ELSEVIER

European Journal of Operational Research 121 (2000) 1–15

EUROPEAN JOURNAL OF OPERATIONAL RESEARCH

www.elsevier.com/locate/orms

Invited Review

Scheduling with limited machine availability ☆

Günter Schmidt *

Lehrstuhl für Informations-und Technologiemanagement, Fachbereich Wirtschaftswissenschaft, Universität des Saarlandes, Postfach 151150, D-66041 Saarbrücken, Germany

Received 1 November 1998

Abstract

This paper reviews results related to deterministic scheduling problems where machines are not continuously available for processing. There might be incomplete information about the points of time machines change availability. The complexity of single and multi machine problems is analyzed considering criteria on completion times and due dates. The review mainly covers intractability results, polynomial optimization and approximation algorithms. In some places too results from enumerative algorithms and heuristics are surveyed. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Scheduling theory; Availability constraints; Algorithms

1. Introduction

In the scheduling theory the basic model assumes that all machines are continuously available for processing throughout the planning horizon. This assumption might be justified in some cases but it does not apply if certain maintenance requirements, breakdowns or other constraints that cause the machines not to be available for processing have to be considered.

Examples of such constraints can be found in many areas. Limited availabilities of machines may result from preschedules which exist mainly because most of the real world resource planning problems are dynamic. This means that the input data are being frequently updated. A natural approach to cope with a dynamic environment is to trigger a new planning horizon when the changes in the data justify it. However, due to many necessities, as process preparation for instance, it is mandatory to take results of earlier plans as fixed which obviously limits availability of resources for any subsequent plan. Consider e.g., MRP-II production planning systems when a rolling horizon approach is used for customer order assignment on a tactical level. Here consecutive time periods overlap where planning decisions taken in earlier

☆ This work has been partially supported by INTAS grant 96-0812.

* Tel.: +49 681 302 4559; fax: +49 681 302 4565; e-mail: gs@itm.uni-sb.de

0377-2217/00/$ – see front matter
PII: S0377-2217(98)00367-1

periods constrain those for later periods. Because of this arrangement, orders related to earlier periods are also assigned to time intervals of later periods causing the resources not to be available during these intervals for orders arriving after the planning decisions have been taken. The same kind of problem may be repeated on the operational level of production scheduling. Here processing of some jobs is fixed in terms of starting and finishing times and machine assignment. When new jobs are released to the shop floor there are already jobs assigned to time intervals and machines while the new ones have to be processed within the remaining free processing intervals.

Another application of limited machine availability comes from operating systems for mono- and multi-processors, where subprograms with higher priority will interfere with the current program executed. A similar problem arises in multi-user computer systems where the load changes during the usage. In big massively parallel systems it is convenient to change the partition of the processors among different types of users according to their requirements for the machine. Fluctuations related to the processing capacity could be modeled by intervals of different processor availability. Numerous other examples exist where the investigation of limited machine availability is of great importance and the practical need to deal with this type of problem has been proven by a growing demand for commercial software packages. Since some time, the analyses of these problems have also attracted many researchers.

In the following we will investigate scheduling problems with limited machine availability in greater detail. The review focuses on deterministic models with complete and incomplete information about the availability constraints. For stochastic scheduling problems with limited machine availability and prior distributions of the problem parameters see Refs. [33,34]. We will survey results for one machine, parallel machine and flow shop scheduling problems in terms of intractability and polynomial time algorithms. In some places also results from enumerative optimization algorithms and heuristics are analyzed. Doing this we will distinguish between non-preemptive and preemptive scheduling.

2. Problem definition

A machine system with limited availability is a set of machines (processors) which does not operate continuously; each machine is ready for processing only in certain time intervals of availability. Let $T = \{T_j | j = 1, \ldots, n\}$ be the set of tasks and $P = \{P_i | i = 1, \ldots, m\}$ be the set of machines with machine P_i only available for processing within S_i given time intervals $[B_i^s,\ F_i^s)$, $s = 1, \ldots, S_i$ and $B_i^{s+1} > F_i^s$ for all $s = 1, \ldots, S_{i-1}$. B_i^s denotes the start time and F_i^s the finish time of sth interval of availability of machine P_i. There might be complete or incomplete information available concerning the intervals. In some cases all B_i^s and F_i^s are known in advance; in other cases only some of them are known. It might also happen that there is no prior knowledge about machine availability at all.

Each task T_j has a processing requirement of p_j time units. In set T precedence constraints among tasks may be defined. $T_i < T_j$ means that the processing of T_i must be completed before T_j can be started. The tasks in set T are called dependent if the order of execution of at least two tasks in T is restricted by this relation. Then these relations may be modeled by a precedence graph. Otherwise, the tasks are called independent.

Each machine may work only on one task at a time, and each task may be performed by only one machine at a time. Machines may be either parallel, i.e., performing the same functions, or dedicated i.e., being specialized for the execution of certain tasks. If all processors P_i from P have equal task processing speeds, then we call them identical. In case of dedicated processors there are three models of processing sets of tasks: flow shop, open shop and job shop. Later we will investigate flow shops. In such a system we assume that tasks form n subsets (chains). Each subset is called a job. That is, job J_j is divided into m tasks, T_{1j}, $T_{2j}, \ldots, T_{mj}$, and task T_{ij} will be performed on processor P_i. In addition, the processing of T_{i-1j} should precede that of T_{ij} for all $i = 2, \ldots, m$ and for all $j = 1, 2, \ldots, n$. The set of jobs will be denoted by J.

Each task (job) may be characterized by the following parameters:

- an arrival time (or ready time) r_j, which is the time at which task T_j (job J_j) is ready for processing; if the arrival times are the same for all tasks from T, then it is assumed that $r_j = 0$ for all T_j;
- a due date d_j, which specifies a time limit by which $T_j(J_j)$ should be completed; usually, penalty functions are defined in accordance with due dates;
- a deadline $\tilde{d}_j$, which is a 'hard' real time limit by which $T_j(J_j)$ must be completed;
- a weight (priority) w_j, which expresses the relative urgency of $T_j(J_j)$.

We want to find a feasible schedule (a time-based assignment of machines from set P to tasks from set T meeting all constraints) if one exists, such that all tasks can be processed within the given intervals of machine availability optimizing some performance criterion. Such measures considered here are completion time and due date related and most of them refer to the maximum completion time, the sum of completion times, and the maximum lateness.

A schedule is called preemptive if each task may be preempted at any time and restarted later at no cost, perhaps on another machine. If preemption of tasks is not allowed we will call the schedule non-preemptive.

In the following we base the discussion on the three fields $\alpha|\beta|\gamma$ classification scheme suggested in Ref. [5], that uses most features of the now classical scheduling theory notation. We add some entry denoting machine availability and we omit entries which are not relevant for the problems investigated here.

The first field $\alpha = \alpha_1\alpha_2\alpha_3$ describes the machine (processor) environment. Parameter $\alpha_1 \in \{\emptyset, P, Q, F\}$ characterizes the machine system used:

- $\alpha_1 = \emptyset$: single machine,
- $\alpha_1 = P$: identical machines (parallel machine system with the same speed factor),
- $\alpha_1 = Q$: uniform machines (parallel machine system with different speed factors),
- $\alpha_1 = F$: dedicated machines (flow shop system).

Parameter $\alpha_2 \in \{\emptyset, k\}$ denotes the number of machines (for parallel machine systems) or the number of stages (for dedicated machine systems):

- $\alpha_2 = \emptyset$: the number of machines (stages) is assumed to be variable,
- $\alpha_2 = k$: the number of machines (stages) is equal to k (k is a positive integer).

In Refs. [41,32] different patterns of availability are discussed for the case of parallel machine systems. These are constant, zigzag, decreasing, increasing and staircase. Let $0 = t_1 < t_2 < \cdots < t_j < \cdots < t_q$ be the points in time where the availability of a certain machine changes and let $m(t_j)$ be the number of machines being available during time interval $[t_j, t_{j+1})$ with $m(t_j) > 0$. It is assumed that the pattern is not changed infinitely often during any finite time interval. According to these cases, parameter $\alpha_3 \in \{\emptyset, NC_{zz}, NC_{inc}, NC_{dec}, NC_{inczz}, NC_{deczz}, NC_{sc}, NC_{win}\}$ denotes the machine availability.

(1) If all machines are continuously available ($t = 0$) then the pattern is called constant ($\alpha_3 = \emptyset$).

(2) If there are only k or $k-1$ machines in each interval available, then the pattern is called zigzag ($\alpha_3 = NC_{zz}$).

(3) A pattern is called increasing (decreasing) if for all j from IN_+

$$m(t_j) \geqslant \max_{1 \leqslant u \leqslant j-1} \{m(t_u)\}$$

$$(m(t_j) \leqslant \min_{1 \leqslant u \leqslant j-1} \{m(t_u)\}),$$

i.e., the number of machines available in interval $[t_{j-1}, t_j)$ is not more (less) than this number in interval $[t_j, t_{j+1})$ ($\alpha_3 \in \{NC_{inc}, NC_{dec}\}$).

(4) A pattern is called increasing (decreasing) zigzag if for all j from IN_+

$$m(t_j) \geqslant \max_{1 \leqslant u \leqslant j-1} \{m(t_u) - 1\}$$

$$(m(t_j) \leqslant \min_{1 \leqslant u \leqslant j-1} \{m(t_u) + 1\})$$

$$(\alpha_3 \in \{NC_{inczz},\ NC_{deczz}\}).$$

(5) A pattern is called staircase if for all intervals the availability of machine P_i implies the availability of machine P_{i-1} ($\alpha_3 = NC_{sc}$). A staircase pattern is shown in the lower part of Fig. 1; dark areas represent intervals of non-availability. Note that patterns (1)–(4) are special cases of (5).

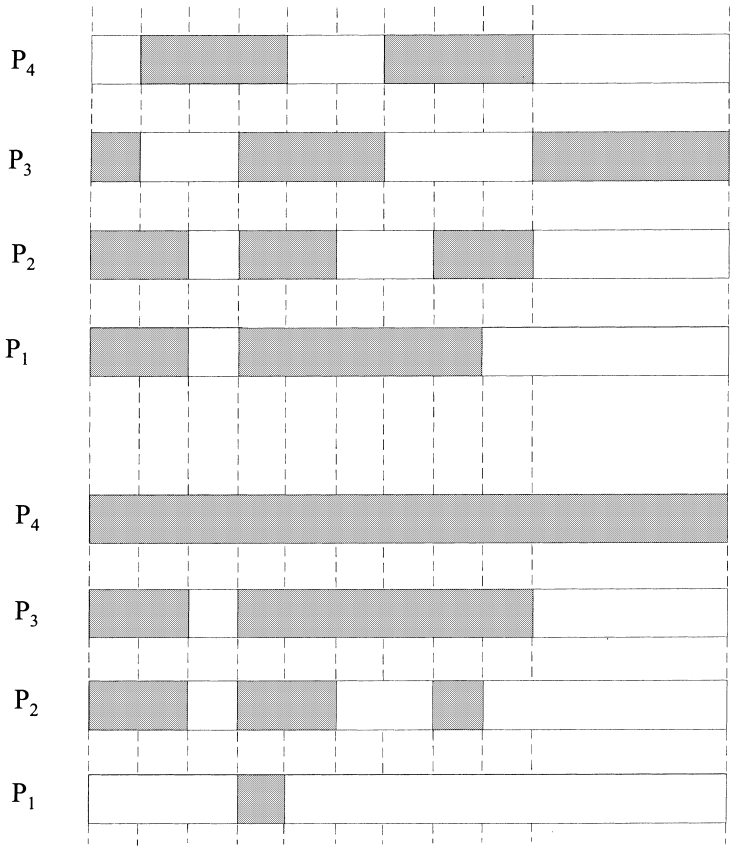


Fig. 1. Rearrangement of arbitrary patterns.

(6) A pattern is called arbitrary if none of the conditions (1)–(5) applies ($\alpha_3 = NC_{\text{win}}$). Such a pattern is shown in the upper part of Fig. 1; patterns defined in (1)–(5) are special cases of the one in (6).

Machine systems with arbitrary patterns of availability can always be translated to a composite machine system forming a staircase pattern [41]. A composite machine is an artificial machine consisting of at most m original machines. The transformation process works in the following way. An arbitrary pattern is separated in as many time intervals as there are distinct points in time where the availability of at least one machine changes. Now in every interval periods of non-availability are moved from machines with smaller index to machines with greater index. If there are $m(t_j)$ machines available in some interval $[t_j, t_j + 1)$ then after the transformation machines $P_1, \ldots, P_{m(t_j)}$ will be available in $[t_j, t_j + 1)$ and $P_{m(t_j)+1}, \ldots, P_m$ will not be available, where $0 < m(t_j) < m$. Doing this for every interval we generate composite machines. Each of them consists of at most m original machines with respect to the planning horizon.

An example for such a transformation considering $m = 4$ machines is given in Fig. 1. Non-availability is represented by the dark areas. Composite machines which do not have intervals of availability can be omitted from the problem description. Then the number of composite machines in each interval is the maximum number of machines simultaneously available. The time complexity of the transformation is $O(qm)$ where q is the number of points in time, where the availability of an original machine is changing. If this

number is polynomial in n or m machine scheduling problems with arbitrary patterns of non-availability can be transformed in polynomial time to a staircase pattern. This transformation is useful as, first, availability at time t is given by the number of available composite machines and, second, some results are obtained assuming this hypothesis.

The second field $\beta = \beta_1, \ldots, \beta_8$ describes task (job) and resource characteristics. We will only refer here to parameters β_1, β_3, β_4, and β_6.

Parameter $\beta_1 \in \{\emptyset, t - \text{pmtn}, \text{pmtn}\}$ indicates the possibilities of preemption:

- $\beta_1 = \emptyset$: no preemption is allowed,
- $\beta_1 = t - \text{pmtn}$: tasks may be preempted, but each task must be processed by only one machine,
- $\beta_1 = \text{pmtn}$: tasks may be arbitrarily preempted.

In Ref. [28] the notion of resumability and non-resumability of tasks is introduced. Resumability corresponds to the case where task preemption is allowed; non-resumability defines the case where preemption is not allowed. Here we assume that not only task ($\beta_1 = t - \text{pmtn}$) but also arbitrary (task and machine) preemptions are possible ($\beta_1 = \text{pmtn}$). If there is only one machine dedicated to each task then task preemptions and arbitrary preemptions become equivalent. For single machine and flow shop problems this difference has not to be considered. Of course the rearrangement of an arbitrary pattern to a staircase pattern is only used when arbitrary preemption is allowed. In what follows, the number of preemptions may be a criterion to appreciate the value of an algorithm. When the algorithm applies to staircase patterns, the number of preemptions for an arbitrary pattern is increased by at most mq.

Parameter $\beta_3 \in \{\emptyset, \text{prec}, \text{tree}, \text{forest}, \text{chains}\}$ reflects the precedence constraints and denotes, respectively, independent tasks, arbitrary precedence constraints, precedence constraints forming a tree a set of trees or a set of chains.

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

- $\beta_4 = \emptyset$: all ready times are zero,
- $\beta_4 = r_j$: ready times differ per task (job).

Parameter $\beta_6 \in \{\emptyset, \tilde{d}\}$ describes deadlines:

- $\beta_6 = \emptyset$: no deadlines are assumed (however, due dates may be defined if a due date involving criterion is used to evaluate schedules),
- $\beta_6 = \tilde{d}$: deadlines are imposed on the performance of a task (job) set.

The third field, γ, denotes an optimality criterion (performance measure), i.e., $\gamma \in \{C_{\max}, \sum C_j, \sum w_j C_j, L_{\max}, \sum U_j, \sum w_j U_j, \emptyset\}$, where $C_{\max}$ refers to minimizing the makespan ($\max\{C_j\}$), $\sum C_j$ to the sum of completion times, $\sum w_j C_j$ to the sum of weighted completion times, $L_{\max}$ to the maximum lateness ($\max\{C_j - d_j\}$), $\sum U_j$ to the sum (or number) of tardy jobs ($C_j > d_j$), $\sum w_j U_j$ to the weighted sum of tardy jobs, and $\emptyset$ means testing for feasibility whenever scheduling to meet deadlines is considered.

In order to solve these problems, different kind of algorithms will be applied. Information about machine availabilities might be complete or incomplete. In an on-line setting machine availabilities are not known in advance. Unexpected machine breakdowns are a typical example of events that arise on-line. Sometimes schedulers have partial knowledge of the availabilities, i.e., they have some look-ahead. They might know of the next time interval where a machine requires maintenance or they might know when a broken machine will be available again. In an off-line setting we assume complete information, i.e., all machine availabilities are known prior to schedule generation. From these possibilities we can differ between different types of algorithms.

- An algorithm is on-line if it proceeds sequentially and at each time t it only needs to know the number of processors available at t, the number of ready tasks at t, their remaining processing time and their deadlines or due dates.
- An algorithm is nearly on-line if it needs in addition at time t the time of the next event, that is either a new task becomes ready for processing or the number of available machines change ([40], extended from [20]).
- An algorithm is off-line if all problem data have to be known in advance. That is, at time 0 it needs all information concerning machine availabilities and task characteristics.

If the machine non-availabilities are due to unexpected breakdowns, on-line algorithms are needed. If the times of machine availability changes are known a little in advance, nearly on-line algorithms will suffice. Otherwise off-line algorithms

will do. Note that it is often assumed in the on-line scheduling literature (see for instance [7]) that even processing times of tasks are not known before the processing begins. This setting is not investigated here. Most of the results which are reviewed later relate to off-line algorithms. In some places we also survey on-line and nearly on-line algorithms.

We investigate the problems according to their computational tractability and use the concepts of complexity as defined in Ref. [10]. We analyze the time complexity of algorithms by $O(g(k))$ where g is some function and k is the input length of a problem instance. We will also report on approximation algorithms in case the scheduling problem cannot be solved to optimality for some reason. For off-line settings we will distinguish between relative and absolute errors of an algorithm. The relative error R_H is defined as

$$R_H(I) = C_{H(I)}/C_{\text{opt}}(I) - 1,$$

where $C_{H(I)}$ is the performance of algorithm H applied to some problem instance I and $C_{\text{opt}}(I)$ is the value of the corresponding optimal solution. The absolute error A_H is defined as

$$A_{H(I)} = C_{H(I)} - C_{\text{opt}}(I).$$

In case we investigate on-line settings we refer to a competitive analysis. Following [43] we call an on-line scheduling algorithm H to be c-competitive if, for all problem instances I, $C_{H(I)} \leqslant c \cdot C_{\text{opt}}^{\text{OFF}}(I)$ where $C_{\text{opt}}^{\text{OFF}}(I)$ is the value of the corresponding optimal off-line solution.

Many of the problems considered later are solved applying simple priority rules which can be executed in $O(n \log n)$ time. The rules order the tasks in some way and then iteratively assign them to the most lightly loaded machine. The following rules are the most prominent.

- Shortest Processing Time (SPT) rule. With this rule the tasks are ordered according to non-decreasing processing times.
- Longest Processing Time (LPT) rule. The tasks are ordered according to non-increasing processing times.
- Earliest Due Date (EDD) rule. Applying this rule all tasks are ordered according to non-decreasing due dates.

3. One machine problems

One machine problems are of fundamental character. They can be interpreted as building blocks for more complex problems. Such formulations may be used to represent bottleneck machines or an aggregation of a machine system. For one machine scheduling problems the only availability pattern which has to be investigated is a special case of zigzag with $k = 1$.

Let us consider first problems where preemption of tasks (jobs) is not allowed. If there is only a single interval of non-availability with $B_i > 0$ and $F_i < \sum_j p_j$ and $\sum C_j$ is the objective $(1, NC_{\text{win}}|| \sum C_j)$ Adiri et al. [1] show that the problem is NP-complete. The SPT rule leads to a tight relative error of $R_{\text{SPT}} \leqslant 2/7$ for this problem [26]. It is easy to see that also problem $1, NC_{\text{win}}||C_{\max}$ is NP-complete [24].

If preemption is allowed the scheduling problem becomes easier. For $1, NC_{\text{win}}|\text{pmtn}|C_{\max}$, it is obvious that every schedule is optimal which starts at time zero and has no unforced idle time, that is, the machine never remains idle while some task is ready for processing. It is trivial to construct such a schedule which is optimal for off-line and on-line settings. Preemption is never useful except when some task cannot be finished before an interval of non-availability occurs. This property is still true for completion time-based criteria if there is no precedence constraint and no release date, as it is assumed in the rest of this section.

While the sum of completion times $(1, NC_{\text{win}}|\text{pmtn}| \sum C_j)$ is minimized by the SPT rule the problem of minimizing the weighted sum $(1, NC_{\text{win}}|\text{pmtn}| \sum w_j C_j)$ is NP-complete [24]. Note that without availability constraints Smith's rule [44] solves the problem. Maximum lateness is minimized by the Earliest Due Date (EDD) rule [24]. If the number of tardy tasks has to be minimized $(1, NC_{\text{win}}|\text{pmtn}| \sum U_j)$ the EDD rule of Moore and Hodgson's algorithm [37] can be modified to solve this problem also in $O(n \log n)$ time [24]. Note that if we add release times or weights for the jobs the problem is NP-complete already for a continuously available machine ([29] or [16]).

4. Parallel machine problems

In this section we cover off-line and on-line formulations of parallel machine scheduling problems with availability constraints. Most results which are presented refer to off-line problems; results for on-line settings are explicitly mentioned.

4.1. Minimizing the sum of completion times

In case of continuous availability of the machines ($P||\sum C_j$) the problem can be solved applying the SPT rule. If machines have only different beginning times B_i (this corresponds to an increasing pattern of availability) the problem can also be solved by the SPT rule [17,30]. If $m = 2$ and there is only one finish time F_i^s on one machine which is smaller than infinity (this corresponds to a zigzag pattern of availability) the problem becomes NP-complete [27]. In the same paper Lee and Liman show that the SPT rule with the following modification leads to a tight relative error of $R_{\mathrm{SPT}} \leqslant 1/2$ for $P2, NC_{\mathrm{zz}}||\sum C_j$ where machine P_1 is continuously available and machine P_2 has one finish time which is smaller than infinity.

Step 1: Assign the shortest task to P_1.

Step 2: Assign the remaining tasks in SPT order alternately to both machines until some time when no other task can be assigned to P_2 without violating F_2.

Step 3: Assign the remaining tasks to P_1.

Fig. 2 illustrates how that bound can be reached asymptotically (when ϵ tends toward 0). The modified SPT rule leads to a large idle time for machine P_1. For fixed m the SPT rule is asymptotic optimal if there is not more than one interval of non-availability for each machine [39].

4.2. Minimizing the makespan

Let us first investigate non-preemptive scheduling. Ullman [45] was the first to study the problem $P, NC_{\mathrm{win}}||C_{\max}$. It is NP-complete in the strong sense for arbitrary m (3-partition is a special case) even if the machines are continuously available. If machines have different beginning times B_i $(P, NC_{\mathrm{inc}}||C_{\max})$ the Longest Processing Time (LPT) rule leads to a relative error of $R_{\mathrm{LPT}} \leqslant 1/2 - 1/(2m)$ or of $R_{\mathrm{MLPT}} \leqslant 1/3$ if the rule is appropriately modified [23]. Both bounds are tight. The modification uses dummy tasks to simulate the different machine starting times B_i. For each machine P_i, a task T_j with processing time $p_j = B_i$ is inserted. The dummy tasks are merged into the original task set and then all tasks are scheduled according to the LPT rule under an additional restriction that only one dummy task is assigned to each machine. After finishing the schedule, all dummy tasks are moved to the head of the machines followed by the remaining tasks assigned to

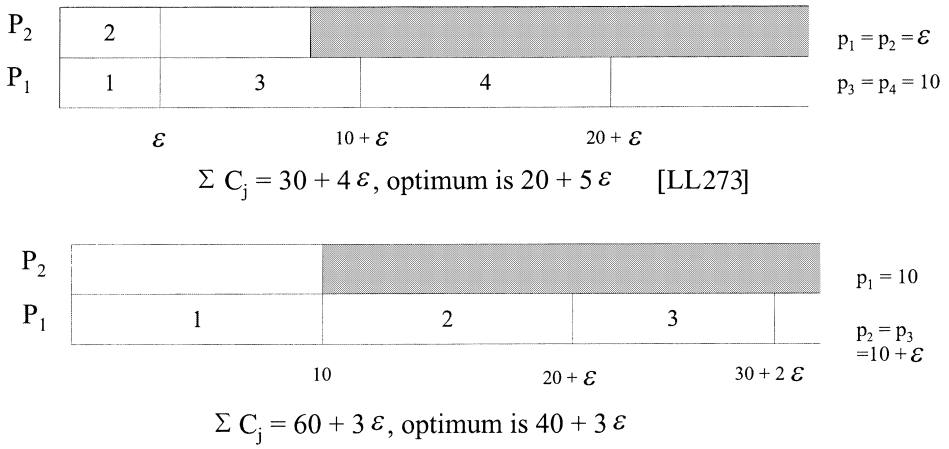


Fig. 2. Examples for the modified SPT rule.

each P_i. The MLPT rule runs in $O((n+m)\log(n+m)+(n+m)m)$ time. In [31] Lee's bound of 1/3 reached by MLPT is improved to 1/4. Note that the LPT algorithm leads to a relative error of $R_{LPT} \leqslant 1/3 - 1/(3m)$ for continuously available machines [11]. Kellerer [18] presents a dual approximation algorithm using a bin packing approach leading to a tight bound of 1/4, too.

Now let us investigate results for preemptive scheduling. If all machines are only available in one and the same time interval $[B,F)$ and tasks are independent the problem is of type $P|\text{pmtn}|C_{\max}$. Following McNaughton [36] it can be shown that there exists a feasible machine preemptive schedule if and only if

$$\max_j\{p_j\} \leqslant (F-B) \text{ and } \sum_j p_j \leqslant m(F-B).$$

There exists an $O(n)$ algorithm which generates at most $m-1$ preemptions to construct this schedule. If all machines are available in an arbitrary number $S=\sum_i S_i$ of time intervals $[B_i^s, F_i^s)$, $s=1,\ldots,S_i$, and the machine system forms a staircase pattern, Schmidt [41] generalizes McNaughton's condition and shows a feasible preemptive schedule exists if and only if the following m conditions are met:

$$\forall k = 1 \to m-1, \sum_{j=1}^{k} p_j \leqslant \sum_{i=1}^{k} PC_i \quad \mathscr{P}(k),$$

$$\sum_{j=1}^{n} p_j \leqslant \sum_{i=1}^{m} PC_i \quad \mathscr{P}(m),$$

with

$$p_1 \geqslant p_2 \geqslant \cdots \geqslant p_n$$

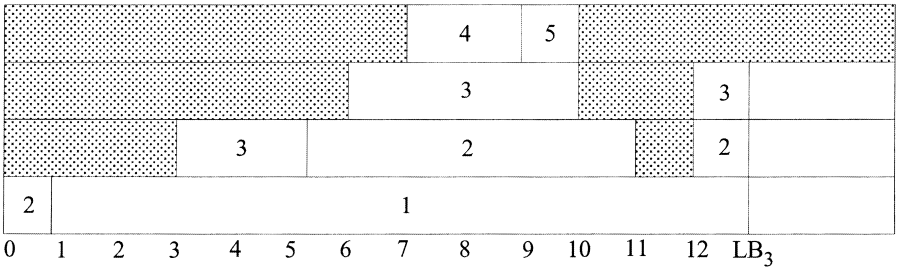
and

$$PC_1 \geqslant PC_2 \geqslant \cdots \geqslant PC_m,$$

where PC_i is the total processing capacity of machine P_i. Such a schedule can be constructed in $O(n+m\log m)$ time after the processing capacities PC_i are computed, with at most $S-1$ preemptions in case of a staircase pattern (remember that any arbitrary pattern of availability can be converted into a staircase one at the price of additional preemptions). Note that in the case of the same availability interval $[B,F)$ for all machines, McNaughton's conditions are obtained from $\mathscr{P}(1)$ and $\mathscr{P}(m)$ alone. This remains true for zigzag patterns as then $\mathscr{P}(2),\ldots,\mathscr{P}(m-1)$ are always verified if $\mathscr{P}(1)$ is true (there is one availability interval for all machines but P_m). In [42] the problem is generalized taking into account different task release times or deadlines. Then it can be solved in $O(nm\log m)$.

The corresponding optimization problem $(P, NC_{\text{sc}}|\text{pmtn}|C_{\max})$ is solved by an algorithm that first computes the lower bounds $LB_1,\ldots,LB_m$ obtained from the conditions above (see Fig. 3). $C_{\max}$ cannot be smaller than LB_k, $k=1\to m-1$, obtained from $\mathscr{P}(k)$. The sum of availabilities of machines $P_1,\ldots,P_k$ during time interval $[0,LB_k)$ may not be smaller than the sum of processing times of tasks $T_1,\ldots,T_k$. The sum of all machine availabilities during time interval $[0,LB_m)$ must also be larger than or equal to the sum of processing times of all tasks. In the example of Fig. 3, $C_{\max}=LB_3$. The number of preemptions is $S-2$.

When precedence constraints are added, Liu and Sanlaville [32] show that problems with chains and arbitrary patterns of non-availability $(P, NC_{\text{win}}|\text{pmtn}, \text{chains}|C_{\max})$ can be solved in polynomial time applying the Longest Remaining Path (LRP) first rule and the processor sharing procedure of [35]. In the same paper it is also shown that the LRP rule could be used to solve problems with decreasing (increasing) zigzag patterns and tasks forming an outforest (inforest) $(P, NC_{\text{deczz}}|\text{pmtn}, \text{out-forest}|C_{\max}$ or $P, NC_{\text{inczz}}|\text{pmtn}, \text{in-forest}|C_{\max})$. In case of only two machines and arbitrary (which means zigzag for $m=2$) patterns of non-availability $(P2, NC_{\text{win}}|\text{pmtn}, \text{prec}|C_{\max})$ this rule also solves problems with arbitrary task precedence constraints with time complexity and number of preemptions of $O(n^2)$. These results are deduced from these for Unit Execution Time scheduling by list algorithms [8,9]. The LRP algorithm is nearly on-line, as are all priority algorithms which extend list algorithms to preemption [21]. Indeed these algorithms first build a schedule admitting processor sharing. These schedules execute tasks of the same priority at the same speed. This property is respected when McNaughton's rule is applied. If a machine availability changes



T_j	1	2	3	4	5
p_j	12	7	7	2	1
LB_i	12	11	38/3	149/12	38/3

Fig. 3. Minimizing the makespan on a staircase pattern.

unexpectedly, the property does not hold any more (see results discussed later in this section).

Applying the LRP rule results in a time complexity of $O(n\log n + nm)$ and a number of preemptions of $O((n+m)^2 - nm)$ which both can be improved. Therefore in Ref. [4] an algorithm is given which solves problem $P, NC_{\text{win}}|\text{pmtn}, \text{chains}|C_{\max}$ with $N < n$ chains in $O(N + m\log m)$ time generating a number of preemptions which is not greater than the number of intervals of availability of all machines. If all machines are only available in one processing interval and all intervals are ordered in a staircase pattern the algorithm generates feasible schedules with at most $m-1$ preemptions. This result is based on the observation that preemptive scheduling of chains for minimizing schedule length can be solved by applying an algorithm for the independent tasks problem. Having more than two machines in the case of arbitrary precedence constraints or an arbitrary number of machines in the case of a tree precedence structure makes the problem NP-complete [4].

If we give up the assumption that all intervals of non-availability are known in advance on-line or nearly on-line algorithms are required for the problem solution. In order to deal with unexpected machine breakdowns on-line algorithms have to be applied. This problem is studied by Kalyanasundaram and Pruhs [14,15]. In Ref. [14] the competitive ratios of on-line algorithms are analyzed for various numbers of faulty machines. The authors assume that if a machine breaks down, the task currently being processed has to be resumed later from the beginning. Also two specific types of breakdowns are considered. In a permanent breakdown a machine does not recover again; in a transient breakdown the machine is available again right after the breakdown. In Ref. [15] it is examined to which extent redundancy can help in on-line scheduling with faulty machines.

In Ref. [2] it is shown that no on-line algorithm can construct optimal makespan schedules if machines change availability at arbitrary time instances. It is also impossible for such an algorithm to guarantee that the solution is within a constant competitive ratio c if there may be time intervals where no machine is available. To see this we use the following argument. Let H be any on-line algorithm. Initially, at time $t = 0$ only one machine is available. We consider n jobs $J_1, \ldots, J_n$, each of which has a processing time of 1 time unit. At time $t = 0$, algorithm H starts processing one job J_{j_0}. Let t' be the first time instance such that H first preempts J_{j_0} or H finishes processing J_{j_0}. At that time t' all machines become available. H's makespan is at least $t' + 1$ because none of the jobs J_j, $j \neq j_0$, has been processed so far. An optimal

off-line algorithm will divide the interval from 0 to t' evenly among the n jobs so that its makespan is $C_{\max}^{\mathrm{OFF}} = t' + 1 - (t'/n)$. To see that a constant c cannot be guaranteed we modify the problem instance so that no machine is available during the interval $(C_{\max}^{\mathrm{OFF}}, c \cdot C_{\max}^{\mathrm{OFF}}]$. The algorithm H cannot finish before $c \cdot C_{\max}^{\mathrm{OFF}}$ because it has jobs left at time $C_{\max}^{\mathrm{OFF}}$.

Albers and Schmidt also report that things look better if the algorithm is allowed to be nearly on-line. In such a case we assume that the algorithm always knows the next point in time when the set of available machines changes. Now optimal schedules can be constructed. The algorithm presented has a running time of $O(qn + S)$, where q is the number of time instances where the set of available machines changes and S is the total number of intervals where machines are available. If at any time at least one machine is available an on-line algorithm can construct schedules which differ by an absolute error A from an optimal schedule for any $A > 0$. This implies that not knowing machine availabilities does not really hurt the performance of an algorithm if arbitrary preemption is allowed.

4.3. Dealing with due date involving criteria

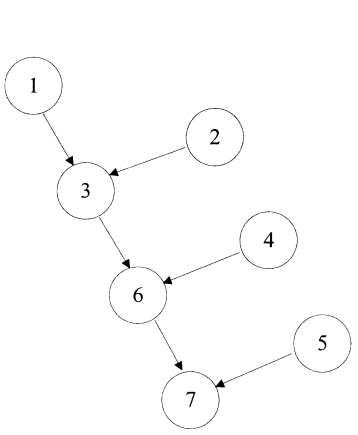
In Ref. [12] it is shown that $P|\mathrm{pmtn}, r_j, \tilde{d}_j|$ can be solved in $O(n^3 \min\{n^2, \log n + \log p_{\max}\})$ time. The same flow-based approach can be coupled with a bisection search to minimize maximum lateness $L_{\max}$ (see Ref. [20], where the method is also extended to uniform machines). A slightly modified version of the algorithm still applies to the corresponding problem where the machines are not continuously available. If the number of changes of machine availabilities during any time interval is linear in the length of the interval this approach can be implemented in $O(n^3 p_{\max}^3(\log n + \log p_{\max}))$ [40]. These algorithms need the knowledge of all the data at time 0 and are hence off-line. When no release dates are given but due dates have to be considered maximum lateness can be minimized $(P, NC_{\mathrm{win}}|\mathrm{pmtn}|L_{\max})$ using the approach suggested by [42] in $O(nm \log n)$ time. The method needs just to know all possible events before the next due date.

If there are not only due dates but also release dates to be considered $(P, NC_{\mathrm{win}}|r_j, \mathrm{pmtn}|L_{\max})$ Sanlaville [40] suggests a nearly on-line priority algorithm with an absolute error of $A \leqslant (m - 1/m)p_{\max}$ if the availability of the machines follows a constant pattern and of $A \leqslant p_{\max}$ if machine availability refers to an increasing zigzag pattern. The priority is calculated according to the Smallest Laxity First (SLF) rule, where laxity (or slack time) is the difference between the task's due date and its remaining processing time. The SLF algorithm runs in $O(n^2 p_{\max})$ and it is optimal in the case of a zigzag pattern and no release dates.

Liu and Sanlaville [32] show that results on $C_{\max}$ minimization for in-forest precedence graphs and increasing zigzag patterns $(P, NC_{\mathrm{inczz}}|\mathrm{pmtn}, \text{in-forest}|C_{\max})$ can be extended to $L_{\max}$, using SLF rule on the modified due dates. Fig. 4 shows an optimal SLF schedule for an in-tree. The modified due date is given by $d'_j = \min(d_j, d'_{s(j)} + p_{s(j)})$ where $T_{s(j)}$ is the successor of T_j when it exists. In the same way, minimizing $L_{\max}$ on two machines with availability constraints is achieved using SLF with a different modification scheme. If there are due dates, release dates and chain precedence constraints to be considered $(P, NC_{\mathrm{win}}|r_j, \mathrm{chains}, \mathrm{pmtn}|L_{\max})$ the problem can be solved using a binary search procedure in combination with a linear programming formulation [3].

Lawler and Martel [22] solved the weighted number of tardy jobs problem on two uniform machines, i.e., $Q2|\mathrm{pmtn}|\sum w_j U_j$. The originality of their paper comes from the fact that they show a stronger result, as the speeds of the processors may change continuously (and even be 0) during the execution. Hence it includes as a special case availability constraints on two uniform machines. They use dynamic programming to propose pseudo-polynomial algorithms ($O(\sum w_j n^2)$, or $O(n^2 p_{\max})$ to minimize the number of tardy jobs). Nothing however is said about the effort needed to compute processing capacity in one interval.

If there are more than two uniform machines to be considered and the problem is to minimize maximum lateness for jobs which have different release dates $(Q, NC_{\mathrm{win}}|r_j, \mathrm{pmtn}|L_{\max})$ the problem can be solved in polynomial time by a combined strategy of binary search and network flow [4]. In



T_j	1	2	3	4	5	6	7
p_j	2	3	2	2	5	3	2
d_j	8	7	8	5	7	9	10
d'_j	4	4	6	5	7	8	10

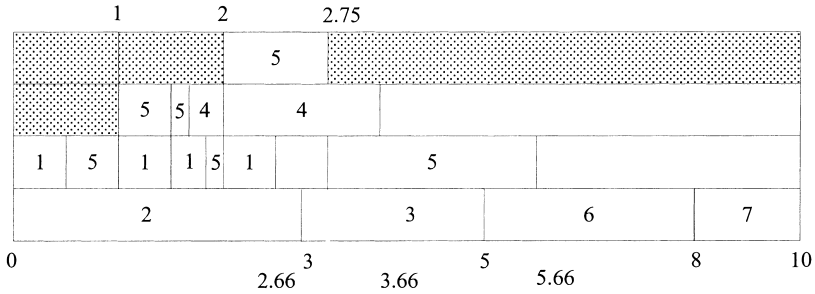


Fig. 4. Minimizing $L_{\max}$ on an increasing zigzag pattern.

the same paper the problem is generalized taking unrelated machines, i.e., machine speeds cannot be represented by constant factors, into account. This problem can also be solved in polynomial time applying a combination of binary search and the two-phase method given in Ref. [5].

5. Flow shop problems

The flow shop scheduling problem for two machines with a constant pattern of availability minimizing $C_{\max}(F2||C_{\max})$ can be solved in polynomial time by Johnson's rule [13]. Lee [25] has shown that this problem becomes already NP-complete if there is a single interval of non-availability on one machine only. He also gives an approximation algorithm which has a relative error of 1/2 if this interval is on machine one or of 1/3 if the interval of non-availability is on machine two. Note that the classical flow shop is symmetrical, but the non-availability interval breaks the symmetry. The approximation algorithms are based on a combination of Johnson's rule and a modification of the ratio rule given in Ref. [38]. Lee also proposes a dynamic programming algorithm for the case with one interval only.

In Ref. [19] it is shown that the existence of approximation algorithms for flow shop scheduling problems with limited machine availability is more of an exception. It is proved that no polynomial time heuristic with a finite worst case bound can exist for $F2, NC_{zz}|t - \text{pmtn}|C_{\max}$ when at least two intervals of non-availability are allowed to occur. Furthermore it is shown that makespan minimization becomes NP-hard in the strong sense if arbitrary number of intervals occur on one machine only. On the other hand there

always exists an optimal schedule where the permutation of jobs scheduled between any two consecutive intervals obeys Johnson's order. However the question which jobs to assign between which intervals remains intractable.

Due to these negative results, a branch and bound algorithm is developed in Ref. [19] to solve $F2, NC_{\text{win}}|\text{pmtn}|C_{\max}$. The approach uses Johnson's order property of jobs scheduled between two consecutive intervals. This property helps to reduce the number of solutions to be enumerated. The results based on experiments with the branch and bound algorithm show that it is easier to deal with intervals of non-availability on the second machine than with these on the first one. This can be explained by the following asymmetry. Any interval of non-availability on the first machine may reduce inventory of jobs waiting in the intermediate buffer between the machines, this, in turn, may result in idleness of the second machine. On the other hand, no interval of non-availability on the second machine might result in idle time of the first machine. The branch and bound algorithms of Ref. [19] outperforms the dynamic programming algorithm proposed by Lee [25]. Nevertheless only 1354 instances out of 2100 could be solved to optimality within a time limit of 1000 seconds.

In order to speed up the solution process a parallel implementation of the branch and bound algorithm is presented in Ref. [6]. Computations have been performed on 1, 2, 3, up to 8 processors. The experiment has been based on instances for which computational times of the sequential version of the algorithm were long. The maximum speed up gained was between 1.2 and 4.8 in comparison to the sequential version for 8 processors being involved in the computation.

Based on these results in Ref. [3] constructive and improvement heuristics are designed for $F2, NC_{\text{win}}|\text{pmtn}|C_{\max}$. They are empirically evaluated using the test data from Ref. [19]. For these instances the optimal solution was not found by the branch and bound algorithm, lower bounds were calculated according to the following problem relaxations.

- Let J_x be a job with shortest processing time on machine P_1. Assume that all the other job processing times on P_1 are zero. A permutation of the n jobs having J_x in the first position gives the lower bound LB_1.
- Let J_y be a job with shortest processing time on machine P_2. Assume that all the other job processing times on P_2 are zero. A permutation of the n jobs having J_y in the last position gives the lower bound LB_2.
- The optimal solution for the problem where the machines are continuously available gives the lower bound LB_3.

Clearly, the minimum makespan cannot be shorter than the maximum of the three lower bounds. From the experiments it turned out that at least 2063 instances out of 2100 could be solved to optimality applying a combination of constructive and improvement heuristics. The time limit to achieve this result was set to 30 seconds for each instance. At least 2055 out of 2100 instances could be solved combining constructive methods only. The average computation time for this experiment was 3.58 seconds per instance. Problem instances which could not be solved to optimality with both kinds of combinations of heuristic algorithms had a worst relative performance of 3.03% and a mean relative performance of 0.197% above the lower bound. Most of the heuristics performed better when the number of jobs was increased. If the intervals of non-availability occurred on machine one the performance of the heuristics was worse than in the case when the intervals occurred on machine two only. This matches with the observations in Ref. [19]. The results in Ref. [3] suggest that heuristic algorithms are very good options for solving flow shop scheduling problems with limited machine availability.

6. Conclusions

We reviewed results on scheduling problems with limited machine availability. The number of results shows that scheduling with availability constraints attracts more and more researchers, as the importance of the applications are recognized. The results presented here are of various kinds. For very few cases there exist optimal on-line algorithms. More cases can be solved by nearly on-

line algorithms but the majority of cases can only be solved to optimality by off-line algorithms. For off-line settings either classical algorithms could be generalized to solve the problem in polynomial time, or it could be shown that the problem becomes NP-complete due to the availability constraints.

In particular, when preemption is not authorized it will logically entail NP-completeness of the problem. Moreover on-line and nearly on-line optimization algorithms do not exist in this case. If one is interested in off-line optimal solutions for non-preemptive problems enumerative algorithms have to be applied; if not approximation algorithms are a good choice. Performance bounds may often be obtained, but their quality will depend on the kind of availability patterns considered. If worst case bounds cannot be found heuristics which can only be evaluated empirically have to be applied. Most of the positive results only hold for single machine and parallel machine systems. Flow shop, open shop and job shop systems mainly require enumerative and heuristic algorithms. Investigating shop systems are a challenging field for further research.

We try to summarize most of the results reviewed in this paper in Tables 1 and 2. Table 1 differs for a given problem type between performance criteria entailing NP-completeness and those for which a polynomial algorithm exists. Table 2 distinguishes between problem types which can be solved to optimality ($c = 1$) in polynomial time by on-line and by nearly on-line algorithms. This table covers only preemptive scheduling problems because it is easy to show that if preemption is not allowed optimality cannot be reached by this type of algorithms.

If availability constraints come from unexpected breakdowns, fully on-line algorithms are needed. But many results of optimality concern at best nearly on-line algorithms (in case of preemptive scheduling). It is an open question to look for competitive ratios for fully on-line algorithms and specific availability patterns.

Table 1
Results for off-line settings

Problem	Polynomial criteria	NP-complete criteria
$1, NC_{\text{win}}$		$\sum C_j, C_{\max}$
$1, NC_{\text{win}}\|\text{pmtn}$	$\sum C_j,\ C_{\max}, L_{\max},$ $\sum U_j$	$\sum w_j C_j,$ $\sum w_j U_j$(constant availability)
P, NC_{inc}	$\sum C_j$	
P, NC_{zz}		$\sum C_j$
$P2, NC_{\text{win}}\|\text{pmtn}, \text{prec}$	$C_{\max}, L_{\max}$	
$P, NC_{\text{zz}}\|\text{pmtn}\|, \text{tree}$	$C_{\max},\ L_{\max}$ (in-tree)	$C_{\max}$ (for NC_{win})
$P, NC_{\text{win}}\|\text{pmtn}, \text{chains}$	$C_{\max},\ L_{\max}$	
$P, NC_{\text{win}}\|\text{pmtn}, \text{r}_j$	$C_{\max},\ L_{\max}$	
$Q, NC_{\text{win}}\|\text{pmtn}, \text{r}_j$	$C_{\max},\ L_{\max}$	
$F2, NC_{\text{win}}\|\text{pmtn}$		$C_{\max}$ (single non-availability interval)

Table 2
Results for on-line and nearly on-line settings

Problem	On-line algorithm	Nearly on-line algorithm
$1, NC_{\text{win}}\|\text{pmtn}$	$\sum C_j,\ C_{\max},\ L_{\max}, \sum U_j$	
$P2, NC_{\text{win}}\|\text{pmtn}, \text{prec}$		$C_{\max},\ L_{\max}$
$P, NC_{\text{zz}}\|\text{pmtn}, \text{tree}$		$C_{\max},\ L_{\max}$(in-tree)
$P, NC_{\text{win}}\|\text{pmtn}, \text{chains}$		$C_{\max},\ L_{\max}$
$P, NC_{\text{win}}\|\text{pmtn}, \text{r}_j$		$C_{\max}$

References

[1] I. Adiri, J. Bruno, E. Frostig, A.H.G. Rinnooy Kan, Single machine flow-time scheduling with a single breakdown, Acta Informatica 26 (1989) 679–696.

[2] S. Albers, G. Schmidt, Scheduling with unexpected machine breakdowns, Technical Report MPI-I-98-1-021, Max Planck Institut für Informatik, Saarbrücken, 1998.

[3] J. Blazewicz, J. Breit, P. Formanowicz, W. Kubiak, G. Schmidt, Heuristics for two machine flow shops with limited machine availability, Discussion Paper B-9802, Fachbereich Wirtschaftswissenschaft, University of Saarland, 1998.

[4] J. Blazewicz, M. Drozdowski, P. Formanowicz, W. Kubiak, G. Schmidt, Scheduling preemtable tasks on parallel processors with limited availability, unpublished.

[5] J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt, J. Weglarz, Scheduling Computer and Manufacturing Processes, Springer, Berlin, 1996.

[6] J. Blazewicz, P. Formanowicz, W. Kubiak, G. Schmidt, A note on a parallel branch and bound algorithm for the flow shop problem with limited machine availability, Working Paper, Poznan Supercomputing and Networking Center, Poznan, 1997.

[7] B. Chen, A. Van Vliet, G.J. Woeginger, A lower bound for randomized on-line scheduling algorithms, Information Processing Letters 51 (1994) 219–222.

[8] D. Dolev, M.K. Warmuth, Scheduling flat graphs, SIAM Journal on Computing 14 (1985) 638–657.

[9] D. Dolev, M.K. Warmuth, Profile scheduling of opposing forests and level orders, SIAM Journal on Algebraic and Discrete Methods 6 (1985) 665–687.

[10] M.R. Garey, D.S. Johnson, Computers and Intractability, Freeman, San Francisco, 1979.

[11] R.L. Graham, Bounds on multiprocessing timing anomalies, SIAM Journal on Applied Mathematics 17 (1969) 263–269.

[12] W.A. Horn, Some simple scheduling algorithms, Naval Research Logistics Quarterly 21 (1974) 177–185.

[13] S.M. Johnson, Optimal two- and three-stage production schedules with setup times included, Naval Research Logistics Quarterly 1 (1954) 61–68.

[14] B. Kalyanasundaram, K.P. Pruhs, Fault-tolerant scheduling, in: Proceedings of the 26th Annual ACM Symposium on the Theory of Computing, 1994, pp. 115–124.

[15] B. Kalyanasundaram, K.P. Pruhs, Fault-tolerant real-time scheduling, in: Proceedings of the Fifth Annual European Symposium on Algorithms (ESA), Springer Lecture Notes in Computer Science, 1997.

[16] R.M. Karp, Reducibility among combinatorial problems, in: R.E. Miller, J.W. Thatcher (Eds.), Complexity of Computer Communications, Plenum Press, New York, 1972, pp. 85–103.

[17] M. Kaspi, B. Montreuil, On the scheduling of identical parallel processes with arbitrary initial processor available time, Research Report 88-12, School of Industrial Engineering, Purdue University, West Lafayette, IN, 1988.

[18] H. Kellerer, Algorithms for multiprocessor scheduling with machine release time, IIE Transactions (to appear).

[19] W. Kubiak, J. Blazewicz, P. Formanowicz, G. Schmidt, A branch and bound algorithm for two machine flow shops with limited machine availability, Research Report RA-001/97, Poznan University of Technology, Institute of Computing Science, Poznan, 1997.

[20] J. Labetoulle, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, Preemptive scheduling of uniform machines subject to due dates, Technical Paper BW 99/79, CWI, Amsterdam, 1979.

[21] E.L. Lawler, Preemptive scheduling of precedence constrained jobs on parallel machines, in: Dempster et al. (Eds.), Deterministic and Stochastic Scheduling, Reidel, Boston, 1982, pp. 101–123.

[22] E.L. Lawler, C.U. Martel, Preemptive scheduling of two uniform machines to minimize the number of late jobs, Operations Research 37 (1989) 314–318.

[23] C.-Y. Lee, Parallel machine scheduling with non-simultaneous machine available time, Discrete Applied Mathematics 30 (1991) 53–61.

[24] C.-Y. Lee, Machine scheduling with an availability constraint, Journal of Global Optimization, Special Issue on Optimization of Scheduling Applications, 9 (1996) 363–384.

[25] C.-Y. Lee, Minimizing the makespan in the two-machine flowshop scheduling problem with an availability constraint, Operations Research Letters 20 (1997) 129–139.

[26] C.-Y. Lee, S.D. Liman, Single machine flow-time scheduling with scheduled maintenance, Acta Informatica 29 (1992) 375–382.

[27] C.-Y. Lee, S.D. Liman, Capacitated two-parallel machine scheduling to minimize sum of job completion times, Discrete Applied Mathematics 41 (1993) 211–222.

[28] C.Y. Lee, L. Lei, M. Pinedo, Current trends in deterministic scheduling, Annals of Operations Research 70 (1997) 1–42.

[29] J.K. Lenstra, A.H.G. Rinnooy Kan, P. Brucker, Complexity of processor scheduling problems, Annals Discrete Mathematics 1 (1977) 343–362.

[30] S. Liman, Scheduling with capacities and due-dates, Ph.D. Thesis, University of Florida, Gainesville, FL, 1991.

[31] G. Lin, Y. He, Y. Yao, H. Lu, Exact bounds of the modified LPT algorithm applying to parallel machines scheduling with nonsimultaneous machine available times, Applied Mathematics Journal of the Chinese University B 12 (1) (1997) 109–116.

[32] Z. Liu, E. Sanlaville, Preemptive scheduling with variable profile, precedence constraints and due dates, Discrete Applied Mathematics 58 (1995) 253–280.

[33] Z. Liu, E. Sanlaville, Profile scheduling of list algorithms, in: P. Chretienne et al. (Eds.), Scheduling Theory and its Applications, Wiley, New York, 1995, pp. 91–110.

[34] Z. Liu, E. Sanlaville, Stochastic scheduling with variable profile and precedence constraints, SIAM Journal on Computing 26 (1997) 173–187.

[35] R. Muntz, E.G. Coffman, Preemptive scheduling of real-time tasks on multiprocessor systems, Journal of the Association for Computing Machinery 17 (1970) 324–338.
[36] R. McNaughton, Scheduling with deadlines and loss functions, Management Science 6 (1959) 1–12.
[37] J.M. Moore, An *n* job one machine sequencing algorithm for minimizing the number of late jobs, Management Science 15 (1968) 102–109.
[38] T.E. Morton, D.W. Pentico, Heuristic Scheduling Systems, Wiley, New York, 1993.
[39] G. Mosheiov, Minimizing the sum of job completion times on capacitated parallel machines, Mathematical Computing and Modelling 20 (1994) 91–99.
[40] E. Sanlaville, Nearly on line scheduling of preemptive independent tasks, Discrete Applied Mathematics 57, 229–241.
[41] G. Schmidt, Scheduling on semi-identical processors, Zeitschrift fur Operations Research A 28 (1984) 153–162.
[42] G. Schmidt, Scheduling independent tasks with deadlines on semi-identical processors, Journal of the Operational Research Society 39 (1988) 271–277.
[43] D.D. Sleator, R.E. Tarjan, Amortized efficiency of list update and paging rules, Communications of the Association for Computing Machinery 28 (1985) 202–208.
[44] W.E. Smith, Various optimizers for single-stage production, Naval Research Logistics Quarterly 3 (1956) 59–66.
[45] J.D. Ullman, NP-complete scheduling problems, Journal of Computer and System Sciences 10 (1975) 384–393.