Chapter 10
Control Rules in Planning

10.1 Introduction

In the Abstract-search procedure in Figure III.1, the purpose of the Prune function is to detect unpromising nodes and prune them from the search space. The efficiency of a planning procedure can often be improved dramatically if a good way can be found to do this—for example, in some cases it may enable a procedure to solve problems in polynomial time that might otherwise take exponential time.

Deciding whether a node can be pruned often involves a collection of highly domain-specific tests to detect situations in which we can be confident that the solutions that lie below a node are less desirable than the solutions that lie below nodes elsewhere in the search space. Here is an example.

Example 10.1 Suppose we want to solve a container-stacking problem (see Section 4.5) using a modified version of Forward-search that includes a Prune function to prune undesirable nodes. Recall that each node corresponds to a state s, and the node's children are produced by applying actions to s. If we consider long plans to be less desirable than short ones, then here are some situations in which some of those children can be pruned.

1. Suppose that in the state s there is a container c whose position is consistent with the goal g. Then c will never need to be moved. For any action a that moves c and any solution plan π that applies a to s, there is a shorter solution π' that does not apply a to s. Thus the state $\gamma(s, a)$ can be pruned.
2. Suppose c's position is inconsistent with g but there is an action a that moves c to a position consistent with g. Then for any action b that moves c to a position inconsistent with g and for any solution plan π that applies b to s, there is a shorter solution π' that applies a to s instead. Thus we can prune $\gamma(s, b)$.

If we modify Forward-search to prune nodes in these situations and a few others, it will search the same search space as the Stack-containers procedure of Section 4.5. Thus it will find near-optimal solutions in low-order polynomial time. ■

In order to write pruning rules like the ones above, we need a language for expressing relationships among properties of the current state and properties of subsequent states. This chapter describes such a language and a planning algorithm based on it. Our presentation is based loosely on ideas developed by Bacchus and Kabanza [33].

Section 10.2 describes a language for writing control rules, Section 10.3 describes a procedure for evaluating control rules written in that language, and Section 10.4 describes a planning procedure based on the inference procedure. Section 10.5 describes ways to extend the approach. Section 10.6 discusses how to handle certain kinds of extended goals.

10.2 Simple Temporal Logic

This section describes a logical formalism that we will call *Simple Temporal Logic* (STL). STL extends first-order logic to include some *modal operators* that express relationships among the current state and subsequent states.

Let $\mathcal{L}$ be any function-free first-order language. Then $\mathcal{L}_\mathcal{T}$, the STL language based on $\mathcal{L}$, includes all the symbols of $\mathcal{L}$ plus the following:

- The propositional constants true, which is an atom that is always true, and false, which is an atom that is always false.
- The modal operators $\cup$ (until), $\Box$ (always), $\Diamond$ (eventually), $\bigcirc$ (next), and GOAL. These operators have the following syntax. If ϕ_1 and ϕ_2 are formulas, then so are $\phi_1 \cup \phi_2$, $\Box\,\phi_1$, $\Diamond\,\phi_1$, and $\bigcirc\,\phi_1$. If ϕ_1 contains no modal operators, then GOAL(ϕ_1) is a formula.

Below are the rules for how to interpret a formula ϕ of $\mathcal{L}_\mathcal{T}$. The interpretation of ϕ involves not just the current state as in classical planning (see Chapter 2) but instead a triple (S, s_i, g), where $S = \langle s_0, s_1, \ldots\rangle$ is an infinite sequence of states, $s_i \in S$ is the current state, and g is a goal formula.

The case that will interest us, of course, is where $\mathcal{L}_\mathcal{T}$ is based on the language $\mathcal{L}$ of some planning domain $\mathcal{D}$, and S is the sequence of states produced by a plan. One difficulty is that a finite plan $\pi = \langle a_1, a_2, \ldots, a_n\rangle$ produces only a finite sequence of states $S_\pi = \langle s_0, \ldots, s_n\rangle$, where $s_1 = \gamma(s_0, a_1), s_2 = \gamma(s_1, a_2), \ldots, s_n = \gamma(s_{n-1}, a_n)$. The work-around is to consider the infinite sequence of states $\hat{S}_\pi = \langle s_0, s_1, \ldots, s_{n-1}, s_n, s_n, s_n, \ldots\rangle$. Intuitively, this is the sequence of states that we get by executing π and then doing nothing forever afterward.

Definition 10.1 Let ϕ be an STL formula, $S = \langle s_0, s_1, \ldots\rangle$ be an infinite sequence of states, and g be a goal formula like the ones in Chapter 2. Then here are the rules that define whether $(S, s_i, g) \models \phi$.

- If ϕ is a ground atom, then $(S, s_i, g) \models \phi$ iff $s_i \models \phi$. In other words, ϕ is true if it is true in the current state.

- Quantifiers and logical connectives have the same semantic rules as in first-order logic. For example, $(S, s_i, g) \models \phi_1 \wedge \phi_2$ iff both $(S, s_i, g) \models \phi_1$ and $(S, s_i, g) \models \phi_2$.
- $(S, s_i, g) \models \Box\phi$ iff $(S, s_j, g) \models \phi$ for every $j \geq i$. For example, if ϕ contains no modal operators, then $\Box\phi$ is true iff ϕ is true in s_i and all subsequent states.
- $(S, s_i, g) \models \bigcirc\phi$ iff $(S, s_{i+1}, g) \models \phi$, i.e., $\bigcirc\phi$ is true iff ϕ is true in s_{i+1}.
- $(S, s_i, g) \models \Diamond\phi$ iff there is a $j \geq i$ such that $(S, s_j, g) \models \phi$, i.e., $\Diamond\phi$ is true iff ϕ is true in some state s_j that is after s_i.
- $(S, s_i, g) \models \phi_1 \cup \phi_2$ iff there is a $j \geq i$ such that $(S, s_k, g) \models \phi_1$ for $k = i, \ldots, j-1$ and $(S, s_j, g) \models \phi_2$, i.e., $\phi_1 \cup \phi_2$ is true if ϕ_1 is true in all states from s_i up until the first state (if any) in which ϕ_2 is true.
- $(S, s_i, g) \models \textsc{goal}(\phi)$ iff $\phi \in g$. In other words, $\textsc{goal}(\phi)$ is true if ϕ is a goal.

If ϕ contains no occurrences of the GOAL operator, then g is irrelevant, and in this case we will write $(S, s_i) \models \phi$ to mean $(S, s_i, g) \models \phi$. ■

Example 10.2 Let $S = \langle s_0, s_1, s_2, \ldots \rangle$ be a sequence of DWR states. Then:

- Each of the following means that on(c1,c2) and on(c2,c3) are true in s_2:

$$s_2 \models \text{on(c1, c2)} \wedge \text{on(c2, c3)}$$
$$(S, s_0) \models \bigcirc\bigcirc(\text{on(c1, c2)} \wedge \text{on(c2, c3)})$$
$$(S, s_1) \models \bigcirc(\text{on(c1, c2)} \wedge \text{on(c2, c3)})$$
$$(S, s_2) \models (\text{on(c1, c2)} \wedge \text{on(c2, c3)})$$

- $(S, s_i) \models \Box\neg\text{holding(crane1, c1)}$ means that holding(crane1, c1) is false in all of $s_i, s_{i+1}, \ldots$.
- $(S, s_i) \models \forall x(\text{on}(x, \text{c1}) \to \Box\,\text{on}(x, \text{c1}))$ means that if another container is on c1 in s_i, then the same container is on c1 in all subsequent states. $(S, s_i) \models \forall x\ \Box(\text{on}(x, \text{c1}) \to \bigcirc\,\text{on}(x, \text{c1}))$ has the same meaning.
- $(S, s_i, g) \models \forall y\,(\text{on}(y, \text{pallet}) \wedge \textsc{goal}(\text{on}(y, \text{pallet})) \to \bigcirc\,\text{on}(y, \text{pallet}))$ means that for every container that is at the bottom of a pile in both s_i and the goal, it is also at the bottom of a pile in s_{i+1}. ■

The next example illustrates how to use STL to write a *control formula* describing situations where states can be pruned.

Example 10.3 Here is how to write an STL formula ϕ that encodes item 1 in Example 10.1. First, here is a formula $\phi_1(c, d, p)$ that holds if it is acceptable for the container c to be on the item d (which may be either a container or a pallet) in the pile p, i.e., if there is no goal requiring c to be in another pile or on top of something else:

$$\begin{aligned}\phi_1(c, d, p) = {} & [\textsc{goal}(\text{in}(c, p)) \vee \neg\exists q\ \textsc{goal}(\text{in}(c, q))] \\ & \wedge [\textsc{goal}(\text{on}(c, d)) \vee \neg\exists e\ \textsc{goal}(\text{on}(c, e))]\end{aligned}$$

Here is a formula $\phi_2(c, p)$ saying that $\mathsf{ok}(c, p)$ holds iff c is in the pile p and c's position is consistent with the goal:

$$\phi_2(c,p) \;=\; \mathsf{ok}(c,p) \leftrightarrow [\mathsf{same}(p, \mathsf{pallet}) \vee \exists d\,(\phi_1(c,d,p) \wedge \mathsf{ok}(d,p))]$$

Here is a formula $\phi_3(c)$ that holds iff c's position is consistent with the goal:

$$\phi_3(c) \;=\; \exists p\,(\phi_2(c,p) \wedge \mathsf{ok}(c,p))$$

Finally, here is a formula ϕ that holds iff for every container c whose position is consistent with g, c will never be moved, i.e., c will always remain in the same pile and on the same item in that pile:

$$\phi \;=\; \forall c\,[\phi_3(c) \rightarrow \exists p \exists d\, \Box(\mathsf{in}(c,p) \wedge \mathsf{in}(c,d))]$$

■

10.3 Progression

In order to use control formulas to prune nodes during planning, one of the key ideas is to compute the *progression* of a control formula ϕ from a state s_i to the next state s_{i+1}. The progression of ϕ is a formula progress (ϕ, s_i) that is true in s_{i+1} iff ϕ is true in s_i.

First, we define a formula progr (ϕ, s_i) recursively as follows. If ϕ contains no modal operators, then:

$$\text{progr}(\phi, s_i) = \begin{cases} \mathsf{true} & \text{if } s_i \models \phi \\ \mathsf{false} & \text{if } s_i \not\models \phi \end{cases}$$

Logical connectives are handled in the usual fashion:

$$\text{progr}(\phi_1 \wedge \phi_2, s_i) = \text{progr}(\phi_1, s_i) \wedge \text{progr}(\phi_2, s_i)$$
$$\text{progr}(\neg\phi, s_i) = \neg\text{progr}(\phi, s_i)$$

Quantifiers are handled as follows. Because $\mathcal{L}_\mathcal{T}$ has no function symbols, its only ground terms are $c_1, \ldots, c_k$. For $j = 1, \ldots, k$, let θ_j be the substitution $\{x \leftarrow c_j\}$. Then:

$$\text{progr}(\forall x\phi, s_i) = \text{progr}(\theta_1(\phi), s_i) \wedge \ldots \wedge \text{progr}(\theta_k(\phi), s_i)$$
$$\text{progr}(\exists x\phi, s_i) = \text{progr}(\theta_1(\phi), s_i) \vee \ldots \vee \text{progr}(\theta_k(\phi), s_i)$$

Modal operators are handled as follows:

$$\text{progr}(\bigcirc\,\phi, s_i) = \phi$$
$$\text{progr}(\phi_1 \cup \phi_2, s_i) = ((\phi_1 \cup \phi_2) \wedge \text{progr}(\phi_1, s_i)) \vee \text{progr}(\phi_2, s_i)$$
$$\text{progr}(\Diamond\,\phi, s_i) = (\Diamond\,\phi) \vee \text{progr}(\phi, s_i)$$
$$\text{progr}(\Box\,\phi, s_i) = (\Box\,\phi) \wedge \text{progr}(\phi, s_i)$$

Next, progress(ϕ, s_i) is the formula produced from progr(ϕ, s) by performing the usual simplifications (i.e., replace true $\wedge\, d$ with d, true $\vee\, d$ with true, $\neg$true with false, false $\wedge\, d$ with false, and so forth).

progr(ϕ, s) and progress(ϕ, s) can be computed in low-order polynomial time[1] by writing algorithms that directly implement their definitions (see Exercise 10.1). Proposition 10.1 says that progress(ϕ, s) means what it is supposed to mean.

Proposition 10.1 $(S, s_i, g) \models \phi$ *iff* $(S, s_{i+1}, g) \models$ progress(ϕ, s_i).

The proof is by induction (Exercise 10.10), but here is a simple example.

Example 10.4 Here is a proof for the special case where $\phi = \Box\, \mathsf{on(c1, c2)}$. There are two cases. The first one is where $s_i \models \mathsf{on(c1, c2)}$. In this case, $s_i \models \phi$ iff $s_{i+1} \models \phi$, so:

$$\begin{aligned} \text{progress}(\phi, s_i) &= \Box\, \mathsf{on(c1,c2)} \wedge \text{progress}(\mathsf{on(c1,c2)}, s_i) \\ &= \Box\, \mathsf{on(c1,c2)} \wedge \mathsf{true} \\ &= \phi \end{aligned}$$

The second case is where $s_i \models \neg\mathsf{on(c1, c2)}$. In this case, $s_i \not\models \phi$, so:

$$\begin{aligned} \text{progress}(\phi, s_i) &= \Box\, \mathsf{on(c1,c2)} \wedge \text{progress}(\mathsf{on(c1,c2)}, s_i) \\ &= \Box\, \mathsf{on(c1,c2)} \wedge \mathsf{false} \\ &= \mathsf{false} \end{aligned}$$

Thus, $s_i \models \phi$ iff $s_{i+1} \models$ progress(ϕ, s_i). ■

If ϕ is a closed formula and $S = \langle s_0, \ldots, s_n \rangle$ is any finite sequence of states (e.g., the sequence of states produced by a plan), then we define progress(ϕ, S) to be the result of progressing ϕ through all of those states:

$$\text{progress}(\phi, S) = \begin{cases} \phi & \text{if } n = 0 \\ \text{progress}(\text{progress}(\phi, \langle s_0, \ldots, s_{n-1} \rangle), s_n) & \text{otherwise} \end{cases}$$

The following two propositions tell how to use a control formula to prune nodes during planning. Proposition 10.2 can be proved by induction (see Exercise 10.11), and Proposition 10.3 follows as a corollary (see Exercise 10.12).

Proposition 10.2 *Let $S = \langle s_0, s_1, \ldots \rangle$ be an infinite sequence of states and ϕ be an STL formula. If $(S, s_0, g) \models \phi$, then for every finite truncation $S' = \langle s_0, s_1, \ldots, s_i \rangle$ of S,* progress$(\phi, S') \neq$ false.

1. Readers who are familiar with model checking will note that this is much lower than the exponential time complexity for progression in model checking.

Proposition 10.3 *Let s_0 be a state, π be a plan applicable to s_0, and $S = \langle s_0, \ldots, s_n \rangle$ be the sequence of states produced by applying π to s_0. If ϕ is an STL formula and* $\mathsf{progress}(\phi, S) = \mathsf{false}$, *then S has no extension $S' = \langle s_0, \ldots, s_n, s_{n+1}, s_{n+2}, \ldots \rangle$ such that $(S', s_0, g) \models \phi$.*

Example 10.5 Let s_0 and g be be the initial state and goal formula, respectively, for a container-stacking problem whose constant symbols are $c_1, \ldots, c_k$. Let ϕ_1, ϕ_2, ϕ_3, and ϕ be as in Example 10.3 (see page 219). Then:

$$\begin{aligned}
\mathsf{progress}(\phi, s_0) &= \mathsf{progress}(\forall c\ [\phi_3(c) \rightarrow \exists p \exists d\ \Box(\mathsf{in}(c,p) \land \mathsf{in}(c,d))], s_0) \\
&= \mathsf{progress}([\phi_3(c_1) \rightarrow \exists p \exists d\ \Box(\mathsf{in}(c_1,p) \land \mathsf{in}(c_1,d))], s_0) \\
&\quad \land \mathsf{progress}([\phi_3(c_2) \rightarrow \exists p \exists d\ \Box(\mathsf{in}(c_2,p) \land \mathsf{in}(c_2,d))], s_0) \\
&\quad \land \ldots \\
&\quad \land \mathsf{progress}([\phi_3(c_k) \rightarrow \exists p \exists d\ \Box(\mathsf{in}(c_k,p) \land \mathsf{in}(c_k,d))], s_0)
\end{aligned}$$

Suppose that in s_0, there is exactly one container c_1 whose position is consistent with g. Then $s_0 \models \phi_3(c_1)$, and $s_0 \not\models \phi_3(c_i)$ for $i = 2, \ldots, k$. Suppose also that in s_0, c_1 is on item d_1 in pile p_1. Then:

$$\begin{aligned}
\mathsf{progress}(\phi, s_0) &= \mathsf{progress}(\exists p \exists d\ \Box(\mathsf{in}(c_1,p) \land \mathsf{in}(c_1,d)), s_0) \\
&= \mathsf{progress}(\Box(\mathsf{in}(c_1,p_1) \land \mathsf{in}(c_1,d_1)), s_0) \\
&= \Box(\mathsf{in}(c_1,p_1) \land \mathsf{in}(c_1,d_1))
\end{aligned}$$

Let a be an action applicable to s_0. If a moves c_1, then c_1 will no longer be on item d_1 in pile p_1, whence $\gamma(s_0, a) \not\models \mathsf{progress}(\phi, s_0)$. Thus $\gamma(s_0, a)$ can be pruned. ■

10.4 Planning Procedure

It is straightforward to modify the Forward-search procedure of Chapter 4 to prune any partial plan π such that $\mathsf{progress}(\phi, S_\pi) = \mathsf{false}$. Furthermore, there is an easy way to optimize the computation of $\mathsf{progress}(\phi, S_\pi)$, by setting $\phi \leftarrow \mathsf{progress}(\phi, s)$ each time through the loop. The optimized procedure, STL-plan, is shown in Figure 10.1.

The soundness of STL-plan follows directly from the soundness of Forward-search. Proposition 10.4 establishes a condition on ϕ that is sufficient to make STL-plan complete.

Proposition 10.4 *Let (O, s_0, g) be the statement of a solvable planning problem $\mathcal{P}$ and ϕ be an STL formula such that $(\hat{S}_\pi, s_0, g) \models \phi$ for at least one solution π of $\mathcal{P}$. Then* $\mathsf{STL\text{-}plan}(O, s_0, g, \phi)$ *is guaranteed to return a solution for $\mathcal{P}$.*

Proof From the condition on ϕ, we know there is at least one solution π for $\mathcal{P}$ such that $(\hat{S}_\pi, s_0, g) \models \phi$. Thus from Proposition 10.2, it follows that STL-plan will not

```
STL-plan($O, s_0, g, \phi$)
    $s \leftarrow s_0$
    $\pi \leftarrow$ the empty plan
    loop
        if $\phi$ = false then return failure
        if $s$ satisfies $g$ then return $\pi$
        $A \leftarrow \{a \mid a$ is a ground instance of an operator in $O$
                and precond($a$) is true in $s\}$
        if $A = \emptyset$ then return failure
        nondeterministically choose an action $a \in A$
        $s \leftarrow \gamma(s, a)$
        $\pi \leftarrow \pi.a$
        $\phi \leftarrow$ progress($\phi, s$)
```

Figure 10.1 STL-plan, a modified version of Forward-search that prunes any plan π such that progress(ϕ, S_π) = false.

prune any state along the path generated by π. Thus at least one of STL-plan's nondeterministic traces will return π. Other nondeterministic traces may possibly return other plans, but it follows from the soundness of Forward-search that any plan returned by STL-plan will be a solution. ∎

In some cases it can be proved that a control formula ϕ satisfies the requirements of Proposition 10.4 (e.g., see Exercise 10.13). However, in general, writing a good control formula is much more of an *ad hoc* activity. A control formula is basically a computer program in a specialized programming language; thus control formulas need to be debugged just like any other computer programs.

10.5 Extensions

Section 2.4 discussed how to augment classical planning to incorporate extensions such as axioms, function symbols, and attached procedures. Similar extensions can be made to STL and to the STL-plan procedure, with a few restrictions as described in this section.

Function Symbols. Recall from Section 10.3 that we defined progr($\forall x \phi$) and progr($\exists x \phi$) to be conjunctions and disjunctions over all of the ground terms of $\mathcal{L}_\mathcal{T}$. If we add function symbols to $\mathcal{L}$, then these definitions become uncomputable because there are infinitely many ground terms. One way around this difficulty is to restrict every universally quantified formula to have the form $\forall x\,(a \rightarrow \phi)$ and every existentially quantified formula to have the form $\exists x\,(a \wedge \phi)$, where a is an atom known to be true for only finitely many values of x. Then instead of evaluating ϕ for every possible value of x, we only need to consider the values of x that make a true.[2]

Axioms. To include axioms, we need a way to perform axiomatic inference. The easiest approach is to restrict the axioms to be Horn clauses, and use a Horn-clause theorem prover. If an operator or method has a positive precondition p, then we take p to be true iff it can be proved from the current state s.

In some planning domains, it is useful to allow negated conditions to occur in the tails of the Horn clauses and in the preconditions of the planning operators. If we do this, then there is a problem with what it means for a condition to be satisfied by a state because there is more than one possible semantics for what logical entailment might mean [500]. However, if we restrict the set of Horn-clause axioms to be a stratified logic program, then the two major semantics for logical entailment agree with each other [46], and in this case the inference procedure will still be sound and complete.

Attached Procedures. It is easy to generalize STL-plan and similar planning procedures to allow some of the function symbols and predicate symbols to be evaluated as attached procedures. In some cases, restrictions can be put on the attached procedures that are sufficient to preserve soundness and completeness. In the most general case, soundness and completeness cannot be preserved, but attached procedures are nevertheless quite useful for practical problem solving.

Time. It is possible to generalize STL-plan and similar procedures to do certain kinds of temporal planning, e.g., to deal with actions that have time durations and may overlap with each other. The details are beyond the scope of this chapter, but some references are given in Section 10.7.

10.6 Extended Goals

By using STL, we can extend the classical planning formalism to allow a much larger class of extended goals than the ones that can be translated into ordinary classical planning problems. We do this by defining an *STL planning problem* to be a 4-tuple (O, s_0, g, ϕ), where ϕ is a control formula expressed in STL. A *solution* to an STL

2. In practice, a similar restriction is normally used even if $\mathcal{L}$ is function-free, in order to alleviate the combinatorial explosion that occurs when a formula contains multiple quantifiers. For example, if $\phi = \forall x_1 \forall x_2 \ldots \forall x_n\, \phi'$ and there are k different possible values for each variable x_i, then progr(ϕ) is a conjunct of k^n different instances of ϕ. The restriction can decrease the complexity of this computation in the same way as if we had decreased the value of k.

planning problem is any plan $\pi = \langle a_1, \ldots, a_k \rangle$ such that the following conditions are satisfied.

- π is a solution to the classical planning problem (O, s_0, g).
- For $i = 1, \ldots, k$, let $s_i = \gamma(s_{i-1}, a_i)$. Then the infinite sequence of states $\langle s_0, s_1, \ldots, s_k, s_k, s_k, \ldots \rangle$ satisfies ϕ.

Here are examples of how to encode several kinds of extended goals in STL planning problems.

- Consider a DWR domain in which there is some location bad-loc to which we never want our robots to move. To express this as a classical planning domain, it is necessary to add an additional precondition to the move operator (see Section 2.4.8). For STL planning, such a change to the move operator is unnecessary because we can instead use the control rule

$$\phi = \Box \neg \mathrm{at(r1, bad\text{-}loc)}.$$

- Suppose that r1 begins at location loc1 in the initial state, and we want r1 to go from loc1 to loc2 and back exactly twice. In Section 2.4.8, we said that this kind of problem cannot be expressed as a classical planning problem. However, it can be expressed as an STL planning problem by using this control rule:

$$\phi = (\bigcirc\, \mathrm{at(loc2)}) \wedge (\bigcirc \bigcirc\, \mathrm{at(loc1)}) \wedge (\bigcirc \bigcirc \bigcirc\, \mathrm{at(loc2)}) \wedge (\bigcirc \bigcirc \bigcirc \bigcirc \Box\, \mathrm{at(loc1)})$$

- To require that every solution reach the goal in five actions or fewer, we can use this control rule:

$$\phi = g \vee (\bigcirc g) \vee (\bigcirc \bigcirc g) \vee (\bigcirc \bigcirc \bigcirc g) \vee (\bigcirc \bigcirc \bigcirc \bigcirc g)$$

- Suppose we want to require that the number of times r1 visits loc1 must be at least three times the number of times it visits loc2. Such a requirement cannot be represented directly in an STL planning problem. However, it can be represented using the same approach we described for classical planning in Section 2.4.8: extend STL planning by introducing the predicate symbols, function symbols, and attached procedures needed to perform integer arithmetic, modify the move operator to maintain atoms in the current state that represent the number of times we visit each location, and modify the goal to include the required ratio of visits. Unlike most classical planning procedures, STL-plan can easily accommodate this extension. The key is to ensure that all of the variables of an attached procedure are bound at the time the procedure is called. This is easy to accomplish in STL-plan because it plans forward from the initial state.

Because STL-plan returns only plans that are finite sequences of actions, there are some kinds of extended goals it cannot handle. Here is an example.

Example 10.6 Consider an STL planning problem $\mathcal{P}$ in which $s_0 = \{p\}$, $g = \emptyset$, $\phi = \Box(\Diamond p \wedge \Diamond \neg p)$, and O contains the following operators:

off	on	no-op
precond: p	precond: ¬p	precond: *(none)*
effects: ¬p	effects: p	effects: *(none)*

Now, consider the infinite sequence of actions:

$$\pi = \langle \text{off}, \text{on}, \text{off}, \text{on}, \text{off}, \text{on}, \ldots \rangle$$

Our formal definitions do not admit π as a solution, nor will STL-plan return it. However, there are situations in which we would want to consider π to be a solution for a planning problem. We might consider modifying STL-plan to print out π one action at a time. However, because STL-plan is nondeterministic, there is no guarantee that it would print out π rather than, say:

$$\langle \text{no-op}, \text{no-op}, \text{no-op}, \ldots \rangle$$

Part V describes some nonclassical planning formalisms and algorithms that address problems like these. ■

10.7 Discussion and Historical Remarks

One of the earliest planning systems to incorporate search-control rules was PRODIGY [105, 524, 525].[3] Unlike the approach described in this chapter, in which the control rules are written by a human, PRODIGY's rules are designed to be learned automatically. They are not written in a logical formalism like STL but instead act like expert-system rules for guiding algorithmic choices made by the search algorithm. Although PRODIGY's planning speed is not competitive with more recent control-rule planners such as TLPlan and TALplanner (discussed next), it can use its learning techniques to improve its efficiency on various classical planning problems [105].

STL is a simplification of the version of the Linear Temporal Logic that Bacchus and Kabanza used in their TLPlan system [33].[4] Similarly, STL-plan is a simplification of TLPlan's planning algorithm. TLPlan incorporates most or all of the extensions we described in Section 10.5. If it is given a good set of domain-specific control rules to guide its search, TLPlan can outperform all of the classical planning systems described in Parts I and II. TLPlan won one of the top two prizes in the 2002 International Planning Competition [195].

STL's modal operators are similar to those used in model checking, but they are used in a different way. In STL-plan, modal formulas are used to prune paths in a forward-search algorithm. In model checking (see Appendix C), the objective is to determine whether a modal formula will remain true throughout all possible

3. For a downloadable copy of PRODIGY, see http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/prodigy/Web/prodigy-home.html.
4. For a downloadable copy of TLPlan, see http://www.cs.toronto.edu/~fbacchus/tlplan.html.

behaviors of a nondeterministic finite-state system, and the algorithms for doing this are rather different than forward search. However, these algorithms can also be used to do planning (see Chapter 17).

Doherty and Kvarnström's TALplanner [160, 342] has a planning algorithm similar to TLPlan's but uses a different temporal logic. TALplanner has capabilities and efficiency similar to those of TLPlan; TALplanner won one of the top two prizes in the AIPS '00 Planning Competition [28]. Both TLPlan and TALplanner can do certain kinds of temporal planning [29, 342]. See Long and Fox [368] for a discussion of some of the issues involved.

Advantages and Disadvantages. Compared to classical and neoclassical planners, the primary advantage of planners such as TLPlan and TALplanner is their sophisticated knowledge representation and reasoning capabilities. They can represent and solve a variety of nonclassical planning problems, and with a good set of control rules they can solve classical planning problems orders of magnitude more quickly than classical or neoclassical planners. The primary disadvantage of TLPlan and TALplanner is the need for the domain author to write not only a set of planning operators but also a set of control rules.

10.8 Exercises

10.1 Write pseudocode for computing progr(ϕ, S), and analyze its worst-case time complexity.

10.2 Consider the painting problem described in Exercise 5.6.

(a) Suppose we run STL-plan on this problem without any control rules. How many nodes will there be at depth 3 of the search space?

(b) Write a control rule to prevent brushes from being dipped first into one can and then into the other.

(c) Suppose we run STL-plan on this problem with your control rule. How many nodes will there be at depth 3 of the search space?

10.3 Consider the washing problem described in Exercise 5.7.

(a) Suppose we run STL-plan on this problem without any control rules. How many nodes will there be at depth 3 of the search space?

(b) Write control rules to ensure that once we start filling an object x, we will not try to fill any other object y until we are finished filling x.

(c) Will your control rule make any difference in what set of states STL-plan visits? Why or why not?

10.4 Write the rest of the rules needed for the pruning function in Example 10.1 (see page 217).

10.5 Encode the second rule in Example 10.1 as a set of STL formulas similar to those in Example 10.3 (see page 219).

10.6 The first rule in Example 10.1 (see page 217) is not strong enough to express the idea that if c's position is consistent with g, then c should not be moved at all. Write a set of STL formulas to represent this.

10.7 For the variable-interchange problem described in Exercise 4.1, do the following.

(a) Write a control rule saying that STL-plan should never assign a value into v1 until it has copied the value of v1 into some other variable.

(b) Generalize the rule so that it applies to both v1 and v2.

(c) Trace the execution of STL-plan on the variable-interchange problem using the control rule you wrote in part (b), going along the shortest path to a goal state. For each node you visit, write the associated control rule.

(d) With the control rule in part (b), are there any paths in STL-plan's search space that do not lead to goal nodes? If so, write a control rule that will be sufficient to prune those paths without pruning the path you found in part (c).

10.8 Give an example of a situation in which both Proposition 10.2 (see page 221) and its converse are true.

10.9 Give an example of a situation in which the converse of Proposition 10.2 (see page 221) is false.

10.10 Prove Proposition 10.1 (see page 221).

10.11 Prove Proposition 10.2 (see page 221).

10.12 Prove Proposition 10.3 (see page 222).

10.13 Let ϕ be the control formula in Example 10.4 (see page 221). Prove that for any container-stacking problem, there is at least one solution π such that $(\hat{S}_\pi, s_0, g) \models \phi$.

10.14 Professor Prune says, "Under the conditions of Proposition 10.10, STL-plan(O, s_0, g, ϕ) will return a solution π such that $(\hat{S}_\pi, s_0, g) \models \phi$." Prove that Professor Prune is wrong.

10.15 Download a copy of TLPlan from http://www.cs.toronto.edu/~fbacchus/tlplan.html.

(a) Write several planning problems for TLPlan in the blocks world (see Section 4.6), including the one in Exercise 2.1, and run TLPlan on them using the blocks-world domain description that comes with it.

(b) Rewrite TLPlan's blocks-world domain description as a domain description for the container-stacking domain (See Section 4.5). Rewrite your blocks-world planning problems as container-stacking problems, and run TLPlan on them.