CHAPTER 11

Hierarchical Task Network Planning

11.1 Introduction

Hierarchical Task Network (HTN) planning is like classical planning in that each state of the world is represented by a set of atoms, and each action corresponds to a deterministic state transition. However, HTN planners differ from classical planners in what they plan for and how they plan for it.

In an HTN planner, the objective is not to achieve a set of goals but instead to perform some set of *tasks*. The input to the planning system includes a set of operators similar to those of classical planning and also a set of *methods*, each of which is a prescription for how to decompose some task into some set of *subtasks* (smaller tasks). Planning proceeds by decomposing *nonprimitive tasks* recursively into smaller and smaller subtasks, until *primitive tasks* are reached that can be performed directly using the planning operators. An HTN planner will apply an operator only if directed to do so by a method; thus in the terminology of Figure III.1, each method is part of the specification of the Branch function.

HTN planning has been more widely used for practical applications than any of the other planning techniques described in this book. This is partly because HTN methods provide a convenient way to write problem-solving "recipes" that correspond to how a human domain expert might think about solving a planning problem. Here is a simple example that we will refer to several times in this chapter.

Example 11.1 Figure 11.1 shows a DWR problem in which three stacks of containers need to be moved from the piles p1a, p2a, and p3a to the piles p1c, p2c, and p3c in a way that preserves the ordering of the containers, i.e., each container should be on top of the same container that it was on originally. Here is an informal description of some methods for solving this problem.

- To accomplish the task of moving all three of the stacks in a way that preserves the containers' ordering, we can do the following for each stack: Move it to an intermediate pile (which will reverse the order of the containers), then move

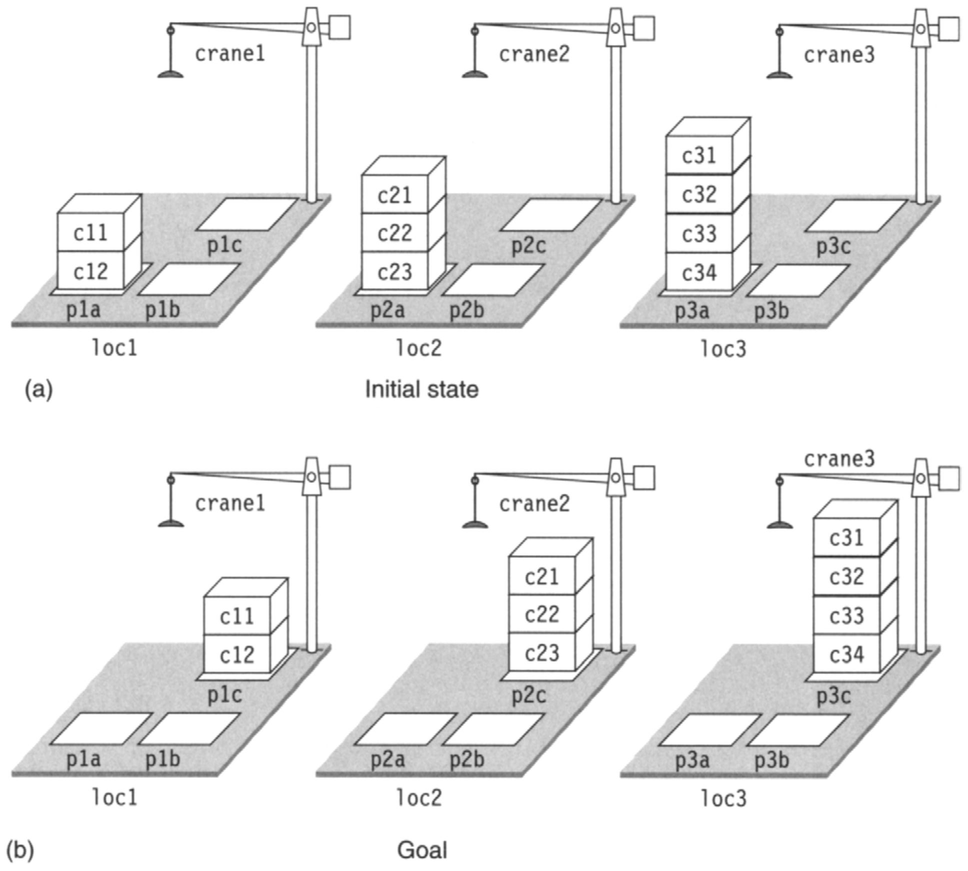

Figure 11.1 A DWR problem in which several stacks of containers need to be moved to different piles.

it to the desired pile (which will put the containers back into their original order).

- To accomplish the task of moving a stack of containers from a pile *p* to another pile *q*, we can do the following until *p* is empty: repeatedly move the topmost container of *p* to *q*.
- To accomplish the task of moving the topmost container of a pile *p* to another pile *q*, we can use the DWR domain's take operator to remove the container from *p* and use the put operator to put it on top of *q*.

An HTN description of the problem would include the DWR operators, the initial state, the task of moving the three stacks in a way that preserves their ordering, and a formal description of the methods just listed. These methods make it much easier to solve the planning problem because they generate only plans that are solutions to the problem. ■

This chapter describes ways to represent HTN planning problems and algorithms for solving the problems. The presentation draws loosely from ideas developed by Sacerdoti [460], Tate [503], Wilkins [549], Erol *et al.* [174], and Nau *et al.* [413, 414].

Sections 11.2 through 11.4 discuss a simplified version of HTN planning that we will call Simple Task Network (STN) planning. Section 11.2 describes the representation scheme, Section 11.3 discusses a planning procedure for a case in which the tasks are totally ordered, and Section 11.4 describes a planning procedure for a more general case in which the tasks may be partially ordered (e.g., to specify that we don't care about the order in which the stacks are moved). Section 11.5 discusses HTN planning, in full generality, including its representation scheme and planning procedures. Section 11.6 discusses the expressivity and complexity of HTN planning; it can express planning problems that cannot be expressed as classical planning problems. Section 11.7 discusses how to extend HTN planning to include extensions like the ones in Section 2.4, and Section 11.8 covers extended goals. Section 11.9 contains discussion and historical remarks.

11.2 STN Planning

This section describes a language for a simplified version of HTN planning called Simple Task Network (STN) planning. Our definitions of terms, literals, operators, actions, and plans are the same as in classical planning. The definition of $\gamma(s, a)$, the result of applying an action a to a state s, is also the same as in classical planning. However, the language also includes tasks, methods, and task networks, which are used in the definitions of a planning problem and its solutions.

11.2.1 Tasks and Methods

One new kind of symbol is a *task symbol*. Every operator symbol is a task symbol, and there are some additional task symbols called *nonprimitive task symbols*. A *task* is an expression of the form $t(r_1, \ldots, r_k)$ such that t is a task symbol, and $r_1, \ldots, r_k$ are terms. If t is an operator symbol, then the task is *primitive*; otherwise, the task is *nonprimitive*. The task is *ground* if all of the terms are ground; otherwise, it is *unground*. An action $a = (\text{name}(a), \text{precond}(a), \text{effects}(a))$ *accomplishes* a ground primitive task t in a state s if $\text{name}(a) = t$ and a is applicable to s.

Definition 11.1 A *simple task network* (or, for short, a *task network*) is an acyclic digraph $w = (U, E)$ in which U is the node set, E is the edge set, and each node $u \in U$ contains a task t_u. w is *ground* if all of the tasks $\{t_u \mid u \in U\}$ are ground; otherwise, w is *unground*. w is *primitive* if all of the tasks $\{t_u \mid u \in U\}$ are primitive; otherwise, w is *nonprimitive*. ■

The edges of w define a partial ordering of U, namely, $u \prec v$ iff there is a path from u to v. If the partial ordering is total, then we say that w is *totally ordered.* In this case, we will sometimes dispense with the graph notation for w and instead write w as the sequence of tasks, namely, $w = \langle t_1, \ldots, t_k \rangle$, where t_1 is the task in the first node of U, t_2 is the task in the second node of U, and so forth. Note that if w is ground, primitive, and totally ordered, and if there are actions $a_1, \ldots, a_k$ whose names are $t_1, \ldots, t_k$, then w corresponds to the plan $\pi_w = \langle a_1, \ldots, a_k \rangle$.

Example 11.2 In the DWR domain, let $t_1 = \mathsf{take(crane2, loc1, c1, c2, p1)}$ and $t_2 = \mathsf{put(crane2, loc2, c3, c4, p2)}$. Let $t_3 = \mathsf{move\text{-}stack}(\mathsf{p1}, q)$, where move-stack is a nonprimitive task symbol. Then t_1 and t_2 are primitive tasks, and t_3 is a nonprimitive task. Let u_1, u_2, and u_3 be nodes such that $t_{u_i} = t_i\ \forall i$. Let $w_1 = (\{u_1, u_2, u_3\}, \{(u_1, u_2), (u_2, u_3)\})$ and $w_2 = (\{u_1, u_2\}, \{(u_1, u_2)\})$. Then w_1 and w_2 are task networks. Since w_2 is totally ordered, we would usually write it as $w_2 = \langle t_1, t_2 \rangle$. Since w_2 also is ground and primitive, it corresponds to the plan $\langle a_1, a_2 \rangle$, where a_1 and a_2 are the actions whose names are t_1 and t_2, respectively. Since we normally use the names of actions to refer to the actions themselves, we usually would say that this is the plan $\langle \mathsf{take(crane2, loc1, c1, c2, p1), put(crane2, loc2, c3, c4, p2)} \rangle$. ■

We also will include in the planning language a new set of symbols called *method symbols.*

Definition 11.2 An *STN method*[1] is a 4-tuple

$$m = (\text{name}(m), \text{task}(m), \text{precond}(m), \text{network}(m))$$

in which the elements are described as follows.

- name(m), the *name* of the method, is a syntactic expression of the form $n(x_1, \ldots, x_k)$, where n is a unique method symbol (i.e., no two methods have the same value for n), and $x_1, \ldots, x_k$ are all of the variable symbols that occur anywhere in m.
- task(m) is a nonprimitive task.
- precond(m) is a set of literals called the method's *preconditions.*
- network(m) is a task network whose tasks are called the *subtasks* of m. ■

In Definition 11.2, name(m) has the same purpose as the name of a classical planning operator: it lets us refer unambiguously to substitution instances of the method, without having to write the preconditions and effects explicitly. task(m) tells what kind of task m can be applied to, precond(m) specifies what conditions the current state must satisfy in order for m to be applied, and network(m) specifies the subtasks to accomplish in order to accomplish task(m).

1. As usual, we will abbreviate this and other terms by omitting adjectives such as STN and HTN when we can do so unambiguously.

A method m is *totally ordered* if network(m) is totally ordered. In this case, rather than specifying the digraph network(m) explicitly, it is simpler to specify subtasks(m), the sequence of subtasks in network(m). For example, if we write

$$\text{subtasks}(m) = \langle t_1, \ldots, t_k \rangle,$$

then this means that

$$\text{network}(m) = (\{u_1, \ldots, u_k\}, \{(u_1, u_2), (u_2, u_3), \ldots, (u_{k-1}, u_k)\})$$

where each u_i is a node such that $t_{u_i} = t_i$.

Rather than writing methods as 4-tuples, we usually will write them in the format shown in the following example.

Example 11.3 Here are formal descriptions of the methods mentioned in Example 11.1 (see page 229). We need two methods for the task of moving a stack of containers: one for the case where the stack is nonempty, and one for the case where it is empty.

The subtasks of the first three methods are totally ordered, so for each of them we specify subtasks(m) rather than network(m). In the last method, the subtasks are partially ordered, so we need to specify its task network explicitly. We do this by naming each of the nodes and using those names to specify the edges.

```
take-and-put($c, k, l_1, l_2, p_1, p_2, x_1, x_2$):
    task:      move-topmost-container($p_1, p_2$)
    precond:   top($c, p_1$), on($c, x_1$),        ; true if $p_1$ is not empty
               attached($p_1, l_1$), belong($k, l_1$),   ; bind $l_1$ and $k$
               attached($p_2, l_2$), top($x_2, p_2$)     ; bind $l_2$ and $x_2$
    subtasks:  ⟨take($k, l_1, c, x_1, p_1$), put($k, l_2, c, x_2, p_2$)⟩

recursive-move($p, q, c, x$):
    task:      move-stack($p, q$)
    precond:   top($c, p$), on($c, x$)    ; true if $p$ is not empty
    subtasks:  ⟨move-topmost-container($p, q$), move-stack($p, q$)⟩
               ;; the second subtask recursively moves the rest of the stack

do-nothing($p, q$)
    task:      move-stack($p, q$)
    precond:   top(pallet, $p$)    ; true if $p$ is empty
    subtasks:  ⟨⟩   ; no subtasks because we are done

move-each-twice()
    task:      move-all-stacks()
    precond:   ; no preconditions
    network:   ; move each stack twice:
               $u_1$ = move-stack(p1a,p1b), $u_2$ = move-stack(p1b,p1c),
               $u_3$ = move-stack(p2a,p2b), $u_4$ = move-stack(p2b,p2c),
```

$u_5 =$ move-stack(p3a,p3b), $u_6 =$ move-stack(p3b,p3c),
$\{(u_1, u_2), (u_3, u_4), (u_5, u_6)\}$ ∎

Definition 11.3 A method instance m *is applicable* in a state s if $\text{precond}^+(m) \subseteq s$ and $\text{precond}^-(m) \cap s = \emptyset$. ∎

For planning, we will be interested in finding method instances that are both applicable in the current state and relevant for some task we are trying to accomplish. We now formalize what we mean by relevance.

Definition 11.4 Let t be a task and m be a method instance (which may be either ground or unground). If there is a substitution σ such that $\sigma(t) = \text{task}(m)$, then m is *relevant* for t, and the *decomposition* of t by m under σ is $\delta(t, m, \sigma) = \text{network}(m)$. If m is totally ordered, we may equivalently write $\delta(t, m, \sigma) = \text{subtasks}(m)$. ∎

Example 11.4 Continuing Example 11.3, let t be the nonprimitive task move-stack (p1a, q), s be the state of the world shown in Figure 11.1(a), and m be the method instance recursive-move(p1a, p1b, c11, c12). Then m is applicable in s, is relevant for t under the substitution $\sigma = \{q \leftarrow \text{p1b}\}$, and decomposes t into:

$$\delta(t, m, \sigma) = \langle \text{move-topmost-container(p1a,p1b)move-stack(p1a,p1b)} \rangle$$

As shown in Figure 11.2, we can draw the decomposition as an AND-branch.[2] The next section discusses how to combine such branches into tree structures that represent derivations of plans. ∎

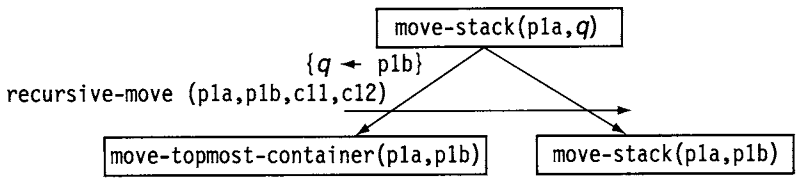
move-stack(p1a,q)
{q ← p1b}
recursive-move (p1a,p1b,c11,c12)
move-topmost-container(p1a,p1b)
move-stack(p1a,p1b)

Figure 11.2 An AND-branch depicting the decomposition of the task move-stack(p1a,q) using the method instance recursive-move(p1a,p1b,c11,c12). The branch is labeled with the substitution and the name of the method instance. The rightward-pointing arrow represents the ordering of the subtasks.

2. For background material on AND/OR trees, see Appendix A.

When we use a method m to decompose a task t, usually this task will be part of a node u in a task network w, in which case we will get a new task network $\delta(w, u, m, \sigma)$ as defined in Definition 11.5. The formal definition of $\delta(w, u, m, \sigma)$ is somewhat complicated, but the intuitive idea is simple: u is removed from w, a copy of subtasks(m) is inserted into w in place of u, and every ordering constraint that formerly applied to u now applies to every node in the copy of subtasks(m).[3]

The complication arises from the need to *enforce* precond(m) in the new task network, i.e., the need to ensure that even after subsequent task decompositions, precond(m) still remains true at the correct place in the network. The point where precond(m) is supposed to be true is just before the first task in subtasks(m), and we need to insert additional ordering constraints into the task network to ensure that this will happen (see Example 11.17 later in this chapter). Because subtasks(m) is partially ordered, there may be more than one possible candidate for the first task in subtasks(m). Thus $\delta(w, u, m, \sigma)$ needs to be a set of alternative task networks: one for each possible candidate for the first task in subtasks(m).

Definition 11.5 Let $w = (U, E)$ be a task network, u be a node in w that has no predecessors in w, and m be a method that is relevant for t_u under some substitution σ. Let $succ(u)$ be the set of all immediate successors of u, i.e., $succ(u) = \{u' \in U \mid (u, u') \in E\}$. Let $succ_1(u)$ be the set of all immediate successors of u for which u is the *only* predecessor. Let (U', E') be the result of removing u and all edges that contain u. Let (U_m, E_m) be a copy of network(m). If (U_m, E_m) is nonempty, then the result of decomposing u in w by m under σ is this set of task networks:

$$\delta(w, u, m, \sigma) = \{(\sigma(U' \cup U_m), \sigma(E_v)) \mid v \in \text{subtasks}(m)\}$$

where

$$E_v = E_m \cup (U_m \times succ(u)) \cup \{(v, u') \mid u' \in succ_1(u)\}$$

Otherwise, $\delta(w, u, m, \sigma) = \{(\sigma(U'), \sigma(E'))\}$ ■

11.2.2 Problems and Solutions

We now discuss STN planning domains, planning problems, and solutions.

Definition 11.6 An *STN planning domain* is a pair

$$\mathcal{D} = (O, M), \tag{11.1}$$

3. The reason for using a copy of subtasks(m) rather than subtasks(m) itself is basically the same as the reason for repeatedly renaming variables in resolution theorem proving: to avoid potential problems if we want to apply m more than once.

where O is a set of operators and M is a set of methods. $\mathcal{D}$ is a *total-order planning domain* if every $m \in M$ is totally ordered. ∎

Definition 11.7 An *STN planning problem* is a 4-tuple

$$\mathcal{P} = (s_0, w, O, M), \tag{11.2}$$

where s_0 is the initial state, w is a task network called the *initial task network*, and $\mathcal{D} = (O, M)$ is an STN planning domain. $\mathcal{P}$ is a *total-order planning problem* if w and $\mathcal{D}$ are totally ordered. ∎

We now define what it means for a plan $\pi = \langle a_1, \ldots, a_n \rangle$ to be a solution for a planning problem $\mathcal{P} = (s_0, w, O, M)$, or equivalently, what it means for π to *accomplish* w. Intuitively, it means that there is a way to decompose w into π in such a way that π is executable and each decomposition is applicable in the appropriate state of the world. The formal definition is recursive and has three cases.

Definition 11.8 Let $\mathcal{P} = (s_0, w, O, M)$ be a planning problem. Here are the cases in which a plan $\pi = \langle a_1, \ldots, a_n \rangle$ is a *solution* for $\mathcal{P}$.

- **Case 1:** w is empty. Then π is a solution for $\mathcal{P}$ if π is empty (i.e., $n = 0$).
- **Case 2:** There is a primitive task node $u \in w$ that has no predecessors in w. Then π is a solution for $\mathcal{P}$ if a_1 is applicable to t_u in s_0 and the plan $\pi = \langle a_2, \ldots, a_n \rangle$ is a solution for this planning problem:

$$\mathcal{P}' = (\gamma(s_0, a_1), w - \{u\}, O, M)$$

  Intuitively, $\mathcal{P}'$ is the planning problem produced by executing the first action of π and removing the corresponding task node from w.
- **Case 3:** There is a nonprimitive task node $u \in w$ that has no predecessors in w. Suppose there is an instance m of some method in M such that m is relevant for t_u and applicable in s_0. Then π is a solution for $\mathcal{P}$ if there is a task network $w' \in \delta(w, u, m, \sigma)$ such that π is a solution for (s_0, w', O, M).

Note that if w is not totally ordered, then w may contain more than one node that has no predecessors. Thus, cases 2 and 3 are not necessarily mutually exclusive. ∎

If π is a solution for (s_0, w, O, M), then for each task node $u \in U$ there is a *decomposition tree* whose leaf nodes are actions of π. The decomposition tree is composed of AND-branches like the ones in Figure 11.12. If u is primitive, then t_u is the name of an action in π, so the decomposition tree consists just of u. Otherwise, if u decomposes into a task network $\delta(w, u, m, \sigma)$ (see case 3 of Definition 11.8),

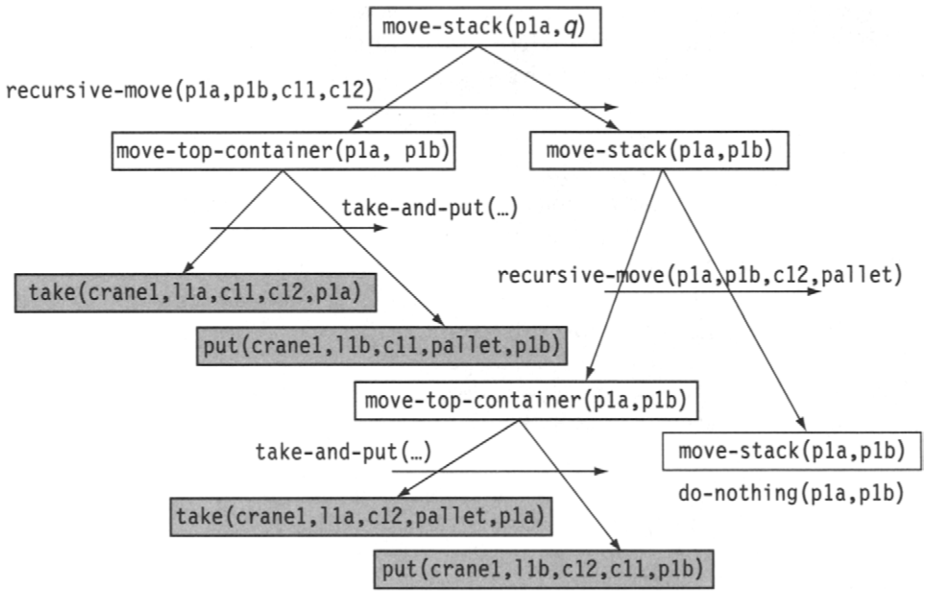
move-stack(p1a,q)
recursive-move(p1a,p1b,c11,c12)
move-top-container(p1a, p1b)
move-stack(p1a,p1b)
take-and-put(...)
take(crane1,l1a,c11,c12,p1a)
put(crane1,l1b,c11,pallet,p1b)
recursive-move(p1a,p1b,c12,pallet)
move-top-container(p1a,p1b)
take-and-put(...)
move-stack(p1a,p1b)
do-nothing(p1a,p1b)
take(crane1,l1a,c12,pallet,p1a)
put(crane1,l1b,c12,c11,p1b)

Figure 11.3 The decomposition tree for the task move-stack(p1a,p1b).The left-to-right arrows below each node indicate the ordering of the children. The plan for the task consists of the shaded nodes. No edges are shown for do-nothing(p1a,p1b) because it has no subtasks.

then the root of the decomposition tree is u, and its subtrees are the decomposition trees for the nodes of $\delta(w, u, m, \sigma)$.

Example 11.5 Let $\mathcal{P} = (s_0, w, O, M)$, where s_0 = the state shown in Figure 11.1 (a), $w = \langle$move-stack(p1a,p1b)$\rangle$, O is the usual set of DWR operators, and M is the set of methods given in Example 11.3. Then there is only one solution for $\mathcal{P}$, namely, the following plan:

$$\pi = \langle \text{take(crane1, l1a, c11, c12, p1a)}, \\ \text{put(crane1, l1b, c11, pallet, p1b)}, \\ \text{take(crane1, l1a, c12, pallet, p1a)}, \\ \text{put(crane1, l1b, c12, c11, p1b)} \rangle$$

The decomposition tree for move-stack(p1a,p1b) is shown in Figure 11.3. Here are the details of how it is produced:

1. Since p1 is nonempty in the initial state, do-nothing is not applicable. However, recursive-move is applicable, producing the subtasks move-topmost-container(p1a,p1b) and move-stack(p1a,p1b).

2. take-and-put is applicable to move-topmost-container(p1a,p1b), producing the subtasks take(crane1,l1a,c11,c12,p1a) and put(crane1,l1b,c11,pallet,p1b). These are primitive tasks and can be accomplished using the corresponding DWR operators (see Example 2.8).
3. Once the above two tasks have been completed, the next task is the move-stack(p1a,p1b) task produced in step 1. As before, recursive-move is applicable, producing the subtasks move-topmost-container(p1a,p1b) and move-stack(p1a,p1b).
4. take-and-put is applicable to move-topmost-container(p1a,p1b), producing the subtasks take(crane1,l1a,c12,pallet,p1a) and put(crane1,l1b,c12,c11,p1b). As before, these are primitive tasks that can be accomplished using the corresponding DWR operators.
5. The last remaining task is the task move-stack(p1a,p1b) produced in step 3. This time, do-nothing is applicable, and it produces no subtasks. ■

11.3 Total-Order STN Planning

Figure 11.4 shows the TFD (Total-order Forward Decomposition) procedure for solving total-order STN problems. The procedure is based directly on the definition of a solution for an STN planning problem; this makes it easy to show that it is sound and complete for all total-order STN planning problems (Exercises 11.6 and 11.7).

Example 11.6 All of the methods in Example 11.3 (see page 233) are totally ordered except for move-each-twice. If we omit this method, we have a total-order STN domain. In this planning domain, suppose we invoke TFD on the planning problem described in Example 11.13. Then one of its execution traces will carry out the steps described in that example, producing the decomposition tree shown in Figure 11.3. ■

Example 11.7 In order to use TFD to solve the planning problem shown in Figure 11.1, we need a totally ordered version of the move-each-twice method. We can write one by specifying an order in which to move the three stacks of containers.

```
move-each-twice()
    task:      move-all-stacks()
    precond: ; no preconditions
    subtasks: ⟨move-stack(p1a,p1b), move-stack(p1b,p1c),
               move-stack(p2a,p2b), move-stack(p2b,p2c),
               move-stack(p3a,p3b), move-stack(p3b,p3c)⟩
```
■

```
TFD($s, \langle t_1, \ldots, t_k \rangle, O, M$)
    if $k = 0$ then return $\langle\rangle$ (i.e., the empty plan)
    if $t_1$ is primitive then
        active $\leftarrow \{(a, \sigma) \mid a$ is a ground instance of an operator in $O$,
                    $\sigma$ is a substitution such that $a$ is relevant for $\sigma(t_1)$,
                    and $a$ is applicable to $s\}$
        if active $= \emptyset$ then return failure
        nondeterministically choose any $(a, \sigma) \in$ active
        $\pi \leftarrow$ TFD($\gamma(s, a), \sigma(\langle t_2, \ldots, t_k \rangle), O, M$)
        if $\pi$ = failure then return failure
        else return $a.\pi$
    else if $t_1$ is nonprimitive then
        active $\leftarrow \{m \mid m$ is a ground instance of a method in $M$,
                    $\sigma$ is a substitution such that $m$ is relevant for $\sigma(t_1)$,
                    and $m$ is applicable to $s\}$
        if active $= \emptyset$ then return failure
        nondeterministically choose any $(m, \sigma) \in$ active
        $w \leftarrow$ subtasks$(m).\sigma(\langle t_2, \ldots, t_k \rangle)$
        return TFD($s, w, O, M$)
```

Figure 11.4 The TFD procedure for total-order STN planning.

Here are some comparisons between TFD and the Forward-search and Backward-search procedures described in Chapter 4.

- Like Forward-search, TFD considers only actions whose preconditions are satisfied in the current state. However, like Backward-search, it considers only operators that are relevant for what it is trying to achieve. This combination can greatly increase the efficiency of the search.
- Like Forward-search, TFD generates actions in the same order in which they will be executed. Thus, each time it plans how to accomplish a task, it has already planned everything that comes before that task, so it knows the current state of the world. Section 11.7 discusses some ways to take advantage of this property.
- Just as Backward-search can create unnecessarily many ground instances of operators, TFD can create unnecessarily many ground instances of methods. Just as before, the cure is to "lift" the procedure, i.e., to modify it so that it only partially instantiates the methods. The resulting Lifted-TFD procedure (see Exercises 11.9 and 11.10) has some properties analogous to those of Lifted-backward-search: it works correctly even when the initial task list is not ground, and it has a smaller branching factor than TFD because it does not instantiate variables except when necessary.

11.4 Partial-Order STN Planning

In Example 11.6, it was easy to rewrite the planning domain of Example 11.3 (see page 233) as a total-order STN domain. However, not all planning domains can be rewritten so easily.

Example 11.8 Consider a DWR problem where we start with the initial state shown in Figure 11.5, and we want to use r1 to move the two containers to location loc2. It is easy (see Exercise 11.3) to write totally ordered methods that produce a decomposition tree like the one in Figure 11.6.

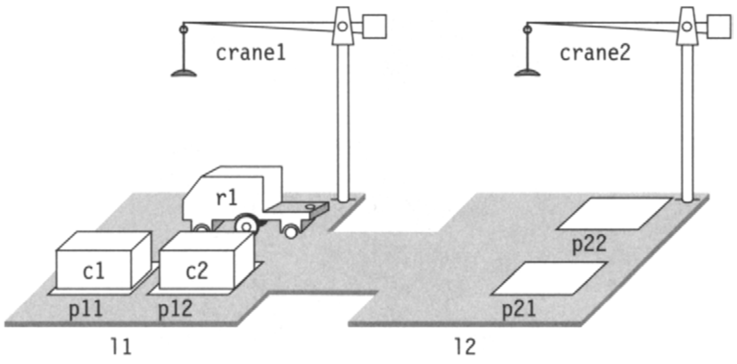

Figure 11.5 Initial state for a DWR problem in which two containers need to be moved from one location to another.

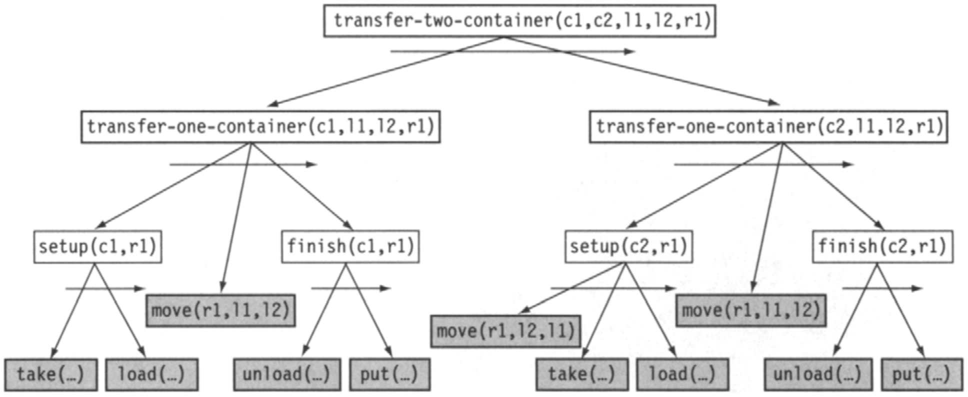

Figure 11.6 Decomposition tree for the DWR problem of Figure 11.5. To keep the figure simple, the labels on the branches have been omitted.

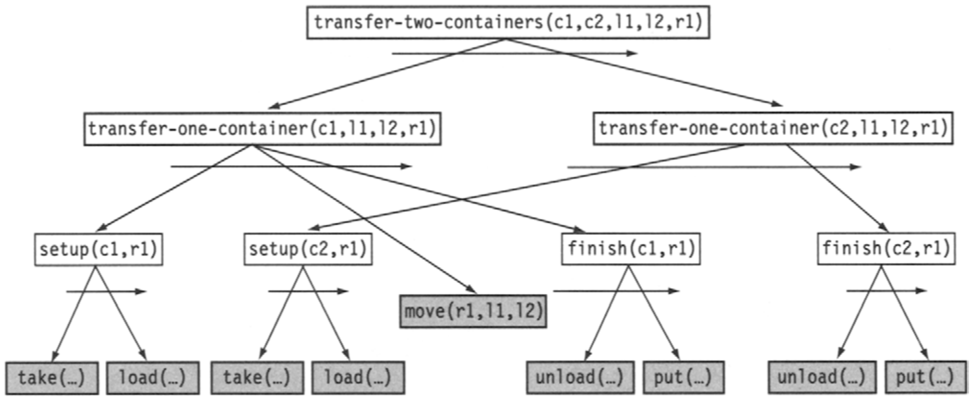

Figure 11.7 An interleaved decomposition tree for the case where r1 can carry more than one container at once. The subtasks of the root are unordered, and their subtasks are interleaved. Decomposition trees like this one can occur in a partial-order STN domain but not in a total-order one.

Next, consider a modified version of the problem in which r1 can carry many containers at once. In this case, we would probably prefer to get a solution tree like the one in Figure 11.7, in which the plans for the two containers are interleaved so that the robot can move both containers at once. Here are some methods for doing that.

```
transfer2($c_1, c_2, l_1, l_2, r$)   ; method to transfer $c_1$ and $c_2$
  task:     transfer-two-containers($c_1, c_2, l_1, l_2, r$)
  precond:  ; none
  subtasks: ⟨transfer-one-container($c_1, l_1, l_2, r$),
             transfer-one-container($c_2, l_1, l_2, r$)⟩

transfer1($c, l_1, l_2, r$)   ; method to transfer c
  task:     transfer-one-container($c, l_1, l_2, r$)
  precond:  ; none
  network:  $u_1$ = setup($c, r$), $u_2$ = move-robot($r, l_1, l_2$),
            $u_3$ = finish(c,r), $\{(u_1, u_2), (u_2, u_3)\}$

move1($r, l_1, l_2$)   ; method to move r if it is not at $l_2$
  task:     move-robot($l_1, l_2$)
  precond:  at($r, l_1$)
  subtasks: ⟨move($r, l_1, l_2$)⟩

move0($r, l_1, l_2$)   ; method to do nothing if r is already at $l_2$
  task:     $u_0$ = move-robot($l_1, l_2$)
  precond:  at($r, l_2$)
  subtasks: ⟨⟩   ; i.e., no subtasks
```

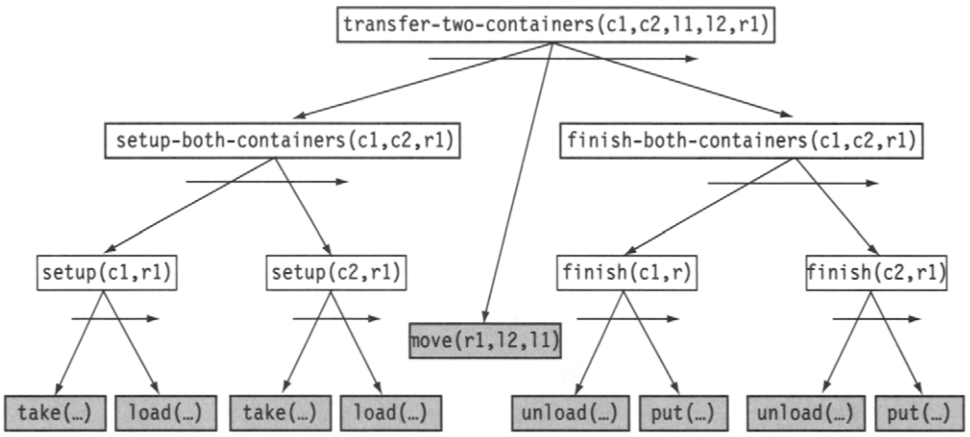

Figure 11.8 A noninterleaved decomposition tree for the case where r1 can carry two containers side by side.

do-setup(c, d, k, l, p, r) *; method to prepare for moving a container*
task: setup(c, r)
precond: on(c, d), in(c, p), belong(k, l), *;attached(p,l), at (r,l)*
network: u_1 = take(k, l, c, d, p), u_2 = load(k, l, c, r), $\{(u_1, u_2)\}$

unload-robot(c, d, k, l, p, r) *; method to finish after moving a container*
task: finish(c, r)
precond: attached(p, l), loaded(r,c), top(d, p), belong(k, l), at (r,l)
network: u_1 = unload(k, l, c, r), u_2 = put(k, l, c, d, p), $\{(u_1, u_2)\}$

Not all of these methods are totally ordered. Furthermore, the decomposition tree in Figure 11.7 cannot be produced by a set of totally ordered methods because the total-ordering constraints would prevent subtasks of different tasks from being interleaved. With totally ordered methods the best one could do would be to write methods that generate a noninterleaved decomposition tree such as the one in Figure 11.8 (see Exercise 11.4). ■

Example 11.9 Here is an example to illustrate the need for the set of edges $\{(v, u') \mid u' \in succ_1(u)\}$ in Definition 11.5 (see page 235).

In the decomposition tree shown in Figure 11.7, consider how to use the transfer1 method from the previous example to decompose the task transfer-one-container(c2,l1,l2,r1) into the subtasks setup(c2,r1) and finish(c2,r1). The transfer1 method has no preconditions—but if it did, they would need to be evaluated in the same state of the world as the preconditions of the first action in the subtree rooted at transfer1(c2). The additional edges inserted by δ provide ordering constraints to guarantee that this will occur. ■

```
PFD($s, w, O, M$)
    if $w = \emptyset$ then return the empty plan
    nondeterministically choose any $u \in w$ that has no predecessors in $w$
    if $t_u$ is a primitive task then
        active ← $\{(a,\sigma) \mid a$ is a ground instance of an operator in $O$,
                      $\sigma$ is a substitution such that name$(a) = \sigma(t_u)$,
                      and $a$ is applicable to $s\}$
        if active $= \emptyset$ then return failure
        nondeterministically choose any $(a,\sigma) \in$ active
        $\pi \leftarrow$ PFD($\gamma(s,a), \sigma(w - \{u\}), O, M$)
        if $\pi$ = failure then return failure
        else return $a.\pi$
    else
        active ← $\{(m,\sigma) \mid m$ is a ground instance of a method in $M$,
                    $\sigma$ is a substitution such that name$(m) = \sigma(t_u)$,
                    and $m$ is applicable to $s\}$
        if active $= \emptyset$ then return failure
        nondeterministically choose any $(m,\sigma) \in$ active
        nondeterministically choose any task network $w' \in \delta(w, u, m, \sigma)$
        return(PFD($s, w', O, M$)
```

Figure 11.9 The PFD procedure for STN planning.

Figure 11.9 shows the PFD (Partial-order Forward Decomposition) procedure. PFD is a generalization of the TFD procedure so that STN planning problems do not have to be total-order problems. PFD is a direct implementation of the definition of a solution to an STN planning problem, so it is not hard to show that it is sound and complete.

Like TFD, PFD can be lifted to produce a Lifted-PFD procedure (see Exercise 11.18) that will search a smaller search space.

Example 11.10 Here is how to use the methods listed previously to produce the decompositions shown in Figure 11.7. First, start with the task network $w_1 = (\{u\}, \emptyset)$, where t_u = transfer-two-containers(c1,c2,l1,l2,r1). Next, apply the method transfer2 to u, to produce the task network $w_2 = (\{u_1, u_2\}, \emptyset)$, where t_{u_1} = transfer-one-container(c1,l1,l2,r1) and t_{u_2} = transfer-one-container(c2,l1,l2,r1). Next, apply the method transfer1 to u_1 to produce the task network $w_3 = (\{u_2, u_3, u_4, u_5\}, \{(u_3, u_4), (u_4, u_5)\})$, where:

$$t_{u_3} = \text{setup(c1,r1)}$$
$$t_{u_4} = \text{move(r1,l1,l2)}$$
$$t_{u_5} = \text{finish(c1,r1)}$$

These are the first two decompositions. The rest of the decomposition tree can be produced in a similar manner. ■

11.5 HTN Planning

In STN planning, we associated two kinds of constraints with a method: ordering constraints and preconditions. We represented ordering constraints explicitly in the task network as edges of a digraph. We did not keep track of these preconditions explicitly in the task network: instead, we *enforced* the preconditions by constructing task networks that were guaranteed to satisfy them. This was the purpose of the set of edges $\{(v, u') \mid u' \in succ_1(u)\}$ in Definition 11.5 (see page 235).

Probably the only kind of planning procedure that is feasible for use with Definition 11.5 is a forward-decomposition procedure such as TFD or PFD. However, there are cases where one may not want to use a forward-decomposition procedure. HTN planning is a generalization of STN planning that gives the planning procedure more freedom about how to construct the task networks.

In order to provide this freedom, a bookkeeping mechanism is needed to represent constraints that the planning algorithm has not yet enforced. The bookkeeping is done by representing the unenforced constraints explicitly in the task network.

11.5.1 Task Networks

The HTN-planning definition of a task network generalizes the definition used in STN planning.

Definition 11.9 A *task network* is a pair $w = (U, C)$, where U is a set of task nodes and C is a set of constraints as described in the text.[4] ■

Each constraint in C specifies a requirement that must be satisfied by every plan that is a solution to a planning problem. Here is some notation that will be useful in describing the constraints. Let π be a solution for w, $U' \subseteq U$ be a set of task nodes in w, and A be the set of all actions $a_i \in \pi$ such that in π's decomposition tree, a_i is a descendant of a node in U'. Then first(U', π) is the action $a_i \in A$ that occurs first, i.e., $i \leq j$ for every $a_j \in A$; and last(U', π) is the action $a_k \in A$ that occurs last, i.e., $k \geq j$ for every $a_j \in A$.

4. It is straightforward to extend this definition to allow C to be a Boolean combination of constraints, but we have restricted it to be a conjunct in order to simplify the presentation.

Here are the kinds of constraints we will consider.[5]

- A *precedence constraint* is an expression of the form $u \prec v$, where u and v are task nodes. Its meaning is identical to the meaning of the edge (u, v) in STN planning: it says that in every solution π for $\mathcal{P}$, the action last$(\{u\}, \pi)$ must precede the action first$(\{v\}, \pi)$. For example, if $t_u =$ move(r2,l2,l3) and $t_v =$ move(r1,l1,l2), then the constraint $u \prec v$ says that r2 must be moved from l2 to l3 before r1 is moved from l1 to l2.
- A *before-constraint* is a generalization of the notion of a precondition in STN planning. It is a constraint of the form before(U', l), where $U' \subseteq U$ is a set of task nodes and l is a literal. It says that in any solution π for $\mathcal{P}$, the literal l must be true in the state that occurs just before first(U', π). For example, suppose u is a task node for which $t_u =$ move(r2,l2,l3). Then the constraint

$$\text{before}(\{u\}, \text{at}(\text{r2}, \text{l2}))$$

  says that r2 must be at l2 just before we move it from l2 to l3.
- An *after-constraint* has the form after(U', l). It is like a before-constraint except that it says l must be true in the state that occurs just after last(U', π).
- A *between-constraint* has the form between(U', U'', l). It says that the literal l must be true in the state just after last(U', π), the state just before first(U'', π), and all of the states in between.

11.5.2 HTN Methods

The definition of a method in HTN planning generalizes the definition used in STN planning.

Definition 11.10 An *HTN method* is a 4-tuple

$$m = (\text{name}(m), \text{task}(m), \text{subtasks}(m), \text{constr}(m))$$

in which the elements are described as follows.

- name(m) is an expression of the form $n(x_1, \ldots, x_k)$, where n is a unique method symbol (i.e., no two methods in the planning domain can have the same method symbol), and $x_1, \ldots, x_k$ are all of the variable symbols that occur anywhere in m.

5. These are merely suggestive of the kinds of constraints that an HTN planner may handle. Some planners do not handle all of the constraints listed here. Others handle various generalizations of the constraints, such as before(first(U'), l) or before(last(U'), l) to say that l must hold before the first or last action in A, or they allow additional kinds of constraints, such as the binding constraints described in Chapter 5.

- task(m) is a nonprimitive task.
- (subtasks(m), constr(m)) is a task network.

■

Suppose that $w = (U, C)$ is a task network, $u \in U$ is a task node, t_u is its task, m is an instance of a method in M, and task(m) $= t_u$. Then m *decomposes* u into subtasks(m'), producing the task network

$$\delta(w, u, m) = ((U - \{u\}) \cup \text{subtasks}(m'), C' \cup \text{constr}(m')),$$

where C' is the following modified version of C.

- For every precedence constraint that contains u, replace it with precedence constraints containing the nodes of subtasks(m'). For example, if subtasks(m') $= \{u_1, u_2\}$, then we would replace the constraint $u \prec v$ with the constraints $u_1 \prec v$ and $u_2 \prec v$.
- For every before-, after-, or between-constraint in which there is a set of task nodes U' that contains u, replace U' with $(U' - \{u\}) \cup$ subtasks(m'). For example, if subtasks(m') $= \{u_1, u_2\}$, then we would replace the constraint before($\{u, v\}, l$) with the constraint before($\{u_1, u_2, v\}, l$).

Example 11.11 Here is a rewrite of the STN methods of Example 11.8 (see page 240) as HTN methods. In each method's task and subtask lists, each u_i is a label for a task; it is used to refer to the task in the method's constraint list.

```
transfer2(c1, c2, l1, l2, r)   ;; method to move c1 and c2 from pile p1 to pile p2
task:        transfer-two-containers(c1, c2, l1, l2, r)
subtasks:    u1 = transfer-one-container(c1, l1, l2, r),
             u2 = transfer-one-container(c2, l1, l2, r)
constr:      u1 ≺ u2

transfer1(c, l1, l2, R)   ;; method to transfer c
task:        transfer-one-container(c, l1, l2, r)
subtasks:    u1 = setup(c, r), u2 = move-robot(r, l1, l2), u3 = finish(c, r)
constr:      u1 ≺ u2, u2 ≺ u3

move1(r, l1, l2)   ;; method to move r if it is not at l2
task:        move-robot(l1, l2)
subtasks:    u1 = move(r, l1, l2)
constr:      before({u1}, at(r, l1))

move0(r, l1, l2)   ;; method to do nothing if r is already at l2
task:        u0 = move-robot(l1, l2)
subtasks:    ;; no subtasks
constr:      before({u0}, at(r1, l2))
```

do-setup(c, d, k, l, p, r) *;; method to prepare for moving a container*
task: setup(c, r)
subtasks: $u_1 =$ take(k, l, c, d, p), $u_2 =$ load(k, l, c, r)
constr: $u_1 \prec u_2$, before$(\{u_1\}$, on$(c, d))$, before$(\{u_1\}$, attached$(p, l))$,
before$(\{u_1\}$, in$(c, p))$, before$(\{u_1\}$, belong(k, l), before$(\{u_1\})$, at$(r, l))$

unload-robot(c, d, k, l, p, r) *;; method to finish after moving a container*
task: finish(c, r)
subtasks: $u_1 =$ unload(k, l, c, r), $u_2 =$ put(k, l, c, d, p)
constr: $u_1 \prec u_2$, before$(\{u_1\}$, attached$(p, l))$, before$(\{u_1\}$, loaded$(r, c))$,
before $(\{u_1\}$, top$(d, p))$ before$(\{u_1\})$, belong(k, l),
before$(\{u_1\}$, at$(r, l))$

Here is how to use the above methods to produce the decompositions shown in Figure 11.7. First, start with a task network $w_1 = (\{u\}, \emptyset)$, where u is a node such that $t_u =$ transfer-two-containers(c1,c2,l1,l2,r1). Next, apply the method transfer2 to u to produce a task network $w_2 = (\{u_1, u_2\}, \emptyset)$ such that $t_{u_1} =$ transfer-one-container(c1,l1,l2,r1) and $t_{u_2} =$ transfer-one-container(c2). Next, apply the instance transfer1 to u_1 to produce a task network $w_3 = (\{u_2, u_3, u_4, u_5\}, C_3)$ such that:

$$t_{u_3} = \text{setup(c1)}$$
$$t_{u_4} = \text{get-r1-to-p2()}$$
$$t_{u_5} = \text{finish(c1)}$$
$$C_3 = \{u_3 \prec u_4,\ u_4 \prec u_5\}$$

These are the first two decompositions. The rest of the decomposition tree can be produced in a similar manner. ■

11.5.3 HTN Problems and Solutions

HTN planning domains are identical to STN planning domains except that they use HTN methods.

Definition 11.11 An *HTN planning domain* is a pair

$$\mathcal{D} = (O, M),$$

and an *HTN planning problem* is a 4-tuple

$$\mathcal{P} = (s_0, w, O, M),$$

where s_0 is the initial state, w is the initial task network, O is a set of operators, and M is a set of HTN methods. ■

We now define what it means for a plan π to be a solution for $\mathcal{P}$. There are two cases, depending on whether w is primitive or nonprimitive.

Definition 11.12 If $w = (U, C)$ is primitive, then a plan $\pi = \langle a_1, a_2, \ldots, a_k \rangle$ is a *solution* for $\mathcal{P}$ if there is a ground instance (U', C') of (U, C) and a total ordering $\langle u_1, u_2, \ldots, u_k \rangle$ of the nodes of U' such that all of the following conditions hold.

- The actions in π are the ones named by the nodes $u_1, u_2, \ldots, u_k$, i.e., $\text{name}(a_i) = t_{u_i}$ for $i = 1, \ldots, k$.
- The plan π is executable in the state s_0.
- The total ordering $\langle u_1, u_2, \ldots, u_k \rangle$ satisfies the precedence constraints in C', i.e., C' contains no precedence constraint $u_i \prec u_j$ such that $j \leq i$.
- For every constraint before(U', l) in C', l holds in the state s_{i-1} that immediately precedes the action a_i, where a_i is the action named by the first node of U' (i.e., the node $u_i \in U'$ that comes earliest in the total ordering $\langle u_1, u_2, \ldots, u_k \rangle$).
- For every constraint after(U', l) in C', l holds in the state s_j produced by the action a_j, where a_j is the action named by the last node of U' (i.e., the node $u_j \in U'$ that comes latest in the total ordering $\langle u_1, u_2, \ldots, u_k \rangle$).
- For every constraint between(U', U'', l) in C', l holds in every state that comes between a_i and a_j, where a_i is the action named by the last node of U', and a_j is the action named by the first node of U''.

If $w = (U, C)$ is nonprimitive (i.e., at least one task in U is nonprimitive), then π is a *solution* for $\mathcal{P}$ if there is a sequence of task decompositions that can be applied to w to produce a primitive task network w' such that π is a solution for w'. In this case, the *decomposition tree* for π is the tree structure corresponding to these task decompositions. ■

Example 11.12 Let $\mathcal{P} = (s_0, (\{u\}, \emptyset), O, M)$, where s_0 is the state shown in Figure 11.5, u is a task node such that t_u = transfer-two-containers(c1,c2), O is the usual set of DWR operators, and M is the set of methods given in Example 11.11. Then the solution plan and its decomposition tree are the same as in Example 11.5 (see page 237). ■

11.5.4 Planning Procedures

HTN planning procedures must both instantiate operators and decompose tasks. Because there are several different ways to do both of these things, the the number of different possible HTN planning procedures is quite large. Abstract-HTN, shown in Figure 11.10, is an abstract procedure that includes many (but not necessarily

```
Abstract-HTN($s, U, C, O, M$)
    if $(U, C)$ can be shown to have no solution
        then return failure
    else if $U$ is primitive then
        if $(U, C)$ has a solution then
            nondeterministically let $\pi$ be any such solution
            return $\pi$
        else return failure
    else
        choose a nonprimitive task node $u \in U$
        active $\leftarrow \{m \in M \mid$ task($m$) is unifiable with $t_u\}$
        if active $\neq \emptyset$ then
            nondeterministically choose any $m \in$ active
            $\sigma \leftarrow$ an mgu for $m$ and $t_u$ that renames all variables of $m$
            $(U', C') \leftarrow \delta(\sigma(U, C), \sigma(u), \sigma(m))$
            $(U', C') \leftarrow$ apply-critic($U', C'$) ;; this line is optional
            return Abstract-HTN($s, U', C', O, M$)
        else return failure
```

Figure 11.10 The Abstract-HTN procedure. An mgu is a most-general unifer (see Appendix B).

all) of them. For example, Abstract-HTN is general enough to accommodate HTN versions of both TFD and PFD.

When Abstract-HTN chooses a nonprimitive task $u \in U$, this is not a nondeterministic choice (i.e., it is not a backtracking point) because every task in U must eventually be decomposed before we can find a solution. The size of the decomposition tree will differ depending on the order in which the tasks are decomposed. Because the smallest possible decomposition tree can be exponentially smaller than the biggest one [517], a good HTN planning procedure will try to choose an ordering that generates a small decomposition tree rather than a large one. As an example, the approach used in TFD and PFD is to decompose the tasks in the same order that they will later be executed.

As written, Abstract-HTN is a "lifted" procedure: when it computes *active*, it does not fully instantiate a method unless it absolutely needs to. We could just as easily have written the procedure to use ground instances of the methods, but the lifted version will usually generate a much smaller search space.

The assignment $(U', C') \leftarrow$ apply-critic(U', C') formalizes the notion of a *critic*, which is a function that can make an arbitrary modification to a task network. Critics can be useful for performing application-specific computations that would be difficult to represent within the planning formalism itself—for example, in a manufacturing planning domain, a critic might invoke a computational package that determines an appropriate trajectory for a manufacturing tool. However, critics must be written carefully if one wants to preserve soundness and completeness. Formally, the Abstract-HTN procedure will be sound if every solution to apply-critic(U', C') is also a solution to (U', C'), and it will be complete if apply-critic(U', C') has at least one solution whenever (U', C') has at least one solution.

11.6 Comparisons

We now compare HTN planning to STN planning and compare both to the control-rule techniques described in Chapter 10. The comparisons involve the expressivity of the formalisms and the computational complexity of the problems they can express.

11.6.1 HTN Planning versus STN Planning

STN planning (and thus HTN planning because it is a generalization of STN planning) can be used to encode undecidable problems. In fact, undecidable problems can even be encoded as STN planning problems that contain no variable symbols. We will skip the details, but the proof involves expressing the problem of determining whether two context-free languages have a nonempty intersection—a problem that is known to be undecidable—as an STN planning problem.

From the above, it follows that STN and HTN planning are more expressive than classical planning because the latter cannot express undecidable problems. Furthermore, STN and HTN planning are more expressive than classical planning even if we require that the tasks in every task network be totally ordered. The details of the proof depend on the theory of formal languages [175]. However, the basic idea is that the set of solutions for a classical planning problem is a regular language, the set of solution for a total-order STN planning problem is a context-free language, and there are context-free languages that are not regular languages.

Example 11.13 Here is a total-order STN planning problem in which the set of solutions is a context-free language that is not a regular language. The initial state is empty, the initial task list is ⟨task1()⟩, and the methods and operators are as follows.

```
method1():
    task:      task1()
    precond:   (no preconditions)
    subtasks:  op1(), task1(), op2()

method2():
    task:      task1()
    precond:   (no preconditions)
    subtasks:  (no subtasks)
```

```
op1():
    precond: (no preconditions)
    effects:  (no effects)

op2():
    precond: (no preconditions)
    effects:  (no effects)
```

The solutions to this problem are as follows:

$$\pi_0 = \langle\rangle$$
$$\pi_1 = \langle \texttt{op1()}, \texttt{op2()} \rangle$$
$$\pi_2 = \langle \texttt{op1()}, \texttt{op1()}, \texttt{op2()}, \texttt{op2()} \rangle$$
$$\pi_3 = \langle \texttt{op1()}, \texttt{op1()}, \texttt{op1()}, \texttt{op2()}, \texttt{op2()}, \texttt{op2()} \rangle$$
$$\ldots$$

■

Example 11.13 depends on the ability of methods (such as method1) to recursively invoke each other or themselves an unlimited number of times. It is possible to formulate an "acyclicity" restriction to ensure that there is a finite bound on the maximum number of times such recursions can occur. However, even with this restriction, total-order STN (and thus HTN) planning is more expressive than classical planning. Given an acyclic total-order STN planning problem, it is possible to rewrite it as an equivalent classical planning problem—but in the worst case, the statement of the classical planning problem will be exponentially larger than the statement of the total-order STN planning problem.

If we restrict total-order STN planning even further, by requiring the initial task list and each method's subtask list to be "regular" (i.e., the list can contain at most only one nonprimitive task, and this task must be at the very end of the list), then this makes the expressive power the same as that of classical planning. Table 11.1 gives the complexity of STN planning in this and several other cases.

11.6.2 HTN Methods versus Control Rules

TFD and PFD are similar in several respects to the STL-plan procedure described in Chapter 10. In both cases, planning is done forward from the initial state, and control knowledge is used to restrict which operators to apply. However, there is a difference in how the control knowledge is represented.

Consider a state space that begins with the initial state and contains all possible sequences of operator applications. STL-plan's rules tell it what parts of this state space to avoid; it can explore any part of the search space that avoids the "bad" states and their successors. In contrast, TFD's and PFD's HTN methods tell them which

Table 11.1 Complexity of PLAN-EXISTENCE for HTN planning.

Restrictions on nonprimitive tasks	*Must the HTNs be totally ordered?*	*Are variables allowed?*	
		No	*Yes*
None	No	Undecidable[a]	Undecidable[a,b]
	Yes	In EXPTIME; PSPACE-hard	in DEXPTIME;[d] EXPSPACE-hard
"Regularity" (≤ 1 nonprimitive task, which must follow all primitive tasks)	Does not matter	PSPACE-complete	EXPSPACE-complete[c]
No nonprimitive tasks	No	NP-complete	NP-complete
	Yes	Polynomial time	NP-complete

[a] Decidable if we impose acyclicity restrictions.
[b] Undecidable even when the planning domain is fixed in advance.
[c] In PSPACE when the planning domain is fixed in advance, and PSPACE-complete for some fixed planning domains.
[d] DEXPTIME means double-exponential time.

parts of this state space to explore. They can apply only actions that are reachable using their HTN methods.

Without the extensions discussed in Sections 10.5, control-rule planning is restricted to solving just classical planning problems. In this case, Section 11.6 shows that HTN planning has more expressive power than control-rule planning. However, several of the extensions in Section 10.5 are used in nearly all control-rule planners, and analogous extensions (see Section 11.7) are used in most HTN planners. With these extensions, both formalisms are capable of representing undecidable problems.

It is hard to say which type of control knowledge (methods or control rules) is more effective. Most researchers would probably agree that both types are useful in different situations and that combining them is a useful topic for future research.

11.7 Extensions

Section 2.4 discussed how to augment classical planning languages to incorporate extensions such as axioms, function symbols, and attached procedures. Both these and several other kinds of extensions have been widely used in HTN planning procedures.

11.7.1 Extensions from Chapter 2

It is especially easy to incorporate the extensions into the TFD and PFD procedures. Since these procedures both plan forward from the initial state, they both know the complete current state of the world at each step of the planning process. This makes it easy to do arbitrary computations involving the current state, including complicated numeric computations, axiomatic inference, and calls to domain-specific software packages such as engineering modelers, database systems, and probabilistic reasoners. Here is a discussion of some of the issues involved.

Function Symbols. If we allow the planning language to contain function symbols, then the arguments of an atom or task are no longer restricted to being constant symbols and variable symbols. Instead, they may be arbitrarily complicated terms. This causes a problem in a planning procedure like TFD and PFD that works with ground instances of operators and methods because there may be infinitely many ground method instances or ground operator instances that are applicable to a given state of the world. In order to make those sets finite, the solution is to use the lifted procedures, Lifted-TFD and Lifted-PFD.

In the lifted versions, the current state still will be completely ground; thus using the lifted versions does not make it any more difficult to incorporate the other extensions mentioned earlier.

Axioms. To incorporate axiomatic inference, we will need to use a theorem prover as a subroutine of the planning procedure. The easiest approach is to restrict the axioms to be Horn clauses and use a Horn-clause theorem prover. If an operator, method, or axiom has a positive precondition p, then we can take p to be true iff p can be proved in the current state s. If we also want to allow negative preconditions, then there is a question about 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. We can modify the precondition evaluation algorithm (which, if we have incorporated axiomatic inference, would be the Horn-clause theorem prover) to recognize that certain terms or predicate symbols are to be evaluated by using attached procedures rather than by using the normal theorem-proving mechanism. In the most general case, such an extension would make it impossible to prove soundness and completeness. However, let us consider two of the main situations in which we might want to use attached procedures: (1) to provide a way to do numeric computations, and (2) to allow queries to external information sources. Provided that certain restrictions are satisfied, both of these things are possible to accomplish in a way that is both sound and complete [157].

Example 11.14 If we extend STN planning to incorporate the extensions described previously, then we can write methods, operators, and axioms that encode the container-stacking procedure of Section 4.5.

In the HTN representation of a container-stacking problem (O, s_0, g), we will want the initial task to be the task of achieving a state in which all of the conditions in g are simultaneously true. For this purpose, we will need to represent a list of the conditions in g. We will use a constant symbol nil to represent the empty list and a binary function symbol cons such that cons(a, l) represents the list in which a is the first element and l is a list of the remaining elements.[6] For example, the list ⟨a, b, c⟩ will be represented as cons(a, cons(b, cons(c, nil))). The Horn-clause axioms for list manipulation are as follows.

```
;; x is a member of any list whose head is x
member(x,cons(x, y))

;; if x is a member of v, then x is a member of cons(u, v)
member(x,cons(u, v)) :- member(x, v)

;; g is the result of removing a from the front of the list cons(a, g)
remove(a,cons(a, g),g)

;; if removing a from g leaves h, then removing a from cons(b, g)
   leaves cons(b, h)
remove(a,cons(b, g),cons(b, h)) :- remove(a, g, h)
```

Here are some axioms to describe situations in which containers need to be moved. All of the above are Horn clauses except for the axiom for different.

```
;; x needs to be moved if there's a goal saying it should be elsewhere
need-to-move(x, h) :- on(x, y), member(on(x, z),h), different(y, z)

;; x needs to be moved if something else should be where x is
need-to-move(x, h) :- on(x, z), member(on(y, z),h), different(x, y)

;; x needs to be moved if x is on y and y needs to be moved
need-to-move(x, h) :- on(x, y), need-to-move(y, h)

same(x, x)                         ;; any term is the same as itself
different(x, y) :- ¬same(x, y)     ;; two terms are different if they are not
   the same

;; find a goal on(x, y) that we can achieve immediately
can-achieve(on(x, y),h, r) :- remove(on(x, y),h, r), top(x, p), top(y, q),
                                ¬need-to-move(y, r)

;; return true if we can proceed, false if we need to move a container out of
   the way
can-proceed(h) :- can-achieve(on(x, y),h, r)
```

The initial task list will contain a single task h of the form cons(a_1,cons(a_2,...)), where $a_1, a_2, \ldots$ are terms that are syntactically identical to the atoms in the goal g. Here are the methods for accomplishing h.

6. The names cons and nil are inspired, of course, by the Lisp programming language.

```
transfer1($h, r, c, d, e, p, q, k, l$) ;; method to achieve a goal on($c, d$) by moving $c$
task:      achieve-goals($h$)
precond:   can-achieve(on($c, d$),$h, r$), need-to-move($c, h$),
subtasks:  take($k, l, c, e, p$), put($k, l, c, d, q$),  ;; move $c$ to its final position
           achieve-goals($r$)                   ;; achieve the rest of the goals

move0($h, c, d, r$)              ;; method for when a goal on($c, d$) is already true
task:      achieve-goals($h$)
precond:   can-achieve(on($c, d$),$h, r$), ¬need-to-move($c, h$)
subtasks:  achieve-goals($r$)      ;; just achieve the rest of the goals

move-out-of-way($h, c, d, p, q, k, l$)   ;; method to move $c$ out of the way
task:      achieve-goals($h$)
precond:   ¬can-proceed($h$), need-to-move($c, h$), ;; there's an impasse
           top($c, p$), ¬on($c$,pallet),           ;; moving $c$ will resolve it
subtasks:  take($k, l, c, d, p$), put($k, l, c$, pallet, $q$), ;; move $c$ to an empty pile $q$
           achieve-goals($h$)                     ;; achieve the remaining goals

do-nothing()                  ;; the "base case," i.e., no goals to achieve
task:      achieve-goals(nil)
precond:   ;; no preconditions
subtasks:  ;; no subtasks
```

■

11.7.2 Additional Extensions

High-Level Effects. Some HTN planners allow the user to declare in the planner's domain description that various nonprimitive tasks, or the methods for those tasks, will achieve various effects. Such planners can use these high-level effects to establish preconditions and can prune the partial plan if a high-level effect threatens a precondition. In practical applications, declarations of high-level effects can be useful for improving the efficiency of planning—but we did not include them in our formal model because their semantics can be defined in several ways, some of which can make it difficult to ensure soundness and completeness.

External Preconditions. Suppose that to decompose some task t, we decide to use some method m. Furthermore, suppose that there is some condition c such that regardless of what subplan we produce below m, at least one action in the subplan will require c as a precondition and the subplan will contain no action that achieves c. In this case, in order for the subplan to appear as part of a solution plan π, the precondition c must somehow be achieved elsewhere in π. Thus, we say that the precondition c is *external* to the method m. Some HTN planning systems allow users to state explicitly in the planner's domain description that certain conditions are external. Also, in some cases it is possible to detect external preconditions (even when they have not been declared) by analyzing the planning domain.

Time. It is possible to generalize PFD and Abstract-HTN 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 11.19.

Planning Graphs. This extension is different from the others because it is an extension to the planning algorithm but not the HTN representation: it is possible to modify Abstract-HTN to make use of planning graphs like those described in Chapter 6. The modified algorithm generates both a decomposition tree and a planning graph. The size of the planning graph is reduced by generating only those actions that match tasks in the decomposition tree, and the size of the decomposition tree is reduced by decomposing a task only if the planning graph can accomplish all of its predecessors. In empirical studies [370], an implementation of such an algorithm performed significantly better than an implementation of Abstract-HTN.

11.8 Extended Goals

HTN planning can easily accommodate (or be extended to accomodate) certain kinds of extended goals. Here are some examples:

- 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 $\neg$ at(r, bad-loc) to the move operator. In HTN planning, such a change to the move operator is unnecessary. We will never put the move operator into a plan unless it appears in the subtasks of one or more methods—thus if we prefer, we can instead put the precondition into the methods that call the move operator rather than into the move operator itself.
- Suppose that r1 begins at location loc1 in the initial state, and we want r1 to go from loc1 to loc2 back exactly twice. In Section 2.4.8, we said that this kind of problem cannot be expressed as a classical planning problem. It can be expressed as an STN planning problem in which the initial task network is ⟨solve-problem()⟩ and the only method relevant for this task is the following.

```
two-round-trips()
    task:      solve-problem()
    precond:   ;;no preconditions
    subtasks:  move(r1,loc1,loc2), move(r1,loc2,loc1),
               move(r1,loc1,loc2), move(r1,loc2,loc1)
```

- Suppose we want to require that every solution reach the goal in five actions or fewer. In HTN planning without any extensions, this can be ensured in a domain-specific way by limiting the number of actions that each method can generate. For example, the two-round-trips method of the previous scenario will never generate a plan of more than four actions. However, this approach is less satisfying than the domain-independent control rule that can be written in STL planning (see Section 10.6). A more general solution would be to extend the planning language to include the predicate symbols, function symbols, and attached procedures needed to perform integer arithmetic. If we do this, then we can put into the initial state the atom count(0) and modify every operator to have the preconditions count(i) and $i \leq 5$. This is basically the same approach that we proposed for classical planning in Section 2.4.8.

- 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 that we described for classical planning in Section2.4.8: extend HTN 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.

Two of the above examples required extending HTN planning to accommodate function symbols and integer arithmetic. 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 planning algorithms such as TFD and PFD that plan forward from the initial state.

Like STL planning (see Example 10.6), HTN planning would have difficulty accommodating extended goals that require infinite sequences of actions.

11.9 Discussion and Historical Remarks

The basic ideas of HTN planning were first developed more than 25 years ago in work by Sacerdoti [460] and in Tate's Nonlin planner [503]. HTN planning has been more widely used in planning applications than any of the other planning techniques described in this book [551]. Examples include production-line scheduling [549], crisis management and logistics [72, 135, 504], planning and scheduling for spacecraft [1, 180, see next page], equipment configuration [3], manufacturing process planning [494], evacuation planning [406], the game of bridge [495], and robotics [401, 402]. Some of these applications will be discussed in Chapters 19 and 22.

In a complex application, an HTN planner may generate plans that contain thousands of nodes. Plans this large are very difficult for humans to understand without a natural pictorial representation. Several HTN planners (e.g., SIPE-2 and O-Plan) provide GUIs to aid in generating plans, viewing them, and following and controlling the planning processes [551]. Particularly useful for visualizing the plan derivation and structure is the ability to view its decomposition tree at various levels of abstraction.

The first steps toward a theoretical model of HTN planning were taken by Yang [558] and Kambhampati and Hendler [301]. A complete model was developed by Erol *et al.* [174]. This model provided the basis for complexity analysis [175] and the first provably correct HTN planning procedure, UMCP [174]. Our model of HTN planning in Section 11.5 is based on theirs, as are our complexity results for HTN planning in Section 11.6.

We have described HTN methods as a modification to the branching function of Abstract-search. An alternative model [49, 53] is to use the branching function of classical planning and consider the methods to be a pruning function. This model is appealing in that it provides a clear relation to classical planning. However, the limitation of this model is that it is only capable of expressing classical planning problems.

The best-known domain-independent HTN planning systems are listed here.

- Nonlin [503][7]is one of the first HTN planning systems.
- SIPE-2 [550][8] has been used in many application domains.
- O-Plan [135, 504][9] has also been used in many application domains.
- UMCP [174][10] is an implementation of the first provably sound and complete HTN planning algorithm.
- SHOP2 [413][11] is an efficient planning system which won one of the top four prizes in the 2002 International Planning Competition [195].

Several of these systems incorporate most of the extensions described in Section 11.7.

Our Lifted-PFD procedure is a simplified version of SHOP2, our Lifted-TFD procedure is a simplified version of SHOP2's predecessor SHOP, and our Abstract-HTN procedure is a simplified version of UMCP.

High-level effects were first described by Tate [503], and the conditions necessary to achieve soundness with them were explored by Bacchus and Yang [36] and Young *et al.* [564]. The semantics of high-level effects are defined in two different ways in the literature: either as effects in addition to the ones asserted by the planning operators (e.g., [36]) or as contraints that must be satisfied in order for a task to be achieved or a method to succeed (e.g., [173]). These two approaches result in very different planning algorithms.

Declarations of external preconditions have been used in the Nonlin [503] and SIPE-2 [550] planning systems. Algorithms for finding external preconditions automatically have been developed for use in the UMCP system [516].

7. A copy of Nonlin can be downloaded at http://www.aiai.ed.ac.uk/project/nonlin.
8. A copy of SIPE-2 can be downloaded at http://www.ai.sri.com/~sipe if the user has a license.
9. A copy of O-Plan can be downloaded at http://www.aiai.ed.ac.uk/oplan.
10. A copy of UMCP can be downloaded at http://www.cs.umd.edu/projects/plus/umcp.
11. A copy of SHOP2 can be downloaded at http://www.cs.umd.edu/projects/shop.

O-Plan, SIPE-2, and SHOP2 can each do certain kinds of temporal planning. For details, see the web sites for O-Plan and SIPE-2, and see Nau *et al.* [413] for SHOP2.

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

Comparing HTNs to the control rules described in Chapter 10, it is hard to say which type of control knowledge is more effective. HTN planners have been much more widely used in practical applications, but that is partly because they have been around longer. HTNs give a planner knowledge about what options to consider, and control rules give a planner knowledge about what options *not* to consider. Probably most researchers would agree with Bacchus and Kabanza [33] that the two types of knowledge are useful in different situations and that combining them is a useful topic for future research.

11.10 Exercises

11.1 Draw the decomposition tree produced by TFD on the planning problem in Figure 11.1, using the methods listed in Example 11.7 (see page 238).

11.2 Since TFD is a nondeterministic procedure, it may have several different execution traces that solve the same planning problem. How many such execution traces are there in Exercise 11.1?

11.3 Write totally ordered methods and operators to produce a decomposition tree similar to the one shown in Figure 11.6.

11.4 Write totally ordered methods to generate the noninterleaved decomposition tree shown in Figure 11.8.

11.5 Let c be any positive integer, and let $\mathcal{P}$ be any HTN planning problem for which the height of $\mathcal{P}$'s search space is no greater than c. Can $\mathcal{P}$ be expressed as a classical planning problem? If so, how? If not, why not?

11.6 Prove that TFD is sound.

11.7 Prove that TFD is complete for all total-order STN planning problems.

11.8 Suppose we write a deterministic implementation of TFD that does a depth-first search of its decomposition tree. Is this implementation complete? Why or why not?

11.9 Write the Lifted-TFD procedure described near the end of Section 11.3.

11.10 Prove that Lifted-TFD is sound and complete for all total-order STN planning problems.

11.11 Give a way to rewrite any classical planning problem as a total-order STN planning problem.

11.12 Trace the operation of PFD on the planning problem in Figure 11.1, using the methods listed in Example 11.3 (see page 233). Draw the decomposition tree.

11.13 Since PFD is a nondeterministic procedure, it may have several different execution traces that solve the same planning problem. How many such execution traces are there in Exercise 11.12?

11.14 To make the painting problem of Exercise 5.6 into a total-order STN problem, let's add these atoms to the initial state:

need-color(b1,red), need-color(b2,red),

need-color(b3,blue), need-color(b4,blue)

The initial task network is $w = \langle$paint1(w), paint1(x), paint1(y), paint1(z)$\rangle$. The operators are unchanged, and there is just one method:

```
method1(b, r, c, k)
  task:      paint1(b)
  precond:   need-color(b, k)
  subtasks:  dip(r, c, k), paint(b, r, k)
```

(a) How many different possible solutions are there?

(b) How many method and operator applications will a depth-first implementation of TFD do in the best case? In the worst case?

(c) How can the domain description be modified to make the worst case more efficient?

(d) To introduce partial ordering into the planning problem, suppose we redefine the initial task network to be $w = (\{u_1, u_2, u_3, u_4\}, \emptyset)$, where $u_1 =$ paint1(b1), $u_2 =$ paint1(b2), $u_3 =$ paint1(b3), and $u_4 =$ paint1(b4). What problem will occur if we do this, and how can it be fixed?

11.15 Consider the washing problem described in Exercise 5.7.

(a) To turn the problem domain into a total-order STN domain, write a method called do-wash that will decompose a task wash(x, y) into the totally ordered sequence of tasks $\langle$start-fill(x), end-fill(x), start-wash(x), end-wash(x, y)$\rangle$. Include preconditions sufficient to guarantee that if do-wash is applicable, then start-fill(x) can be accomplished.

(b) Is do-wash's precondition sufficient to guarantee that if do-wash is applicable, then all of its subtasks can be accomplished? Explain your answer.

(c) Suppose we run TFD with the method you wrote in part (a), the same initial state and operators as in Exercise 5.7, and the initial task network

⟨wash(wm,clothes), wash(dw,dishes), wash(bt,dan)⟩. Are there any solution plans that TFD will be unable to find that PSP was able to find? Explain your answer.

(d) Suppose we run PFD with the same information as in part (c), except that the intial task network is an unordered set of tasks {wash(wm,clothes), wash(dw,dishes), wash(bt,dan)}. Are there any solution plans that PFD will be unable to find that PSP was able to find? Explain your answer.

(e) Is do-wash's precondition still sufficient to guarantee that if do-wash is applicable, then all of its subtasks can be accomplished? Explain your answer.

11.16 In Example 11.3 (see page 233), suppose we allow the initial state to contain an atom need-to-move(p, q) for each stack of containers that needs to be moved from some pile p to some other pile q. Rewrite the methods and operators so that instead of being restricted to work on three stacks of containers, they will work correctly for an arbitrary number of stacks and containers.

11.17 Prove that PFD is sound and complete for all STN planning problems.

11.18 Write the Lifted-PFD procedure described near the end of Section 11.4. (Hint: Take a look at the Abstract-HTN procedure.)

11.19 Prove that Lifted-PFD is sound and complete for all STN planning problems.

11.20 Modify PFD to work for HTN planning problems in which there are no constraints of the form after(U', l) and between(U', U'', l).

11.21 Modify the procedure you developed in Exercise 11.20 to work for HTN planning problems in which there are constraints of the form after(U', l) and between(U', U'', l).

11.22 Trace how Example 11.14 (see page 253) would work on the DWR version of the Sussman anomaly (Example 4.3).

11.23 Using the domain description given in Example 11.14 (see page 253), is there ever a possibility of TFD finding any redundant solutions? Why or why not?

11.24 Is there a way to rewrite Example 11.14 (see page 253) so that all of the axioms are Horn clauses? If so, what is it? If not, why not?

11.25 In Exercise 11.16, suppose the planning language includes a single binary function symbol. Do the exercise without allowing the additional atoms in the initial state. (Hint: Use the function symbol to create a task whose argument represents a list of containers to be moved.)

11.26 Download a copy of SHOP2 from http://www.cs.umd.edu/projects/shop.

(a) Write several planning problems for it in the blocks world (see Section 4.6), including the one in Exercise 2.1, and run SHOP2 on them using the blocks-world domain description that comes with it.

(b) Encode the domain description of Example 11.8 (see page 240) as a SHOP2 domain description. Run it on several problems in that domain, including the one in Example 11.1.