨⟩ then
2           return success
        a ← pop(π)
3       if a ∉ Applicable(s) then return failure
        perform action a
```

Algorithm 2.1. Run-Plan, a simple procedure to run a plan.

where Σ is a state-transition system, s_0 is a state called the *initial state*, and S_g is a set of states called *goal states*. A *solution* for P is any plan $\pi = \langle a_1, \ldots, a_n \rangle$ such that $\gamma(s_0, \pi) \in S_g$. The solution π is *minimal* if no subsequence of π is also a solution, *shortest* if there is no solution π' such that $|\pi'| < |\pi|$, and *optimal* if

$$\text{cost}(\pi) = \min\{\text{cost}(\pi') \mid \pi' \text{ is a solution for } P\}. \tag{2.12}$$

Example 2.3. Suppose a planning problem P has three solution plans:

$$\pi_1 = \langle a_1 \rangle; \qquad \pi_2 = \langle a_2, a_3, a_4, a_5 \rangle; \qquad \pi_3 = \langle a_2, a_3, a_1 \rangle.$$

If each action's cost is 1, then π_1 is a minimal, shortest, and cost-optimal solution, π_2 is a minimal solution but is neither shortest nor cost-optimal, and π_3 is neither minimal nor shortest nor cost-optimal. □

2.2.3 Acting with a Plan

Algorithm 2.1, Run-Plan, is a simple procedure for running a plan. If π is applicable in the initial observed state,[6] then ideally it will produce the sequence of states $\widehat{\gamma}(s_0, \pi)$, and Run-Plan will return success. However, recall from Part I that Σ is not necessarily a perfect model of the actor's environment. Execution errors or unpredicted exogenous events may sometimes cause Run-Plan to encounter states in which the next action of π is not applicable, in which case Line 3 will return failure.

Run-Plan can be adapted to test whether π has achieved a desired goal S_g, by adding S_g to its argument list and adding the following line before Line 2:

if $s \notin S_g$ **then return** failure

Section 2.6 will discuss some ways for an actor to recover from failures, either by re-executing parts of π or acquiring a new solution plan. Chapter 3 will discuss several algorithms to produce solution plans.

[6] Although we call this the observed state, it is more likely to be an abstraction of the state that the actor observes, with various low-level details omitted that are irrelevant for planning.

2.3 State-Variable Representation

In Section 2.2, states were an abstract set $S = \{s_0, s_1, \ldots, \}$. Explicit enumeration of S can often be quite large; even trivially simple examples such as Figure 2.2 can have hundreds or thousands of states. Furthermore, names such as $s_0, s_1, \ldots$ tell us nothing about the states' internal structure.

To represent complex domains, we will want a more expressive representation that gives information about relationships among objects in the actor's environment. For example, to describe the state shown in Figure 2.2—which will be a running example in this section—we might want to write

$$\mathsf{loc}(\mathsf{r1}) = \mathsf{d1} \tag{2.13}$$

to mean that robot r1's location is d1.

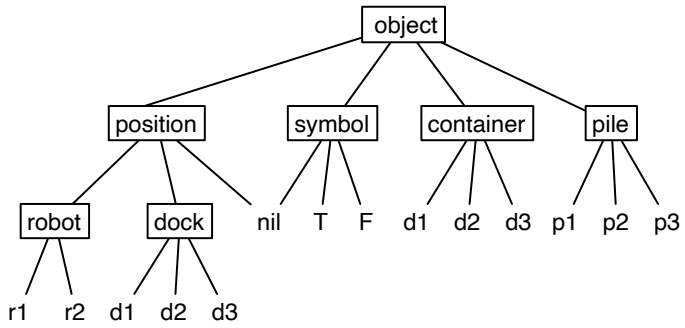


Figure 2.3. A type hierarchy for the objects in Figure 2.2. Boxed words are types, other words are objects.

In Equation 2.13, d1 and r1 are called *objects* or *object constants*. We will organize objects into sets and subsets using a type hierarchy. For example, Figure 2.3 is a type hierarchy for our running example, and it corresponds to the following sets of objects:

$$\mathcal{H} = \begin{cases} \mathit{Objects} = \mathit{Positions} \cup \mathit{Containers} \cup \mathit{Piles} \cup \mathit{Symbols}; \\ \mathit{Positions} = \mathit{Robots} \cup \mathit{Docks} \cup \{\mathsf{nil}\}; \\ \mathit{Symbols} = \{\mathsf{T}, \mathsf{F}, \mathsf{nil}\}; \quad \mathit{Containers} = \{\mathsf{c1}, \mathsf{c2}, \mathsf{c3}\}; \\ \mathit{Piles} = \{\mathsf{p1}, \mathsf{p2}, \mathsf{p3}\}; \quad \mathit{Robots} = \{\mathsf{r1}, \mathsf{r2}\}; \\ \mathit{Docks} = \{\mathsf{d1}, \mathsf{d2}, \mathsf{d3}\}. \end{cases} \tag{2.14}$$

With a slight abuse in terminology, we not distinguish between types and and the corresponding sets of constants. For example, we will call $\mathcal{H}$ a type hierarchy.

We will have typed variables called *object variables*. In our running example, an object variable r of type robot has $\mathit{Range}(r) = \mathit{Robots}$ (or less formally, $r \in \mathit{Robots}$).

A domain usually has a set of *rigid* properties that do not change over time, such as its topology and connectivity. In our running example, robots and containers can be moved around, but the locations of piles and docks are rigid. To represent such properties we will define *rigid relations* over the types. In our running example, the

rigid relations are adjacent $\subseteq$ *Docks* $\times$ *Docks*, a symmetric relation telling which pairs of loading docks have roads between them, and at $\subseteq$ *Piles* $\times$ *Docks*, which gives each pile's location:

$$\begin{aligned}\mathsf{adjacent} &= \{(\mathsf{d1},\mathsf{d2}),(\mathsf{d2},\mathsf{d1}),(\mathsf{d2},\mathsf{d3}),(\mathsf{d3},\mathsf{d2}),(\mathsf{d3},\mathsf{d1}),(\mathsf{d1},\mathsf{d3})\};\\ \mathsf{at} &= \{(\mathsf{p1},\mathsf{d1}),(\mathsf{p2},\mathsf{d2}),(\mathsf{p3},\mathsf{d2})\}.\end{aligned}$$

Properties of the domain that change over time, possibly under the effect of the actor's activity, we will describe using functional terms called *state variables*.[7] State variables have zero or more arguments, each of which may be either an object or an object variable. Each state variables is typed; for example, $\mathsf{loc}(\mathsf{r1}) \in$ *Docks*.

2.3.1 Representing States

In each state, every ground state variable has a value that we will represent as an *assignment*, written as an equation similar to Equation 2.13, which assigns to $\mathsf{loc}(\mathsf{r1})$ the value d1. We also will find it useful to refer to *lifted assignments*, e.g., $\mathsf{loc}(\mathsf{r1}) = l$, where r and l are object variables to be instantiated later (see Definition 2.5).

Here are the state variables. Their parameter types are $r \in$ *Robots*, $d \in$ *Docks*, $c \in$ *Containers*, $p \in$ *Piles*.

- $\mathsf{cargo}(r) \in$ *Containers* $\cup \{\mathsf{nil}\}$ is either the container that r is carrying, or nil if r is not carrying anything. Each robot can hold at most one container.
- $\mathsf{loc}(r) \in$ *Docks* is the loading dock where r is located.
- $\mathsf{occupied}(d) \in \{\mathsf{T},\mathsf{F}\}$ is T if there is a robot at d, and F otherwise. At most one robot can be at each loading dock.
- $\mathsf{pile}(c) \in$ *Piles* $\cup \{\mathsf{nil}\}$ is the pile that c is in, or nil if c is not in a pile.
- $\mathsf{pos}(c) \in$ *Robots* $\cup$ *Piles* $\cup \{\mathsf{nil}\}$ is c's position, which may be a robot, another container if c is in a pile, or nil if c is at the bottom of a pile.
- $\mathsf{top}(p) \in$ *Containers* $\cup \{\mathsf{nil}\}$ is the container at the top of p, with $\mathsf{top}(p) = \mathsf{nil}$ if p is empty.

Example 2.4. The following total assignment is the state shown in Figure 2.2:

$$\begin{aligned}s_0 = \{&\mathsf{cargo}(\mathsf{r1})=\mathsf{nil}, && \mathsf{cargo}(\mathsf{r2})=\mathsf{nil}, &&\\ &\mathsf{loc}(\mathsf{r1})=\mathsf{d1}, && \mathsf{loc}(\mathsf{r2})=\mathsf{d2}, &&\\ &\mathsf{occupied}(\mathsf{d1})=\mathsf{T}, && \mathsf{occupied}(\mathsf{d2})=\mathsf{T}, && \mathsf{occupied}(\mathsf{d3})=\mathsf{F},\\ &\mathsf{pile}(\mathsf{c1})=\mathsf{p1}, && \mathsf{pile}(\mathsf{c2})=\mathsf{p1}, && \mathsf{pile}(\mathsf{c3})=\mathsf{p2},\\ &\mathsf{pos}(\mathsf{c1})=\mathsf{c2}, && \mathsf{pos}(\mathsf{c2})=\mathsf{nil}, && \mathsf{pos}(\mathsf{c3})=\mathsf{nil},\\ &\mathsf{top}(\mathsf{p1})=\mathsf{c1}, && \mathsf{top}(\mathsf{p2})=\mathsf{c3}, && \mathsf{top}(\mathsf{p3})=\mathsf{nil}\}.\end{aligned}$$

□

Usually some total assignments are nonsensical in the domain that the state variables are intended to represent. In our running example, it is nonsensical to have a state in which both $\mathsf{pos}(\mathsf{c1}) = \mathsf{r1}$ and $\mathsf{cargo}(\mathsf{r1}) = \mathsf{F}$. In principle, one could exclude such

[7] The terms *state variable* and *fluent* are considered synonymous [965]. However, in much of the published literature, fluents have only Boolean values, and state variables have no parameters. We use a more flexible representation that includes both parameters and non-Boolean values.

things from the set of states S by writing a set of constraints that every state must satisfy. However, we will instead will leave these "unreal" states in S, and enforce the constraints implicitly by writing actions that always map real states to other real states (see Section 2.3.3).

Definition 2.5. The following terminology is borrowed loosely from first-order logic:

- An *atom* (short for *atomic formula*) or *positive literal* is either a rigid-relation assertion $rel(z_1, \ldots, z_n)$, or a state-variable assignment $x(z_1, \ldots, z_{n-1}) = z_n$, where *rel* or x is the name of the relation or state variable, and $z_1, \ldots, z_n$ are objects or object variables.
- A *negated* atom or *negative literal* is an atom with a negation sign in front of it, such as $\neg rel(z_1, \ldots, z_n)$ or $\neg\, x(z_1, \ldots, z_{n-1}) = z_n$. We usually will write the latter as $x(z_1, \ldots, z_{n-1}) \neq z_n$.
- Let e be any syntactic expression that contains literals. Then e is *ground* if it contains no object variables, *lifted* if it contains object variables but no objects, and *partially instantiated* if it includes both objects and object variables.
- If z is an object variable in e, then *instantiating* z means replacing z with either an object in $Range(z)$, or another object variable z' such that $Range(z') \subseteq Range(z)$. Instantiating e means instantiating zero or more of the object variables in e. The resulting expression is an *instance* of e. □

Remark 2.6. Although state variables and rigid relations may have arguments that are object constants or object variables, we do not—for now—allow the arguments to be other state variables, as in an expression such as $\mathsf{at}(\mathsf{p1}, \mathsf{loc}(\mathsf{r1}))$.[8] This restriction, and the restriction in Definition 2.2 that the cost function cannot depend on s, are needed to accommodate the requirements of some, but not all, of the algorithms in Parts I, II, and VI. Cases in which these restrictions can be relaxed or discarded are discussed in Sections 2.7.2 and 3.6.7 and the first paragraph of Chapter 5. □

2.3.2 Action Schemas and Actions

Definition 2.7. Given a type hierarchy $\mathcal{H}$, an *action schema* (or *action template*) is a tuple

$$\alpha = (\text{head}(\alpha), \text{pre}(\alpha), \text{eff}(\alpha), \text{cost}(\alpha)) \quad \text{or} \quad \alpha = (\text{head}(\alpha), \text{pre}(\alpha), \text{eff}(\alpha)),$$

where:

- $\text{head}(\alpha)$ is an expression of the form $\mathit{name}(z_1, \ldots, z_k)$, where *name* is a name and $(z_1, \ldots, z_k)$ is a list of zero or more object variables that are α's *parameters*. So that *name* will uniquely identify α, no other action can have the same name.
- $\text{pre}(\alpha) = \{p_1, \ldots, p_m\}$ is a set of zero or more *preconditions*, each of which is a literal. Within each literal p_i, every argument must be either an object or one of the parameters $z_1, \ldots, z_k$.

[8] Were it not for this restriction, the definition of instantiation in Definition 2.5 would also allow substituting a state variable x for an object variable z if $Range(x) \subseteq Range(z)$.

- $\text{eff}(\alpha) = \{x_1 = v_1, \ldots, x_n = v_n\}$ is a set of zero or more *effects*, each of which is a state-variable assignment for a different state variable. In each effect, the assigned value v_i must be either an object or one of $z_1, \ldots, z_k$.
- $\text{cost}(\alpha)$ is a positive number denoting the cost of applying actions that are instances of α. If it is omitted from the schema, it defaults to 1.
- Every parameter and state variable in α has a range that is one of the sets in $\mathcal{H}$.

Notation and terminology. To emphasize that each effect $x_i = v_i$ changes a state variable's value, we usually will instead write it as $x_i \leftarrow v_i$. Furthermore, we usually will write α in the following pseudocode format, omitting the last line if $c = 1$:

$$\begin{aligned} &\mathit{name}(z_1, z_2, \ldots, z_k) \\ &\quad \text{pre: } p_1, \ldots, p_m \\ &\quad \text{eff: } x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n \\ &\quad \text{cost: } c \end{aligned}$$

We will often refer to α by writing just its name, and to instances of α by writing just their heads, as in the following two examples. Such references are unambiguous because α's name is unique and its only variables are its parameters. □

Example 2.8. Continuing Example 2.1, here are three action schemas, where $c \in \mathit{Containers}$, $c' \in \mathit{Containers} \cup \{\text{nil}\}$, $d, d' \in \mathit{Docks}$, $p \in \mathit{Piles}$, and $r \in \mathit{Robots}$:

$$\begin{aligned} &\text{take}(r, c, c', p, d) && \textit{// } r \textit{ takes } c \textit{ off of } p \\ &\quad \text{pre: } \text{at}(p, d), \text{cargo}(r) = \text{nil}, \text{loc}(r) = d, \text{pos}(c) = c', \text{top}(p) = c \\ &\quad \text{eff: } \text{cargo}(r) \leftarrow c, \text{pile}(c) \leftarrow \text{nil}, \text{pos}(c) \leftarrow r, \text{top}(p) \leftarrow c' \end{aligned}$$

$$\begin{aligned} &\text{put}(r, c, c', p, d) && \textit{// } r \textit{ puts } c \textit{ onto } p \\ &\quad \text{pre: } \text{at}(p, d), \text{pos}(c) = r, \text{loc}(r) = d, \text{top}(p) = c' \\ &\quad \text{eff: } \text{cargo}(r) \leftarrow \text{nil}, \text{pile}(c) \leftarrow p, \text{pos}(c) \leftarrow c', \text{top}(p) \leftarrow c \end{aligned}$$

$$\begin{aligned} &\text{move}(r, d, d') && \textit{// } r \textit{ moves from } d \textit{ to } d' \\ &\quad \text{pre: } \text{adjacent}(d, d'), \text{loc}(r) = d, \text{occupied}(d') = \text{F} \\ &\quad \text{eff: } \text{loc}(r) \leftarrow d', \text{occupied}(d) \leftarrow \text{F}, \text{occupied}(d') \leftarrow \text{T} \end{aligned}$$

In take and put, one might be tempted to replace the preconditions $\text{at}(p, d)$ and $\text{loc}(r) = d$ with a single precondition $\text{at}(p, \text{loc}(r))$. In other situations, one might want to put computational formulas in the preconditions and effects of action schemas. The restriction in Remark 2.6 prevents these things, but later in the book we will discuss cases where the restriction can be relaxed. □

Let a be *ground instance* of an action schema, that is, a is an expression produced by substituting object variables for all of the action schema's parameters. If a is a ground instance of an action schema and $\text{eff}(a)$ assigns at most one value to each state variable, then a represents an action. If a state s satisfies $\text{pre}(a)$, then a is *applicable* in s, and applying it produces the following state:

$$\gamma(s, a) = \{\text{an assignment } x = w \text{ for each effect } x \leftarrow w \text{ in } \text{eff}(a)\} \cup \{\text{every}$$

assignment $x = w$ in s such that $\text{eff}(a)$ does not assign a value to $x\}$. (2.15)

Thus for every assignment $x = v$ in s,

$$\gamma(s,a) \text{ contains } \begin{cases} x = w & \text{if } \text{eff}(a) \text{ contains } x \leftarrow w \text{ for some } w, \\ x = v & \text{otherwise.} \end{cases}$$

If a isn't applicable in s, then $\gamma(s,a)$ is undefined.

Example 2.9. Let $\mathcal{A}$ be the set of action schemas in Example 2.8, and let $a_1 = \text{take}(\text{r1}, \text{c1}, \text{c2}, \text{p1}, \text{d1})$. Then

$$\begin{aligned} \text{pre}(a_1) &= \{\text{at}(\text{p1}, \text{d1}), \text{cargo}(\text{r1}) = \text{nil}, \text{loc}(\text{r1}) = \text{d1}, \text{pos}(\text{c1}) = \text{c2}, \text{top}(\text{p1}) = \text{c1}\}; \\ \text{eff}(a_1) &= \{\text{cargo}(\text{r1}) \leftarrow \text{c1}, \text{pile}(\text{c1}) \leftarrow \text{nil}, \text{pos}(\text{c1}) \leftarrow \text{r1}, \text{top}(\text{p1}) \leftarrow \text{c2}\}. \end{aligned}$$

It follows that a_1 is applicable to the state s_0 in Example 2.4. The state $\gamma(s_0, a_1)$ is shown in Figure 2.4. It is identical to s_0 except for the following changes:

$$\text{cargo}(\text{r1}) = \text{c1},\ \text{pile}(\text{c1}) = \text{nil},\ \text{pos}(\text{c1}) = \text{r1},\ \text{top}(\text{p1}) = \text{c2}. \qquad \square$$

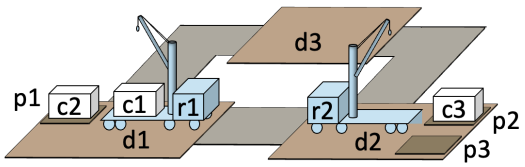


Figure 2.4. The state $\gamma(s_0, a_1)$, where s_0 and a_1 are as in Example 2.9.

2.3.3 Representing Planning Problems

In Section 2.2 we defined planning problems using a set of goal states S_g. To represent S_g we will use a set of ground literals g called a *goal formula*, with S_g being the set of all states that satisfy g, that is, $S_g = \{s \in S \mid s \models g\}$.[9]

For notational convenience, we will usually write a call to a planning algorithm as

$$\textit{Planner}(\Sigma, s_0, g) \quad \text{or} \quad \textit{Planner}(\Sigma, s_0, S_g),$$

where *Planner* is the name of the planning algorithm and (Σ, s_0, g) or (Σ, s_0, S_g) is the planning problem. However, as we explained at the start of Section 2.3, what the planner needs is not an exhaustive list of everything in Σ, but instead a compact representation with which it can quickly compute the parts of Σ that it needs. In most cases, the following information is sufficient:

- a type hierarchy $\mathcal{H}$,
- a set R of rigid relations,

[9] Obviously this places some limitations on what states can be in S_g. A widely used work-around is to add state variables to the domain to make it easier to represent important sets of states. An example is the cargo state variable in Example 2.8.

- a set X of state variables, including specifications of their ranges,
- a set $\mathcal{A}$ of action schemas,
- an initial state s_0,
- a goal formula g.

More specifically:

Definition 2.10. $(\mathcal{H}, R, X, \mathcal{A})$ is a *state-variable representation* of a state-transition system $\Sigma = (S, A, \gamma, \text{cost})$ in which S contains all total assignments of the state variables, A is the set of all actions represented by the action schemas in $\mathcal{A}$, and γ is given by Equation 2.15. Similarly, $(\mathcal{H}, R, X, \mathcal{A}, s_0, g)$ is a state-variable representation of the planning problem $P = (\Sigma, s_0, g)$.

Terminology. When using a state-variable representation for Σ, we will sometimes call Σ a *state-variable planning domain*. In this case, Σ is *lifted* if both X and $\mathcal{A}$ are lifted, and *ground* if both X and $\mathcal{A}$ are ground. □

In a state-variable representation, some of the total assignments in S may be nonsensical. For example, let $(\mathcal{H}, R, X, \mathcal{A})$ be the state-variable representation developed in Examples 2.1 and 2.8. In the environment that $(\mathcal{H}, R, X, \mathcal{A})$ is intended to represent, it would be nonsensical to have a state in which both $\mathsf{cargo}(\mathsf{r1}) = \mathsf{c1}$ and $\mathsf{loc}(\mathsf{c1}) = \mathsf{d3}$. The representation allows such states, but none of them can ever be reached from states such as the one in Figure 2.2.

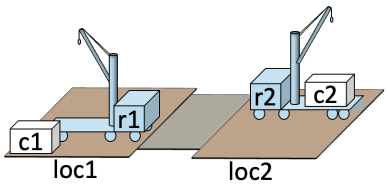
In principle, one could formulate constraints to exclude nonsensical states from S. However, if the initial state s_0 and the action schemas in $\mathcal{A}$ are defined properly, then no plan that begins at s_0 will ever produce a nonsensical state. Thus such constraints generally are unnecessary. In the AI planning literature, most of the classical-planning formulations do not use constraints on states.

2.4 Classical Representation

Classical representation[10] is an alternative to state-variable representation that has been widely used in the literature on automated planning. It differs from state-variable representation primarily in the following respects. All atoms have a name-and-arguments syntax, with no '=' or '←' symbols. Each state s is represented as the set of all atoms that are true in s, hence any atom not in this set is false in s. Each *planning operator* (the classical version of an action schema) has preconditions and effects that contain atoms and negated atoms.

Figure 2.5 shows state-variable and classical representations of a simple DWR planning problem. In pedagogical examples like the one in the figure, there usually are no type declarations in the classical-planning domain, depending instead on the actions to produce sensible values for the variables. However, computer implementations of these examples usually do include type declarations (see Section 2.4.1).

[10] This is also sometimes called "STRIPS representation" because it is similar (though not identical) to the representation used in the STRIPS planning system [358, 854].



(a) Initial state s_0

Types:
Robots = {r1,r2}, *Containers* = {c1,c2},
Locs = {loc1,loc2}, *Booleans* = {T, F}
$r \in$ *Robots*; $c \in$ *Containers*; $l, m \in$ *Locs*;
loc(r) $\in$ *Locs*;
pos(c) $\in$ *Locs* $\cup$ *Robots*;
loaded(r) $\in$ *Booleans*

take(r, c, l)
pre: loc(r, l), pos(c, l), ¬loaded(r)
eff: pos(c, l), ¬pos(c, r), loaded(r)

put(r, c, l)
pre: loc(r, l), pos(c, r)
eff: pos(c, l), ¬pos(c, r), ¬loaded(r)

move(r, l, m)
pre: loc(r, l)
eff: loc(r, m), ¬loc(r, l)

s_0 = {loc(r1,loc1), loc(r2,loc2),
pos(c1,loc1), pos(c2,r2),
loaded(r2)}

g = {pos(c1,loc2)}

(b) Classical representation

take(r, c, l)
pre: loc(r) = l, pos(c) = l, loaded(r) = F
eff: pos(c) ← r, loaded(r) ← T

put(r, c, l)
pre: loc(r) = l, pos(c) = r
eff: pos(c) ← l, loaded(r) ← F

move(r, l, m)
pre: loc(r) = l
eff: loc(r) ← m

s_0 = {loc(r1) = loc1, loc(r2) = loc2,
pos(c1) = loc1, pos(c2) = r2,
loaded(r1) = F, loaded(r2) = T}

g = {pos(c1) = loc2}

(c) State-variable representation

Figure 2.5. State-variable and classical representations of a simple planning problem. It is similar to Example 2.8, but with the following differences: there are no piles, containers cannot be stacked on each other, and both robots may be at the same location.

Instead of a list of positive and negative effects, classical planning operators sometimes are written with lists of atoms to add and delete from the current state. For example, in the take operator in Figure 2.5(b), the 'eff:' line may be replaced with

add: loaded(r), pos(c, l)
del: pos(c, r)

Classical and state-variable representations have equivalent expressive power. Each can be translated into the other with at most a polynomial increase in the size of the representation. Because of this, the computational complexity results in Section 2.5 are independent of whether the planning problems are represented in classical or state-variable representation.

2.4.1 PDDL Example

PDDL, the Planning Domain Definition Language, is based on classical representation but uses a LISP-like syntax. As an example, Figure 2.6 shows a PDDL version of the planning problem in Figure 2.5. The purpose of the `requirements` clause at the beginning of the domain definition is to specify what capabilities a planning system

```
(define (domain example-domain)

  (requirements
     :negative-preconditions)
  (:action take
     :parameters (?r ?l ?c)
     :precondition
         (and (loc ?r ?l)
              (loc ?c ?l)
              (not (loaded ?r)))
     :effect
         (and (not (loc ?c ?l))
              (loc ?c ?r)
              (loaded ?r)))
  (:action put
     :parameters (?r ?l ?c)
     :precondition
         (and (loc ?r ?l)
              (loc ?c ?r))
     :effect
         (and (loc ?c ?l)
              (not (loc ?c ?r))
              (not (loaded ?r))))
  (:action move
     :parameters (?r ?l ?m)
     :precondition
         (and (loc ?r ?l)
              (adjacent ?l ?m))
     :effect
         (and (not (loc ?r ?l))
              (loc ?r ?m))))


(define (problem example-prob)
  (:domain example-domain))
  (:init
      (adjacent loc1 loc2)
      (adjacent loc2 loc1)
      (loc c1 loc1)
      (loc c2 r2)
      (loc r1 loc1)
      (loc r2 loc2)
  (:goal (loc c1 loc2)))
```

Figure 2.6. PDDL representation of the classical planning problem in Figure 2.5.

Figure 2.7. Complexity of classical planning problems.

Is P lifted or ground?	Is Σ fixed in advance?	Complexity of PLAN EXISTENCE	Complexity of PLAN LENGTH
lifted	no	EXPSPACE-complete	NEXPTIME-complete
ground	no	PSPACE-complete	PSPACE-complete
lifted	yes	PSPACE	PSPACE
ground	yes	Constant time	Constant time

will need. Here, it specifies that the planner must be able to reason about negative preconditions such as `(not (loaded ?r))` in the `take` operator.

The domain definition in Figure 2.6 does not include a type hierarchy like the one in Figure 2.5(b). However, PDDL provides an option for specifying one, by including `:typing` in the `requirements` clause. PDDL also includes ways to write axioms for inferring properties that are not stated explicitly, preferences on which goals to achieve, elementary numeric operations, certain kinds of temporal constraints and deterministic exogenous events. For tutorial expositions of these and other features, see [480].

2.5 Computational Complexity

Computational complexity results are normally given for *decision problems*, where each decision problem is an infinite set questions that have yes/no answers. Here two

decision problems in which P may be any state-variable planning problem:

- PLAN EXISTENCE: does P have a solution?
- PLAN LENGTH: does P have a solution of length $\leq k$?

Figure 2.7 shows how the computational complexity of each decision problem P depends on whether the problem is lifted or ground, and whether the planning domain is given in the planner's input or fixed in advance (thus allowing a domain-specific planner to be used). Section 2.7.1 provides additional information.

The lower computational complexity values when P is ground do not mean that grounding a decision problem will make it easier to solve. Computational complexity is relative to the size of the problem representation, which is much larger for the grounded version of a problem than the lifted version, so grounding a problem makes makes its computational complexity smaller even though the amount of computation to solve it remains roughly the same.

Although these complexity results may look intimidating, they are *worst-case* results. There are many planning domains in which the time complexity is much lower (for example, many are polynomial in the average case, and some are polynomial even in the worst case). Furthermore, there are planning algorithms (such as variations of GBFS in Section 3.1.6) that can often find near-optimal solutions very quickly.

Because of those considerations, it might be more useful to think of the complexity results as indications of the expressivity of state-variable representation. Despite all of its restrictions, it is capable of expressing problems that are very hard to solve.

2.6 Acting

Suppose an actor calls a planning algorithm on a planning problem $P = (\Sigma, s_0, S_g)$, and the planner returns a solution plan π. If Σ were a perfect model of the environment, π would be guaranteed to produce the state $\gamma(s_0, \pi)$. However, because it is very unlikely that Σ will model the environment perfectly, unexpected outcomes might occur. These may be caused, for example, by problems with the actor's execution platform, incorrect information in the actor's model of the world, or exogenous events. In such situations, an actor can sometimes react by selectively choosing which parts of π to execute, as described in the next section. In other cases, the actor may need to call a planner to get a new plan, as discussed in Section 2.6.2.

2.6.1 Reactive Plan Execution

Sometimes an actor can react to unexpected events during plan execution by repeatedly choosing which parts of π to execute—for example, by re-executing some actions or skipping some actions. Algorithm 2.2, Reactive-Execution, is a procedure to do this. In the **for** loop at Line 2, it searches for a suffix $\langle a_i, \ldots, a_n \rangle$ of π that can achieve g from the current state s. It returns failure if the search is unsuccessful, and otherwise it executes a_i, gets the new current state, and repeats the search. The **for** loop at Line 2 is inefficient because it recomputes many of the same state transitions at each iteration of the loop, but Exercise 2.5 looks at some ways to speed it up.

```
Reactive-Execution(Σ, π, g)
    let ⟨a_1, ..., a_n⟩ be the actions in π
1   while True do
        s ← observe current state
        if s ⊨ g then return success
        a ← nil
2       for i ← n down to 1 do
            if γ(s, ⟨a_i, ..., a_n⟩) ⊨ g then
                a ← a_i
                exit the for loop
        if a = nil then return failure
        perform a
```

Algorithm 2.2. Reactive-Execution is an acting procedure that selects and executes parts of a plan π, repeating until it either achieves a goal g or fails.

Reactive-Execution can react very quickly in situations where parts of π are still capable of achieving the goal. In other situations, it may be necessary for the actor to acquire a new or modified plan. We now will discuss some acting procedures that do this by calling a planner that can be used online.

2.6.2 Acting with Lookahead

```
Run-Lookahead(Σ, g)
    while True do
        s ← observed current state
        if s ⊨ g then return success
        π ← Lookahead(Σ, s, g)
        if π = failure then return failure
        a ← pop-first-action(π)      // remove and return π's first action
        trigger the execution of a
```

Algorithm 2.3. Run-Lookahead, which replans before each action.

This section discusses two procedures based on the receding-horizon approach described in Section 1.1.4. Both procedures use an online planning algorithm, *Lookahead*, that is not required to return an entire solution plan. The plan may go part of the way to a goal state, or may even be a single action. Section 2.6.3 will discuss some ways to modify the planning algorithms in Chapter 3 to do this.

The first procedure is Algorithm 2.3, Run-Lookahead. Until it reaches a goal state, it repeatedly calls *Lookahead* to get a plan, performs the first action of the plan, and calls *Lookahead* again. This can be useful if the environment often changes

unpredictably in ways that can cause plans to fail, because it immediately detects such changes and replans. However, it might be impractical if *Lookahead* has a large running time, and it might be unnecessary if plan failures are infrequent.

```
Run-Lazy-Lookahead(Σ, g)
    π ← ⟨⟩
    while True do
        s ← observed state
        if s ⊨ g then return success
1       if π = ⟨⟩ or Simulate(Σ, s, g, π) = failure then
2           π ← Lookahead(Σ, s, g)
            if π = failure then return failure
        a ← pop-first-action(π)      // remove and return π's first action
        perform a
```

Algorithm 2.4. Run-Lazy-Lookahead, which replans only when necessary.

The second procedure is Algorithm 2.3, Run-Lazy-Lookahead. Repeatedly, it gets a plan π from *Lookahead* and executes π until either it reaches a goal and exits, or π ends, or *Simulate*, a plan simulator, says the rest of π will not work properly.

The purpose of *Simulate* is to detect potential future problems before they occur. A simple *Simulate* program could return success if $\gamma(s, \pi) \models g$ and failure otherwise. To detect more subtle problems, *Simulate* could instead do a more detailed test such as a physics-based simulation.

Compared to Run-Lookahead, Run-Lazy-Lookahead eliminates the overhead of planning at every step and replaces it with the overhead of running *Simulate* at every step. If *Simulate* is not too complicated, this will probably reduce the total overhead since it evaluates a single plan rather than a large space of plans. However, a potential advantage of Run-Lookahead is that it can respond to exogenous changes that make a better plan available, such as an unexpected opportunity to take a faster route to a destination or get a higher score in a game.

Both Run-Lookahead and Run-Lazy-Lookahead interleave acting and planning. It is also possible to write procedures in which the acting and planning processes run concurrently. This is more complicated because of the need to coordinate the two processes, but it can be useful in rapidly changing environments.

2.6.3 Interacting with an Online Planner

We emphasized earlier that *Lookahead* should be an online planner: the actor may call it frequently to get updated plans, and it may need to produce plans quickly so that the actor doesn't have to make long pauses between actions. However, most of the planning algorithms to be discussed in Chapter 3 are designed to run offline: they return only when they have found a solution plan or verified that no solution exists. In rapidly changing environments, the soundness of such planners can no longer be guaranteed: by the time that the planner returns a plan, changes to the environment

may already have invalidated it. To use such a planner online, one may want to modify the planner or how the actor interacts with it. We now discuss some possible modifications.

Subgoaling. One way that the actor can reduce the amount of time used by the planner is to call it on smaller planning problems. Instead of giving a planner the ultimate goal g, the actor may instead give the planner a *subgoal* that needs to be achieved in order to achieve g. Once the subgoal has been achieved, the actor may formulate its next subgoal and call the planner again.

In practical settings, formulating these subgoals is usually done in a domain-specific manner. However, one possible domain-independent approach may be to compute an ordered set of landmarks (see Section 3.2.3) and choose the earliest one as a subgoal.

Limited-horizon planning. Recall that in the receding-horizon technique, the planner starts at the current state and searches until it either reaches a goal or exceeds the planning horizon, then it returns the best solution or partial solution it has found. Several of the planning algorithms in Chapter 3 can be modified to do this.

The term *partial solution* is somewhat misleading because there is no guarantee that the plan will actually lead the actor toward a goal. However, even a complete solution plan will not always enable an actor to reach the goal, because the actor may encounter problems that are not in the planner's domain model.

Sampling. In a *sampling* search, the planner uses a modified version of greedy search (Section 3.1.2) in which the node selection is randomized. The choice can be purely random, or it can be weighted according to a heuristic evaluation (see Chapter 3). The modified algorithm can do this several times to generate multiple solutions, and either return the one that looks best or return the n best solutions so that the actor can evaluate them further. The UCT and UPOM algorithms in Chapters 9 and 15 use this technique.

2.6.4 Acting with Plan Repair

When an actor runs into problems while executing a plan π, sometimes it is preferable to repair π instead of calling *Lookahead* to get a new one. This can reduce the runtime needed for planning, and can also improve the plan's *stability*, that is, the amount of the original plan π that is retained in the repaired plan. While executing π, the actor might have made commitments to other actors that would be difficult to cancel, or may have obtained resources that are needed later in π and are important not to waste. Plan stability can also be important for human interaction, as users may be confused if an actor makes radical changes to π in response to trivial problems.

To modify Run-Lazy-Lookahead to try to repair plans, Line 2 can be replaced with

$$\pi \leftarrow \textit{Lookahead-Repair}(\Sigma, s, g, \pi)$$

where *Lookahead-Repair* should attempt to repair π, and return a new plan only if its repair attempts fail. Section 3.5 discusses some possible plan-repair algorithms.

2.7 Discussion and Bibliographic Notes

2.7.1 Classical and State-Variable Representations

Although problem representations based on state variables have long been used in control-system design [303, 439, 990] and operations research [7, 518, 1073], their use in automated-planning research came much later [68, 70, 390]. Instead, most automated-planning research has used representation and reasoning techniques derived from mathematical logic. This began with the early work on GPS [841] and the situation calculus [769] and continued with the STRIPS planning system [358] and the classical representation described in Section 2.4 [409, 713, 854, 881, 965]. Classical representation is sometimes called STRIPS representation, but it is somewhat simpler than the representation used in the STRIPS planner [358].

In classical and state-variable planning domains, it is possible to encode (rather awkwardly) arithmetic relations among finite sets of numbers [480]. State-variable representations can be extended to include real numeric state variables, but this incurs a sharp increase in computational complexity [488].

The PDDL representation language was first published in 1998 [408] for use in the AIPS-98 planning competition [733], the first of a long series of International Planning Competitions.[11] The language has gone through several updates and extensions, but has remained static since 2008. For an excellent exposition of its features, see [480].

The complexity results in Section 2.5, and several other related results, are proved in [327]. The proofs are stated using classical representation, but can easily be translated to state-variable representation.

Ground representations. A classical representation is *ground* if it contains no unground atoms. With this restriction, the planning operators have no parameters; hence each planning operator represents just a single action. Ground classical representations usually are called *propositional* representations [192], because the ground atoms can be rewritten as propositional variables.

Every classical representation can be translated into an equivalent propositional representation by replacing each planning operator with all of its ground instances (all of the actions that it represents), but this incurs a combinatorial explosion in the size of the representation. If a planning operator has p parameters and each parameter has v possible values, then there are v^p ground instances. In a ground classical representation, each instance must be written explicitly, thus increasing the size of the representation by a multiplicative factor of v^p.

A *ground state-variable representation* is one in which all of the state variables are ground. Each ground state variable can be rewritten as a state variable that has no arguments (like an ordinary mathematical variable) [70, 489, 939]. Every state-variable representation can be translated into an equivalent ground state-variable representation, with a combinatorial explosion like the one in the classical-to-propositional conversion. If an action schema has p parameters and each parameter has v possible values, then the ground representation is larger by a factor of v^p.

[11] See https://www.icaps-conference.org/competitions/

The propositional and ground state-variable representation schemes are both PSPACE-equivalent [69, 191]. They can represent exactly the same set of planning problems as classical and state-variable representations; but as we just discussed, they may require exponentially more space to do so. This lowers the complexity class because computational complexity is expressed as a function of the size of the input.

In a previous work [409, Section 2.5.4], we claimed that propositional and ground state-variable representations could each be converted into the other with at most a linear increase in size, but that claim was only partially correct. Propositional actions can be converted to ground state-variable actions with at most a linear increase in size, using a procedure similar to the one we used to convert planning operators to action schemas. For converting in the reverse direction, the worst-case increase in size is polynomial but superlinear.[12]

The literature contains several examples of cases in which the problem representation and the computation of heuristic functions can be done more easily with ground state variables than with propositions [490, 939]. Helmert [489, Section 1.3] advances a number of arguments for considering ground state-variable representations superior to propositional representations.

2.7.2 Generalized Domain Models

If we ignore Remark 2.6, state-variable representation can be generalized to let states be arbitrary data structures, and an action schema's preconditions, effects, and cost be arbitrary computable functions operating on those data structures. Analogous generalizations can be made to the classical representation in Section 2.4 by allowing a predicate's arguments to be functional terms whose values are calculated procedurally rather than inferred logically [480]. Such generalizations make some kinds of planning algorithms and search heuristics inapplicable (see Section 3.6.7), but can make the domain models applicable to a much larger variety of application domains,

There are several other ways to generalize the action models in Section 2.3.2, such as explicit models of time requirements or multiple possible outcomes. Parts III, IV, V, and VI discuss several such generalizations.

2.7.3 Online Planning

The automated planning literature started very early to address the problems of integrating a planner in the acting loop of an agent. Concomitant to the seminal paper on STRIPS [358], Fikes [357] proposed a program called Planex for monitoring the execution of a plan and revising planning when needed, and our Reactive-Execution algorithm is inspired by the "triangle table" data structure used in that work. Numerous contributions followed (e.g., [36, 175, 463, 824, 907, 1022, 1123]). As we

[12] We believe it is a multiplicative factor in the interval $[\lg v, v]$, where v is the maximum size of any state variable's range. The lower bound, $\lg v$, follows from the observation that if there are n state variables, then representing the states may require $n \lg v$ propositions, with commensurate increases in the size of the planning operators. The upper bound, v, follows from the existence of a conversion procedure that replaces each action's effect $x(c_1, \ldots, c_n) \leftarrow d$ with the following set of literals:
$$\{p_x(c_1, \ldots, c_n, d)\} \cup \{\neg x(c_1, \ldots, c_n, d') \mid d' \in \mathit{Range}(x(c_1, \ldots, c_n)) \setminus \{d\}\}.$$

remarked at the beginning of Section 2.6.3, a limitation of all these works is that their soundness cannot be guaranteed if the environment changes too rapidly [61].

Problems involving integration of classical planning algorithms into the control architecture of specific systems, such as spacecraft, robots, or Web services, have been extensively studied. However, many of these contributions have assumed, as we did tacitly in Section 2.6, that the plans generated by the planning algorithms were directly executable, an assumption that is often unrealistic. Part V will discuss the integration of planning and acting with refinement of actions into commands, and ways to react to events.

The receding-horizon technique has been widely used in control theory, specifically in model-predictive control. The survey by Garcia et al. [379] traces its implementation back to the early sixties. The general idea is to use a predictive model to anticipate over a given horizon the response of a system to some control and to select the control such that the response has some desired characteristics. Optimal control seeks a response that optimizes a criterion. The use of these techniques together with task planning has been explored by Dean and Wellman [283].

Subgoaling has been used in the design of several problem-solving and search algorithms (e.g., [639, 669]). It is especially useful if the goals are *serialized*, that is, ordered in a sequence such that each one can be achieved without negating the ones that were previously achieved. A set of goals is *serializable* if they can be serialized [639],[13] and serializable goals can be further classified as *trivially* serializable (for example, goals that are fully independent) and *laboriously* serializable [85].

In practical applications, subgoaling often involves domain-specific techniques. For example, the video game *Killzone 2* [212, 1135] does subgoal planning with the planner running several times per second, concurrently with acting, for short-term objectives such as "get to shelter" for its computerized opponents.

Sampling techniques are widely used for handling stochastic models of uncertainty and nondeterminism (see Part III).

2.8 Exercises

2.1. Let $P_1 = (\Sigma, s_0, g_1)$ and $P_2 = (\Sigma, s_0, g_2)$ be two classical planning problems with the same planning domain and initial state. Let $\pi_1 = \langle a_1, \ldots, a_n \rangle$ and $\pi_2 = \langle b_1, \ldots, b_n \rangle$ be solutions for P_1 and P_2, respectively. Let $\pi = \langle a_1, b_1, \ldots, a_n, b_n \rangle$.

(a) If π is applicable in s_0, then is it a solution for P_1? For P_2? Why or why not?
(b) E_1 be the set of all state variables in $\text{eff}(a_1), \ldots, \text{eff}(a_n)$, and E_2 be the set of all state variables in $\text{eff}(b_1), \ldots, \text{eff}(b_n)$. If $E_1 \cap E_2 = \varnothing$, then is π applicable in s_0? Why or why not?
(c) Let P_1 be the set of all state variables that occur in $\text{pre}(a_1), \ldots, \text{pre}(a_n)$, and P_2 be the set of all state variables that occur in the preconditions of $\text{pre}(b_1), \ldots, \text{pre}(b_n)$. If $P_1 \cap P_2 = \varnothing$ and $E_1 \cap E_2 = \varnothing$, then is π applicable in s_0? Is it a solution for P_1? For P_2? Why or why not?

[13] For example, Figure 2.8 is a nonserializable planning problem called the Sussman anomaly [1143].

pickup(x)
  pre: loc(x) = table, top(x) = nil,
    holding = nil
  eff: loc(x) ← hand, holding ← x

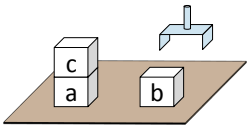
putdown(x)
  pre: holding = x
  eff: loc(x) ← table, holding ← nil

unstack(x, y)
  pre: loc(x) = y, top(x) = nil,
    holding = nil
  eff: loc(x) ← hand, top(y) ← nil,
    holding ← x

stack(x, y)
  pre: holding = x, top(y) = nil
  eff: loc(x) ← y, top(y) ← x,
    holding ← nil

(a) action schemas, where $x, y \in$ *Blocks*

Objects = *Blocks* ∪ {hand, table, nil}
Blocks = {a, b, c}



s_0 = {top(a) = c, top(b) = nil,
top(c) = nil, holding = nil,
loc(a) = table, loc(b) = table,
loc(c) = a}

g = {loc(a) = b, loc(b) = c}

(b) objects, initial state, and goal

Figure 2.8. A blocks-world planning domain and a planning problem.

2.2. Give a classical planning problem P_1 and a solution π_1 for P_1 such that π_1 is minimal but not shortest. Give a classical planning problem P_2 and a solution π_2 for P_2 such that π_2 is acyclic but not minimal.

2.3. Let $\Sigma = (S, A, \gamma)$ be the state-transition system represented by $\mathcal{H}$, R X, and $\mathcal{A}$ in Examples 2.1 and 2.8.

(a) How many states are in S? How many actions are in A? Briefly describe them.
(b) Let S' be the set of all states reachable from s_0, that is,

$$S' = \{\gamma(s_0, \pi) \mid \pi \text{ is a plan that is applicable in } s_0\}.$$

How many states are in S'? Give an example of a state in S that is not in S'.
(c) Do the states in S all have sensible meanings? Do the states in S'?
(d) Let $P = (\Sigma, s_0, g)$, where s_0 is as in Example 2.1 and g = {pos(c1) = d2}. Give a shortest solution for P. Give a solution that is minimal but not shortest. How many minimal solutions are there?

2.4. The *blocks world* is a well-known classical planning domain[14] in which a set of cubical blocks, *Blocks* = {a, b, c, . . .}, are arranged in stacks of varying size on an infinitely large table, table. To move the blocks, there is a robot hand, hand, that can hold at most one block at a time.

Figure 2.8(a) gives the action schemas. For each block x, loc(x) is x's location, which may be table, hand, or another block; and top(x) is the block (if any) that is on x, with top(x) = nil if nothing is on x. Finally, holding tells what block the robot hand is holding, with holding = nil if the hand is empty.

[14] More accurately, because the number of blocks may vary, it is a set of planning domains.

(a) Why are there four action schemas rather than just two?
(b) Is the state variable holding really needed? Why or why not?
(c) In the planning problem in Figure 2.8(b), how many states satisfy g?
(d) Give necessary and sufficient conditions for a set of blocks-world atoms to be a state.
(e) Is every blocks world planning problem solvable? Why or why not?

2.5. This exercise involves the **for** loop at Line 2 of Reactive-Execution.

(a) Give the loop's big-O time complexity as a function of n and k, where n is the number of actions in π, and k is the maximum number of preconditions and effects of each action in π.
(b) The **for** loop can be made much faster by using a table that relates each action's preconditions to effects of previous actions and the initial state, and relates each action's effects to the preconditions of subsequent actions and the goal. Write such a data structure, rewrite the **for** loop to use it, and analyze the resulting time complexity.

2.6. Suppose an actor starts in state s_0 of the planning problem shown in Figure 2.5, using Run-Lazy-Lookahead with a *Lookahead* algorithm that always returns the shortest possible solution plan. The first call to *Lookahead* returns

$$\pi = \{\text{take(r1,c1,loc1)}, \text{ move(r1,loc1,loc2)}, \text{ put(r1,c1,loc2)}\}.$$

(a) Suppose that after the actor has performed take(r1,c1,loc1) and move(r1,loc1,loc2), monitoring reveals that c1 fell off of the robot and is still back at loc1. Tell what will happen, step by step. Assume that *Lookahead*(P) will always return the best solution for P.
(b) Repeat part (a) assuming that c1 will fall off of the robot every time it performs move(r1,loc1,loc2).
(c) Repeat part (a) using Run-Lookahead.
(d) Suppose that after the actor has performed take(r1,c1,loc1), monitoring reveals that r1's wheels have stopped working, hence r1 cannot move from loc1. What should the actor do to recover? How would you modify Run-Lazy-Lookahead and Run-Lookahead to accomplish this?

2.7. Consider the planning domain in Examples 2.1 and 2.8.

(a) Rewrite the planning domain using classical representation.
(b) Rewrite it in PDDL.