CHAPTER 15

Planning and Resource Scheduling

15.1 Introduction

Scheduling addresses the problem of how to perform a given set of actions using a limited number of resources in a limited amount of time. A *resource* is an entity that one needs to borrow or to consume (e.g., a tool, a machine, or energy) in order to perform an action. An action may have the choice between several alternate resources, and a resource may be shared between several actions. The main decision variables for scheduling a given action are which resources should be allocated to the action and when to perform it. A solution *schedule*[1] has to meet a number of constraints (e.g., on deadlines, on the ordering of actions, on the type and amount of resources they require, and on the availability of resources). In addition, there is usually an optimization requirement in scheduling: one would like to minimize a cost criteria such as achieving all actions as early as possible or using the least costly resources.

Planning and scheduling are closely related problems. In a simple decomposition scheme, planning appears to be an upstream problem that needs to be solved before scheduling. Planning focuses on the causal reasoning for finding the set of actions needed to achieve the goal, while scheduling concentrates on time and resource allocation for this set of actions. Indeed, a plan given as output by a planner is usually a structured or partially ordered set of actions that does not specify the resources and a precise schedule for its actions.

Along with this decomposition scheme, a scheduled plan is synthesized in two steps (see Figure 15.1): (1) *what* to do, and (2) *when* and *how* to do it. Planning addresses the issue of what has to be done, while scheduling focuses on when and how to do it.

1. In the usual scheduling terminology, an action is called an *activity* or a *task*, and a solution is a *schedule*. We will keep here the planning terminology for actions, but we will use the word *schedule* to emphasize a plan with scheduled resources.

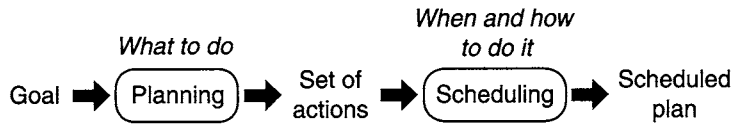

Figure 15.1 Planning and scheduling: a simple decomposition scheme.

Very often, planning and scheduling have been addressed as two separate problems.

1. In "*pure scheduling*" problems (e.g., for organizing jobs in a factory), there is a limited number of possible types of jobs. The set of actions required to process a job is known because the *what* issue has been addressed separately (e.g., as part of the initial design of the factory). The problem of interest is to find an optimal schedule for the jobs at hand.
2. In "*pure planning*" problems (e.g., for controlling an autonomous agent), the usual agent has few intrinsic resources, such as for moving around and handling objects. There is not much choice about the resources to allocate to actions and not much concurrency and resource sharing between actions. The concern is more about finding the right actions for achieving a goal than about finding an optimal schedule for sharing resources between actions and agents.

However, few practical planning problems are "pure planning" problems, and there is more and more demand for scheduling problems to handle planning issues as well. There is a need for both planning and scheduling, when the problem at hand is to organize the activity of a complex system (e.g., one with alternate resources, resource-sharing requirements, and a large set of possible goals), in particular when it is not feasible to cache the plans for each type of goal beforehand. For example, this is the case in factory organization with flexible manufacturing, in management of large infrastructure operations such as a rescue operation, and in the control of autonomous agents whose actions require and share a variety of resources.

Some of these complex applications may possibly be addressed along the simple decomposition scheme of Figure 15.1. However, considering planning and scheduling as two successive independent steps is, in general, too restrictive and oversimplifying. On the one hand, the decomposition scheme may not work when the constraints are too tight (e.g., when there are many plans but very few that have a feasible schedule). On the other hand, modeling resources and taking them into account as fluents can be highly inefficient.

A given set of objects in a planning domain can be modeled either as a resource or as a set of constants. In the latter case, these objects will be referred to *individually*, through the names of the constants, whereas in the former case they will be referred to *indiscriminately*, as the number of objects available at some time point. For example, if a location in a DWR scenario contains five identical cranes, then planning a load action with a particular crane introduces a decision point that

may be unnecessary: if one is not interested in distinguishing between these five cranes, backtracking on this choice is meaningless. If this set of cranes is modeled as a resource, then a load action reduces during its duration the *number* of available cranes by one. The choice of which of the five cranes will be used is not a decision point at planning time. The set of robots in the DWR domain can also be modeled as a resource if they are considered to be identical or as several resources corresponding to the different types of available robots (see Section 20.4). Containers, on the other hand, cannot be modeled as a resource as long as one needs to refer to a container individually: its current location and its desired goal location.

The main difference between a state variable and a resource is not the types of their domains (usually symbolic and numeric, respectively).[2] It is the fact that an action modifies the value of a state variable in an *absolute* way, whereas an action changes a resource variable in a *relative* way. As an example of an action that can have effects on both state variables and resources, move(r, l, l') modifies the robot's location from the absolute value l to l', while it reduces the level of energy available on the robot in a relative way, from its current level before the action to some final level.

Furthermore, several *concurrent* actions may use the same resource; their cumulative effects shape the total evolution of that resource. For example, this is the case if the robot performs, while moving, some surveillance and communication actions that are not strictly required for its motion but that draw on its available energy.

We will be pursuing in this chapter an integrated approach that searches for a scheduled plan in which the resource allocation and the action scheduling decision are taken into account while planning. This approach, presented in Section 15.4, will be developed as a continuation of the temporal planning techniques of Chapter 14, where we addressed the temporal issues of planning and scheduling. This state-variable approach is not the only satisfactory way to tackle the integration of planning and scheduling. Other techniques, in particular HTN-based techniques, have been pursued successfully and are discussed in Section 15.5.

Before that, we introduce scheduling terminology and concepts (Section 15.2). We very briefly survey different types of machine scheduling problems and some of the mathematical programming techniques used specifically for scheduling (Section 15.3). These two sections remain quite informal; more precise notations and definitions are given in Section 15.4. The chapter ends with a discussion and exercises.

15.2 Elements of Scheduling Problems

A scheduling problem is specified by giving

- a set of resources and their future availability,
- a set of actions that needs to be performed and their resource requirements,

2. Hence, the extension of the PDDL to numerical variables does not strictly cover resources.

- a set of constraints on those actions and resources, and
- a cost function.

A schedule is a set of allocations of resources and start times to actions that meet all resource requirements and constraints of the problem. An optimal schedule optimizes the cost function.

There are different types of scheduling problems depending on the nature of the resources, the type of actions and their resource requirements, the type of constraints used, and the uncertainty explicitly modeled in the problem specification. Let us review informally some of the main classes of scheduling problems while focusing on the deterministic case, i.e., without uncertainty.

15.2.1 Actions

Usually in scheduling, actions have more restricted models than in planning. In particular, the state-transition function, which is essential in planning models, is usually implicit in scheduling, the emphasis being more on the resource and time needed by an action.

An action a is simply specified with its resource requirements (discussed in Section 15.2.2) and three variables ranging over the real numbers: its start time $s(a)$, its end time $e(a)$, and its duration $d(a)$. Usually $s(a)$ and $e(a)$ are specified within upper and lower bounds: $s(a) \in [s_{min}(a), s_{max}(a)]$ and $e(a) \in [e_{min}(a), e_{max}(a)]$.

Two types of actions are considered in scheduling: *preemptive actions* and *nonpreemptive actions*. A nonpreemptive action has to be executed continuously, without interruption; in that case, $d(a) = e(a) - s(a)$.

A preemptive action can be interrupted a number of times and resumed after a while. While interrupted, a preemptive action releases its resources that can be allocated to another action. Here $d(a) = \sum_I d_i(a) \leq e(a) - s(a)$, where the values of $d_i(a)$ are the durations of intervals during which a is executed. These execution intervals are not given as input (otherwise, a decomposes trivially into a finite sequence of actions). They have to be found as part of a solution schedule. Constraints can be associated with these execution intervals, e.g., fixed constraints such as "a requires daylight; if needed, the action must be interrupted during the night," and resource constraints such as "any action using resource r for an hour has to release it for at least half an hour." There can also be a cost associated with each interruption. In the rest of this chapter, we will consider only nonpreemptive actions, except when stated otherwise.

15.2.2 Resources

A resource is something needed in order to achieve an action. In addition to the usual preconditions and effects, the resource requirements of an action specify which

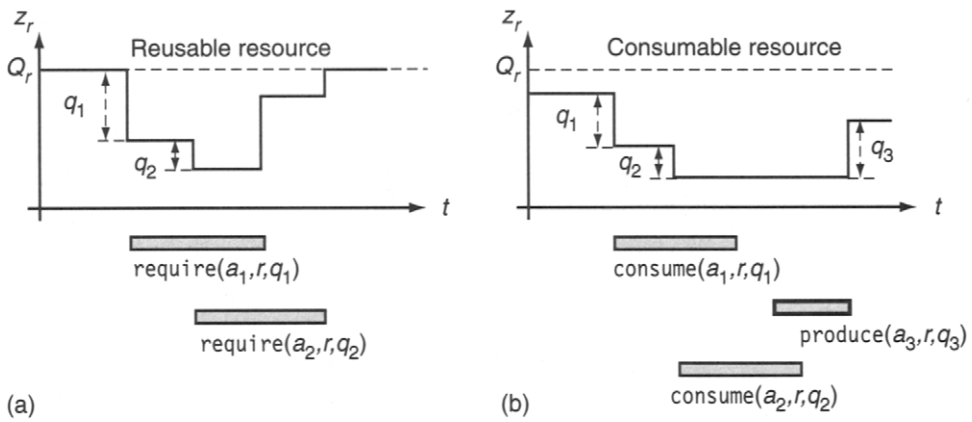

Figure 15.2 Examples of resource profiles.

resources and what quantities of each resource are needed for that action. Two main classes of resources can be distinguished: *reusable resources* and *consumable* resources.

A reusable resource is "borrowed" by an action during its execution. The resource is released, unchanged, when the action is completed or is interrupted. Typical examples of reusable resources are tools, machines, and cranes in some locations of the DWR domain. A reusable resource r has a total capacity Q_r and a current level $z_r(t) \in [0, Q_r]$. For example, if there is a total of five cranes at some location, then $Q_r = 5$ for that resource; its level $5 \geq z_r(t) \geq 0$ gives the number of available cranes at time t in that location.

An action a that requires during its execution a quantity q of the resource r decreases the current level of r at time $s(a)$ by an amount q; at time $e(a)$ it increases the current level by the same amount. For a given schedule, the function $z_r(t)$ is referred to as the *resource profile* of r. Figure 15.2 (a) illustrates a profile of a resource r that has two requirements, require(a_1, r, q_1) and require(a_2, r, q_2), by two overlapping actions. A reusable resource can be *discrete* (e.g., a finite set of tools) or *continuous*, (e.g., the electrical power in an outlet). A particular case is when r has a unit capacity, i.e., $Q_r = 1$: only one action at a time may use it. In the DWR domain, a single crane attached to a location can be modeled as a unary discrete resource.

A consumable resource is consumed by an action during its execution. Bolts, energy, and fuel are typical examples of consumable resources. A consumable resource may also be produced by other actions (e.g., refueling, which increases the amount of fuel in the tank of a robot). Such a resource r can be modeled as a *reservoir* of maximal capacity Q_r and of current level $z_r(t) \in [0, Q_r]$. An action a that consumes a quantity q of r reduces z_r by q. For example, this reduction can be modeled as a step function at the start time $s(a)$ if that resource is consumed all at once or as a linear function from $s(a)$ to $e(a)$ if the consumption takes place

continuously during the action execution. Similarly for an action that produces a quantity q of r: z_r is augmented by q at $e(a)$ or as a linear function between $s(a)$ and $e(a)$. The resource profile of a consumable resource is, similarly, the evolution $z_r(t)$ for a given schedule, as shown in Figure 15.2 (b).

It is important to stress that an action changes a resource variable r in a *relative way* by reducing or augmenting the current level z_r. The reader is invited to compare the graphs in Figure 15.2 with those of Figure 14.5, which plot the evolution of state variables along time. But for unary discrete resources, several concurrent actions may share or use the same resource and contribute to the definition of its profile.

The resource requirements of an action can be stated as a conjunction of assertions: $\mathsf{require}(a, r_i, q_i) \wedge \ldots \wedge \mathsf{consume}(a, r_j, q_j) \wedge \ldots \wedge \mathsf{produce}(a, r_k, q_k)$. Unless one assumes that the consumption and production of a resource happen as step functions at the beginning and at the end of an action, respectively, the third argument of consume or produce needs be given as a function of time, specifying how the resource is consumed or produced between the time points $s(a)$ and $e(a)$.

Furthermore, alternative resources can be specified for an action as a disjunction of assertions: $\mathsf{require}(a, r_1, q_1) \vee \ldots \vee \mathsf{require}(a, r_m, q_m)$. More generally, a set of alternative resources can be specified for all actions by defining a class of equivalent resources: $\mathsf{resource\text{-}class}(s) = \{r_1, \ldots, r_m\}$. A requirement such as $\mathsf{require}(a, s, q)$ is equivalent to the disjunction $\mathsf{require}(a, r_1, q) \vee \ldots \vee \mathsf{require}(a, r_m, q)$. It can be met by allocating to a a quantity q of any resource in this set. In the case of alternate resources, one has to distinguish between the resource requirements of an action a and the resources actually allocated to it by a schedule.

A resource may have a finite set of possible states. For example, this is the case for a refrigerated container that has different temperature settings, or a machine tool that can be set with different tools. A requirement for such a resource has to specify which particular state of the resource is required. Two actions may share the resource only if they require it to be in the same state. Changing the state of a resource has a cost and takes time.

Finally, an efficiency parameter can be associated with a resource r: the duration of an action using r varies as a function of that parameter.

15.2.3 Constraints and Cost Functions

Temporal constraints in scheduling problems are usually expressed within a framework equivalent to the quantitative temporal constraint networks (Section 13.4.2). Most often, simple temporal constraints (STPs, see Section 13.4.1) are used. Bounds on the distance between start points and end points of actions are specified in either of two ways:

- With respect to an absolute reference time, e.g., in order to specify a *deadline* for an action a, $e(a) \leq \delta_a$ (i.e., $e(a) \in (-\infty, \delta_a]$), or a *release* date for a before which the action cannot start, $s(a) \geq \rho_a$ (i.e., $s(a) \in [\rho_a, \infty)$)

- With respect to relative instants, e.g., in order to specify the *latency* between two actions a and b, $s(b) - e(a) \in [\lambda_{a,b}, \mu_{a,b}]$, or the total extent of a preemptive action c, $e(c) - s(c) \leq \nu_c$

Constraints can be associated with a resource, such as a reduced availability during some period. For example, a radio link resource of a satellite has *visibility windows*, which are intervals of time during which communication with the satellite is possible. Generally, an initial availability profile can be expressed for each resource by giving initial values to z_r as a function of time. Other constraints may link resource allocations to time bounds. For example, if two actions a and b use the same resource r, then there will be a latency, or required period of time between the two actions, of at least δ_r: if allocate(r, a) and allocate(r, b) then $s(b) - e(a) \geq \delta_r$ or $s(a) - e(b) \geq \delta_r$.

Several types of costs can be considered in scheduling. The cost for using or consuming a resource can be fixed or can be a function of the quantity and duration required by an action. A resource can have an additional *setup cost*, or cost of making the resource available after a use, which may depend on the previous action that used the resource or the previous state of the resource. For example, if action b follows action a in using the unary resource r, then this allocation has a cost of $c_r(a, b)$ and takes a duration of $d_r(a, b)$, i.e., $s(b) \geq e(a) + d_r(a, b)$. Different penalty costs can be specified, usually with respect to actions that do not meet their deadlines.

The objective criteria in scheduling is to minimize a function f of the various costs and/or the end time of actions in a schedule. The most frequent criteria to minimize are:

- the makespan or maximum ending time of the schedule, i.e., $f = max_i\{e(a_i) \mid a_i \in A\}$,
- the *total weighted completion time*, i.e., $f = \Sigma_i w_i e(a_i)$, where the constant $w_i \in \Re^+$ is the weight of action a_i,
- the maximum tardiness, i.e., $f = max\{\tau_i\}$, where the tardiness τ_i is the time distance to the deadline δ_{a_i} when the action a_i is late, i.e., $\tau_i = max\{0, e(a_i) - \delta_{a_i}\}$,
- the total weighted tardiness, i.e., $f = \Sigma_i w_i \tau_i$,
- the total number of late actions, i.e., for which $\tau_i > 0$,
- the weighted sum of late actions, i.e., $f = \Sigma_i w_i u_i$, where $u_i = 1$ when action i is late and $u_i = 0$ when i meets its deadline,
- the total cost of the schedule, i.e., the sum of the costs of allocated resources, of setup costs, and of penalties for late actions,
- the peak resource usage, and
- the total number of resources allocated.

Other criteria may also be considered, such as scheduling a maximum number of the given set A of actions, while taking into account all constraints, including the deadlines.

15.3 Machine Scheduling Problems

Machine scheduling is a well-studied generic class of scheduling problems. The class includes the *flow-shop*, the *open-shop*, and the *job-shop* scheduling problems. This section presents machine scheduling problems and their various special cases. It then discusses the complexity of machine scheduling and introduces briefly approaches for solving these problems and for integrating planning and machine scheduling.

15.3.1 Classes of Machine Scheduling Problems

A *machine* is a resource of unit capacity that is either available or not available at some time point. A *job* j is a partially ordered set of one or several actions, $a_{j1}, \ldots, a_{jk}$. It is analogous to what we have been calling a *plan*; it is partially ordered, and the resources needed by its actions, i.e., machines, are left uninstantiated. In a machine scheduling problem, we are given n jobs and m machines. A schedule specifies a machine i for each action a_{jk} of a job j and a time interval during which this machine processes that action. A machine cannot process two actions at the same time, and a job cannot be processed by two machines at once. In other words, two time intervals corresponding to the same machine or to the same job should not overlap.

Actions in different jobs are completely independent—they can be processed in any order. But actions in the same job cannot be processed concurrently, and they may have ordering constraints. One can think of a job as the same physical object, e.g., an engine in an assembly line, being processed at different stages through different actions or operations performed by different machines. This explains the constraint of no overlap in time of two actions of the same job.

In *single-stage* machine scheduling problems, each job corresponds to a single action, i.e., it requires a single operation and can be processed by *any* of the m machines. One may consider the following types of machines.

- For *identical parallel machines*, the processing time p_j of job j is independent of the machine allocated to j. The m machines can be modeled as a single discrete resource of capacity m.
- For *uniform parallel machines*, a machine i has a speed $\mathsf{speed}(i)$; the processing time of job j on the machine i is the ratio of p_j to the speed of i.

- For *unrelated parallel machines*, the $n \times m$ processing times p_{ij} have independent values.

In *multiple-stage* machine scheduling problems, a job corresponds to several actions, each requiring a particular machine. Here, the m machines have different functions. There are two particular cases and one general case of multi-stage machine scheduling problems.

- In *flow-shop* problems, each job j has exactly m actions. Action a_{ji} needs to be processed by machine i, and the actions have to be processed in the order $1, 2, \ldots, m$.
- *Open-shop problems*, are similar to flow-shop problems, but the m actions of a job can be processed in any order.
- *Job-shop problems* correspond to the general case in which each job has a number of actions with specific requirements on the needed machines for each action and the processing order of actions.

In addition to these features, a machine scheduling problem is characterized by the constraints associated with the set of jobs. One may specify deadlines and/or release dates for each job, setup times for the actions of a job j that depend on the allocated machine and/or the preceding action scheduled on this machine, and precedence constraints between jobs.

Finally, a machine scheduling problem is also characterized by the optimization criteria specified for that problem, e.g., makespan, maximum tardiness, weighted completion time, weighted tardiness, etc. However, here deadlines and end times of interest in the optimization criteria are those of jobs, not of their individual actions.

Example 15.1 Let us consider a job-shop problem with three machines (m_1, m_2, and m_3) and five jobs ($j_1, \ldots, j_5$). The first job j_1 has three actions. It requires successively the machine m_2 during three units of time for its first action, then the machine m_1 during three units for its second action, and then m_3 during six units for its last action. This is denoted as:

$j_1 : \langle m_2(3), m_1(3), m_3(6) \rangle$

The four other jobs are similarly specified:

$j_2 : \langle m_2(2), m_1(5), m_2(2), m_3(7) \rangle$
$j_3 : \langle m_3(5), m_1(7), m_2(3) \rangle$
$j_4 : \langle m_2(4), m_3(6), m_1(7), m_2(4) \rangle$
$j_5 : \langle m_2(6), m_3(2) \rangle$

A schedule for this problem is given in Figure 15.3. The horizontal axes represent time; each bar denotes which machine is allocated to a job and when it is allocated.

■

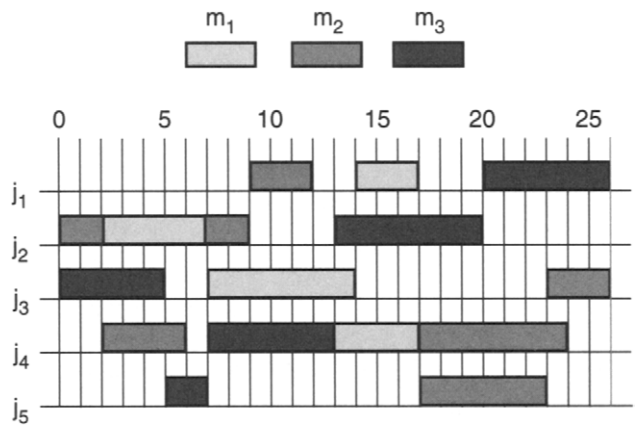

Figure 15.3 A schedule for a job-shop problem.

A standard notation, introduced in the survey of Graham *et al.* [247], is widely used in the literature for designating different machine scheduling problems. It has three descriptors, noted as $\alpha|\beta|\gamma$, where:

- α gives the type of the problem, denoted as P, U, and R for, respectively, identical parallel, uniform parallel, and unrelated parallel machines, and F, O, and J for, respectively, flow-shop, open-shop, and job-shop problems. α also gives the number of machines when the class of problem is specified with a fixed number of machines.
- β indicates the characteristics of the jobs, i.e., the jobs' deadlines, setup times, and precedence constraints. This descriptor field is left empty when there are no constraints on jobs, i.e., $\alpha||\gamma$.
- γ gives the objective function.

For example, $Pm|\delta_j|\Sigma_j w_j e_j$ is the class of problems with a fixed number of m parallel machines, deadlines on jobs, and the objective of minimizing the weighted completion time. $J|prec|makespan$ is the class of job-shop problems on an arbitrary number of machines with precedence constraints between jobs and the objective of minimizing the makespan of the schedule.

15.3.2 Complexity of Machine Scheduling

Only a few machine scheduling problems have been solved efficiently with polynomial algorithms. For example, the single-stage, one-machine problem, $1||max\text{-}tardiness$, can be solved in $O(n\log n)$: its solution is a sequence of jobs

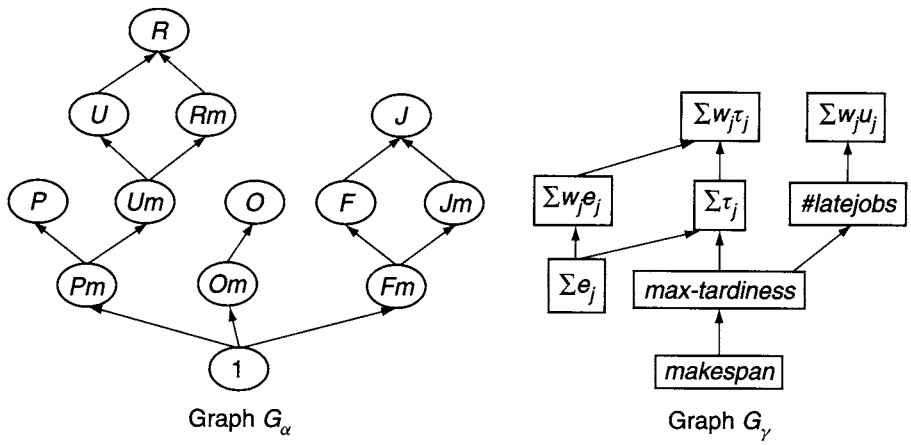

Figure 15.4 Reductions between machine scheduling problems.

as processed by the available machine; the optimal sequence is simply obtained by ordering the jobs in nondecreasing due dates. Similarly, the problem $1||\Sigma_j w_j e_j$ is solved by ordering jobs in nondecreasing ratios p_j/w_j. The flow-shop problem $F2||$*makespan* or the open-shop problem $O2||$*makespan* are also polynomial.

Other than these few cases, most machine scheduling problems have been shown to be of high complexity. For example, the problems $1|$*release-dates*$|$*max-tardiness*, $J2||$*makespan*, $J2||$*max-tardiness*, and $O3||$*makespan* are NP-hard.

As one may easily imagine, there are many polynomial reductions[3] among the various types of machine scheduling problems. A partial view of these reductions is illustrated through the two graphs shown in Figure 15.4, which correspond, respectively, to the α and γ descriptors introduced earlier: vertices in graph G_α are labeled with the α descriptor, and vertices in graph G_γ are labeled with the γ descriptor. If two classes of problems $P = \alpha|\beta|\gamma$ and $P' = \alpha'|\beta|\gamma$ are such that there is a path in G_α from the vertex labeled α to the vertex labeled α', then the class P reduces to the class P'. For example, the class $P_m|\beta|\gamma$ reduces to the class $U|\beta|\gamma$, and similarly for two classes of problems $P = \alpha|\beta|\gamma$ and $P' = \alpha|\beta|\gamma'$ with respect to paths in graphs G_γ. If P reduces to P', then when P' is polynomial so is P, and when P is NP-hard, so is P'. This general property simplifies the characterization of the complexity of machine scheduling problems.

Using these two graphs, and others for the β descriptors, and relying on known results of the complexity of some classes of problems, it is possible to derive automatically the complexity status of many problems. The survey of Lawler *et al.* [355] refers to a database of 4,536 classes of machine scheduling problems, out of which

3. See Appendix A, Section A.5.

417 are known to be solvable in polynomial time and 3,821 have been proven NP-hard.

15.3.3 Solving Machine Scheduling Problems

Exact algorithms for machine scheduling rely often on the *branch-and-bound* schema. In a few cases, clever branching rules and lower bounds enable the technique to be scaled up to address problems of large size, such as several NP-hard classes of single-stage one-machine problems [106]. Other approaches, such as dynamic programming and integer programming, have been found useful in some particular problems.

However, in general, machine scheduling problems are hard combinatorial optimization problems. For example, the state of the art for exact solutions of the $J||makespan$ problem is still open for instances of 10 to 20 jobs and 10 machines. Hence, approximation techniques are the most widely used ones for addressing practical problems, among which local search methods and randomized algorithms have been found successful (see Anderson *et al.* [21] for a survey on local search in machine scheduling). The characterization of these approximation techniques aims at establishing bounds of the quality of the solution with respect to the optimum.

15.3.4 Planning and Machine Scheduling

Some planning problems can be modeled as machine scheduling problems. This usually requires several simplifying assumptions and a significant change of focus. The planning actors intended to perform the actions of the plan are considered here to be machines. Instead of focusing on finding what each actor should do, machine scheduling focuses on deciding which machine should be allocated to which job and when. This is easily illustrated in the so-called logistic planning domains, such as transportation problems, where a set of trucks, airplanes, and other means of transportation is used to carry parcels between cities and locations. The DWR domain can provide a good example.

Example 15.2 A very simple version of the DWR domain, viewed in machine scheduling terms, considers the problem of using m identical robots to transport n containers between distinct locations. Here, a machine is a robot. A job, denoted by container-transportation(c, l, l'),[4] consists of moving a container c from l to l'. A job requires and is assumed to be entirely processed by a single robot. The robot allocated to a job has to go to location l, pick up and load container c, move to location l', and unload the container. There are no precedence constraints between jobs, but there can be release dates, e.g., a container is available for transportation

4. This notation is similar to the one used in Chapter 11 to denote tasks in HTN planning.

only when the ship carrying it arrives at the harbor. Jobs may also have deadlines, e.g., for the expected departure times of ships.

The duration of a job is the time it takes to go from l to l', plus the time for loading and unloading. To model the time it takes a robot to move from the destination of job k to the origin location of the next job j, we can introduce a setup time t_{ikj} if the robot i is allocated to a job j after performing a job k. Finally, if we are interested in minimizing the weighted completion time of the schedule, then this DWR problem is modeled as a problem of the class $P|r_j\delta_j t_{ikj}|\Sigma_j w_j e_j$, where r_j, δ_j, and t_{ikj} denote, respectively, the release date, the deadline, and the setup times of job j. ■

This simplified formulation of the DWR domain as a machine scheduling problem does not take into account cranes for loading and unloading containers nor the arrangement of containers into piles. The following example introduces a richer specification of the domain.

Example 15.3 Here the job container-transportation(c, l, l') decomposes into four actions: goto(l), load(c), move(l, l'), and unload(c). There are two types of resources: m robots and k_l cranes in location l. The four actions in each job require a robot, and the load and unload actions require cranes at locations l and l', respectively. The *same* robot should be allocated to the four actions of a job j because it does not make sense to use different robots for the same container transportation job.

Furthermore, precedence constraints between jobs can be used to take into account the relative positions of containers in piles: whenever c is on c', the transportation job of c should be ordered to start before that of c'. Similarly, one may specify precedence constraints on the end times of jobs to reflect the final positions of containers. This is not a general translation of the relation on(c, c') but only a limited way to handle it in scheduling terms. These constraints do not model the case of interfering containers that are in the initial or final piles but do not need to be transported between two locations. In order to do that, one has to introduce some *ad hoc* jobs, such as unstack(c, p), to remove c from a pile p, and stack(c, p), assuming unlimited free space in each location. These jobs should be added for every interfering container, in addition to precedence constraints between jobs that model the required initial and final positions of containers in piles. ■

Several remarks on the two previous examples are in order. First, machine scheduling handles requirements about which machines are needed for which actions in a job. There is also the requirement that although any robot can be allocated to the actions of a job it should be the *same* robot for *all* actions in a job is beyond the standard classes of machine scheduling problems.

Also note that not only are the precedence constraints on end times of jobs beyond standard classes, but it is hardly acceptable in practice. Even if one accepts a deterministic model without uncertainty in the durations of actions, at execution

time it makes sense to control the start times of actions, but it is quite difficult to control their end times. A way to avoid this problem is to assume that the final position of containers within a pile is irrelevant. If these positions are important for the problem at hand, then the earlier machine scheduling formulation is not adequate.

Even this *ad hoc* machine scheduling model of the DWR domain ignores several resource constraints, such as the space in a location being limited to at most one robot at a time.

In conclusion, machine scheduling is a fairly restricted model with respect to the needs of planning. However, these well-studied classes of problems, which already have a high computational complexity, are very rich in heuristics and approximation techniques that can be beneficial for planning with resources using the approach presented in the next section.

15.4 Integrating Planning and Scheduling

We already discussed in the introduction of this chapter the limitations of separating planning from scheduling, and we motivated the need for an integrated approach when resources are involved in the problem at hand. The two frameworks presented in Chapter 14—temporal databases and chronicles—are both suitable for developing a planner that handles resources. In both cases the flaw-repair paradigm in the plan space within a CSP-based approach is applicable. Let us present in this section a *chronicle-based planner* that extends the material presented in Chapter 14 in order to integrate causal reasoning and resource reasoning.[5]

15.4.1 Representation

Recall from Chapter 14 that a chronicle representation is based on a set $X = \{x_1, \ldots, x_n\}$ of *state variables*. Each x_i is a function from time to some finite domain describing how the value of a state property evolves along time. To take resources into account, we further consider as part of the definition of a domain a finite set $Z = \{z_1, \ldots, z_m\}$ of *resource variables*. Each resource r is described by a resource variable z_r, which is a function of time that gives the resource profile of r.[6] In the rest of this chapter, we will focus on discrete *reusable* resources. Thus, each resource variable is a function z_r : time $\rightarrow \{0, 1, \ldots, Q_r\}$, where Q_r is the total capacity of the resource r.

As we did with state variables, it can be convenient to describe generically a set of resource variables with a function of several arguments, of the form

5. This section relies on the material presented in the two previous chapters, in particular in Sections 13.3.2 and 14.3.

6. As for state variables, the time argument of the function z_r is not explicit in our notation.

$f : D \times \text{time} \rightarrow \{0, 1, \ldots, Q\}$, such that D is a finite domain, and for any $v \in D, f(v)$ is a resource variable of Z. For example, in a DWR domain in which every location has Q identical cranes, we can model cranes as resources and use a function $\text{cranes}(l)$ to describe the number of available cranes at location l at some time point t.

Because we are considering only discrete reusable resources, a resource is not consumed but only borrowed at the beginning of an action and released after its use. We will model the use of resources in a chronicle with temporal assertions, similar to those used for state variables, that express the amount of and the time at which the available quantity of a resource changes, or the interval during which some amount is used.

Definition 15.1 A *temporal assertion* on a resource variable z whose total capacity is Q is either:

- the instantaneous *decrease* of the resource profile of z at a time t by a *relative* quantity q, where q is an integer in the range $1 \leq q \leq Q$, when an amount q of z is borrowed at t,
- the instantaneous *increase* of the resource profile of z at a time t by a *relative* quantity q, when an amount q of z is released at t, or
- the *use* of a quantity q of the resource z during an interval $[t, t'), t < t'$, when an amount q is borrowed during the interval $[t, t')$.

These temporal assertions are denoted, respectively:

$$z@t : -q \qquad z@t : +q \qquad z@[t, t') : q$$

■

As for state variables, these types of assertions are directly related:

$$z@[t, t') : q \text{ is equivalent to } z@t : -q \wedge z@t' : +q$$

Furthermore, borrowing a resource *ad infinitum* is like consuming it: one may express the instantaneous *decrease* of the available quantity of a resource as a *use* of that resource over an infinite interval of time:

$$z@t : -q \text{ is equivalent to } z@[t, \infty) : q$$

Similarly, the *increase* of z at t by an amount $+q$ is equivalent to considering that a higher initial capacity $Q' = Q + q$ at time 0 and that the amount q of z was in use during the interval $[0, t)$:

$$z@t : +q \text{ is equivalent to } z@0 : +q \wedge z@[0, t) : q$$

For example, in Figure 15.2 (b) we can rewrite $\text{produce}(a_3, r, q_3)$ as an initial capacity of $Q_r + q_3$ together with $\text{require}(a', r, q_3)$ for a virtual action a' that starts at time 0 and ends at the end of action a_3. Note that this does not change the profile of the resource r nor in particular its maximal value.

Consequently, every set of temporal assertions about a resource z can be expressed in a uniform way as a set of *use assertions* over intervals of the form $\{z@[t_1, t'_1):q_1, \ldots, z@[t_n, t'_n):q_n\}$. This is a standard way to represent resource assertions.

From now on, let us assume that all assertions on resource variables are of this form. Let us also extend the notion of temporal assertions in a chronicle (Definition 14.6) to include both temporal assertions on state variables and temporal assertions on resource variables. The definition of a chronicle on a set of state and resource variables is unchanged, i.e., it is a pair $\Phi = (\mathcal{F}, \mathcal{C})$, where $\mathcal{F}$ is a set of temporal assertions on state and resource variables and $\mathcal{C}$ is a set of object constraints and temporal constraints. However, the definition of a consistent chronicle (Definition 14.8) now has to be extended with specific requirements for the consistency of a set of temporal assertions about resource variables.

We are assuming that distinct resources are completely independent: drawing on a resource z does not affect another resource z'. Every assertion on a resource concerns a single resource variable.[7] Hence, two assertions on two distinct resource variables can never be mutually inconsistent. Consistency requirements are needed only for assertions about the same resource variable. Let $\Phi = (\mathcal{F}, \mathcal{C})$ be a chronicle; z is one of its resource variables, and $R_z = \{z@[t_1, t'_1):q_1, \ldots, z@[t_n, t'_n):q_n\}$ is the set of all temporal assertions in $\mathcal{F}$ on the resource variable z. A pair of assertions $z@[t_i, t'_i):q_i$ and $z@[t_j, t'_j):q_j$ in R_z is *possibly intersecting* iff the two intervals $[t_i, t'_i)$ and $[t_j, t'_j)$ are possibly intersecting.[8] This is the case iff there is no constraint in $\mathcal{C}$ or entailed from $\mathcal{C}$ that makes the two intervals disjoint, i.e., $\mathcal{C} \not\models (t'_i < t_j)$, and $\mathcal{C} \not\models (t'_j < t_i)$.

Similarly, a set of assertions $\{z@[t_i, t'_i):q_i \mid i \in I\} \subseteq R_z$ is possibly intersecting iff the corresponding set of intervals $\{[t_i, t'_i) \mid i \in I\}$ is possibly intersecting.

Definition 15.2 A set R_z of temporal assertions about the resource variable z is *conflicting* iff there is a possibly intersecting set of assertions $\{z@[t_i, t'_i):q_i \mid i \in I\} \subseteq R_z$ such that $\sum_{i \in I} q_i > Q$. ■

Intuitively there is a conflict if there exists a possible instantiation of Φ such that there is a time point at which some actions are attempting to use more of a resource than its total capacity. In other words, the set R_z is conflicting if there is a set of values for the temporal variables in R_z that are consistent with $\mathcal{C}$ such that at some point in $\bigcap_{i \in I}[t_i, t'_i)$, the total quantities of the resource z used according to R_z exceed the total capacity Q of z. It is interesting to characterize a conflicting set from the properties of *pairs* of assertions in this set, as in the following proposition.

7. An action that needs two resources z and z' leads to two distinct assertions.
8. See Definitions 13.1 and 13.2 and Propositions 13.4 and 13.5 for possibly intersecting intervals.

Proposition 15.1 *A set R_z of temporal assertions on the resource variable z is conflicting iff there is a subset $\{z@[t_i, t_i'):q_i \mid i \in I\} \subseteq R_z$ such that every pair $i, j \in I$ is possibly intersecting, and $\sum_{i \in I} q_i > Q$.*

Proof The proof for Proposition 15.1 follows directly from Proposition 13.5. ■

Definition 15.3 A chronicle $\Phi = (\mathcal{F}, \mathcal{C})$ is *consistent* iff all its temporal assertions on state variables are consistent (in the sense specified in Definition 14.8) and none of its sets of temporal assertions on resource variables is conflicting. ■

This definition of the consistency of chronicles extends Definition 14.8. We now can apply, unchanged, Definitions 14.9 through 14.11, to chronicles with resources supporting other chronicles. This is due to the fact that a temporal assertion on a resource variable does not need an enabler—it requires only a nonconflicting set of assertions. Hence, all our constructs about planning operators and plans as chronicles (Section 14.3.2) still hold with resource variables.

Example 15.4 Let us consider a DWR domain where the space available in each location for loading and unloading robots is described as a resource that gives the number of available loading/unloading spots in a location. Let us denote by space(l) the corresponding resource variable in location l, i.e., the number of loading/unloading spots available at location l. Each load/unload operation requires one such spot. If space(l) > 1, then there can be more than one robot in location l. The move operator for this domain (see Figure 15.5) is:

$$\begin{aligned}\text{move}(t_s, t_e, t_1, t_2, r, l, l') = \{&\text{robot-location}(r)@t_s : (l, \text{routes}),\\ &\text{robot-location}(r)@[t_s, t_e] : \text{routes},\\ &\text{robot-location}(r)@t_e : (\text{routes}, l'),\\ &\text{space}(l)@t_1 : +1,\\ &\text{space}(l')@t_2 : -1,\\ &t_s < t_1 < t_2 < t_e,\\ &\text{adjacent}(l, l') \ \}\end{aligned}$$

This operator says that when the robot leaves location l there is one more spot available in l at t_1 than the current value of space(l) before applying the move operator, and when the robot arrives at l' there is one position less in l' at t_2 than before t_2. ■

In Example 15.4, robot-location(r) is a state variable, and space(l) is a resource variable. It is important to note that the value of a state variable can be an object variable, e.g., the value of robot-location(r) is l or l'. However, we are restricting the relative values of increases, decreases, and uses of a resource variable to be integer constants.

An additional element needs to be specified in the example: the total capacity of each resource. One way to do this is to specify a fixed total capacity for the

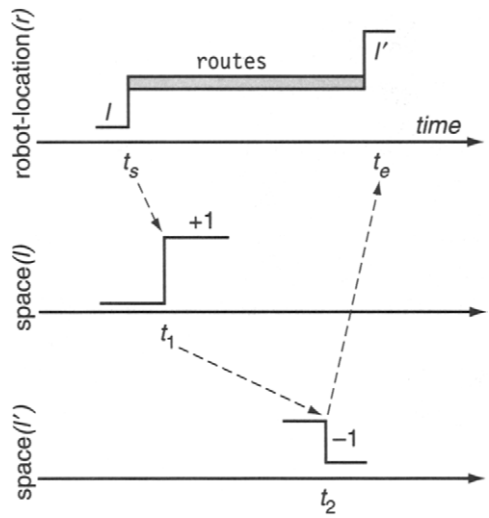

Figure 15.5 A move operator with resource variables.

set of resources {space(l) | l being a location}, e.g., $Q_{\text{space}} = 12$; i.e., there are at most 12 loading/unloading positions in each location. If a location, say loc1, has only 4 positions, then the initial chronicle will contain the temporal assertion space(loc1)@$[0, \infty)$:8. Another equivalent way to specify the total capacity of each resource is to allow this capacity to be given as a function of the object variable l for the resource space(l), i.e., $Q_{\text{space}}(\text{loc1}) = 4$, $Q_{\text{space}}(\text{loc2}) = 12$, etc.

A formal language for temporal planning domains with resources will have to integrate either form of these specifications as part of some initial declarations of object variables, state variables and their respective domains, and resource variables and their total capacities. Additional constraints such as rigid relations, initial resource profiles, etc., are specified as part of the initial chronicle.

Given these definitions, our previous specifications of planning domains and problems (Section 14.3.3) stand almost unchanged. A planning *domain* with time and resources is a pair $\mathcal{D} = (\Lambda_\Phi, \mathcal{O})$, where $\mathcal{O}$ is a set of temporal planning operators with resources (as in Example 15.4), and Λ_Φ is the set of all chronicles that can be defined with temporal assertions and constraints on the state and resource variables of X and Z, respectively, using the constants and the temporal and object variables of the representation.

A planning *problem* on $\mathcal{D}$ is a tuple $\mathcal{P} = (\mathcal{D}, \Phi_0, \Phi_g)$, where Φ_0 is a consistent chronicle that represents an initial scenario describing the initial state of the domain, the initial resource profiles, and the expected evolution that will take place independently of the actions to be planned, and Φ_g is a consistent chronicle that represents the goals of the problem. The *statement* of a problem $\mathcal{P}$ is given by $P = (\mathcal{O}, \Phi_0, \Phi_g)$.

A *solution plan* for the problem P is a set $\pi = \{a_1, \ldots, a_n\}$ of actions, each being an instance of an operator in $\mathcal{O}$, such that there is a chronicle in $\gamma(\Phi, \pi)$ that entails Φ_g.

Note that this formulation of a planning problem does not consider optimization criteria; it only seeks a feasible plan. Consequently, the approach of the chronicle planning procedure CP (Figure 14.7) is applicable here with an important extension: instead of dealing with only two types of flaws—open goals and threats—we also have to detect and manage *resource conflict flaws.*

15.4.2 Detecting Resource Conflicts

The chronicle planning procedure CP maintains a current chronicle $\Phi = (\mathcal{F}, \mathcal{C})$, which is refined successively for various types of flaws. Let us first focus on how to detect resource conflict flaws in Φ.

A resource conflict flaw is by definition a conflicting set of temporal assertions on a resource variable in the chronicle $\Phi = (\mathcal{F}, \mathcal{C})$. Let $R_z = \{z@[t_1, t'_1):q_1, \ldots, z@[t_n, t'_n):q_n\}$ be the set of all temporal assertions in $\mathcal{F}$ about the resource variable z. Let us associate with the variable z an undirected graph $H_z = (V, E)$ such that the following hold.

- A vertex $v_i \in V$ corresponds to an assertion $z@[t_i, t'_i):q_i$ of R_z.
- There is an edge $(v_i, v_j) \in E$ iff the two intervals $[t_i, t'_i)$ and $[t_j, t'_j)$ are possibly intersecting, given the constraints in $\mathcal{C}$.

H_z is called the graph of *Possibly Intersecting Assertions* (PIA) for z.

A set of vertices $U \subseteq V$ is *overconsuming* when $\sum_{i \in U} q_i \geq Q$, Q being the total capacity of the resource z. Consequently, the set R_z is conflicting iff H_z has an *overconsuming clique.* Recall that a clique in a graph is a subset of vertices that are pairwise adjacent. If U is a clique, then any subset of U is also a clique.

In order to solve resource conflicts, we are interested not in detecting all overconsuming cliques but in detecting only those that are minimal, in the set inclusion sense. This is will become clear in Section 15.4.3.

Definition 15.4 A *Minimal Critical Set* (MCS) for a resource z is a set of vertices $U \subseteq V$ such that U is an overconsuming clique and no proper subset $U' \subset U$ is overconsuming. ■

Example 15.5 Consider the following set of assertions for a resource z whose total capacity is $Q = 100$:

$$R_z = \{\, z@[t_1, t'_1):50, \quad z@[t_2, t'_2):60, \quad z@[t_3, t'_3):20, \quad z@[t_4, t'_4):50,$$
$$z@[t_5, t'_5):50, \quad z@[t_6, t'_6):70, \quad z@[t_7, t'_7):40\}$$

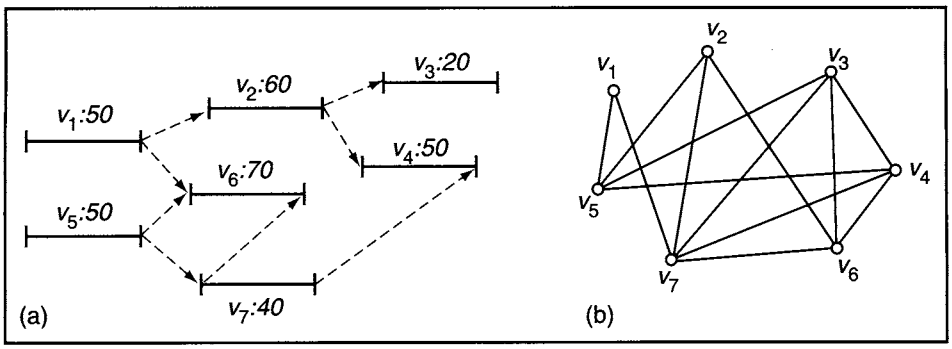

Figure 15.6 A possible intersection graph for a set of assertions for a resource whose total capacity is $Q = 100$. All the precedence constraints are represented as dotted lines in (a), i.e., v_7 ends before *or* after v_4 starts, but necessarily before the end of v_4.

Suppose that $\mathcal{C}$ contains the constraints $t'_1 < t_2$, $t'_1 < t_6$, $t'_2 < t_3$, $t'_2 < t_4$, $t'_5 < t_6$, $t'_5 < t_7$, $t_7 < t'_6$, and $t'_7 < t'_4$, in addition to the constraints on the end points of intervals, $\forall i, t_i < t'_i$. These assertions and constraints are depicted in Figure 15.6 (a). The corresponding PIA graph H_z is given in Figure 15.6 (b). In this figure an assertion $z@[t_i, t'_i):q_i$ is denoted as a temporal interval labeled $v_i:q_i$, for the corresponding vertex v_i of H_z. Note that $\{v_1, v_5\}$ is a clique, but it is not overconsuming; $\{v_3, v_4, v_6, v_7\}$ is an overconsuming clique, but it is not minimal because $\{v_6, v_7\}$, $\{v_4, v_6\}$ and $\{v_3, v_4, v_7\}$ are MCSs for the resource z. ■

The algorithm MCS-expand (Figure 15.7) performs a depth-first greedy search that detects all MCSs in a PIA graph, if any. Its main data structure is a pair (clique(p), pending(p)) for each node p of the search tree, where clique(p) is the

```
MCS-expand(p)
    for each v_i ∈ pending(p) do
        add a new node m_i successor of p
        pending(m_i) ← {v_j ∈ pending(p) | j < i and (v_i, v_j) ∈ E}
        clique(m_i) ← clique(p) ∪ {v_i}
        if clique(m_i) is overconsuming then MCS ← MCS ∪ clique(m_i)
        else if pending(m_i) ≠ ∅ then MCS-expand(m_i)
end
```

Figure 15.7 Searching for minimal critical sets.

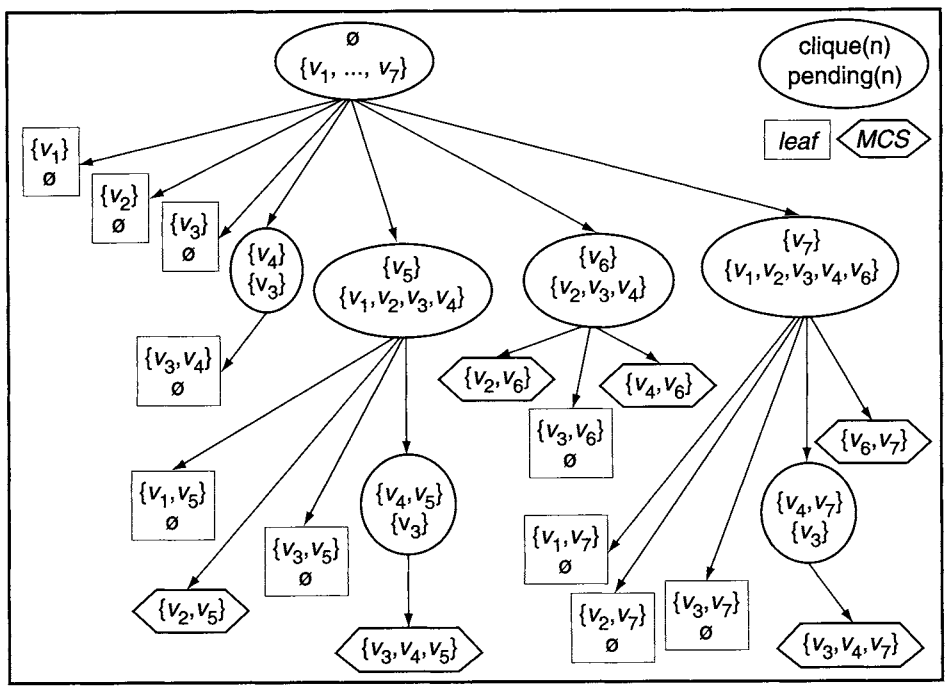

Figure 15.8 Search tree for detecting MCSs.

current clique being examined, and pending(p) is the set of candidate vertices of H_z that will be used in successor nodes to expand this current clique. The root node is associated with the pair ($\emptyset$, V). For a leaf node n, either pending(n) is an empty set of vertices (in which case n is a dead end for the search) or clique(n) is an MCS. The global variable MCS is a set of MCSs. It is initially empty; when the algorithm stops, it contains all found MCSs. To simplify the notations, the set of vertices in H_z is supposed to be totally ordered in some arbitrary way, e.g., in the increasing order of the index i. This is used in the update of pending(m_i) in MCS-expand.

Example 15.6 The search tree of algorithm MCS-expand for the PIA graph of Example 15.5 is given in Figure 15.8. Each internal node p is labeled by the two sets clique(p) and pending(p). Leaves corresponding to MCSs are drawn in hexagons. The algorithm MCS-expand stops in this example with MCS = $\{\{v_2, v_5\}, \{v_3, v_4, v_5\}, \{v_2, v_6\}, \{v_4, v_6\}, \{v_3, v_4, v_7\}, \{v_6, v_7\}\}$. ■

Proposition 15.2 *Algorithm MCS-expand is sound and complete: it stops and returns all minimal critical sets in a PIA graph.*

15.4.3 Managing Resource Conflict Flaws

Let $U = \{z@[t_i, t_i'):q_i \mid i \in I\}$ be an MCS for the resource z in a chronicle Φ. Any constraint such as $t_i' < t_j$, for $i, j \in I$, is a resolver of this flaw in Φ. Indeed, such a constraint makes the two intervals $[t_i, t_i')$ and $[t_j, t_j')$ disjoint. Adding such a constraint removes the edge (v_i, v_j) in H_z. Hence U is not a clique anymore, and the assertions in U are no longer conflicting.

Furthermore, if U is the only MCS of z, then a constraint $t_i' < t_j$ makes the set R_z *nonconflicting*. This is the case because for any other overconsuming clique U': U' is not minimal, and $U \subset U'$. Hence, U' ceases being a clique when U is not a clique. This is the advantage of relying on MCSs instead of using any overconsuming cliques to deal with resource conflicts.

Each pair of vertices (v_i, v_j) in an MCS U corresponds to a potential resolver for U: $t_i' < t_j$. However, if U has k vertices, we do not necessarily have $k \times (k-1)$ consistent and independent resolvers for U. Some constraints $t_i' < t_j$ may not be consistent with $\mathcal{C}$. Other such constraints can be overconstraining in the sense that they entail with $\mathcal{C}$ more constraints than strictly needed to resolve the flaw U. An overconstraining resolver may not be desirable in the search strategy.

Example 15.7 Consider the MCS $U = \{v_3, v_4, v_7\}$ in Example 15.5 (see page 367). There are six possible constraints of the form $t_i' < t_j$, for $i, j \in \{3, 4, 7\}$, $i \neq j$, as potential resolvers for U. But the constraint $t_4' < t_7$ is inconsistent with $\mathcal{C}$ because $\mathcal{C}$ contains the constraints $t_7' < t_4'$ and $t_i < t_i', \forall i$. Furthermore, the constraint $t_4' < t_3$ is overconstraining because it implies $t_7' < t_3$ (through $t_7' < t_4'$). A solution for the resource conflict U with the constraint $t_4' < t_3$ is necessarily a particular case of a solution of U with the constraint $t_7' < t_3$. Hence, we can remove $t_4' < t_3$ from the set of resolvers of U. Finally, we end up with the following set of resolvers: $\{t_3' < t_4, t_3' < t_7, t_7' < t_3, t_7' < t_4\}$. ■

Let $U = \{z@[t_i, t_i'):q_i \mid i \in I\}$ be an MCS for the resource z in a chronicle Φ, and let $\rho = \{t_i' < t_j \mid i, j \in I, i \neq j \text{ and } t_i' < t_j \text{ consistent with } \mathcal{C}\}$. Each constraint of ρ, when added to $\mathcal{C}$ in a refinement step, resolves the flaw U. However, the constraints in ρ are not all independent. As illustrated in Example 15.7, a constraint may entail another one with respect to $\mathcal{C}$. A set of constraints ρ' is *equivalent* to ρ, given $\mathcal{C}$, iff each constraint of ρ when added to $\mathcal{C}$ entails a constraint in ρ', and, symmetrically, each constraint in ρ' when added to $\mathcal{C}$ entails some constraint in ρ. It is possible to show that there exists a unique subset of ρ that is equivalent to ρ and is minimal under set inclusion. This minimal set of resolvers is desirable because it leads to a smaller branching factor for resolving the resource flaw characterized by U. It can be found in time $O(|U|^3)$ by a procedure that removes from the set ρ the overconstraining resolvers. Note, however, that any set of consistent resolvers is sufficient for resolving U, even if some of its elements are overconstraining.

We are now ready to extend the chronicle planning procedure CP of Chapter 14 into a procedure CPR that manages resources (Figure 15.9). Here CPR maintains

```
CPR(Φ, G, K, M, π)
    if G = K = M = ∅ then return(π)
    perform the three following steps in any order
        if G ≠ ∅ then do
          select any α ∈ G
          if θ(α/Φ) ≠ ∅ then return(CPR(Φ, G − {α}, K ∪ θ(α/Φ), M, π))
          else do
              relevant ← {a | a applicable to Φ and has a provider for α}
              if relevant = ∅ then return(failure)
              nondeterministically choose a ∈ relevant
              M' ← the update of M with respect to Φ ∪ (F(a), C(a))
              return(CPR(ΦU(F(a), C(a)), GUF(a), KU{θ(a/Φ)}, M', πU{a}))
        if K ≠ ∅ then do
            select any C ∈ K
            threat-resolvers ← {φ ∈ C | φ consistent with Φ}
            if threat-resolvers = ∅ then return(failure)
            nondeterministically choose φ ∈ threat-resolvers
            return(CPR(Φ ∪ φ, G, K − C, M, π))
        if M ≠ ∅ then do
            select U ∈ M
            resource-resolvers ← {φ resolver of U | φ is consistent with Φ}
            if resource-resolvers = ∅ then return(failure)
            nondeterministically choose φ ∈ resource-resolvers
            M' ← the update of M with respect to Φ ∪ φ
            return(CPR(Φ ∪ φ, G, K, M', π))
end
```

Figure 15.9 CPR, a procedure for chronicle planning with resources.

a tuple $\Omega = (\Phi, G, \mathcal{K}, \mathcal{M}, \pi)$. As in the previous chapter, $\Phi = (\mathcal{F}, \mathcal{C})$ is the current chronicle, G is a set of assertions corresponding to the current open goals, $\mathcal{K} = \{C_1, \ldots, C_i\}$ is a set of pending sets of enablers, and π is the current plan.

The new component in Ω is $\mathcal{M}$, the current set of MCSs in Φ. Every update in Φ, such as adding new time points and constraints, may change $\mathcal{M}$. The set $\mathcal{M}$ can be maintained incrementally. If an update does not modify R_z and does not add a new constraint on a time point in R_z, then the MCSs of z in $\mathcal{M}$ are unchanged.

Each recursion of the procedure CPR on the current $\Omega = (\Phi, G, \mathcal{K}, \mathcal{M}, \pi)$ involves three main steps:

- Solving open goal flaws when $G \neq \emptyset$

- Solving threat flaws when $\mathcal{K} \neq \emptyset$
- Solving resource conflict flaws when $\mathcal{M} \neq \emptyset$

Note that the consistency test for threat resolvers is restricted to state variables, i.e., it is not concerned with resource conflicts.

As in Section 14.3.4, it is desirable to handle resource conflicts with a *resource constraints manager*. Its tasks will be to maintain incrementally $\mathcal{M}$ with respect to any update in Φ and to find a minimal set ρ_U of consistent resolvers for each MCS U. Here also a meta-CSP can be used to handle the set $\mathcal{M}$ of MCSs. A variable of that CSP corresponds to an MCS U; its domain is ρ_U. A backtracking in CPR occurs when that domain is empty. A constraint propagation and an update take place when that domain contains a single resolver. Otherwise, ρ_U is maintained according to the updates in the time-map manager and consequently in $\mathcal{M}$. Eventually ρ_U can be filtered out with other pending MCSs with an arc-consistency algorithm.

The discussion on the control issues of CP (Section 14.3.5) applies here as well. We can use heuristics similar to the variable ordering heuristics and the value selection heuristics (Section 8.4.1). These heuristics lead the algorithm to select as the next resource flaw the most constrained MCS, i.e., the one that has the smallest number of resolvers, and to choose as a resolver for the selected flaw a least constraining one.

Finally, the partial-order scheme, mentioned for state variables, can also be applied to state and resource variables. From the analysis of planning operators, it is possible to define a partial order on state and resource variables. Flaws are processed according to this order: no flaw is considered for a variable y, which can be either a state variable or a resource variable, until all flaws for all variables that precede y have been addressed. The partial order can guarantee that the resolution of flaws on y does not introduce new flaws for state variables that precede y.

15.5 Discussion and Historical Remarks

Scheduling is a broad research area. It has been a very active field within operation research for over 50 years. There is probably much more published material on the single topic of deterministic machine scheduling problems than published material on AI planning. Sections 15.2 and 15.3 are only a brief introduction to the field. Numerous books (e.g., [43, 200]) devote more space to the details and the techniques of scheduling than allowed here. Comprehensive surveys are also available (e.g., [21, 247, 355]).

The purpose of this chapter was only to give to the reader interested in extending automated planning to actions with resources the background and necessary entry points.

More and more planning systems rely on operation research tools and approaches such as linear programming [556], integer programming [539], and mixed ILP [155]. Many scheduling systems [60, 115, 177, 197, 492] deal with concerns close

to those of planning and draw on AI techniques in order to handle more flexible activity models. But, as discussed in Smith *et al.* [484], the gap between planning and scheduling requires further research.

One of the first planners to offer resource-handling capabilities was probably FORBIN [143], an HTN planner. Other contributions involving scheduling techniques within HTNs have been pursued, notably within the O-Plan [162] and SIPE-2 planners [550, 552]. For example, in SIPE-2 the temporal and resource constraints are processed by plan critics during the critic phase. The OPIS system [491] is used, among others, as a scheduling critic. A critic can trigger a backtrack if constraints are unsatisfiable, or it can instantiate or further constrain variables. This is done together with the management of links in the task network.

The time-oriented view for temporal planning opened a promising avenue of development based on constraint satisfaction techniques. Systems such as HSTS and following developments for space applications [198, 199, 408] and the IxTeT system [346] rely on CSPs for handling resource and temporal constraints. The MCS approach for handling resource conflicts (Section 15.4.2) was developed in the context of IxTeT, drawing on maximal clique algorithms for particular graphs [212, 265]. This approach has been further extended for continuous resources or reservoirs and improved [344, 345].

It can be expected that this area of research will become essential in automated planning. The international planning conferences, AIPS and then ICAPS,[9] have been renamed to emphasize planning and scheduling. However, more work is needed to disseminate to the planning community benchmarks involving resources and action scheduling, as well as tools for handling them efficiently.

15.6 Exercises

15.1 In Exercise 14.11, suppose we modify the planning problem by assuming that the water flow is a resource with a maximum capacity of 2, and each fill operation uses 1 unit of this resource during the time it is executing.

(a) Draw the PIA graph. What minimal overconsuming cliques does it have?

(b) How many different ways are there to resolve the overconsumption? What does this tell us about the number of different possible solution plans?

(c) Draw a chronicle in which the overconsumption is resolved.

(d) Write the chronicle as a set of temporal assertions and a set of constraints.

15.2 Suppose we extend the DWR domain by assuming that for each container c there is a numeric value weight(c). For each robot r there is a resource variable called

9. The International Conference on Automated Planning and Scheduling; see http://www.icaps-conference.org/.

capacity(r), which represents the amount of weight that the robot can carry, with capacity(r) = 10 when the robot is completely unloaded. A robot may hold more than one container if the sum of the weights of the containers does not exceed its capacity.

(a) Extend the load and unload operators of Exercise 14.8 so they model this new situation.

(b) Write a chronicle that contains the following actions, starting with a robot r1 that is initially empty.

Load container c1 onto r1 at time t_1.
Unload container c1 from r1 at time t_1'.
Load container c2 onto r1 at time t_2.
Unload container c2 from r1 at time t_2'.
Load container c3 onto r1 at time t_3.
Unload container c3 from r1 at time v_3'.
Load container c4 onto r1 at time t_4.
Unload container c4 from r1 at time t_4'.

The weights of the containers are weight($c1$) = 5, weight($c2$) = 3, weight($c3$) = 3, and weight($c4$) = 4. There are the following time constraints: $t_i < t_i' \; \forall i$, and $t_2' < t_3$.

(c) Draw the intersection graph for part (b) above, and find all maximal cliques.

(d) Describe each of the possible ways to resolve the overconsumption(s) in part (c).

(e) Draw a chronicle in which the overconsumption is resolved.

(f) Write the chronicle as a set of temporal assertions and a set of constraints.