Contents

1

Deterministic machine scheduling problems

Eugene L. Lawler
University of California, Berkeley

Jan Karel Lenstra
Centrum Wiskunde & Informatica

Alexander H.G. Rinnooy Kan
University of Amsterdam

David B. Shmoys
Cornell University

Scheduling theory is something of a jungle, encompassing a bewildering variety of problem types. In this book we shall be concerned with only a limited variety of the animals in this jungle, namely those which are *deterministic machine scheduling problems*. With the right techniques, some of these animals are easily domesticated. But others are quite intractable and submit to the taming techniques of *combinatorial optimization* with great difficulty. Before learning how to train these animals, we must be able to identify them and describe them. That is the purpose of this chapter.

1.1. Machines, jobs, and schedules

The number of pans to be heated in the preparation of a gourmet dinner exceeds the number of burners available. More ships are in a harbor than there are quays for unloading them. Tasks assigned to a multiprogrammed computer system compete for processing on a timesharing basis. Planes requesting permission to use an airport runway are assigned places in a queue. Jobs in a job shop contend for processing by various machines.

Each of these situations suggests a problem in which limited resources must be allocated over time to a set of activities. Adopting the terminology of the job shop, the resources can be described as *machines* and the activities as *jobs*. Pans, ships, tasks and planes can be thought of as jobs, and burners, quays, central processing units and runways as machines. Generally speaking, such problems involving machines and jobs are in the domain of *machine scheduling theory*.

Thus every instance of a scheduling problem involves a set of m machines $1, 2, ..., m$, comprising the *machine environment*, and a set of n jobs $1, 2, ..., n$, each requiring processing by one or more of the machines. As we shall explain more precisely later on, jobs have fixed *processing requirements*, which determine how long they must be processed. A processing requirement does not change in time and is unaffected by when the processing of a job is performed or by when the processing of other jobs occurs.

Also, as we shall describe, there may be constraints, in the form of *release dates* and *deadlines*, on the time period in which a given job is available for processing. There may be *precedence constraints* on the order in which jobs are processed. And one may allow *preemption*, or interruptions in the processing of jobs. But whatever the nature of these *job characteristics*, it is always understood that at any given point in time *no machine can process more than one job* and *no job can be processed by more than one machine*.

In this book we shall deal with only *deterministic* scheduling problems, in which all data are known precisely. There will be no uncertainty about the processing requirements of the jobs or about the other job characteristics, which are all known in advance of scheduling. Otherwise, we would be dealing with *stochastic* scheduling problems, which would involve us in *probability theory* and, most particularly, with *queueing theory* rather than *combinatorial optimization*. In Chapter 14, we provide a small taste of stochastic scheduling.

The output of a scheduling procedure is a *schedule*, which specifies exactly what job, if any, each machine works on at each point in time. A schedule can be represented by a *Gantt chart*, a device employed by managers and industrial engineers since the First World War. In the chart shown in Figure 1.1, the horizontal axis indicates time and each band is identified with a machine. The intervals in which a machine is assigned to no job are shaded; such periods are known as *idle time*.

Some schedules are better than others, and we seek to find a best schedule with respect to a specified *optimality criterion*. In any schedule there is a well defined *completion time* C_j for each job j, the time at which it is last processed. A value

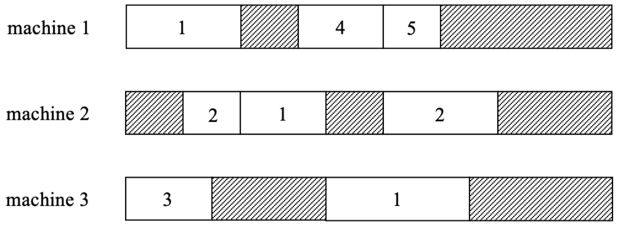


Figure 1.1. Gantt chart.

$F(C_1, C_2, ..., C_n)$ is assigned to each possible schedule, and our goal is to find a schedule for which this value is minimized. We shall only be concerned with optimality criteria for which the function F is *monotonic*. That is:

$$\text{if } C_j \leq C'_j \ (j = 1, ..., n), \text{ then } F(C_1, ..., C_n) \leq F(C'_1, ..., C'_n). \tag{1.1}$$

Now that we know something about what all the machine scheduling problems we shall deal with have in common, it is time to proceed with a systematic description and classification of problem types. As we shall elaborate in the remainder of this chapter, our description is based on three components: machine environment, job characteristics, and optimality criterion. Unless stated otherwise, all numerical data used in specifying a problem instance will be assumed to be integral.

1.2. Single machines and parallel machines

Every machine scheduling problem has a specified environment of m machines i $(i = 1, ..., m)$ and a specified set of n jobs j $(j = 1, ..., n)$, requiring processing by the machines. For each job j there is a specified *processing requirement* p_j $(j = 1, ..., n)$. (Later, in multi-operation models, we shall have a processing requirement for each of the separate operations comprising a job.) Recall that the processing requirement is independent of when the job is processed in a schedule. In particular, the processing requirement does not depend on the identity of the jobs that precede it and follow it. This means that we do not allow for the possibility of *sequence dependent setup and change-over times*. To do so would take us into the realm of the traveling salesman problem, which we now should like to avoid; see Exercise 1.1.

On occasion, we shall consider problems in which the processing requirements can take on only a restricted set of values. The simplest such case is that in which each $p_j = 1$. Clearly, imposing such a constraint can only make a problem easier to solve. Hence a *unit-time* scheduling problem is a *specialization* of the problem with *general* processing requirements, in that any solution procedure for the latter can be used to solve the former. Ordinarily, we shall make the following assumption:

Processing requirements are arbitrary nonnegative integral values. (1.2)

Condition (1.2) is an assumption that we shall make *by default* about any given problem, unless we explicitly state otherwise. Other default assumptions about job characteristics are:

All jobs are available for processing at time 0, but not before. (1.3)

(All jobs have *release dates* equal to 0.)

It is feasible to process any job at any time after time 0. (1.4)

(There are *no deadlines*.)

No constraints are imposed on the order in which jobs may be processed. (1.5)

(Jobs are *independent*; there are *no precedence constraints*.)

Once a machine begins processing a job, it must process it to completion. (1.6)

(Scheduling is *nonpreemptive*.)

The simplest machine environment is simply that of a *single machine* $(m = 1)$, which is the subject of Chapters 3–6 of this book. Here the processing requirement p_j simply indicates the amount of time for which the single machine must process job j.

Because of assumption (1.6), there is a well-defined sequence in which the jobs are processed in any given schedule, with each job in the sequence being completed before the next job is started. There are clearly an infinite number of schedules for the same sequence. But for *any given sequence* of the jobs, there is a *unique best schedule* in which the completion time of each and every job is as early as it is in any other schedule for the same sequence. We call this a *left-justified* schedule. For example, if the jobs are processed in the sequence $1,2,...,n$, it is obtained by processing job 1 in the time interval $[0, p_1]$, job 2 in $[p_1, p_1 + p_2], ...$, and job n in $[p_1 + ... + p_{n-1}, p_1 + ... + p_n]$. It follows from the monotonicity condition (1.1) that if any schedule for this sequence is optimal, this one is. Thus the single-machine scheduling problem reduces to a *sequencing* problem $-$ that of determining which one of $n!$ possible sequences yields a schedule that is best with respect to the given optimality criterion.

The single-machine environment is a special case of *identical parallel machines*. In the case of this machine environment, a job may be performed on any one of the $m \geq 1$ available machines and each machine i requires the same time p_j to process job j. A computing center with m identical computers provides an example of such a machine environment.

More generally, the m parallel machines may have different *speeds* $s_i \geq 0$ $(i = 1,...,m)$, with machine i requiring p_j/s_i time units to perform job j. These are known as *uniform parallel machines*. A computer center with m computers, fully compatible and identical in all essential characteristics except for speed, provides an example of such a machine environment. Note that identical parallel machines are a special case of uniform parallel machines, in that it is understood that $s_i = 1$ for each machine i.

A still more general case of parallel machines is that of *unrelated parallel machines*, in which there are mn specified parameters $s_{ij} \geq 0$ $(i = 1,...,m, j = 1,...,n)$, with machine i requiring p_j/s_{ij} time units to perform job j. A computer center with m different computers, some good for certain tasks and less good for others, provides an example of such a machine environment. Identical, uniform and unrelated parallel machines are the subject of Chapters 7–10 of this book.

Note that the unit-time restriction does indeed provide meaningful specializations of identical and uniform parallel machine problems. However, in the case of unrelated parallel machines, the specialization is bogus: every unrelated parallel machine problem with arbitrary processing requirements p_j and speeds s_{ij} is equivalent to a unit-time problem with $p'_j = 1$ and $s'_{ij} = s_{ij}/p_j$. Unrelated parallel machine problems are in general much more difficult than identical and uniform machine problems and demand very different solution techniques. They also have a close relation with the multi-operation models we shall describe in Chapters 11–13, and some discussion of unrelated parallel machines is deferred to Chapters 7–10.

Again, because of condition (1.6), each schedule for a set of parallel machines provides a well-defined sequence in which the jobs are processed by any given machine. By carrying out the same kind of analysis we did for single machines, we see that parallel machine scheduling also reduces to a sequencing problem of sorts: we need consider only left-justified schedules, and there are as many of those as there are ways to assign the n jobs to the m machines and to sequence the jobs assigned to each of the machines. As we see in Exercise 1.2, there are exactly $(n+m-1)!/(m-1)!$ possibilities in the case of m distinct machines.

Exercises

1.1. The legendary traveling salesman has to leave his home city and visit each of $n-1$ other cities exactly once, returning home at the end of his tour. He seeks to find a tour of minimum total length, given that the distance from city j to city k is a known positive value c_{jk}. Formulate the problem as a single-machine sequencing problem with $n+1$ jobs, where the time p_{jk} required to perform a given job k depends upon the identity of the job j that precedes it. The objective should be to find a sequence of jobs that minimizes the total length of the schedule.

1.2. Show that there are $(n+m-1)!/(m-1)!$ ways to assign n jobs to m distinct machines and to sequence the jobs for each machine. (*Hint*: How many ways are there to permute a set of $n+m-1$ objects, n of which are distinct and $m-1$ of which are identical?) How many possibilities are there if there are m_i identical machines of each of k different types (where $\sum_{i=1}^{k} m_i = m$)?

1.3. Release dates, deadlines, and precedence constraints

We have spoken of unit-time problems (all $p_j = 1$) as *specializations* that are obtained by departing from our default assumption (1.2). Now let us consider some *generalizations* that are obtained by departing from assumptions (1.3) and (1.4).

In (1.3) we assumed by default that all jobs are available for processing at time 0. Rather than restricting ourselves to such a *static* model, we might like to examine a *dynamic* model in which jobs are released for processing at various points in time. Each job j can have a *release date* $r_j \geq 0$ specified for it; in order for a schedule to be feasible, it must satisfy the constraint that $S_j \geq r_j$, where S_j is the *starting time* of job j, the time at which it is first processed. We shall assume that all release dates are given to us at the time we are to prepare a schedule. In practice, of course, this may not be realistic since release dates may not be known with much certainty. Keeping this in mind, we shall occasionally investigate to what extent a solution procedure actually requires full information about release dates and processing requirements of jobs that are to become available in the future. A scheduling procedure that at any time t does not require information about the jobs j with $r_j > t$ is said to be *on line* or *real time*.

Just as the starting time S_j of job j may be constrained by a release date r_j, its completion time C_j may be constrained by a *deadline* $\bar{d}_j$ with the requirement that $C_j \leq \bar{d}_j$, contrary to the default assumption (1.4). As we shall see, the presence of release dates and deadlines can only make problems more difficult to solve. However, any solution procedure that is effective for a scheduling problem with release dates and deadlines can be applied to the same problem when they are not present. So we are dealing with a strict generalization.

The observations we have made about the reduction of single and parallel machine scheduling problems remain valid under the imposition of release dates and deadlines. For any sequence there is a unique left-justified schedule, in which each job completion time is as early as in any other schedule consistent with the sequence. Thus, if jobs $1,...,n$ are to be processed by a certain machine in that order, each job should begin processing as early as possible, subject to the completion of its predecessor: job 1 should be processed in the time interval $[S_1, C_1]$, where $S_1 = r_1$ and $C_1 = S_1 + p_1$, and job j should be processed in the interval $[S_j, C_j]$, where $S_j = \max\{r_j, C_{j-1}\}$ and $C_j = S_j + p_j$, for $j = 2,...,n$. The difficulty is that a sequence might be infeasible, because one or more of the completion times in this schedule may violate a deadline.

Additional feasibility constraints may arise if the jobs are not *independent* but related by *precedence constraints*, contrary to (1.5). For example, a sizeable project like building a house may be broken down into a number of jobs, such as laying the foundation, erecting the walls, building the roof, which must be done in a certain order. If job j must precede job k, then we write $j \to k$. This is interpreted as meaning that job j must be completed before job k is started: $C_j \leq S_k$.

If precedence constraints exist, it is convenient to represent them by means of a *precedence digraph* G, with vertices $1,...,n$ corresponding to jobs. The existence of an arc (j,k) implies that $j \to k$. The number of arcs directed out of (into) a given vertex is said to be the *outdegree* (*indegree*) of the vertex. In order for the precedence constraints to be consistent, G must be *acyclic*, i.e., contain no directed cycles.

A digraph can be represented by its *adjacency matrix* $A = (a_{jk})$, where $a_{jk} = 1$ if (j,k) is an arc and $a_{jk} = 0$ otherwise. As is well known, a digraph is acyclic if and only if it is possible to number its vertices in such a way that each arc in the digraph extends from a lower numbered vertex to a higher numbered one. It follows that, for an appropriate numbering of the vertices, the adjacency matrix of a precedence digraph is *upper triangular*, with all 1's above the main diagonal.

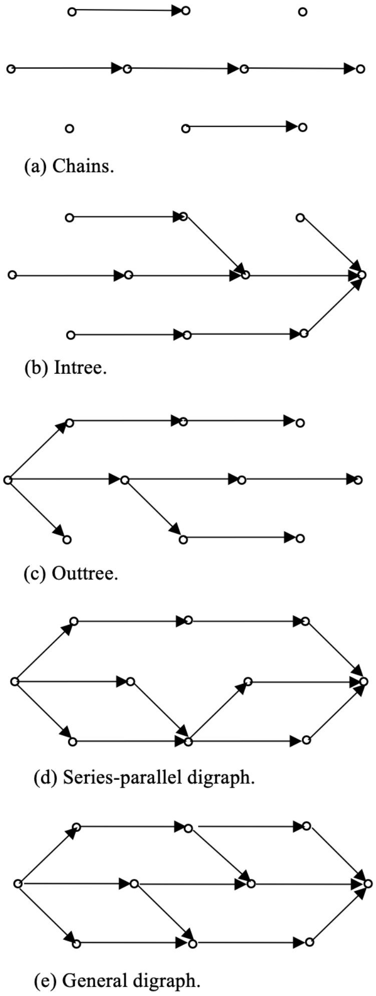
A digraph is said to be *transitive* if the existence of arcs (j,k) and (k,l) implies the existence of an arc (j,l). The *transitive closure* of a digraph G is the digraph G' obtained from G by adding all arcs missing from G that are implied by transitivity. It is easy to see that the transitive closure of any digraph is unique. On the other hand, G' is said to be a *transitive reduction* if G' can be obtained from G by successively removing arcs that are implied by transitivity until no more such arcs remain. In general, a digraph may have several transitive reductions, resulting from various choices in the successive arcs that are removed. For acyclic digraphs this is not the case; as we will see in Exercise 1.3, the transitive reduction of an acyclic digraph is unique.

There are various restricted classes of precedence constraints that we shall want to consider (cf. Figure 1.2). We say that precedence constraints are in the form of *chains* if they can be represented by a (transitively reduced) digraph in which each vertex has indegree at most 1 and outdegree at most 1. Chain-type precedence constraints are a special case of *intree* constraints and of *outtree* constraints, which can be represented by (transitively reduced) digraphs in which each vertex has outdegree at most 1 and indegree at most 1, respectively.

Suppose a widget is to be assembled from its component parts. One job is to bolt parts A and B together to form subassembly C, another is to fit parts D, E and F together to form subassembly G. After this is done, there is the job of putting subassemblies C and G together to form a larger subassembly H, and so on. It is readily seen that the jobs in this assembly process are related by intree precedence constraints. Similarly, the jobs involved in disassembly are related by outtree constraints.

More generally, we shall simply say that precedence constraints are *tree* constraints if they are either intree or outtree constraints. That is, the class of tree digraphs is the union of the classes of intree and outtree digraphs, whereas the class of chain digraphs is their intersection. Later on, in Chapter 5, we shall define a still more general class of precedence constraints, called *series-parallel*.

Note that our observations about the reduction of single-machine scheduling problems to sequencing problems remain essentially valid, even if release dates, deadlines and precedence constraints are all imposed. It is just that certain sequences may be infeasible because they violate precedence constraints. And some of the sequences that are consistent with the precedence constraints may be infeasible because deadlines are violated, even when each successive job in the sequence is scheduled to start as early as possible. See Exercise 1.4.



(a) Chains.

(b) Intree.

(c) Outtree.

(d) Series-parallel digraph.

(e) General digraph.

Figure 1.2. Several types of precedence constraints.

Exercises

1.3. (a) Give an example of a digraph that has more than one transitive eduction. (Three vertices suffice.)

(b) Show that the transitive reduction of an acyclic digraph is unique.

1.4. We have seen in Exercise 1.2 that we may limit our search for the solution to a parallel machine scheduling problem to the $(n+m-1)!/(m-1)!$ possible ways in which it is possible to assign jobs to machines and to sequence the jobs assigned to each machine. Some of these possibilities may be infeasible because they violate precedence constraints.

(a) Show that precedence constraints are not necessarily satisfied if they are independently satisfied by each of the m sequences of jobs that are to be performed by the machines.

(b) Show that, if a given set of m sequences of jobs on machines satisfies the precedence constraints, left-justification yields a unique best schedule, in which each job completion time is as early as it is in any other schedule consistent with the sequences.

1.4. Preemption

Up to this point we have considered only *nonpreemptive scheduling*, in accord with default condition (1.6). Sometimes it is realistic to permit *preemption* or *job splitting*. If preemption is allowed, we will take that to imply that the processing of a job may be interrupted arbitrarily often and resumed at a later time on the same machine, or in the case of parallel machines at any time on a different machine. We assume there is no cost or loss of efficiency associated with preemption. All that matters is that the total processing requirement of a job be satisfied. Thus in the most general case of unrelated parallel machines, if x_{ij} is the total amount of time that machine i processes job j, then it is necessary that

$$\sum_{i=1}^{m} s_{ij}x_{ij} = p_j \text{ for } j = 1,...,n.$$

An obvious example of preemption is timesharing by jobs in a multiprogrammed computer system. In this case, the overhead involved in job swapping is nonneglible. But it is relatively small compared with the total amount of actual processing done, and for the purpose of scheduling it is not unreasonable to ignore this overhead.

Every feasible nonpreemptive schedule is, of course, also a feasible schedule when preemption is permitted. And there are scheduling problems for which there is no advantage to preemption, in the sense that there always exists a nonpreemptive schedule as good as any schedule involving preemption. For example, there is no advantage to preemption in a single-machine problem in which all release dates are 0; see Exercise 1.5. On the other hand, a problem may be such that no feasible nonpreemptive schedule exists, even though there are many feasible preemptive ones;

see Exercise 1.6.

It follows that preemptive scheduling problems are not simply sequencing problems: optimization may require a search over a much larger number of possible schedules. There is no general rule for predicting whether the nonpreemptive version of a problem is either easier or harder than the preemptive version of the same problem. And certainly one cannot make a general statement that a nonpreemptive optimization procedure can be applied to solve a preemptive problem, or vice versa. Thus permitting preemption neither generalizes nor specializes the default assumption (1.6).

In general, it is a difficult problem to find, from among all optimal preemptive schedules, a schedule with the smallest possible number of preemptions. However, we shall be interested in keeping the number of preemptions down to a small number, in spite of our assumption that they have no cost. From a practical point of view this makes evident sense.

In the case that preemption is permitted, we will not consider the specialization to unit processing requirements. The reason for this is simply that preemptive unit-time models tend to be rather artificial and have not spawned any results of independent interest.

Exercises

1.5. Prove that there is no advantage to preemption in any single-machine oroblem in which all release dates are 0. Specifically, show that any feasible preemptive schedule with completion times C_j can be modified to obtain a feasible nonpreemptive schedule with completion times C'_j with $C'_j \leq C_j$ $(j = 1,...,n)$.

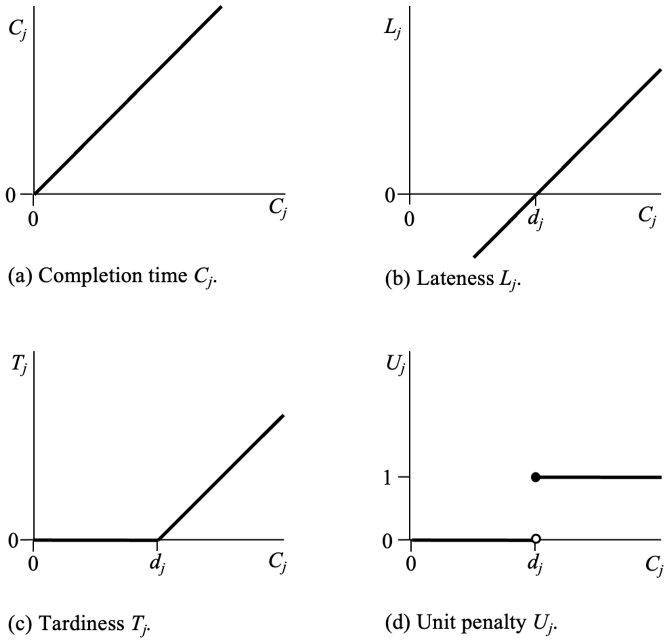
1.6. Give an example of a single-machine problem with release dates and deadlines in which all feasible schedules are preemptive.

1.5. Optimality criteria

As we have noted, each schedule is assigned a value $F(C_1,...,C_n)$, depending only on the job completion times, and we seek to find a feasible schedule for which this value is minimized. It should also be noted that we may restrict our attention to left-justified schedules: because of the monotonicity assumption (1.1), there will always be such a schedule among the optimal ones.

We shall deal with functions F that result from cost functions f_j assigned to the individual jobs j. We then distinguish between *minmax* problems, in which $F(C_1,...,C_n)$ is given by

$$f_{\max} = \max_{j=1,...,n} f_j(C_j),$$



(a) Completion time C_j.

(b) Lateness L_j.

(c) Tardiness T_j.

(d) Unit penalty U_j.

Figure 1.3. Four special choices of the cost function f_j.

and *minsum* problems, in which $F(C_1,...,C_n)$ is given by

$$\sum f_j = \sum_{j=1}^{n} f_j(C_j).$$

Both types of problems will be studied for general cost functions f_j, as well as for a number of special choices (cf. Figure 1.3).

The simplest choice for f_j is $f_j(C_j) = C_j (j = 1,...,n)$. The minmax problem then amounts to the minimization of *maximum completion time* $C_{\max} = \max_{j=1,...,n} C_j$. $C_{\max}$ is sometimes referred to as *makespan* or *schedule length*, since it represents the time required to process all jobs. This is a very natural and, not surprisingly, by far the most frequently encountered criterion.

The minsum problem in this case is to minimize *total completion time* $\sum C_j$, another natural choice. It is a special case of *total weighted completion time* $\sum w_j C_j$, where the *weight* w_j of job j reflects its relative importance. If $w_j = 1/n$ for all j, minimizing total weighted completion time reduces to minimizing *mean completion time*, which is equivalent to simply minimizing $\sum C_j$.

Now suppose each job j is assigned a *due date* d_j, which serves as a yardstick by which the job completion cost is measured. A due date d_j is quite distinct from a deadline $\bar{d}j$. If $C_j > \bar{d}_j$, the schedule is infeasible. If $C_j > d_j$, the schedule is not necessarily infeasible; it may just cost more. We say that job j is *early* if $C_j < d_j$ and *late* if $C_j > d_j$. A common cost function induced by due dates is *lateness*,

$$L_j = C_j - d_j.$$

For this choice, the minmax problem involves the minimization of *maximum lateness* $L_{\max} = \max_{j=1,\ldots,n} L_j$, another popular criterion. Note that minimizing $C_{\max}$ is equivalent to minimizing $L_{\max}$ in the special case that all due dates are 0. So $L_{\max}$ is a proper generalization of $C_{\max}$.

We could generalize the $L_{\max}$ criterion by placing weights on the jobs. However, the minimization of maximum weighted lateness seems neither particularly attractive nor useful, so we shall ignore that possibility. Moreover, so far as minimizing total weighted lateness is concerned, observe that $\sum w_j L_j = \sum w_j C_j - \sum w_j d_j$. Since $\sum w_j d_j$ is a schedule-independent constant, $\sum w_j L_j$ and $\sum w_j C_j$ are equivalent, as are $\sum L_j$ and $\sum C_j$. It follows that we have no need to consider $\sum w_j L_j$ and $\sum L_j$ as separate criteria.

If job j is early, its lateness L_j becomes negative. Sometimes a more realistic cost function to consider is *tardiness*,

$$T_j = \max\{0, C_j - d_j\}.$$

T_j becomes nonzero only if job j is actually late. As we will see in Exercise 1.8, schedules that minimize $L_{\max}$ also minimize $T_{\max}$. However, the converse is not true: minimizing $T_{\max}$ does not necessarily minimize $L_{\max}$. We shall not be concerned with $T_{\max}$ as a criterion.

Minimization of *total tardiness* $\sum T_j$ and *total weighted tardiness* $\sum w_j T_j$ do provide interesting and challenging problems. Note that these criteria reduce to $\sum C_j$ and $\sum w_j C_j$ respectively if $d_j = 0$ for all j.

Finally, we may be interested in a different sort of measure of success in meeting due dates, in which we simply count the number of jobs that are late. In this case we assign a unit penalty for each late job:

$$U_j = \begin{cases} 0 & \text{if } C_j \leq d_j, \\ 1 & \text{if } C_j > d_j. \end{cases}$$

We shall consider both the minimization of the *number of late jobs* $\sum U_j$ and of the *weighted number of late jobs* $\sum w_j U_j$.

Exercises

1.7. Suppose that, in scheduling a single machine, we are interested in minimizing the total machine idle time prior to the completion of the last job. The minimization of total idle time is clearly equivalent to the minimization of $C_{\max}$. For which parallel machine environments is this observation also true?

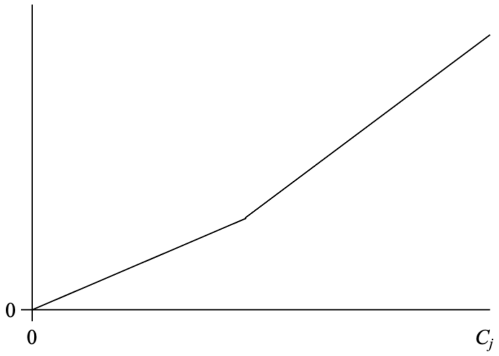


Figure 1.4. Cost function for Exercise 1.11.

1.8. (a) Prove that a schedule that minimizes $L_{\max}$ also minimizes $T_{\max}$.
(b) Construct a left-justified single-machine schedule showing that the converse is not true.
1.9. Prove that a schedule that minimizes $L_{\max}$ also minimizes $U_{\max}$.
1.10. Yet another cost function is *earliness* $E_j = d_j - C_j$. Do $E_{\max}$ and $\sum E_j$ satisfy the monotonicity assumption (1.1)?
1.11. Consider the cost function depicted in Figure 1.4 and show that it can be expressed as an appropriately weighted sum of completion time and tardiness.

1.6. Multi-operation models

Let us now consider *multi-operation* models, the subject of Chapters 11–13 of this book. In these models, we will write J_j for job j $(j = 1,...,n)$ and M_i for machine i $(i = 1,...,m)$. Each job J_j consists of $\mu(j)$ *operations* $O_{1j}, O_{2j}, ..., O_{\mu(j)j}$, with the requirement that each operation O_{ij} must be performed on a specified machine $M_{\iota(i,j)}$ $(1 \leq \iota(i,j) \leq m)$ for an amount of time equal to its processing requirement p_{ij}. No two of the operations of J_j can be performed at the same time, and each operation must be completed before J_j is considered to be finished. The completion time of a job is thus the maximum of the completion times of its operations.

In the case of the *open shop* model, each job J_j has exactly m operations, and each O_{ij} must be performed on M_i. That is, $\mu(j) = m$ and $\iota(i,j) = i$ $(i = 1,...,m, j = 1,...,n)$. There is no restriction on the order in which the operations of a given job may be performed. As an example, consider a large automotive repair shop that is divided into several smaller shops: an engine shop, a radiator shop, a body shop, etc. A car is to have its engine tuned, its radiator repaired and its fender straightened. It is unimportant in which order these operations are performed, but no two of them can be carried out at the same time.

A *flow shop* is like an open shop except that the operations of each job J_j must be carried out in the same fixed sequence: first O_{1j}, then $O_{2j}, ...,$ and finally O_{mj}. The completion time of J_j is the completion time of O_{mj}. A small print shop, with a single typesetting machine, a single printing press and a single binding machine is an example of a flow shop.

The *job shop* is a generalization of the flow shop model, in the sense that the parameters $\mu(j)$ and $\iota(i,j)$ are unrestricted. Each job requires the services of some or all of the machines in its own fixed sequence, with repetitions allowed. For example, one job might require operations on M_1, M_3 in that order, and another might require operations on M_3, M_2, M_1, M_2, M_3 in that order. The completion time of J_j is the completion time of $O_{\mu(j)j}$, its last operation. Machine shops provide the classical example of a job shop. One job might require the use of the lathe, then the drill press, then the grinder, but another job might require the same machines in a different order.

These three types of multi-operation models are illustrated in Figure 1.5. Their treatment in Chapters 11–13 will be restricted to models with general processing times and without release dates, deadlines and precedence constraints. Moreover, we shall only be concerned with the minimization of $C_{\max}$ (with a single excursion to $L_{\max}$). A few interesting results for problems outside this class are included as exercises.

It is easy to think of a common generalization of the parallel-machine model and the multi-operation model. Rather than assume the existence of a single machine that can perform a given operation, we could assume that a given operation can be performed on any one of a set of identical, uniform or unrelated parallel machines. However, we shall not pursue such a generalization.

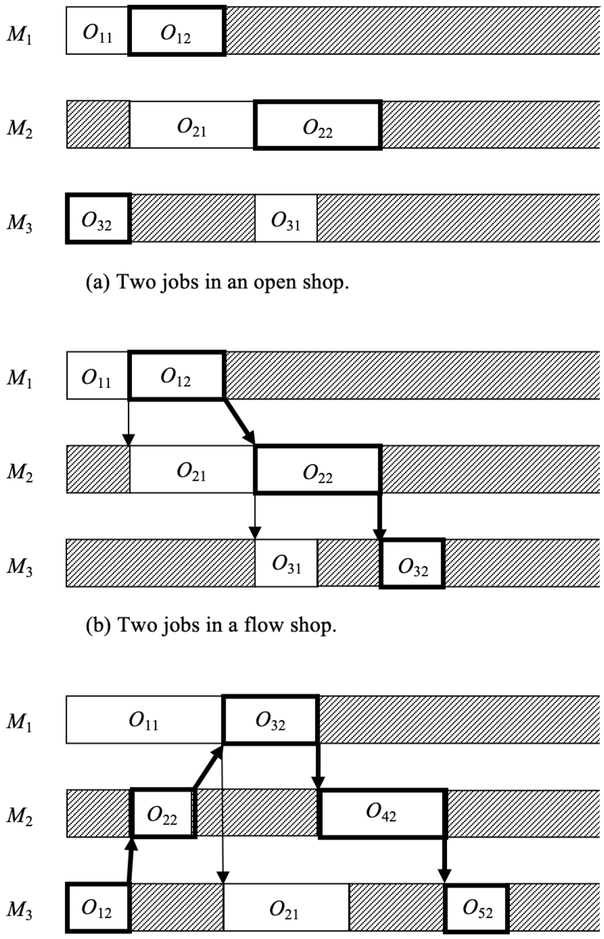
1.7. Problem classification

We are now ready to explain our system of description and classification. Each problem type we shall consider will be described as $\alpha|\beta|\gamma$, where α denotes the machine environment, β indicates job characteristics, and γ specifies the optimality criterion.

First the α-field. We shall use the following mnemonics to indicate machine environments:

P	identical parallel machines;
Q	uniform parallel machines;
R	unrelated parallel machines;
O	open shop;
F	flow shop;
J	job shop.

Each of these letters is followed by a constant, by an m, or by no symbol at all. In the first case, the constant indicates the number of machines, which is thereby specified



(a) Two jobs in an open shop.

(b) Two jobs in a flow shop.

(c) Two jobs in a job shop.

Figure 1.5. Shop schedules.

as part of the problem type. For example, $F2|\beta|\gamma$ denotes a problem involving a two-machine flow shop. The second case indicates that the number of machines is an unspecified constant. Thus, $Fm|\beta|\gamma$ denotes a flow shop problem with some fixed number of machines. If there is no symbol following the letter, the number of machines is a variable, the value of which is part of the problem instance. Hence, $F|\beta|\gamma$ is a flow shop problem in which the number of machines will be specified together with the other numerical data. (We will discuss this distinction further in Chapter 3.) And what about single-machine problems? These are simply denoted by the numeral 1, i.e., $1|\beta|\gamma$. (Note that $P1$, $Q1$, $R1$, $O1$, $F1$ and $J1$ are all single-machine environments.)

Now for the β-field. This field indicates deviations from the default assumptions (1.2–1.6). So if nothing at all appears in this field, each of these assumptions applies. Thus $1||\gamma$ indicates a single-machine problem with all release dates equal to 0, no deadlines, no precedence constraints, and no preemption permitted. We shall employ the following mnemonics to indicate deviations from the default assumptions:

$pmtn$	preemption is permitted;
$chain$	chain-type precedence constraints;
$intree$	intree-type precedence constraints;
$outtree$	outtree-type precedence constraints;
$tree$	tree-type precedence constraints;
$sepa$	series-parallel precedence constraints;
$prec$	general precedence constraints;
r_j	release dates;
$\bar{d}_j$	deadlines;
$p_j = 1$	unit processing requirements.

For example, $1|pmtn, prec, r_j|\gamma$ indicates a preemptive single-machine problem with general precedence constraints and release dates (but without deadlines and with general processing requirements). We will occasionally use the β-field to indicate other job characteristics, which, in certain situations, appear to merit a separate investigation. Some examples are:

$\mu(j) \leq 2$	each job has one or two operations;
$d_j = d$	all due dates are equal;
$p_j \in \{1,2\}$	the processing requirements are equal to 1 or 2.

We have introduced ten criteria that are of interest:

$f_{\max}$	general minmax criterion;
$C_{\max}$	maximum completion time;
$L_{\max}$	maximum lateness;
$\sum f_j$	general minsum criterion;
$\sum C_j$	total completion time;
$\sum w_j C_j$	total weighted completion time;
$\sum T_j$	total tardiness;
$\sum w_j T_j$	total weighted tardiness;
$\sum U_j$	number of late jobs;
$\sum w_j U_j$	weighted number of late jobs.

We conclude this chapter by repeating that not all problems generated by our classification scheme are of equal interest and that some combinations of choices are excluded *a priori*. As has been indicated above, in the case of unrelated parallel machines and in the case that preemption is permitted, we will not consider the specialization to unit processing requirements, and other restrictions apply to the investigation of multi-operation models. We also exclude the combination of deadlines with $C_{\max}$ or $L_{\max}$: the resulting problems are in some sense equivalent to the $L_{\max}$ problem without deadlines.

Exercises

1.12. The first paragraph of Section 1.1 mentions five practical scheduling situations. Formulate each of these in terms of the problem classification. In each case, consider the relevance of the possible machine environments, job characteristics, and optimality criteria.

Notes

1.1. *Machines, jobs, and schedules.* Conway, Maxwell, and Miller (1967) wrote the first book on scheduling theory. Their text is still remarkable for the way it combines deterministic scheduling with queueing and simulation. Other books on deterministic scheduling include the undergraduate texts by Baker (1974) and French (1982), the collection of advanced expository reviews edited by Coffman (1976), and the dissertations of Rinnooy Kan (1976) and Lenstra (1977). The proceedings volume edited by Dempster, Lenstra, and Rinnooy Kan (1982) provides surveys of the broader area of deterministic and stochastic scheduling and emphasizes developments on the interface between scheduling and queueing theory.

The present book is to some extent an outgrowth of the comprehensive survey papers by Graham, Lawler, Lenstra, and Rinnooy Kan (1979), Lawler, Lenstra, and Rinnooy Kan (1982), and Lawler, Lenstra, Rinnooy Kan, and Shmoys (1993). More tutorial surveys are given by Lawler and Lenstra (1982), who consider the influence of precedence constraints, and Lawler (1983), who concentrates on polynomial algorithms and open problems. We further mention the NP-completeness column on multiprocessor scheduling by Johnson (1983), the annotated bibliography of the scheduling literature covering the period 1981–1984 by Lenstra and Rinnooy Kan (1985), the discussions of new directions in scheduling by Lenstra and Rinnooy Kan (1984), Blazewicz (1987) and Blazewicz, Finke, Haupt, and Schmidt (1988), and the overviews of single-machine scheduling by Gupta and Kyparisis (1987) and of multiprocessor and flow shop scheduling by Kawaguchi and Kyan (1988). Graves (1981) reviews the broader area of production scheduling, which includes machine scheduling as well as lot sizing, with particular attention for practical aspects.

Henry Laurence Gantt (1861–1919) was a mechanical and industrial engineer. A disciple of Frederick W. Taylor, 'the father of scientific management', he held slightly more enlightened views on the social impacts of his work. He developed his famous charts during the First World War at the Ordnance Bureau of the United States Army, in order to provide methods for a simple, fast and accurate comparison between a production plan and its realization. The origin of the Gantt chart is reviewed by Porter (1968). Gantt discussed the underlying principles in his paper 'Efficiency and democracy' (Gantt, 1919A), which he presented in December 1918 at the Annual Meeting of The American Society of Mechanical Engineers; see also his monograph *Organizing for Work* (i.e., for production, not for profit) (Gantt, 1919B). His life and work are described by Alford (1934) and Rathe (1961), and his charts by Clark (1922) in a book that has been translated into twelve languages. It is fair to say that the Gantt charts as we use them are a gross simplification of the originals, both in purpose and design.

1.2. *Single machines and parallel machines.* The traveling salesman problem is dealt with in more detail by Lawler, Lenstra, Rinnooy Kan, and Shmoys (1985).

Left-justified schedules are also referred to as 'semi-active' schedules in part of the scheduling literature.

1.3. *Release dates, deadlines, and precedence constraints.* The investigation of precedence constraints presupposes an elementary knowledge of the theory of directed graphs, for which we refer to standard books such as those by Wilson (1972) and Bondy and Murty (1976). Transitive closures and transitive reductions are discussed by Aho, Hopcroft, and Ullman (1974).

1.5. *Optimality criteria.* Objective functions that are monotonic in the job completion times are sometimes said to be 'regular'.

1.7. *Problem classification.* The classification scheme for deterministic machine scheduling problems was developed by Graham, Lawler, Lenstra, and Rinnooy Kan (1979). It is based on the classification scheme for scheduling and queueing problems introduced by Conway, Maxwell, and Miller (1967).