3 Definition, Analysis and Classification of Scheduling Problems

Throughout this book we are concerned with scheduling computer and manufacturing processes. Despite the fact that we deal with two different areas of applications, the same model could be applied. This is because the above processes consist of complex activities to be scheduled, which can be modeled by means of tasks (or jobs), relations among them, processors, sometimes additional resources (and their operational functions), and parameters describing all these items in greater detail. The purpose of the modeling is to find optimal or sub-optimal schedules in the sense of a given criterion, by applying best suited algorithms. These schedules are then used for the original setting to carry out the various activities. In this chapter we introduce basic notions used for such a modeling of computer and manufacturing processes.

3.1 Definition of Scheduling Problems

In general, scheduling problems considered in this book[1] are characterized by three sets: set $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$ of *n tasks*, set $\mathcal{P} = \{P_1, P_2, \cdots, P_m\}$ of *m processors* (*machines*) and set $\mathcal{R} = \{R_1, R_2, \cdots, R_s\}$ of *s* types of *additional resources* $\mathcal{R}$. Scheduling, generally speaking, means to assign processors from $\mathcal{P}$ and (possibly) resources from $\mathcal{R}$ to tasks from $\mathcal{T}$ in order to complete all tasks under the imposed constraints. There are two general constraints in classical scheduling theory. Each task is to be processed by at most one processor at a time (plus possibly specified amounts of additional resources) and each processor is capable of processing at most one task at a time. In Chapters 6 and 13 we will show some new applications in which the first constraint will be relaxed.

We will now characterize the processors. They may be either *parallel*, i.e. performing the same functions, or *dedicated* i.e. specialized for the execution of certain tasks. Three types of parallel processors are distinguished depending on their speeds. If all processors from set $\mathcal{P}$ have equal task processing speeds, then we call them *identical*. If the processors differ in their speeds, but the *speed* b_i of

[1] The notation presented in this section is extended in the following chapters of the book.

J. Blazewicz et al., *Handbook on Scheduling*, International Handbooks on Information Systems, https://doi.org/10.1007/978-3-319-99849-7_3

each processor is constant and does not depend on the task in $\mathcal{T}$, then they are called *uniform*. Finally, if the speeds of the processors depend on the particular task processed, then they are called *unrelated*.

In case of dedicated processors there are three models of processing sets of tasks: *flow shop*, *open shop* and *job shop*. To describe these models more precisely, we assume that tasks form *n subsets*[2] (*chains* in case of flow- and job shops), each subset called a *job*. That is, job J_j is divided into n_j tasks, T_{1j}, $T_{2j},\cdots,T_{n_j j}$, and two adjacent tasks are to be performed on different processors. A set of jobs will be denoted by $\mathcal{J}$. In an open shop the number of tasks is the same for each job and is equal to m, i.e. $n_j = m$, $j = 1, 2,\cdots,n$. Moreover, T_{1j} should be processed on P_1, T_{2j} on P_2, and so on. A similar situation is found in flow shop, but, in addition, the processing of $T_{i-1\,j}$ should precede that of T_{ij} for all $i = 1,\cdots,n_j$ and for all $j = 1, 2,\cdots,n$. In a general job shop system the number n_j is arbitrary. Usually in such systems it is assumed that buffers between processors have unlimited capacity and a job after completion on one processor may wait before its processing starts on the next one. If, however, buffers are of zero capacity, jobs cannot wait between two consecutive processors, thus, a *no-wait property* is assumed.

In general, task $T_j \in \mathcal{T}$ is characterized by the following data.

1. *Vector of processing times* $\boldsymbol{p}_j = [p_{1j}, p_{2j},\cdots,p_{mj}]^{\mathrm{T}}$, where p_{ij} is the time needed by processor P_i to process T_j. In case of identical processors we have $p_{ij} = p_j$, $i = 1, 2,\cdots,m$. If the processors in $\mathcal{P}$ are uniform, then $p_{ij} = p_j/b_i$, $i = 1, 2,\cdots,m$, where p_j is the *standard processing time* (usually measured on the slowest processor) and b_i is the *processing speed factor* of processor P_i. In case of shop scheduling, the vector of processing times describes the processing requirements of particular tasks comprising one job; that is, for job J_j we have $\boldsymbol{p}_j = [p_{1j}, p_{2j},\cdots,p_{n_j j}]^{\mathrm{T}}$, where p_{ij} denotes the processing time of T_{ij} on the corresponding processor.

2. *Arrival time* (or *ready time*) r_j, which is the time at which task T_j is ready for processing. If the arrival times are the same for all tasks from $\mathcal{T}$, then it is assumed that $r_j = 0$ for all j.

3. *Due date* d_j, which specifies a time limit by which T_j should be completed; usually, penalty functions are defined in accordance with due dates.

4. *Deadline* $\tilde{d}_j$, which is a "hard" real time limit by which T_j must be completed.

5. *Weight* (*priority*) w_j, which expresses the relative urgency of T_j.

6. *Resource request* (if any), as defined in Chapter 13.

---

[2] Thus, the number of tasks in $\mathcal{T}$ is assumed to be $\geq n$.

Unless stated otherwise we assume that all these parameters, p_j, r_j, d_j, $\tilde{d}_j$, and w_j, are integers. In fact, this assumption is not very restrictive, since it is equivalent to permitting arbitrary rational values. We assume moreover, that tasks are assigned all required resources whenever they start or resume their processing and that they release all the assigned resources whenever they are completed or preempted. These assumptions imply that deadlock cannot occur.

Next, some definitions concerning task preemptions and precedence constraints among tasks are given. A schedule is called *preemptive* if each task may be preempted at any time and restarted later at no cost, perhaps on another processor. If preemption of all the tasks is not allowed we will call the schedule *non-preemptive*.

In set $\mathcal{T}$ *precedence constraints* among tasks may be defined. $T_i \prec T_j$ means that the processing of T_i must be completed before T_j can be started. In other words, in set $\mathcal{T}$ a precedence relation $\prec$ is defined. The tasks in set $\mathcal{T}$ are called *dependent* if the order of execution of at least two tasks in $\mathcal{T}$ is restricted by this relation. Otherwise, the tasks are called *independent*. A task set with precedence relation is usually represented as a directed graph (a digraph) in which nodes correspond to tasks and arcs to precedence constraints (a *task-on-node graph*). It is assumed that no transitive arcs exist in precedence graphs. An example of a set of dependent tasks is shown in Figure 3.1.1(a) (nodes are denoted by T_j/p_j). Several special types of precedence graphs have already been described in Section 2.3.2. Let us notice that in the case of dedicated processors (except in open shop systems) tasks that constitute a job are always dependent, but the jobs themselves can be either independent or dependent. There is another way of representing task dependencies which is useful in certain circumstances. In this so-called *activity network*, precedence constraints are represented as a *task-on-arc graph*, where arcs represent tasks and nodes time events. Let us mention here a special graph of this type called *uniconnected activity network* (*uan*), which is defined as a graph in which any two nodes are connected by a directed path in one direction only. Thus, all nodes are uniquely ordered. For every precedence graph one can construct a corresponding activity network (and vice versa), perhaps using dummy tasks of zero length. The corresponding activity network for the precedence graph from Figure 3.1.1(a), is shown in Figure 3.1.1(b). Note that we will show in Section 5.1.1. the equivalence of the uniconnected activity network and the interval order task-on-node representation (cf. also [BK02]).

Task T_j will be called *available* at time t if $r_j \leq t$ and all its predecessors (with respect to the precedence constraints) have been completed by time t.

Now we will give the definitions concerning schedules and optimality criteria. A *schedule* is an assignment of processors from set $\mathcal{P}$ (and possibly resources from set $\mathcal{R}$) to tasks from set $\mathcal{T}$ in time such that the following conditions are satisfied:

– at every moment each processor is assigned to at most one task and each task is processed by at most one processor [3],
– task T_j is processed in time interval $[r_j, \infty)$,
– all tasks are completed,
– if tasks T_i, T_j are in relation $T_i \prec T_j$, the processing of T_j is not started before T_i is completed,
– in the case of non-preemptive scheduling no task is preempted (then the schedule is called *non-preemptive*), otherwise the number of preemptions of each task is finite [4] (then the schedule is called *preemptive*),
– resource constraints, if any, are satisfied.

To represent schedules we will use the so-called *Gantt charts*. An example schedule for the task set of Figure 3.1.1 on three parallel, identical processors is shown in Figure 3.1.2. The following parameters can be calculated for each task $T_j, j = 1, 2, \cdots, n$, processed in a given schedule:

completion time C_j,

flow time $F_j = C_j - r_j$, being the sum of waiting and processing times;

lateness $L_j = C_j - d_j$,

tardiness $D_j = \max\{C_j - d_j, 0\}$;

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

For the schedule given in Figure 3.1.2 one can easily calculate the two first parameters. In vector notation these are $\boldsymbol{C} = [3, 4, 5, 6, 1, 8, 8, 8]$ and $\boldsymbol{F} = \boldsymbol{C}$. The other two parameters could be calculated, if due dates would be defined. Suppose that due dates are given by the vector $\boldsymbol{d} = [5, 4, 5, 3, 7, 6, 9, 12]$. Then the latenesses, tardinesses and earliness for the tasks in the schedule are: $\boldsymbol{L} = [-2, 0, 0, 3, -6, 2, -1, -4]$, $\boldsymbol{D} = [0, 0, 0, 3, 0, 2, 0, 0]$, $\boldsymbol{E} = [2, 0, 0, 0, 6, 0, 1, 4]$.

To evaluate schedules we will use three main *performance measures* or *optimality criteria*:

Schedule length (*makespan*) $C_{max} = \max\{C_j\}$,

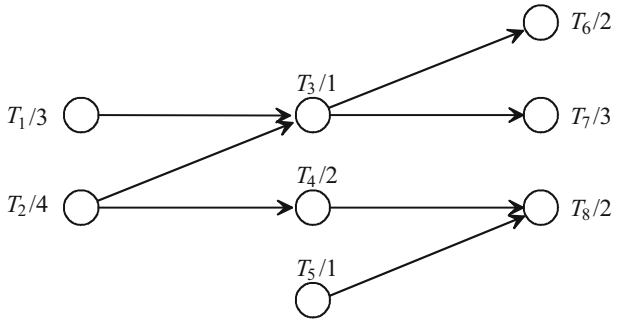
mean flow time $\overline{F} = \frac{1}{n}\sum_{j=1}^{n} F_j$,

or *mean weighted flow time* $\overline{F}_w = \sum_{j=1}^{n} w_j F_j / \sum_{j=1}^{n} w_j$,

maximum lateness $L_{max} = \max\{L_j\}$.

[3] As we mentioned, this assumption can be relaxed.
[4] This condition is imposed by practical considerations only.

(a)



(b)

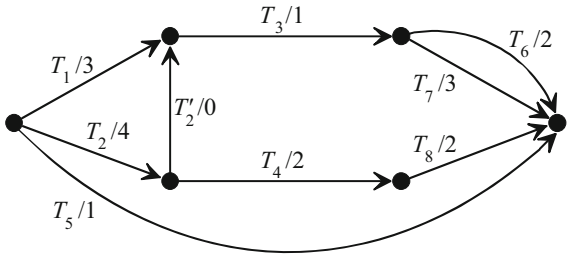


Figure 3.1.1 *An example task set*
(a) *task-on-node representation*
(b) *task-on-arc representation (dummy tasks are primed).*

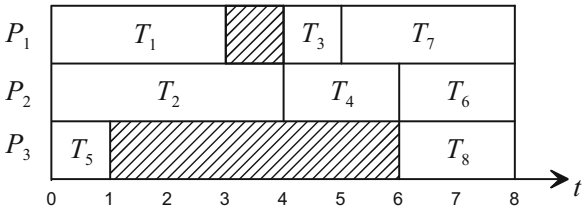


Figure 3.1.2 *A schedule for the task set given in Figure* 3.1.1.

In some applications, other related criteria may be used, as for example: *mean tardiness* $\bar{D} = \frac{1}{n}\sum_{j=1}^{n} D_j$, *mean weighted tardiness* $\bar{D}_w = \sum_{j=1}^{n} w_j D_j / \sum_{j=1}^{n} w_j$, *mean earliness* $\bar{E} = \frac{1}{n}\sum_{j=1}^{n} E_j$, *mean weighted earliness* $\bar{E}_w = \sum_{j=1}^{n} w_j E_j / \sum_{j=1}^{n} w_j$, *number of tardy tasks* $U = \sum_{j=1}^{n} U_j$, where $U_j = 1$ if $C_j > d_j$, and 0 otherwise, or *weighted number of tardy tasks* $U_w = \sum_{j=1}^{n} w_j U_j$.

Again, let us calculate values of particular criteria for the schedule in Figure 3.1.2. They are: schedule length $C_{max} = 8$, mean flow time $\overline{F} = 43/8$, maximum lateness $L_{max} = 3$, mean tardiness $\overline{D} = 5/8$, mean earliness $\overline{E} = 13/8$, and number of tardy jobs $U = 2$. The other criteria can be evaluated if weights of tasks are specified.

A schedule for which the value of a particular performance measure γ is at its minimum will be called *optimal*, and the corresponding value of γ will be denoted by γ^*.

We may now define the *scheduling problem* Π as a set of parameters described in this subsection [5] not all of which have numerical values, together with an optimality criterion. An *instance I* of problem Π is obtained by specifying particular values for all the problem parameters.

We see that scheduling problems are in general of optimization nature (cf. Section 2.2.1). However, some of them are originally formulated in decision version. An example is scheduling to meet deadlines, i.e. the problem of finding, given a set of deadlines, a schedule with no late task. However, both cases are analyzed in the same way when complexity issues are considered.

A *scheduling algorithm* is an algorithm which constructs a schedule for a given problem Π. In general, we are interested in optimization algorithms, but because of the inherent complexity of many problems of that type, approximation or heuristic algorithms will be discussed (cf. Sections 2.2.2 and 2.5).

Scheduling problems, as defined above, may be analyzed much in the same way as discussed in Chapter 2. However, their specificity raises some more detailed questions which will be discussed in the next section.

3.2 Analysis of Scheduling Problems and Algorithms

Deterministic scheduling problems are a part of a much broader class of combinatorial optimization problems. Thus, the general approach to the analysis of these problems can follow similar lines, but one should take into account their peculiarities. It is rather obvious that very often the time we can devote to solving particular scheduling problems is seriously limited so that only low order polynomial time algorithms may be used. Thus, the examination of the complexity of these problems should be the basis of any further analysis.

It has been known for some time [Coo71, Kar72] (cf. Section 2.2) that there exists a large class of combinatorial optimization problems for which most probably no *efficient optimization* algorithms exist. These are the problems whose decision counterparts (i.e. problems formulated as questions with "yes" or "no" an-

[5] Parameters are understood generally, including e.g. relation $\prec$.

swers) are NP-complete. The optimization problems are called NP-hard in this case. We refer the reader to [GJ79] and to Section 2.2 for a comprehensive treatment of the NP-completeness theory, and in the following we assume knowledge of its basic concepts like NP-completeness, NP-hardness, polynomial time transformation, etc. It follows that the complexity analysis answers the question whether or not an analyzed scheduling problem may be solved (i.e. an optimal schedule found) in time bounded from above by a polynomial in the input length of the problem (i.e. in polynomial time). If the answer is positive, then an optimization polynomial time algorithm must have been found. Its usefulness depends on the order of its worst-case complexity function and on the particular application. Sometimes, when the worst-case complexity function is not low enough, although still polynomial, a mean complexity function of the algorithm may be sufficient. This issue is discussed in detail in [AHU74]. On the other hand, if the answer is negative, i.e. when the decision version of the analyzed problem is NP-complete, then there are several other ways of further analysis.

First, one may try to relax some constraints imposed on the original problem and then solve the relaxed problem. The solution of the latter may be a good approximation to the solution of the original problem. In the case of scheduling problems such a relaxation may consist of

- allowing preemptions, even if the original problem dealt with non-preemptive schedules,
- assuming unit-length tasks, when arbitrary-length tasks were considered in the original problem,
- assuming certain types of precedence graphs, e.g. trees or chains, when arbitrary graphs were considered in the original problem, etc.

Considering computer applications, especially the first relaxation can be justified in the case when parallel processors share a common primary memory. Moreover, such a relaxation is also advantageous from the viewpoint of certain optimality criteria.

Second, when trying to solve NP-hard scheduling problems one often uses approximation algorithms which tend to find an optimal schedule but do not always succeed. Of course, the necessary condition for these algorithms to be applicable in practice is that their worst-case complexity function is bounded from above by a low-order polynomial in the input length. Their sufficiency follows from an evaluation of the difference between the value of a solution they produce and the value of an optimal solution. This evaluation may concern the worst case or a mean behavior. To be more precise, we use here notions that have been introduced in Section 2.5, i.e. absolute performance ratio R_A and asymptotic performance ratio R_A^∞ of an approximation algorithm A.

These notions define a measure of "goodness" of approximation algorithms; the closer R_A^∞ is to 1, the better algorithm A performs. However, for some combinatorial problems it can be proved that there is no hope of finding an approxima-

tion algorithm of a certain accuracy, i.e. this question is as hard as finding a polynomial time algorithm for any NP-complete problem.

Analysis of the worst-case behavior of an approximation algorithm may be complemented by an analysis of its mean behavior. This can be done in two ways. The first consists in assuming that the parameters of instances of the considered problem Π are drawn from a certain distribution, and then the *mean performance* of algorithm A is analyzed. One may distinguish between the *absolute error* of an approximation algorithm, which is the difference between the approximate and optimal values and the *relative error*, which is the ratio of these two (cf. Section 2.5). Asymptotic optimality results in the stronger (absolute) sense are quite rare. On the other hand, asymptotic optimality in the relative sense is often easier to establish. It is rather obvious that the mean performance can be much better than the worst case behavior, thus justifying the use of a given approximation algorithm. A main obstacle is the difficulty of proofs of the mean performance for realistic distribution functions. Thus, the second way of evaluating the mean behavior of approximation algorithms, consisting of experimental studies, is still used very often. In the latter approach, one compares solutions, in the sense of the values of an optimality criterion, constructed by a given approximation algorithm and by an optimization algorithm. This comparison should be made for a large, representative sample of instances.

In this context let us mention the most often used approximation scheduling algorithm which is the so-called *list scheduling algorithm* (which is in fact a general approach). In this algorithm a certain list of tasks is given and at each step the first available processor is selected to process the first available task on the list. The accuracy of a particular list scheduling algorithm depends on the given optimality criterion and the way the list has been constructed.

The third and last way of dealing with hard scheduling problems is to use exact enumerative algorithms whose worst-case complexity function is exponential in the input length. However, sometimes, when the analyzed problem is not NP-hard in the strong sense, it is possible to solve it by a pseudopolynomial optimization algorithm whose worst-case complexity function is bounded from above by a polynomial in the input length and in the maximum number appearing in the instance of the problem. For reasonably small numbers such an algorithm may behave quite well in practice and it can be used even in computer applications. On the other hand, "pure" exponential algorithms have probably to be excluded from this application, but they may be used sometimes for other scheduling problems which can be solved by off-line algorithms.

The above discussion is summarized in a schematic way in Figure 3.2.1. In the following chapters we will use the above scheme when analyzing scheduling problems.

3.3 Motivations for Deterministic Scheduling Problems

In this section, an interpretation of the assumptions and results in deterministic scheduling theory which motivate and justify the use of this model, is presented. We will underline especially computer applications, but we will also refer to manufacturing systems, even if the practical interpretation of the model is not for this application area. In a manufacturing environment deterministic scheduling is also known as *predictive*. Its complement is *reactive scheduling*, which can also be regarded as deterministic scheduling with a shorter planning horizon.

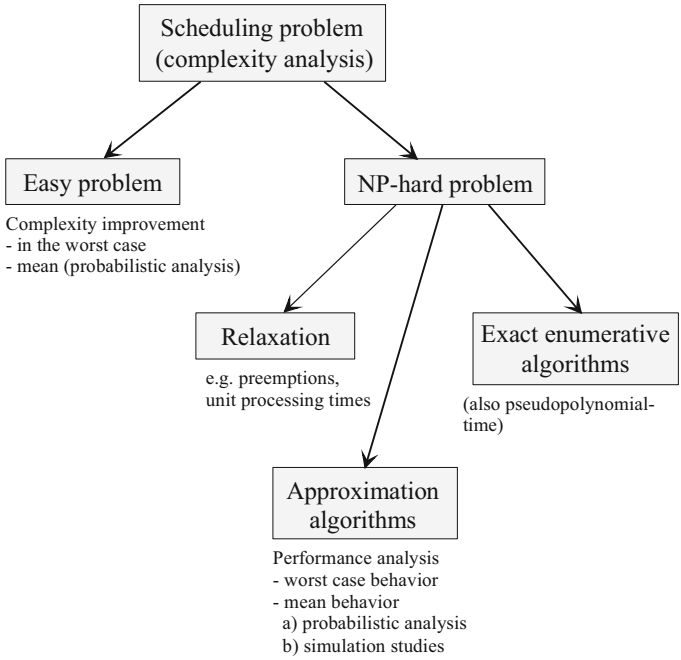


Figure 3.2.1 *An analysis of a scheduling problem - schematic view.*

Let us begin with an analysis of processors (machines). *Parallel processors* may be interpreted as central processors which are able to process every task (i.e. every program). *Uniform processors* differ from each other by their speeds, but they do not prefer any type of tasks. *Unrelated processors*, on the contrary, are specialized in the sense that they prefer certain types of tasks, for example numerical computations, logical programs, or simulation procedures. The processors may have different instruction sets, but they are still of comparable processing capacity so they can process tasks of any type, only processing times may be different.

In manufacturing systems, pools of machines exist where all the machines have the same capability (except possibly speed) to process tasks.

Completely different from the above are *dedicated* processors (dedicated machines) which may process only certain types of tasks. The interpretation of this model for manufacturing systems is straightforward but it can also be applied to computer systems. As an example let us consider a computer system consisting of an input processor, a central processor and an output processor. It is not difficult to see that such a system corresponds to a flow shop with $m = 3$. On the other hand, a situation in which each task is to be processed by an input/output processor, then by a central processor and at the end again by the input/output processor, can easily be modeled by a job shop system with $m = 2$. As far as an open shop is concerned, there is no obvious computer interpretation. But this case, like the other shop scheduling problems, has great significance in other applications, especially in an industrial environment.

By an *additional resource* we understand in this book a "facility" besides processors the tasks to be performed compete for. The competition aspect in this definition should be stressed, since "facilities" dedicated to only one task will not be treated as resources in this book. In computer systems, for example, messages sent from one task to another specified task will not be considered as resources. In manufacturing environments tools, material, transport facilities, etc. can be treated as additional resources.

Let us now consider the assumptions associated with the task set. As mentioned in Section 3.1, in deterministic scheduling theory a priori knowledge of ready times and processing times of tasks is usually assumed. As opposed to other practical applications, the question of a priori knowledge of these parameters in computer systems needs a thorough comment.

Ready times are obviously known in systems working in an off-line mode and in control systems in which measurement samples are taken from sensing devices at fixed time moments.

As far as *processing times* are concerned, they are usually not known a priori in computer systems. Despite this fact the solution of a deterministic scheduling problem may also have an important interpretation in these systems. First, when scheduling tasks to meet deadlines, the only approach (when the task processing times are not known) is to solve the problem with assumed upper bounds on the processing times. Such a bound for a given task may be implied by the worst case complexity function of an algorithm connected with that task. Then, if all deadlines are met with respect to the upper bounds, no deadline will be exceeded for the real task processing times [6]. This approach is often used in a broad class of computer control systems working in a hard real time environment, where a certain set of control programs must be processed before taking the next sample from the same sensing device.

---

[6] However, one has to take into account list scheduling anomalies which will be explained in Section 5.1.

Second, instead of exact values of processing times one can take their mean values and, using the procedure described by Coffman and Denning in [CD73], calculate an optimistic estimate of the mean value of the schedule length.

Third, one can measure the processing times of tasks *after* processing a task set scheduled according to a certain algorithm A. Taking these values as an input in the deterministic scheduling problem, one may construct an optimal schedule and compare it with the one produced by algorithm A, thus evaluating the latter.

Apart from the above, optimization algorithms for deterministic scheduling problems give some indications for the construction of heuristics under weaker assumptions than those made in stochastic scheduling problems, cf. [BCSW86].

The existence of *precedence constraints* in computer systems also requires an explanation. In the simplest case the results of certain programs may be the input data for others. Moreover, precedence constraints may also concern parts of the same program. A conventional, serially written program, may be analyzed by a special procedure looking for parallel parts in it (see for example [RG69, Rus69], or [Vol70]). These parts may also be defined by the programmer who can use special programming languages supporting parallel concepts. Apart from this, a solution of certain reliability problems in operating systems, as for example the *determinacy problem* (see [ACM70, Bae74, Ber66]), requires an introduction of additional precedence constraints.

We will now discuss particular *optimality criteria* for scheduling problems from their practical significance point of view. Minimizing *schedule length* is important from the viewpoint of the owner of a set of processors (machines), since it leads to both, the maximization of the processor utilization factor (within schedule length C_{max}), and the minimization of the maximum in-process time of the scheduled set of tasks. This criterion may also be of importance in a computer control system in which a task set arrives periodically and is to be processed in the shortest time.

The *mean flow time* criterion is important from the user's viewpoint since its minimization yields a minimization of the mean response time and the mean in-process time of the scheduled task set.

Due date involving criteria are of great importance in manufacturing systems, especially in those that produce to specific customer orders. Moreover, the *maximum lateness* criterion is of great significance in computer control systems working in the hard real time environment since its minimization leads to the construction of a schedule with no task late whenever such schedules exist (i.e. when $L^*_{max} \leq 0$ for an optimal schedule).

The criteria mentioned above are basic in the sense that they require specific approaches to the construction of schedules.

3.4 Classification of Deterministic Scheduling Problems

The great variety of scheduling problems we have seen from the preceding section motivates the introduction of a systematic notation that could serve as a basis for a classification scheme. Such a notation of problem types would greatly facilitate the presentation and discussion of scheduling problems. A notation proposed by Graham et al. [GLL+79] and Błażewicz et al. [BLRK83] will be presented next and then used throughout the book.

The notation is composed of three fields $\alpha \mid \beta \mid \gamma$. They have the following meaning: The first field $\alpha = \alpha_1, \alpha_2$ describes the processor environment. Parameter $\alpha_1 \in \{\varnothing, P, Q, R, O, F, J\}$ characterizes the type of processor used:

$\alpha_1 = \varnothing$: single processor [7],

$\alpha_1 = P$: identical processors,

$\alpha_1 = Q$: uniform processors,

$\alpha_1 = R$: unrelated processors,

$\alpha_1 = O$: dedicated processors: open shop system,

$\alpha_1 = F$: dedicated processors: flow shop system,

$\alpha_1 = J$: dedicated processors: job shop system.

Parameter $\alpha_2 \in \{\varnothing, k\}$ denotes the number of processors in the problem:

$\alpha_2 = \varnothing$: the number of processors is assumed to be variable,

$\alpha_2 = k$: the number of processors is equal to k (k is a positive integer).

The second field $\beta = \beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6, \beta_7, \beta_8$ describes task and resource characteristics. Parameter $\beta_1 \in \{\varnothing, pmtn\}$ indicates the possibility of task preemption:

$\beta_1 = \varnothing$: no preemption is allowed,

$\beta_1 = pmtn$: preemptions are allowed.

Parameter $\beta_2 \in \{\varnothing, \text{res}\}$ characterizes additional resources:

$\beta_2 = \varnothing$: no additional resources exist,

$\beta_2 = res$: there are specified resource constraints; they will be described in detail in Chapter 13.

Parameter $\beta_3 \in \{\varnothing, prec, uan, tree, chains\}$ reflects the precedence constraints:

$\beta_3 = \varnothing, prec, uan, tree, chains$: denotes respectively independent tasks, general precedence constraints, uniconnected activity networks, precedence constraints forming a tree or a set of chains.

Parameter $\beta_4 \in \{\varnothing, r_j\}$ describes ready times:

---

[7] In this notation $\varnothing$ denotes an empty symbol which will be omitted in presenting problems.

$\beta_4 = \varnothing$: all ready times are zero,

$\beta_4 = r_j$: ready times differ per task.

Parameter $\beta_5 \in \{\varnothing, p_j = p, \underline{p} \le p_j \le \overline{p}\}$ describes task processing times:

$\beta_5 = \varnothing$: tasks have arbitrary processing times,

$\beta_5 = (p_j = p)$: all tasks have processing times equal to p units,

$\beta_5 = (\underline{p} \le p_j \le \overline{p})$: no p_j is less than $\underline{p}$ or greater than $\overline{p}$.

Parameter $\beta_6 \in \{\varnothing, \widetilde{d}\}$ describes deadlines:

$\beta_6 = \varnothing$: no deadlines are assumed in the system (however, due dates may be defined if a due date involving criterion is used to evaluate schedules),

$\beta_6 = \widetilde{d}$: deadlines are imposed on the performance of a task set.

Parameter $\beta_7 \in \{\varnothing, n_j \le k\}$ describes the maximal number of tasks constituting a job in case of job shop systems:

$\beta_7 = \varnothing$: the above number is arbitrary or the scheduling problem is not a job shop problem,

$\beta_7 = (n_j \le k)$: the number of tasks for each job is not greater than k.

Parameter $\beta_8 \in \{\varnothing, \textit{no-wait}\}$ describes a no-wait property in the case of scheduling on dedicated processors:

$\beta_8 = \varnothing$: buffers of unlimited capacity are assumed,

$\beta_8 = \textit{no-wait}$: buffers among processors are of zero capacity and a job after finishing its processing on one processor must immediately start on the consecutive processor.

The third field, γ, denotes an optimality criterion (performance measure), i.e. $\gamma \in \{C_{max}, \Sigma C_j, \Sigma w_j C_j, L_{max}, \Sigma D_j, \Sigma w_j D_j, \Sigma E_j, \Sigma w_j E_j, \Sigma U_j, \Sigma w_j U_j, -\}$, where $\Sigma C_j = \overline{F}$, $\Sigma w_j C_j = \overline{F}_w$, $\Sigma D_j = \overline{D}$, $\Sigma w_j D_j = \overline{D}_w$, $\Sigma E_j = \overline{E}$, $\Sigma w_j E_j = \overline{E}_w$, $\Sigma U_j = U$, $\Sigma w_j U_j = U_w$ and "–" means testing for feasibility whenever scheduling to meet deadlines is considered.

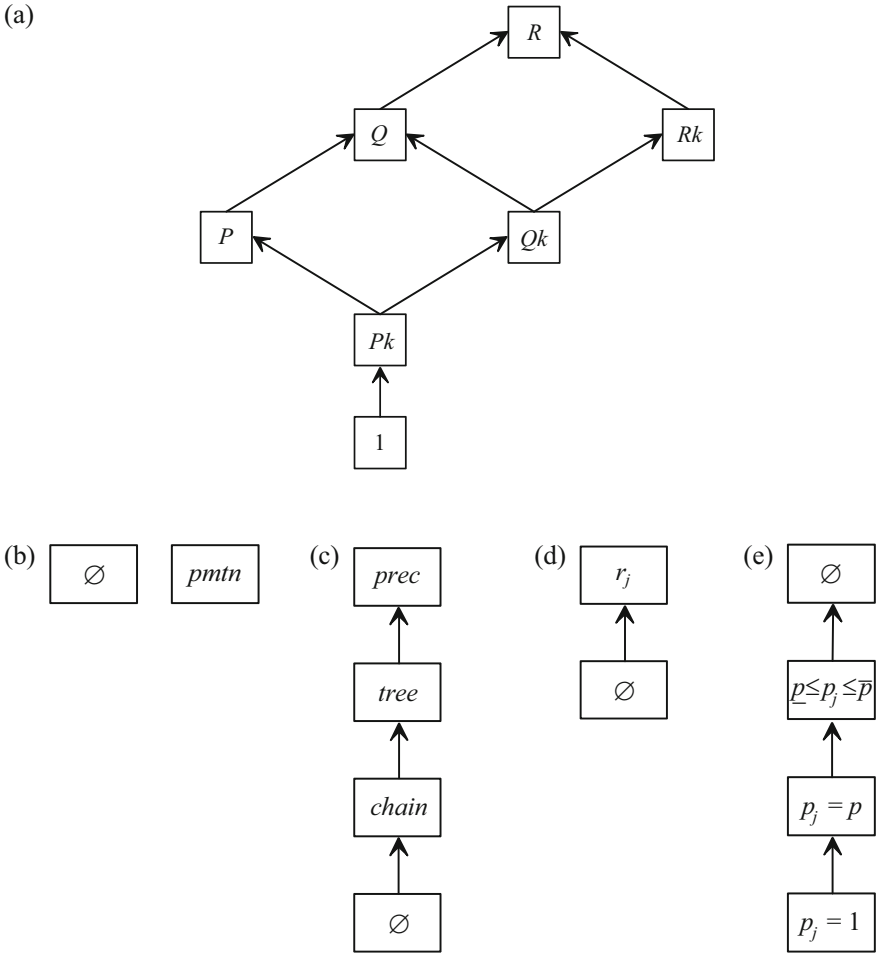
The use of this notation is illustrated by Example 3.4.1.

Example 3.4.1

(a) Problem $P||C_{max}$ reads as follows: *Scheduling of non-preemptable and independent tasks of arbitrary processing times (lengths), arriving to the system at time* 0, *on parallel, identical processors in order to minimize schedule length.*

(b) $O3|pmtn, r_j|\Sigma C_j$ stands for: *Preemptive scheduling of arbitrary length tasks arriving at different time moments in the three machine open shop, where the objective is to minimize mean flow time.* □

At this point it is worth mentioning that scheduling problems are closely related in the sense of polynomial transformation [8]. Some basic polynomial transformations between scheduling problems are shown in Figure 3.4.1. For each graph in the figure, the presented problems differ only by one parameter (e.g. by type and number of processors, as in Figure 3.4.1(a)) and the arrows indicate the direction of the polynomial transformation. These simple transformations are very useful in many situations when analyzing new scheduling problems. Thus, many of the results presented in this book can immediately be extended to cover a broader class of scheduling problems.



[8] This term has been explained in Section 2.2.

(f)

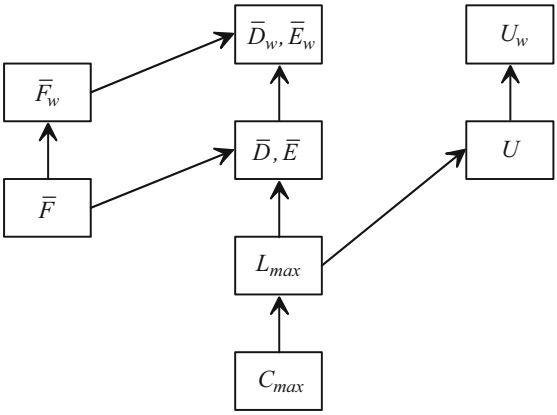


Figure 3.4.1 *Graphs showing interrelations among different values of particular parameters*
(a) *processor environment*
(b) *possibility of preemption*
(c) *precedence constraints*
(d) *ready times*
(e) *processing times*
(f) *optimality criteria.*

References

ACM70 ACM Record of the project MAC conference on concurrent system and parallel computation, Wood's Hole, Mass., 1970.

AHU74 A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

Bae74 J. L. Baer, Optimal scheduling on two processors of different speeds, in: E. Gelenbe, R. Mahl (eds.), *Computer Architecture and Networks*, North-Holland, Amsterdam, 1974.

BCSW86 J. Błażewicz, W. Cellary, R. Słowiński, J. Węglarz, *Scheduling under Resource Constraints: Deterministic Models*, J. C. Baltzer, Basel, 1986.

Ber66 A. J. Bernstein, Analysis of programs for parallel programming, *IEEE Trans. Comput.* EC-15, 1966, 757-762.

BK02 J. Blazewicz, D. Kobler, Review of properties of different precedence graphs for scheduling problems, *Eur. J. Oper. Res.* 142, 2002, 435-443.

BLRK83 J. Błażewicz, J. K. Lenstra, A. H. G. Rinnooy Kan, Scheduling subject to resource constraints: classification and complexity, *Discret Appl. Math.* 5, 1983, 11-24.

CD73 E. G. Coffman, Jr., P. J. Denning, *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

Coo71 S. A. Cook, The complexity of theorem proving procedures, *Proceedings of the 3rd ACM Symposium on Theory of Computing,* 1971, 151-158.

GJ79 M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

GLL+79 R. L. Graham, E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, Optimization and approximation in deterministic sequencing and scheduling theory: a survey, *Annals of Discrete Mathematics* 5, 1979, 287-326.

Kar72 R. M. Karp, Reducibility among combinatorial problems, in: R. E. Miller, J. W. Thatcher (eds.), *Complexity of Computer Computations*, Plenum Press, New York, 1972, 85-103.

RG69 C. V. Ramamoorthy, M. J. Gonzalez, A survey of techniques for recognizing parallel processable streams in computer programs, *AFIPS Conference Proceedings, Fall Joint Computer Conference,* 1969, 1-15.

Rus69 E. C. Russel, *Automatic Program Analysis*, Ph.D. thesis, Department of Engineering, University of California, Los Angeles, 1969.

Vol70 S. Volansky, *Graph Model Analysi and Implementation of Computational Sequences*, Ph.D. thesis, Report No. UCLA-ENG-7048, School of Engineering Applied Sciences, University of California, Los Angeles, 1970.