5 Scheduling on Parallel Processors

This chapter is devoted to the analysis of scheduling problems in a parallel processor environment. As before the three main criteria to be analyzed are schedule length, mean flow time and lateness. Then, some more developed models of multiprocessor systems and lot size scheduling are described. Corresponding results are presented in the four following sections.

5.1 Minimizing Schedule Length

In this section we will analyze the schedule length criterion. Complexity analysis will be complemented, wherever applicable, by a description of the most important approximation as well as enumerative algorithms. The presentation of the results will be divided into subcases depending on the type of processors used, the type of precedence constraints, and to a lesser extent task processing times and the possibility of task preemption.

5.1.1 Identical Processors

Problem $P||C_{max}$

The first problem considered is $P||C_{max}$ where a set of independent tasks is to be scheduled on identical processors in order to minimize schedule length. We start with complexity analysis of this problem which leads to the conclusion that the problem is not easy to solve, since even simple cases such as scheduling on two processors can be proved to be NP-hard [Kar72].

Theorem 5.1.1 *Problem $P2||C_{max}$ is NP-hard.*

Proof. As a known NP-complete problem we take PARTITION [Kar72] which is formulated as follows.

Instance: Finite set $\mathcal{A}$ and a size $s(a_i) \in \mathbb{N}$ for each $a_i \in \mathcal{A}$.

Answer: "Yes" if there exists a subset $\mathcal{A}' \subseteq \mathcal{A}$ such that

$$\sum_{a_i \in \mathcal{A}'} s(a_i) = \sum_{a_i \in \mathcal{A} - \mathcal{A}'} s(a_i).$$

Otherwise "No".

J. Blazewicz et al., *Handbook on Scheduling*, International Handbooks on Information Systems, https://doi.org/10.1007/978-3-319-99849-7_5

Given any instance of PARTITION defined by the positive integers $s(a_i)$, $a_i \in \mathcal{A}$, we define a corresponding instance of the decision counterpart of $P2||C_{max}$ by assuming $n = |\mathcal{A}|$, $p_j = s(a_j)$, $j = 1, 2, \cdots, n$, and a threshold value for the schedule length, $y = \frac{1}{2}\sum_{a_i \in \mathcal{A}} s(a_i)$. It is obvious that there exists a subset $\mathcal{A}'$ with the desired property for the instance of PARTITION if and only if, for the corresponding instance of $P2||C_{max}$, there exists a schedule with $C_{max} \leq y$ (cf. Figure 5.1.1). This proves the theorem. □

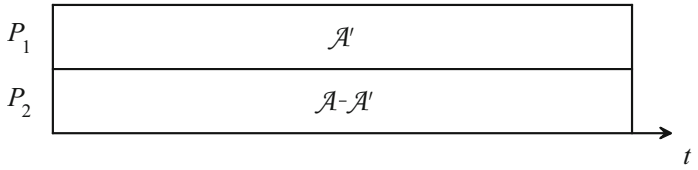


Figure 5.1.1 *A schedule for Theorem* 5.1.1.

Since there is no hope of finding an optimization polynomial time algorithm for $P||C_{max}$, one may try to solve the problem along the lines presented in Section 3.2. First, one may try to find an approximation algorithm for the original problem and evaluate its worst case as well as its mean behavior. We will present such an analysis below.

One of the most often used general approximation strategies for solving scheduling problems is *list scheduling*, whereby a priority list of the tasks is given, and at each step the first available processor is selected to process the first available task on the list [Gra66] (cf. Section 3.2). The accuracy of a given list scheduling algorithm depends on the order in which tasks appear on the list. One of the simplest algorithms is the *LPT algorithm* in which the tasks are arranged in order of non-increasing p_j.

Algorithm 5.1.2 *LPT Algorithm for* $P||C_{max}$.

begin
Order tasks on a list in non-increasing order of their processing times;
  -- i.e. $p_1 \geq \cdots \geq p_n$
for $i = 1$ **to** m **do** $s_i := 0$;
  -- processors P_i are assumed to be idle from time $s_i = 0$ on, $i = 1, \cdots, m$
$j := 1$;
repeat
  $s_k := \min\{s_i\}$;
  Assign task T_j to processor P_k at time s_k;
    -- the first non-assigned task from the list is scheduled on the first processor
    -- that becomes free

$s_k := s_k + p_j$; $j := j+1$;
until $j = n+1$; -- all tasks have been scheduled
end;

It is easy to see that the time complexity of this algorithm is $O(n\log n)$ since its most complex activity is to sort the set of tasks. The worst case behavior of the *LPT* rule is analyzed in Theorem 5.1.3.

Theorem 5.1.3 [Gra69] *If the LPT algorithm is used to solve problem* $P||C_{max}$, *then*

$$R_{LPT} = \frac{4}{3} - \frac{1}{3m}. \tag{5.1.1}$$

□

Space limitations prevent us from including here the proof of the upper bound in the above theorem. However, we will give an example showing that this bound can be achieved. Let $n = 2m+1$, $\boldsymbol{p} = [2m-1, 2m-1, 2m-2, 2m-2, \cdots, m+1, m+1, m, m, m]$. For $m = 3$, Figure 5.1.2 shows two schedules, an optimal one and an *LPT* schedule.

We see that in the worst case an *LPT* schedule can be up to 33% longer than an optimal schedule. However, one is led to expect better performance from the *LPT* algorithm than is indicated by (5.1.1), especially when the number of tasks becomes large. In [CS76] another absolute performance ratio for the *LPT* rule was proved, taking into account the number k of tasks assigned to a processor whose last task terminates the schedule.

Theorem 5.1.4 *For the assumptions stated above, we have*

$$R_{LPT}(k) = 1 + \frac{1}{k} - \frac{1}{km}. \tag{5.1.2}$$

□

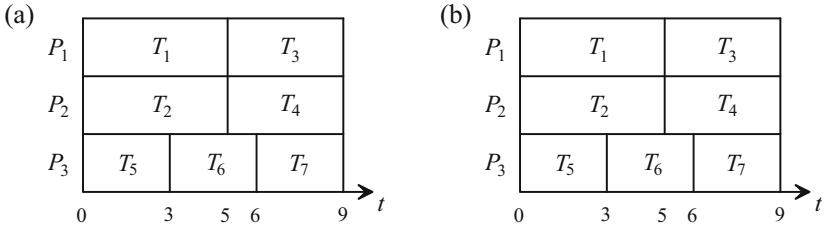


Figure 5.1.2 *Schedules for Theorem* 5.1.3
(a) *an optimal schedule,*
(b) *LPT schedule.*

This result shows that the worst-case performance bound for the *LPT* algorithm approaches one as fast as $1+1/k$.

On the other hand, it would be of interest to know how good the *LPT* algorithm is on the average. Such a result was obtained by [CFL84], where the relative error was found for two processors on the assumption that task processing times are independent samples from the uniform distribution on $[0, 1]$.

Theorem 5.1.5 *Under the assumptions already stated, we have the following bounds for the mean value of schedule length for the LPT algorithm,* $E(C_{max}^{LPT})$, *for problem* $P2 \,||\, C_{max}$.

$$\frac{n}{4} + \frac{1}{4(n+1)} \leq E(C_{max}^{LPT}) \leq \frac{n}{4} + \frac{e}{2(n+1)}, \tag{5.1.3}$$

where $e = 2.7\cdots$ *is the base of the natural logarithm.* □

Taking into account that $n/4$ is a lower bound on $E(C_{max}^{*})$ we get

$$E(C_{max}^{LPT})/E(C_{max}^{*}) < 1 + O(1/n^2).$$

Therefore, as n increases, $E(C_{max}^{LPT})$ approaches the optimum no more slowly than $1 + O(1/n^2)$ approaches 1. The above bound can be generalized to cover also the case of m processors for which we have [CFL83]:

$$E(C_{max}^{LPT}) \leq \frac{n}{2m} + \left(\frac{m}{n}\right).$$

Moreover, it is also possible to prove [FRK86, FRK87] that $C_{max}^{LPT} - C_{max}^{*}$ almost surely converges to 0 as $n \to \infty$ if the task processing time distribution has a finite mean and a density function f satisfying $f(0) > 0$. It is also shown that if the distribution is uniform or exponential, the rate of convergence is $O(\log(\log n)/n)$. This result, obtained by a complicated analysis, can also be guessed from simulation studies. Such an experiment was reported by Kedia [Ked70] and we present the summary of the results in Table 5.1.1. The last column presents the ratio of schedule lengths obtained by the *LPT* algorithm and the optimal preemptive one. Task processing times are drawn from the uniform distribution of the given parameters.

To conclude the above analysis we may say that the *LPT* algorithm behaves quite well and may be useful in practice. However, if one wants to have better performance guarantees, other approximation algorithms should be used, as for example *MULTIFIT* introduced by Coffman et al. [CGJ78] or the algorithm proposed by Hochbaum and Shmoys [HS87]. A comprehensive treatment of approximation algorithms for this and related problems is given by Coffman et al. [CGJ84].

n, m		Intervals of task processing time distribution	C_{max}	$C_{max}^{LPT} / C_{max}^{*}$
6	3	1, 20	20	1.00
9	3	1, 20	32	1.00
15	3	1, 20	65	1.00
6	3	20, 50	59	1.05
9	3	20, 50	101	1.03
15	3	20, 50	166	1.00
8	4	1, 20	23	1.09
12	4	1, 20	30	1.00
20	4	1, 20	60	1.00
8	4	20, 50	74	1.04
12	4	20, 50	108	1.02
20	4	20, 50	185	1.01
10	5	1, 20	25	1.04
15	5	1, 20	38	1.03
20	5	1, 20	49	1.00
10	5	20, 50	65	1.06
15	5	20, 50	117	1.03
25	5	20, 50	198	1.01

Table 5.1.1 *Mean performance of the LPT algorithm.*

We now pass to the second way of analyzing problem $P||C_{max}$. Theorem 5.1.1 gave a negative answer to the question about the existence of an optimization polynomial time algorithm for solving $P2||C_{max}$. However, we have not proved that our problem is NP-hard in the strong sense and we may try to find a pseudo-polynomial optimization algorithm. It appears that, based on a dynamic programming approach, such an algorithm can be constructed using ideas presented by Rothkopf [Rot66]. Below the algorithm is presented for $P||C_{max}$; it uses Boolean variables $x_j(t_1, t_2, \cdots, t_m)$, $j = 1, 2, \cdots, n$, $t_i = 0, 1, \cdots, C$, $i = 1, 2, \cdots, m$, where C denotes an upper bound on the optimal schedule length C_{max}^{*}. The meaning of these variables is the following

$$x_j(t_1, t_2, \cdots, t_m) = \begin{cases} \texttt{true} \text{ if tasks } T_1, T_2, \cdots, T_j \text{ can be scheduled on} \\ \quad \text{processors } P_1, P_2, \cdots, P_m \text{ in such a way that } P_i \\ \quad \text{is busy in time interval } [0, t_i],\ i = 1, 2, \cdots, m, \\ \texttt{false} \text{ otherwise.} \end{cases}$$

Now, we are able to present the algorithm.

Algorithm 5.1.6 *Dynamic programming for* $P||C_{max}$ [Rot66].

begin

for all $(t_1, t_2, \cdots, t_m) \in \{0, 1, \cdots, C\}^m$ **do** $x_0(t_1, t_2, \cdots, t_m) :=$ **false;**

$x_0(0, 0, \cdots, 0) :=$ **true;**

-- initial values for Boolean variables are now assigned

for $j = 1$ **to** n **do**

for all $(t_1, t_2, \cdots, t_m) \in \{0, 1, \cdots, C\}^m$ **do**

$$x_j(t_1, t_2, \cdots, t_m) = \bigvee_{i=1}^{m} x_{j-1}(t_1, t_2, \cdots, t_{i-1}, t_i - p_j, t_{i+1}, \cdots, t_m); \tag{5.1.4}$$

$$C^*_{max} := \min\{\max\{t_1, t_2, \cdots, t_m\} \mid x_n(t_1, t_2, \cdots, t_m) = true\}; \tag{5.1.5}$$

-- optimal schedule length has been calculated

Starting from the value C^*_{max}, assign tasks $T_n, T_{n-1}, \cdots, T_1$ to appropriate processors using formula (5.1.4) backwards;

end;

The above procedure solves problem $P||C_{max}$ in $O(nC^m)$ time; thus for fixed m it is a pseudopolynomial time algorithm. As a consequence, for small values of m and C the algorithm can be used even in computer applications. To illustrate the use of the above algorithm let us consider the following example.

Example 5.1.7 Let $n = 3$, $m = 2$ and $\boldsymbol{p} = [2, 1, 2]$. Assuming bound $C = 5$ we get the cube given in Figure 5.1.3(a) where particular values of variables $x_j(t_1, t_2, \cdots, t_m)$ are stored. In Figures 5.1.3(b) through 5.1.3(e) these values are shown, respectively, for $j = 0, 1, 2, 3$ (only true values are depicted). Following Figure 5.1.3(e) and equation (5.1.5), an optimal schedule is constructed as shown in Figure 5.1.3(f). □

The interested reader may find a survey of some other enumerative approaches for the problem in question in [LLR+93].

Problem $P|pmtn|C_{max}$

Now one may try the third way of analyzing the problem $P||C_{max}$ (as suggested in Section 3.2), i.e. on may relax some constraints imposed on problem $P||C_{max}$ and allow preemptions of tasks. It appears that problem $P|pmtn|C_{max}$ can be solved very efficiently. It is easy to see that the length of a preemptive schedule cannot be smaller than the maximum of two values: the maximum processing time of a task and the mean processing requirement on a processor [McN59], i.e.:

$$C^*_{max} = \max\left\{\max_j\{p_j\}, \frac{1}{m}\sum_{j=1}^{n} p_j\right\}. \tag{5.1.6}$$

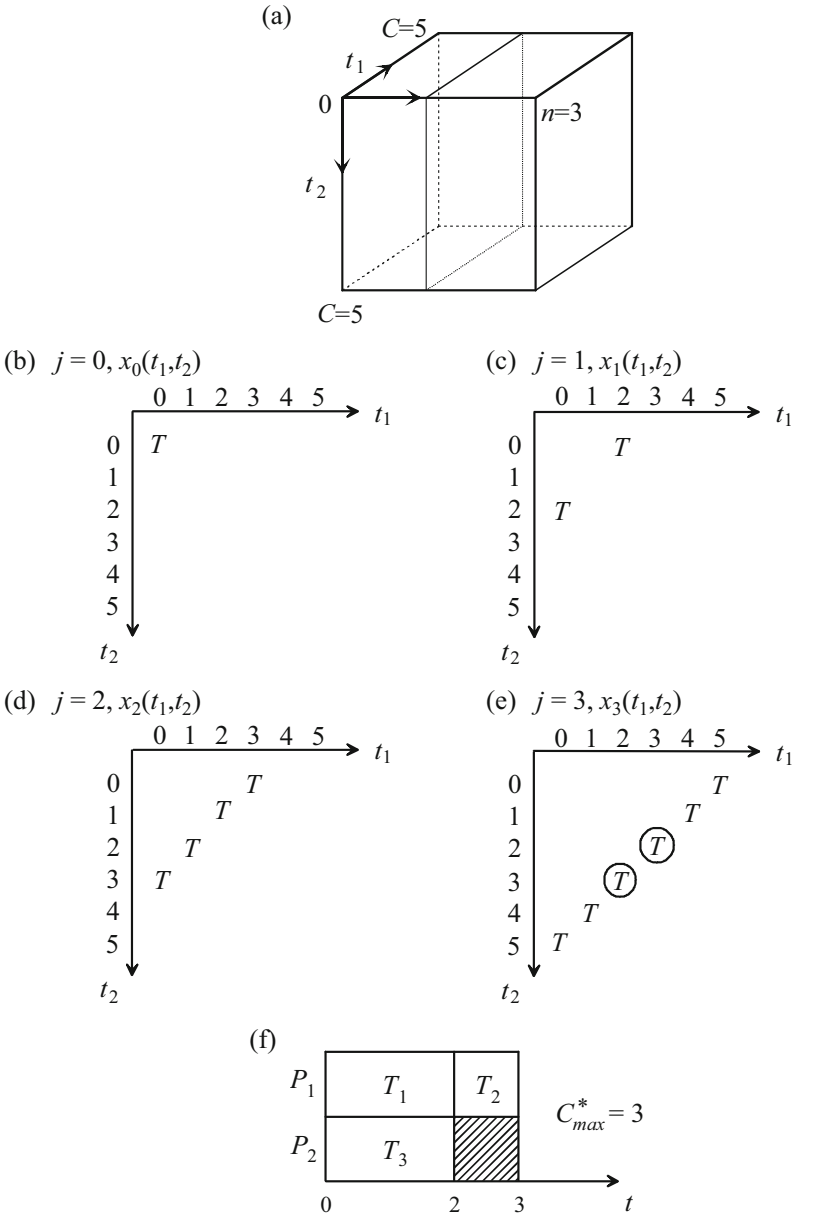


Figure 5.1.3 *An application of dynamic programming for Example* 5.1.7
(a) *a cube of Boolean variables,*
(b)-(e) *values of* $x_j(t_1,t_2)$ *for* $j = 0, 1, 2, 3$, *respectively (here T stands for true),*
(f) *an optimal schedule.*

The following algorithm given by McNaughton [McN59] constructs a schedule whose length is equal to C^*_{max}.

Algorithm 5.1.8 *McNaughton's rule for* $P\,|\,pmtn\,|\,C_{max}$ [McN59].

begin

$C^*_{max} := \max\{\sum_{j=1}^{n} p_j/m, \max_j\{p_j\}\}$; -- minimum schedule length

$t := 0$; $i := 1$; $j := 1$;

repeat

  if $t + p_j \le C^*_{max}$

  then

    begin

    Assign task T_j to processor P_i, starting at time t;

    $t := t + p_j$; $j := j + 1$;

      -- task T_j can be fully assigned to processor P_i,

      -- assignment of the next task will continue at time $t + p_j$

    end

  else

    begin

    Starting at time t, assign task T_j for $C^*_{max} - t$ units to processor P_i;

      -- task T_j is preempted at time C^*_{max},

      -- processor P_i is now busy until C^*_{max},

      -- assignment of T_j will continue on the next processor at time 0

    $p_j := p_j - (C^*_{max} - t)$; $t := 0$; $i := i + 1$;

    end;

until $j = n + 1$; -- all tasks have been scheduled

end;

Note that the above algorithm is an optimization procedure since it always finds a schedule whose length is equal to C^*_{max}. Its time complexity is $O(n)$.

We see that by allowing preemptions we made the problem easy to solve. However, there still remains the question of practical applicability of the solution obtained this way. One has to ask if this model of preemptive task scheduling can be justified, because it cannot be expected that preemptions are free of cost. Generally, two kinds of preemption costs have to be considered: time and finance. Time delays originating from preemptions are less crucial if the delay caused by a single preemption is small compared to the time the task continuously spends on the processor. Financial costs connected with preemptions, on the other hand, reduce the total benefit gained by preemptive task execution; but again, if the profit gained is large compared to the losses caused by the preemptions the schedule will be more useful and acceptable. These circumstances suggest the introduction of a scheduling model where task preemptions are only allowed af-

ter the tasks have been processed continuously for some given amount k of time. The value for k (*preemption granularity*) should be chosen large enough so that the time delay and cost overheads connected with preemptions are negligible. For given granularity k, upper bounds on the preemption overhead can easily be estimated since the number of preemptions for a task of processing time p is limited by $\lfloor p/k \rfloor$. In [EH93] the problem $P\,|\,pmtn\,|\,C_{max}$ with k-restricted preemptions is discussed: If the processing time p_j of a task T_j is less than or equal to k, then preemption is not allowed; otherwise preemption may take place after the task has been continuously processed for at least k units of time. For the remaining part of a preempted task the same condition is applied. Notice that for $k = 0$ this problem reduces to the "classical" preemptive scheduling problem. On the other hand, if for a given instance the granularity k is larger than the longest processing time among the given tasks, then no preemption is allowed and we end up with non-preemptive scheduling. Another variant is the *exact-k-preemptive* scheduling problem where task preemptions are only allowed at those moments when the task has been processed exactly an integer multiple of k time units. In [EH93] it is proved that, for $m = 2$ processors, both the k-preemptive and the exact-k-preemptive scheduling problems can be solved in time $O(n)$. For $m > 2$ processors both problems are NP-hard.

Problem $P\,|\,prec\,|\,C_{max}$

Let us now pass to the case of dependent tasks. At first tasks are assumed to be scheduled non-preemptively. It is obvious that there is no hope of finding a polynomial time optimization algorithm for scheduling tasks of arbitrary length since $P\,||\,C_{max}$ is already NP-hard. However, one may try again list scheduling algorithms. Unfortunately, this strategy may result in an unexpected behavior of constructed schedules, since the schedule length for problem $P\,|\,prec\,|\,C_{max}$ (with arbitrary precedence constraints) may increase if:

- the number of processors increases,
- task processing times decrease,
- precedence constraints are weakened, or
- the priority list changes.

Figures 5.1.4 through 5.1.8 indicate the effects of changes of the above mentioned parameters. These *list scheduling anomalies* have been discovered by Graham [Gra66], who has also evaluated the maximum change in schedule length that may be induced by varying one or more problem parameters. We will quote this theorem since its proof is one of the shortest in that area and illustrates well the technique used in other proofs of that type. Let there be defined a task set $\mathcal{T}$ together with precedence constraints $\prec$. Let the processing times of the tasks be given by vector $\boldsymbol{p}$, let $\mathcal{T}$ be scheduled on m processors using list L, and

let the obtained value of schedule length be equal to C_{max}. On the other hand, let the above parameters be changed: a vector of processing times $\boldsymbol{p}' \leq \boldsymbol{p}$ (for all the components), relaxed precedence constraints $\prec' \subseteq \prec$, priority list L' and the number of processors m'. Let the new value of schedule length be C'_{max}. Then the following theorem is valid.

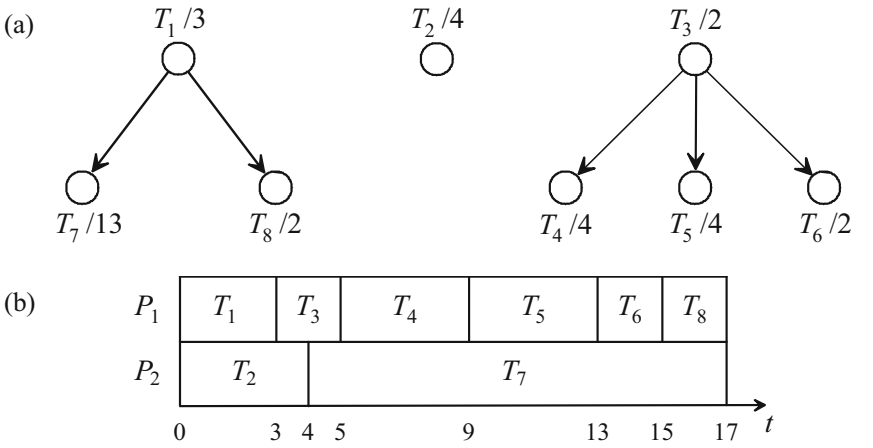


Figure 5.1.4 **(a)** *A task set, $m = 2$, $L = (T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8)$,* **(b)** *an optimal schedule.*

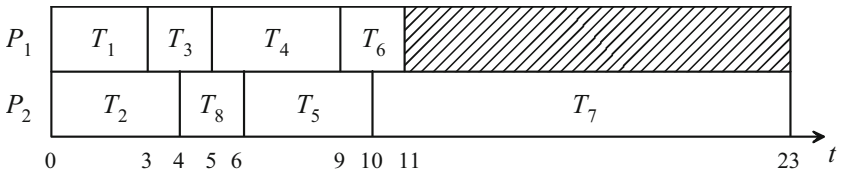


Figure 5.1.5 *Priority list changed: A new list $L' = (T_1, T_2, T_3, T_4, T_5, T_6, T_8, T_7)$.*

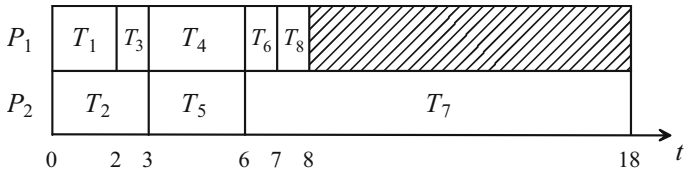


Figure 5.1.6 *Processing times decreased; $p'_j = p_j - 1, j = 1, 2, \cdots, n$.*

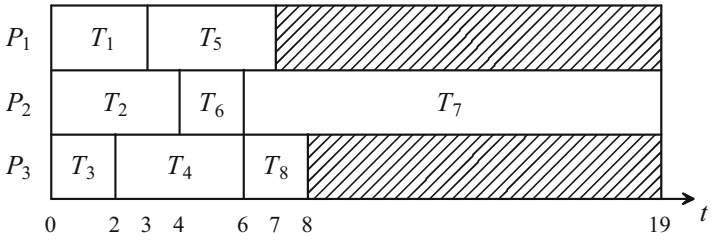


Figure 5.1.7 *Number of processors increased, $m = 3$.*

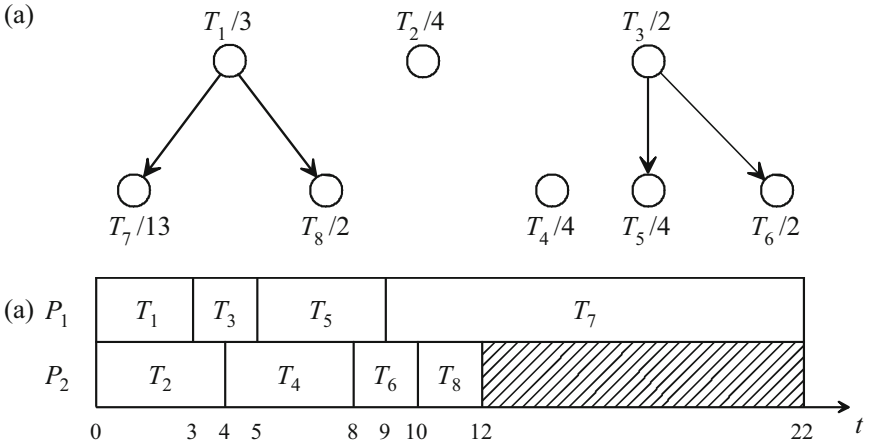


Figure 5.1.8 **(a)** *Precedence constraints weakened,*
(b) *a resulting list schedule.*

Theorem 5.1.9 [Gra66] *Under the above assumptions*,

$$\frac{C'_{max}}{C_{max}} \leq 1 + \frac{m-1}{m'}. \tag{5.1.7}$$

Proof. Let us consider schedule S' obtained by processing task set $\mathcal{T}$ with primed parameters. Let the interval $[0, C'_{max})$ be divided into two subsets, $\mathcal{A}$ and $\mathcal{B}$, defined in the following way:

$$\mathcal{A} = \{t \in [0, C'_{max}) \mid \text{all processors are busy at time } t\},\ \mathcal{B} = [0, C'_{max}) - \mathcal{A}.$$

Notice that both $\mathcal{A}$ and $\mathcal{B}$ are unions of disjoint half-open intervals. Let T_{j_1} denote a task completed in S' at time C'_{max}, i.e. $C_{j_1} = C'_{max}$. Two cases may occur:

1. The starting time s_{j_1} of T_{j_1} is an interior point of $\mathcal{B}$. Then by the definition of $\mathcal{B}$ there is some processor P_i which for some $\varepsilon > 0$ is idle during interval $[s_{j_1} - \varepsilon, s_{j_1})$. Such a situation may only occur if we have $T_{j_2} \prec' T_{j_1}$ and $C_{j_2} = s_{j_1}$ for some task T_{j_2}.
2. The starting time of T_{j_1} is not an interior point of $\mathcal{B}$. Let us also suppose that $s_{j_1} \neq 0$. Define $x_1 = \sup\{x \mid x < s_{j_1}, \text{ and } x \in \mathcal{B}\}$ or $x_1 = 0$ if set $\mathcal{B}$ is empty. By the construction of $\mathcal{A}$ and $\mathcal{B}$, we see that $x_1 \in \mathcal{A}$, and processor P_i is idle in time interval $[x_1 - \varepsilon, x_1)$ for some $\varepsilon > 0$. But again, such a situation may only occur if some task $T_{j_2} \prec' T_{j_1}$ is processed during this time interval.

It follows that either there exists a task $T_{j_2} \prec' T_{j_1}$ such that $y \in [C_{j_2}, s_{j_1})$ implies $y \in \mathcal{A}$ or we have: $x < s_{j_1}$ implies either $x \in \mathcal{A}$ or $x < 0$.

The above procedure can be inductively repeated, forming a chain T_{j_3}, $T_{j_4}, \cdots$, until we reach task T_{j_r} for which $x < s_{j_r}$ implies either $x \in \mathcal{A}$ or $x < 0$. Hence there must exist a chain of tasks

$$T_{j_r} \prec' T_{j_{r-1}} \prec' \cdots \prec' T_{j_2} \prec' T_{j_1} \tag{5.1.8}$$

such that at each moment $t \in \mathcal{B}$, some task T_{j_k} is being processed in S'. This implies that

$$\sum_{\phi' \in S'} p'_{\phi'} \leq (m'-1) \sum_{k=1}^{r} p'_{j_k} \tag{5.1.9}$$

where the sum on the left-hand side is made over all idle-time tasks ϕ' in S'. But by (5.1.8) and the hypothesis $\prec' \subseteq \prec$ we have

$$T_{j_r} \prec T_{j_{r-1}} \prec \cdots \prec T_{j_2} \prec T_{j_1}. \tag{5.1.10}$$

Hence,

$$C_{max} \geq \sum_{k=1}^{r} p_{j_k} \geq \sum_{k=1}^{r} p'_{j_k}. \tag{5.1.11}$$

Furthermore, by (5.1.9) and (5.1.11) we have

$$C'_{max} = \frac{1}{m'}\Big(\sum_{k=1}^{n} p'_k + \sum_{\phi' \in S'} p'_{\phi'}\Big) \leq \frac{1}{m'}(m\, C_{max} + (m'-1)\, C_{max}). \tag{5.1.12}$$

It follows that

$$\frac{C'_{max}}{C_{max}} \leq 1 + \frac{m-1}{m'}$$

and the theorem is proved. □

From the above theorem, the *absolute performance ratio* for an arbitrary list scheduling algorithm solving problem $P||C_{max}$ can be derived.

Corollary 5.1.10 [Gra66] *For an arbitrary list scheduling algorithm LS for* $P||C_{max}$ *we have*

$$R_{LS} = 2 - \frac{1}{m}. \tag{5.1.13}$$

Proof. The upper bound of (5.1.13) follows immediately from (5.1.7) by taking $m' = m$ and by considering the list leading to an optimal schedule. To show that this bound is achievable let us consider the following example: $n = (m-1)m+1$, $\boldsymbol{p} = [1, 1, \cdots, 1, 1, m]$, $\prec$ is empty, $L = (T_n, T_1, T_2, \cdots, T_{n-1})$ and $L' = (T_1, T_2, \cdots, T_n)$. The corresponding schedules for $m = 4$ are shown in Figure 5.1.9. □

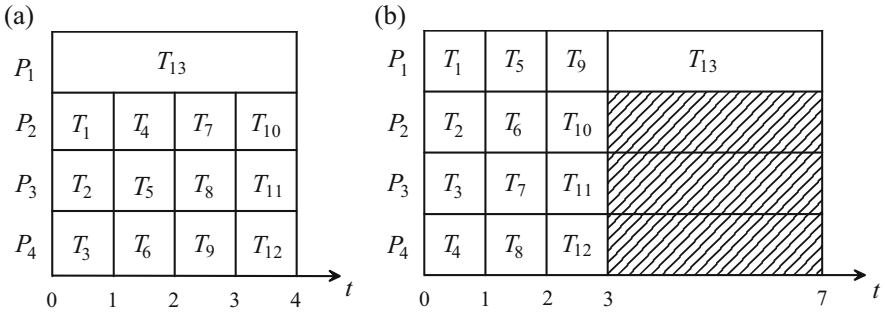


Figure 5.1.9 *Schedules for Corollary* 5.1.10
(a) *an optimal schedule,*
(b) *an approximate schedule.*

It follows from the above considerations that an arbitrary list scheduling algorithm can produce schedules almost twice as long as optimal ones. However, one can solve optimally problems with tasks of unit lengths.

Problem $P|prec, p_j = 1|C_{max}$

The first algorithm has been given for scheduling *forests*, consisting either of *in-trees* or of *out-trees* [Hu61]. We will first present Hu's algorithm for the case of an in-tree, i.e. for the problem $P|in\text{-}tree, p_j = 1|C_{max}$. The algorithm is based on the notion of a *task level* in an in-tree which is defined as the number of tasks on the path to the root of the graph. The algorithm by Hu, which is also called *level algorithm* or *critical path algorithm* is as follows.

Algorithm 5.1.11 *Hu's algorithm for* $P \mid in\text{-}tree, p_j = 1 \mid C_{max}$ [Hu61].

begin
Calculate levels of the tasks;
$t := 0$;
repeat
Construct list L_t consisting of all the tasks without predecessors at time t;
-- all these tasks either have no predecessors
-- or their predecessors have been assigned in time interval $[0, t-1]$
Order L_t in non-increasing order of task levels;
Assign m tasks (if any) to processors at time t from the beginning of list L_t;
Remove the assigned tasks from the graph and from the list;
$t := t+1$;
until all tasks have been scheduled;
end;

The algorithm can be implemented to run in $O(n)$ time. An example of its application is shown in Figure 5.1.10.

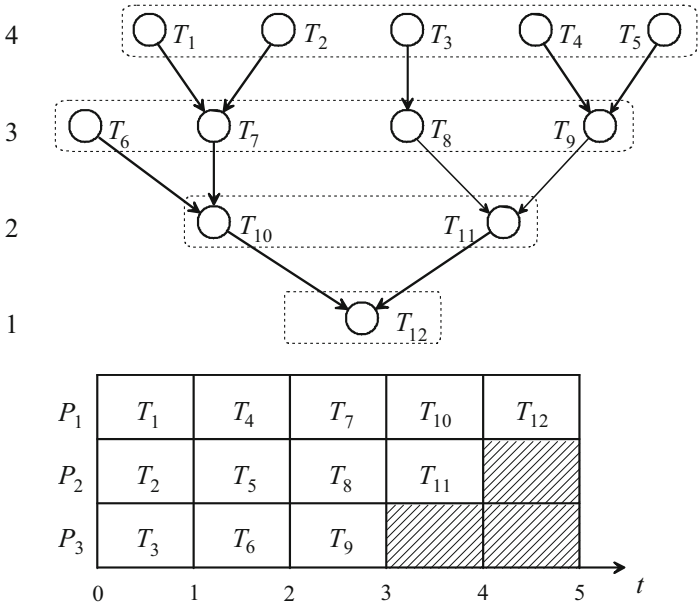


Figure 5.1.10 *An example of the application of Algorithm* 5.1.11 *for three processors.*

A *forest* consisting of in-trees can be scheduled by adding a dummy task that is an immediate successor of only the roots of in-trees, and then by applying Algorithm 5.1.11. A schedule for an *out-tree* can be constructed by changing the ori-

entation of arcs, applying Algorithm 5.1.11 to the obtained in-tree and then reading the schedule backwards, i.e. from right to left.

It is interesting to note that the problem of scheduling *opposing forests* (that is, combinations of in-trees and out-trees) on an arbitrary number of processors is NP-hard [GJTY83]. However, if the number of processors is limited to 2, the problem is easily solvable even for arbitrary precedence graphs [CG72, FKN69, Gab82]. We present the algorithm given by Coffman and Graham [CG72] since it can be further extended to cover the preemptive case. The algorithm uses *labels* assigned to tasks, which take into account the levels of the tasks and the numbers of their immediate successors. The following algorithm assigns labels to the tasks, and then uses them to find the shortest schedule for problem $P2\,|\,prec, p_j = 1\,|\,C_{max}$.

Algorithm 5.1.12 *Algorithm by Coffman and Graham for* $P2\,|\,prec, p_j = 1\,|\,C_{max}$ [CG72].

begin
Assign label 1 to any task T_0 for which isucc(T_0) = $\emptyset$;
  -- recall that isucc(T) denotes the set of all immediate successors of T
$j := 1$;
repeat
  Construct set $\mathcal{S}$ of all unlabeled tasks whose successors are labeled;
  for all $T \in \mathcal{S}$ **do**
    begin
    Construct list $L(T)$ consisting of labels of tasks belonging to isucc(T);
    Order $L(T)$ in decreasing order of the labels;
    end;
  Order these lists in increasing lexicographic order $L(T_{[1]}) \prec \cdots \prec L(T_{[|\mathcal{S}|]})$;
  -- see Section 2.1 for definition of $\prec$
  Assign label $j+1$ to task $T_{[1]}$;
  $j := j+1$;
until $j = n+1$; -- all tasks have been assigned labels
call Algorithm 5.1.11;
  -- here the above algorithm uses labels instead of levels when scheduling tasks
end;

A careful analysis shows that the above algorithm can be implemented to run in time which is almost linear in n and in the number of arcs in the precedence graph [Set76]; thus its time complexity is practically $O(n^2)$. An example of the application of Algorithm 5.1.12 is given in Figure 5.1.11.

It must be stressed that the question concerning the complexity of problem $Pm\,|\,prec, p_j = 1\,|\,C_{max}$ with a fixed number m of processors is still open despite the fact that many papers have been devoted to solving various subcases of prec-

edence constraints. If tasks with unit processing times are considered, the following results are available. Problems $P3\,|\,opposing\ forest,\ p_j = 1\,|\,C_{max}$ and $Pk\,|\,opposing\ forest,\ p_j = 1\,|\,C_{max}$ are solvable in time $O(n)$ [GJTY83] and $O(n^{2k-2}\log n)$ [DW85], respectively. On the other hand, if the number of available processors is variable, then this problem becomes NP-hard. Some results are also available for the subcases in which task processing times may take only two values. Problems $P2\,|\,prec, p_j = 1\ or\ 2\,|\,C_{max}$ and $P\,|\,prec, p_j = 1\ or\ k\,|\,C_{max}$ are NP-hard [DL88], while problems $P2\,|\,tree,\ p_j = 1\ or\ 2\,|\,C_{max}$ and $P2\,|\,tree,\ p_j = 1\ or\ 3\,|\,C_{max}$ are solvable in time $O(n\log n)$ [NLH81] and $O(n^2\log n)$ [DL89], respectively. Arbitrary processing times result in strong NP-hardness even for the case of chains scheduled on two processors (problem $P2\,|\,chains\,|\,C_{max}$) [DLY91].

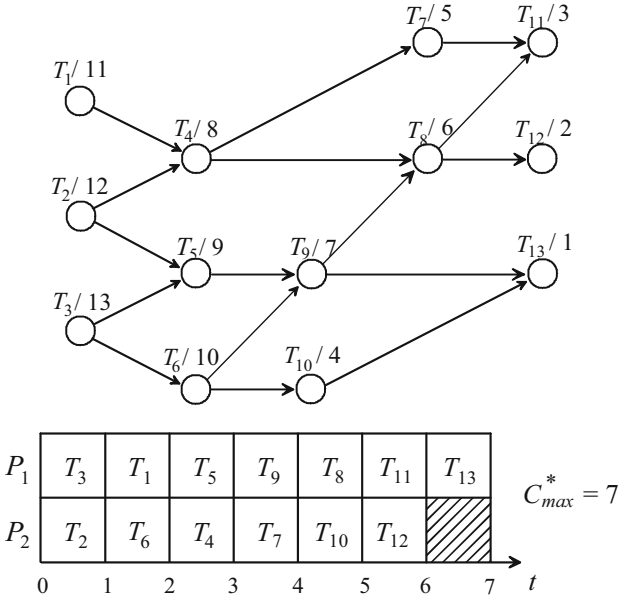


Figure 5.1.11 *An example of the application of Algorithm* 5.1.12 (*tasks are denoted by* T_j/*label*).

Furthermore, several papers deal with approximation algorithms for $P\,|\,prec, p_j = 1\,|\,C_{max}$ and more general problems. We quote some of the most interesting results. The application of the level algorithm (Algorithm 5.1.11) to solve $P\,|\,prec, p_j = 1\,|\,C_{max}$ has been analyzed by Chen and Liu [CL75] and Kunde [Kun76]. The following bound has been proved.

$$R_{\text{level}} = \begin{cases} \frac{4}{3} & \text{for } m = 2 \\ 2 - \frac{1}{m-1} & \text{for } m \geq 3\,. \end{cases}$$

Algorithm 5.1.12 is slightly better, its bound is $R = 2 - \frac{2}{m} - \frac{m-3}{m \cdot C^*_{max}}$ for $m \geq 3$ [BT94]. In this context one should not forget the results presented in Theorems 5.1.9 and 5.1.10, where list scheduling anomalies have been analyzed.

Problem P | pmtn, prec | C_{max}

The analysis also showed that preemptions can be profitable from the viewpoint of two factors. First, they can make problems easier to solve, and second, they can shorten the schedule. Coffman and Garey [CG91] proved that for problem $P2\,|\,prec\,|\,C_{max}$ the least schedule length achievable by a non-preemptive schedule is no more than 4/3 the least schedule length achievable when preemptions are allowed. While the proof of this fact seems to be tedious, a very simple example showing that this bound is met can easily be given for a set of three independent tasks of equal length (cf. Figure 5.1.12).

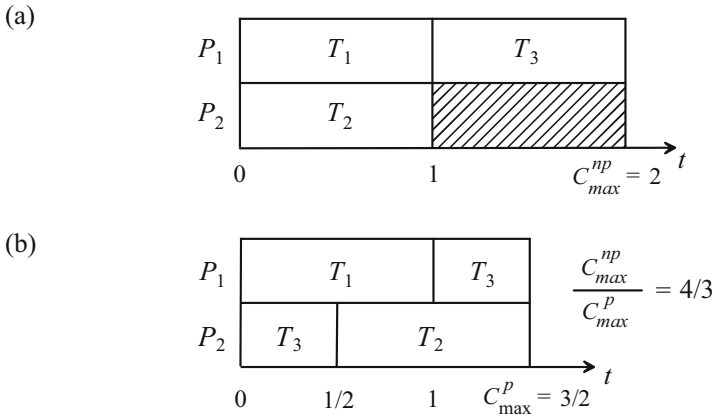


Figure 5.1.12 *An example of* 4/3 *conjecture*
(a) *non-preemptive scheduling,*
(b) *preemptive scheduling.*

In the general case of dependent tasks scheduled on processors in order to minimize schedule length, one can construct optimal preemptive schedules for tasks of arbitrary length and with other parameters the same as in Algorithm 5.1.11 or 5.1.12. The approach again uses the notion of the *level* of task T_j in a precedence graph, by which is now understood the sum of processing times (including p_j) of

tasks along the longest path between T_j and a terminal task (a task with no successors). Let us note that the level of a task being executed is decreasing. We have the following algorithm [MC69, MC70] for the problems $P2\,|\,pmtn, prec\,|\,C_{max}$ and $P\,|\,pmtn, forest\,|\,C_{max}$. The algorithm uses a notion of a *processor shared schedule*, in which a task receives some fraction β (≤ 1) of the processing capacity of a processor.

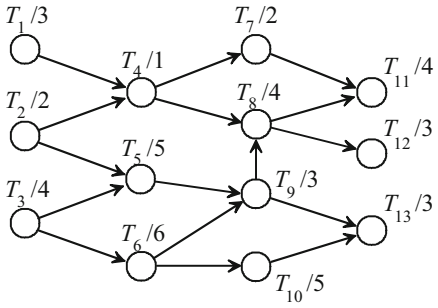
Algorithm 5.1.13 *Algorithm by Muntz and Coffman for* $P2\,|\,pmtn, prec\,|\,C_{max}$ *and* $P\,|\,pmtn, forest\,|\,C_{max}$ [MC69, MC70].

begin
for all $T \in \mathcal{T}$ **do** Compute the level of task T;
$t := 0$; $h := m$;
repeat
  Construct set $\mathcal{Z}$ of tasks without predecessors at time t;
  while $h > 0$ **and** $|\mathcal{Z}| > 0$ **do**
    begin
    Construct subset $\mathcal{S}$ of $\mathcal{Z}$ consisting of tasks at the highest level;
    if $|\mathcal{S}| > h$
    then
      begin
      Assign $\beta := h/|\mathcal{S}|$ of a processing capacity to each of the tasks from $\mathcal{S}$;
      $h := 0$; -- a processor shared partial schedule is constructed
      end
    else
      begin
      Assign one processor to each of the tasks from $\mathcal{S}$;
      $h := h - |\mathcal{S}|$; -- a "normal" partial schedule is constructed
      end;
    $\mathcal{Z} := \mathcal{Z} - \mathcal{S}$;
    end; -- the most "urgent" tasks have been assigned at time t
  Calculate time τ at which **either** one of the assigned tasks is finished **or** a point is reached at which continuing with the present partial assignment means that a task at a lower level will be executed at a faster rate β than a task at a higher level;
  Decrease levels of the assigned tasks by $(\tau - t)\beta$;
  $t := \tau$; $h := m$;
    -- a portion of each assigned task equal to $(\tau - t)\beta$ has been processed
until all tasks are finished;
call Algorithm 5.1.8 to re-schedule portions of the processor shared schedule to get a normal one;
end;

The above algorithm can be implemented to run in $O(n^2)$ time. An example of its application to an instance of problem $P2\,|\,pmtn, prec\,|\,C_{max}$ is shown in Figure 5.1.13.

At this point let us also consider another class of the precedence graphs for which the scheduling problem can be solved in polynomial time. To do this we have to present precedence constraints in the form of an activity network (task-on-arc precedence graph, viz. Section 3.1) whose nodes (events) are ordered in such a way that the occurrence of node i is not later than the occurrence of node j, if $i < j$.

(a)



(b)

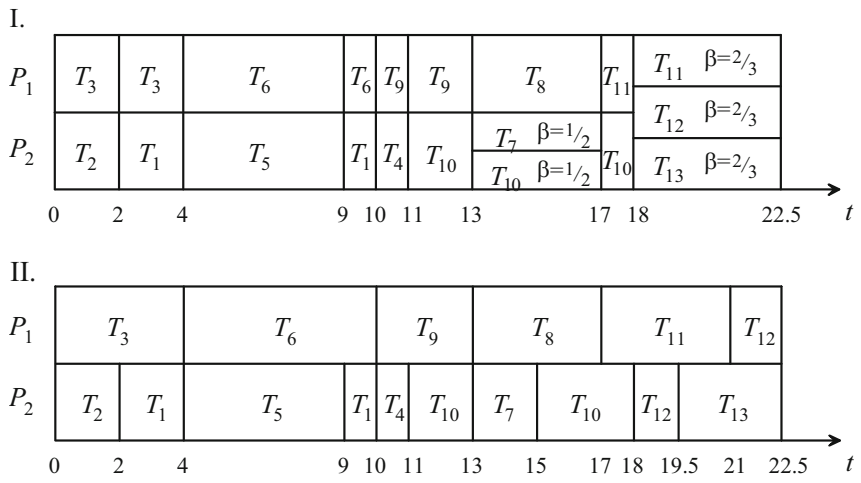


Figure 5.1.13 *An example of the application of Algorithm* 5.1.13
(a) *a task set* (*nodes are denoted by* T_j/p_j),
(b) I: *a processor-shared schedule,* II: *an optimal schedule.*

Now, let $\mathcal{S}_I$ denote the set of all the tasks which may be performed between the occurrence of event (node) I and $I+1$. Such sets will be called *main sets*. Let us consider *processor feasible sets*, i.e. those main sets and those subsets of the main sets whose cardinalities are not greater than m, and number these sets from 1 to some K. Now, let Q_j denote the set of indices of processor feasible sets in which task T_j may be performed, and let x_i denote the duration of the i^{th} feasible set. Then, a linear programming problem can be formulated in the straightforward way [WBCS77, BCSW76b] (another *LP* formulation for unrelated processors is presented in Section 5.1.2 as the first phase of a two-phase method):

$$\textit{Minimize} \qquad C_{max} = \sum_{i=1}^{K} x_i \tag{5.1.14}$$

$$\begin{aligned} \textit{subject to} \qquad & \sum_{i \in Q_j} x_i = p_j, \qquad j = 1,2,\cdots,n, \\ & x_i \geq 0, \qquad\quad i = 1,2,\cdots,K. \end{aligned} \tag{5.1.15}$$

It is clear that the solution of the *LP* problem depends on the order of nodes in the activity network; hence an optimal solution is found when this topological order is unique. Such a situation takes place for a *uniconnected activity network* (*uan*), i.e. one in which any two nodes are connected by a directed path in only one direction. An example of a uniconnected activity network together with the corresponding precedence graph is shown in Figure 5.1.14. On the other hand, the number of variables in the above *LP* problem depends polynomially on the input length, when the number of processors m is fixed. We may then use a non-simplex algorithm (e.g. from [Kha79] or [Kar84]) which solves any *LP* problem in time polynomial in the number of variables and constraints. Hence, we may conclude that the above procedure solves problem $Pm\,|\,pmtn, uan\,|\,C_{max}$ in polynomial time.

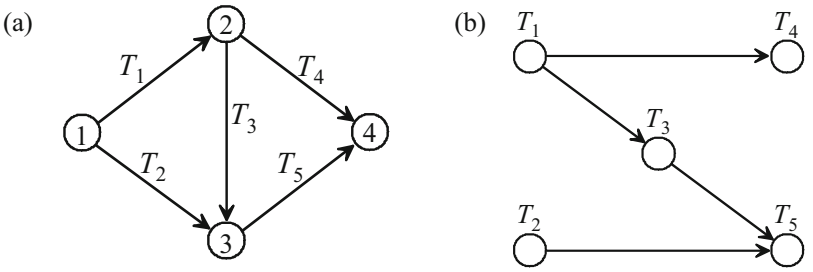


Figure 5.1.14 **(a)** *An example of a simple uniconnected activity network,* **(b)** *The corresponding precedence graph.* *Main sets* $S_1 = \{T_1, T_2\}$, $S_2 = \{T_2, T_3, T_4\}$, $S_3 = \{T_4, T_5\}$.

Recently another *LP* formulation has been proposed which enables one to solve problem $P \mid pmtn, uan \mid C_{max}$ in polynomial time, regardless of a number of processors [JMR+04].

As we already mentioned the uniconnected activity network has a task-on-node equivalent representation in a form of the interval order. Below, we present a sketch of the proof [BK02]. Let us start with the following theorem which will be given without a proof.

Theorem 5.1.14 *Let G be an activity network (task-on-arc graph). G is uniconnected if and only if G has a Hamiltonian path.* □

Now, the following theorems may be proved [BK02].

Theorem 5.1.15 *If G is a uan, then G is a task-on-arc representation of an interval order.*

Proof. By Theorem 5.1.14, $G = (V, A)$ is composed of a Hamiltonian path $W = (v_1, \ldots, v_n)$ with possibly some additional arcs of the form (v_i, v_j) with $i < j$. The interval order we are looking for is defined by the following collection of intervals $(I_a)_{a \in A}$. For every arc $a = (v_i, v_j)$ of A, we put the interval $[i, j)$ into the collection.

We have now to show that $I_a = [i, j)$ is entirely to the left of $I_{a'} = [i', j')$ if and only if a has to precede a' in the task precedence constraints represented by G. This is easy to show, since:

$I_a = [i, j)$ is entirely to the left of $I_{a'} = [i', j')$

$\Leftrightarrow$ $j \leq i'$

$\Leftrightarrow$ there is a path from v_j to $v_{i'}$ in G (along W)

$\Leftrightarrow$ a with head j has to precede a' with tail i'. □

If dummy tasks are not allowed, an interval order does not necessarily have a task-on-arc representation. Indeed, if we consider the collection of intervals $\{[1,2), [1,3), [2,4), [3,4)\}$, its task-on-node representation is graph N in Figure 2.3.1. It implies that this partial order does not have a task-on-arc representation without dummy tasks. But the equivalence of task-on-node and task-on-arc representations can be obtained through the use of dummy tasks. Since we allow them also here, the following result can be proved.

Theorem 5.1.16 *Any interval order has a task-on-arc representation with a Hamiltonian path (and therefore corresponds to a uan).*

Proof. Consider any collection of intervals $(I_a)_{a \in A}$ with $I_a = [b_a, e_a)$. We define the following graph $G=(V,E)$. Set

$$V = \{ b_a \mid a \in A \} \cup \{ e_a \mid a \in A \}.$$

For any v in V, let *next(v)* be the vertex $w > v$ such that there is no x in V with $v \pi x \pi w$ (*next*(v) is not defined for the largest e_a). Set

$$A' = \{(v, next(v)) \mid v \in V \text{ and } next(v) \text{ defined}\}$$

and

$$E = A' \cup \{(b_a, e_a) \mid a \in A\}.$$

The arcs in A' represent dummy tasks. This graph G has indeed a Hamiltonian path, starting with the smallest b_a ($\min_{a \in A} e_a$), following the arcs in A' and ending at the largest e_a ($\max_{a \in A} e_a$). It remains to show that $I_a = [b_a, e_a)$ is entirely to the left of $I_{a'} = [b_{a'}, e_{a'})$ if and only if arc (b_a, e_a) has to precede arc $[b_{a'}, e_{a'})$ in the task precedence constraints represented by G. We do not have to deal with arcs in A' since they represent dummy tasks:

$I_a = [b_a, e_a)$ is entirely to the left of $I_{a'} = [b_{a'}, e_{a'})$

$\Leftrightarrow e_a \le b_{a'}$

$\Leftrightarrow$ there is a path from e_a to $b_{a'}$ in G (using the arcs in A')

$\Leftrightarrow (b_a, e_a)$ with head e_a has to precede $(b_{a'}, e_{a'})$ with tail $b_{a'}$.

□

The following corollary is a direct consequence of Theorems 5.1.15 and 5.1.16

Corollary 5.1.17 *Let Q be a partial order. If dummy tasks are allowed, Q is an interval order if and only if Q can be represented as a uan.*

We may now conclude the above considerations with the following result:

$P \mid pmtn, interval\ order \mid C_{max}$ is solvable in polynomial time.

For general precedence graphs, however, we know from Ullman [Ull76] that the problem is NP-hard. In that case a heuristic algorithm such as Algorithm 5.1.13 my be chosen. The worst-case behavior of Algorithm 5.1.13 applied in the case of $P \mid pmtn, prec \mid C_{max}$ has been analyzed by Lam and Sethi [LS77]:

$$R_{\text{Alg.5.1.13}} = 2 - \frac{2}{m}, \qquad m \ge 2.$$

5.1.2 Uniform and Unrelated Processors

Problem $Q \mid p_j = 1 \mid C_{max}$

Let us start with an analysis of independent tasks and non-preemptive scheduling. Since the problem with arbitrary processing times is already NP-hard for identical processors, all we can hope to find is a polynomial time optimization algorithm for tasks with unit standard processing times only. Such an approach

has been given by Graham et al. [GLL+79] where a transportation network formulation has been presented for problem $Q|p_j = 1|C_{max}$. We describe it briefly below.

Let there be n sources j, $j = 1, 2, \cdots, n$, and mn sinks (i, k), $i = 1, 2, \cdots, m$ and $k = 1, 2, \cdots, n$. Sources correspond to tasks and sinks to processors and positions of tasks on them. Let $c_{ijk} = k/b_i$ be the cost of arc $(j, (i, k))$; this value corresponds to the completion time of task T_j processed on P_i in the k^{th} position. The arc flow x_{ijk} has the following interpretation:

$$x_{ijk} = \begin{cases} 1 & \text{if } T_j \text{ is processed in the } k^{\text{th}} \text{ position on } P_i \\ 0 & \text{otherwise.} \end{cases}$$

The min-max transportation problem can be now formulated as follows:

$$\textit{Minimize} \qquad \max_{i,j,k} \{c_{ijk} x_{ijk}\} \tag{5.1.16}$$

$$\textit{subject to} \qquad \sum_{i=1}^{m} \sum_{k=1}^{n} x_{ijk} = 1 \qquad \text{for all } j, \tag{5.1.17}$$

$$\sum_{j=1}^{n} x_{ijk} \le 1 \qquad \text{for all } i, k, \tag{5.1.18}$$

$$x_{ijk} \ge 0 \qquad \text{for all } i, j, k. \tag{5.1.19}$$

This problem can be solved by a standard transportation procedure (cf. Section 2.3) which results in $O(n^3)$ time complexity, or by a procedure due to Sevastjanov [Sev91]. Below we sketch this last approach. It is clear that the minimum schedule length is given as

$$C^*_{max} = \sup \{t \mid \sum_{i=1}^{m} \lfloor t b_i \rfloor < n\}. \tag{5.1.20}$$

On the other hand, a lower bound on the schedule length for the above problem is

$$C' = n / \sum_{i=1}^{m} b_i \le C^*_{max}. \tag{5.1.21}$$

Bound C' can be achieved e.g. by a preemptive schedule. If we assign $k_i = \lfloor C' b_i \rfloor$ tasks to processor P_i, $i = 1, 2, \cdots, m$, respectively, then these tasks may be processed in time interval $[0, C']$. However, $l = n - \sum_{i=1}^{m} k_i$ tasks remain unassigned. Clearly $l \le m - 1$, since $C' b_i - \lfloor C' b_i \rfloor < 1$ for each i. The remaining l tasks are then assigned one by one to those processors P_i for which $\min_i \{(k_i + 1) / b_i\}$ is attained at a given stage, where, of course, k_i is increased by one after the assignment of a task to a particular processor P_i. This procedure is repeated until all tasks are

assigned. We see that this approach results in an $O(m^2)$-algorithm for solving problem $Q\,|\,p_j = 1\,|\,C_{max}$.

Example 5.1.18 To illustrate the above algorithm let us assume that $n = 9$ tasks are to be processed on $m = 3$ uniform processors whose processing speeds are given by the vector $\boldsymbol{b} = [3, 2, 1]$. We get $C' = 9/6 = 1.5$. The numbers of tasks assigned to processors at the first stage are, respectively, 4, 3, and 1. A corresponding schedule is given in Figure 5.1.15(a), where task T_9 has not yet been assigned. An optimal schedule is obtained if this task is assigned to processor P_1, cf. Figure 5.1.15(b). □

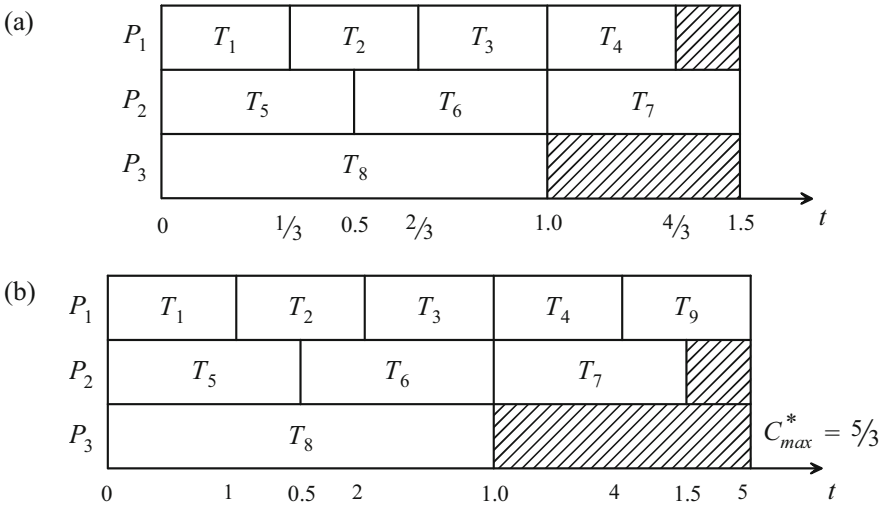


Figure 5.1.15 *Schedules for Example* 5.1.18
(a) a *partial schedule,*
(b) *an optimal schedule.*

Problem $Q\,|\,|\,C_{max}$

Since other problems of non-preemptive scheduling of independent tasks are NP-hard, one may be interested in applying certain heuristics. One heuristic algorithm which is a list scheduling algorithm, has been presented by Liu and Liu [LL74a]. Tasks are ordered on the list in non-increasing order of their processing times and processors are ordered in non-increasing order of their processing speeds. Now, whenever a machine becomes free it gets the first non-assigned task of the list; if there are two or more free processors, the fastest is chosen. The worst-case behavior of the algorithm has been evaluated for the case of an $m+1$ processor system, m of which have processing speed factor equal to 1 and the remaining processor has processing speed factor b. The bound is as follows.

$$R = \begin{cases} \frac{2(m+b)}{b+2} & \text{for } b \le 2 \\ \frac{m+b}{2} & \text{for } b > 2\,. \end{cases}$$

It is clear that the algorithm does better if, in the first case ($b \le 2$), m decreases faster than b, and if b and m decrease in case of $b > 2$. Other algorithms have been analyzed by Liu and Liu [LL74b, LL74c] and by Gonzalez et al. [GIS77].

Problem $Q\,|\,pmtn\,|\,C_{max}$

By allowing preemptions, i.e. for the problem $Q\,|\,pmtn\,|\,C_{max}$, one can find optimal schedules in polynomial time. We present an algorithm given by Horvath et al. [HLS77] despite the fact that there is a more efficient one by Gonzalez and Sahni [GS78]. We do this because the first algorithm covers also precedence constraints, and it generalizes the ideas presented in Algorithm 5.1.13. The algorithm is based on two concepts: the *task level*, defined as previously as processing requirement of the unexecuted portion of a task, but now expressed in terms of a standard processing time, and *processor sharing*, i.e. the possibility of assigning only a fraction β ($0 \le \beta \le \max\{b_i\}$) of processing capacity to some task. Let us assume that tasks are indexed in order of non-increasing p_j's and processors are in order of non-increasing values of b_i. It is quite clear that the minimum schedule length can be estimated by

$$C^*_{max} \ge C = \max\Big\{ \max_{1 \le k \le m} \{\frac{X_k}{B_k}\},\ \{\frac{X_n}{B_m}\} \Big\} \tag{5.1.22}$$

where X_k is the sum of processing requirements (i.e. standard processing times p_j) of the first k tasks, and B_k is the collective processing capacity (i.e. the sum of processing speed factors b_i) of the first k processors. The algorithm presented below constructs a schedule of length equal to C for the problem $Q\,|\,pmtn\,|\,C_{max}$.

Algorithm 5.1.19 *Algorithm by Horvath, Lam and Sethi for $Q\,|\,pmtn\,|\,C_{max}$* [HLS77].

```
begin
for all T ∈ 𝒯 do Compute level of task T;
t := 0; h := m;
repeat
  while h > 0 do
    begin
    Construct subset 𝒮 of 𝒯 consisting of tasks at the highest level;
        -- the most "urgent" tasks are chosen
```

```
      if |S| > h
      then
        begin
        Assign the tasks of set S to the h remaining processors to be processed
            at the same rate β = Σ_{i=m−h+1}^{m} b_i/|S|;
        h := 0;      -- tasks from set S share the h slowest processors
        end
      else
        begin
        Assign tasks from set S to be processed at the same rate β on the fastest
            |S| processors;
        h := h−|S|;      -- tasks from set S share the fastest |S| processors
        end;
      end;          -- the most urgent tasks have been assigned at time t
    Calculate time moment τ at which either one of the assigned tasks is finished
      or a point is reached at which continuing with the present partial assign-
      ment causes that a task at a lower level will be executed at a faster rate β
      than a higher level task;
          -- note, that the levels of the assigned tasks decrease during task execution
    Decrease levels of the assigned tasks by (τ−t)β;
    t := τ ; h := m;
              -- a portion of each assigned task equal to (τ−t)β has been processed
until all tasks are finished;
    -- the schedule constructed so far consists of a sequence of intervals during each
    -- of which certain tasks are assigned to the processors in a shared mode.
    -- In the next loop task assignment in each of these intervals is determined
for each interval of the processor shared schedule do
  begin
  Let y be the length of the interval;
  if g tasks share g processors
  then Assign each task to each processor for y/g time units
  else
    begin
      Let p be the processing requirement of each of the g tasks in the inter-
          val;
      Let b be the processing speed factor of the slowest processor;
    if p/b < y
    then call Algorithm 5.1.8
          -- tasks can be assigned as in McNaughton's rule,
          -- ignoring different processor speeds
    else
      begin
      Divide the interval into g subintervals of equal lengths;
```

```
            Assign the g tasks so that each task occurs in exactly h intervals, each
              time on a different processor;
          end;
        end;
      end;
            -- a normal preemptive schedule has now been constructed
end;
```

The time complexity of Algorithm 5.1.19 is $O(mn^2)$. An example of its application is shown in Figure 5.1.16.

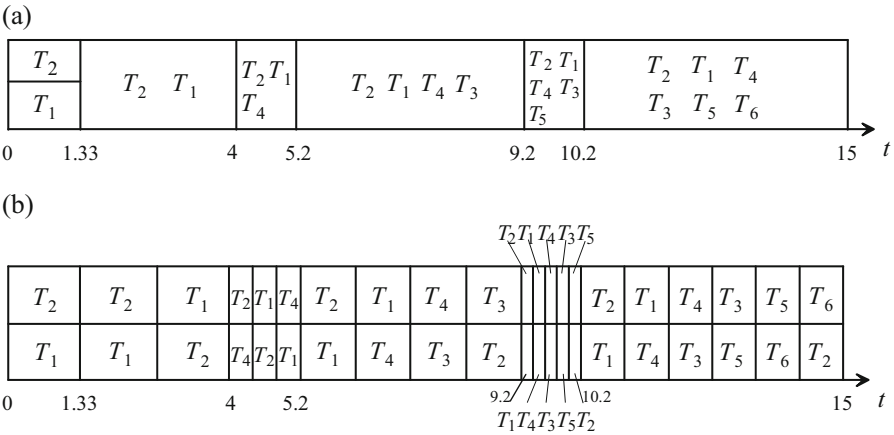


Figure 5.1.16 *An example of the application of Algorithm* 5.1.19: $n = 6$, $m = 2$, $\boldsymbol{p} = [20, 24, 10, 12, 5, 4]$, $\boldsymbol{b} = [4, 1]$
(a) *a processor shared schedule,*
(b) *an optimal schedule.*

Problem $Q\,|\,pmtn, prec\,|\,C_{max}$

When considering dependent tasks, only preemptive polynomial time optimization algorithms are known. Algorithm 5.1.19 also solves problem $Q2\,|\,pmtn, prec\,|\,C_{max}$, if the level of a task is understood as in Algorithm 5.1.13 where standard processing times for all the tasks were assumed. When considering this problem one should also take into account the possibility of solving it for uniconnected activity networks and interval orders via the slightly modified linear programming approach (5.1.14)-(5.1.15). It is also possible to solve the problem by using another *LP* formulation which is described for the case of $R\,|\,pmtn\,|\,C_{max}$.

It is also possible to solve problem $Q\,|\,pmtn, prec\,|\,C_{max}$ approximately by the two machine aggregation approach, developed in the framework of flow shop

scheduling [RS83] (cf. Chapter 8). In this case the two fastest processors are used only, and the worst case bound is

$$\frac{C_{max}}{C^*_{max}} \leq \begin{cases} \sum_{i=1}^{m/2} \max\{b_{2i-1}/b_1, b_{2i}/b_2\} & \text{if } m \text{ is even,} \\ \sum_{i=1}^{\lfloor m/2 \rfloor} \max\{b_{2i-1}/b_1, b_{2i}/b_2\} + b_m/b_1 & \text{if } m \text{ is odd.} \end{cases}$$

Problem $R \mid pmtn \mid C_{max}$

Let us pass now to the case of unrelated processors. This case is the most difficult. We will not speak about unit-length tasks, because unrelated processors with unit length tasks would reduce to the case of identical or uniform processors. Hence, no polynomial time optimization algorithms are known for problems other than preemptive ones. Also, very little is known about approximation algorithms for this case. Some results have been obtained by Ibarra and Kim [IK77], but the obtained bounds are not very encouraging. Thus, we will pass to the preemptive scheduling model.

Problem $R \mid pmtn \mid C_{max}$ can be solved by a *two-phase method*. The first phase consists in solving a linear programming problem formulated independently by Błażewicz et al. [BCSW76a, BCW77] and by Lawler and Labetoulle [LL78]. The second phase uses the solution of this *LP* problem and produces an optimal preemptive schedule.

Let $x_{ij} \in [0, 1]$ denote the part of T_j processed on P_i. The *LP* formulation is as follows:

$$\textit{Minimize} \qquad C_{max} \tag{5.1.23}$$

$$\textit{subject to} \qquad C_{max} - \sum_{j=1}^{n} p_{ij} x_{ij} \geq 0, \qquad i = 1, 2, \cdots, m \tag{5.1.24}$$

$$C_{max} - \sum_{i=1}^{m} p_{ij} x_{ij} \geq 0, \qquad j = 1, 2, \cdots, n \tag{5.1.25}$$

$$\sum_{i=1}^{m} x_{ij} = 1, \qquad j = 1, 2, \cdots, n. \tag{5.1.26}$$

Solving the above problem, we get $C_{max} = C^*_{max}$ and optimal values x^*_{ij}. However, we do not know how to schedule the task parts, i.e. how to assign these parts to processors in time. A schedule may be constructed in the following way. Let $\boldsymbol{T} = [t^*_{ij}]$ be the $m \times n$ matrix defined by $t^*_{ij} = p_{ij} x^*_{ij}$, $i = 1, 2, \cdots, m$, $j = 1, 2, \cdots, n$. Notice that the elements of $\boldsymbol{T}$ reflect optimal values of processing times of particular tasks on the processors. The j^{th} column of $\boldsymbol{T}$ corresponding to task T_j

will be called *critical* if $\sum_{i=1}^{m} t^*_{ij} = C^*_{max}$. By $\boldsymbol{Y}$ we denote an $m \times m$ diagonal matrix whose element y_{kk} is the total idle time on processor P_k, i.e. $y_{kk} = C^*_{max} - \sum_{j=1}^{n} t^*_{kj}$. Columns of $\boldsymbol{Y}$ correspond to dummy tasks. Let $\boldsymbol{V} = [\boldsymbol{T},\boldsymbol{Y}]$ be an $m \times (n+m)$ matrix. Now set $\mathcal{U}$ containing m positive elements of matrix $\boldsymbol{V}$ is defined as having exactly one element from each critical column and at most one element from other columns, and having exactly one element from each row. We see that $\mathcal{U}$ corresponds to a task set which may be processed in parallel in an optimal schedule. Thus, it may be used to construct a partial schedule of some length $\delta > 0$. An optimal schedule is then produced as the union of the partial schedules. This procedure is summarized in Algorithm 5.1.20 [LL78].

Algorithm 5.1.20 *Construction of an optimal schedule corresponding to LP solution for* $R\,|\,pmtn\,|\,C_{max}$.

begin
$C := C^*_{max}$;
while $C > 0$ **do**
  begin
  Construct set $\mathcal{U}$;
    -- thus a subset of tasks to be processed in a partial schedule has been chosen
  $v_{\min} := \min_{v_{ij} \in \mathcal{U}} \{v_{ij}\}$;
  $v_{max} := \max_{j \in \{j' \mid v_{ij'} \notin \mathcal{U} \text{ for } i = 1,\ldots,m\}} \{\sum_i v_{ij}\}$;
  if $C - v_{\min} \geq v_{max}$
  then $\delta := v_{\min}$
  else $\delta := C - v_{max}$;
    -- the length of the partial schedule is equal to δ
  $C := C - \delta$;
  for each $v_{ij} \in \mathcal{U}$ **do** $v_{ij} := v_{ij} - \delta$;
    -- matrix $\boldsymbol{V}$ is changed; notice that due to the way δ is defined,
    -- the elements of $\boldsymbol{V}$ can never become negative
  end;
end;

The proof of correctness of the algorithm can be found in [LL78].

Now we only need an algorithm that finds set $\mathcal{U}$ for a given matrix $\boldsymbol{V}$. One of the possible algorithms is based on the network flow approach. In this case the network has m nodes corresponding to machines (rows of $\boldsymbol{V}$) and $n+m$ nodes corresponding to tasks (columns of $\boldsymbol{V}$), cf. Figure 5.1.17. A node i from the first group is connected by an arc to a node j of the second group if and only if $v_{ij} > 0$. Arc

flows are constrained by b from below and by $c = 1$ from above, where the value of b is 1 for arcs joining the source with processor-nodes and critical task nodes with the sink, and $b = 0$ for the other arcs. Obviously, finding a feasible flow in this network is equivalent to finding set $\mathcal{U}$. The following example illustrates the second phase of the described method.

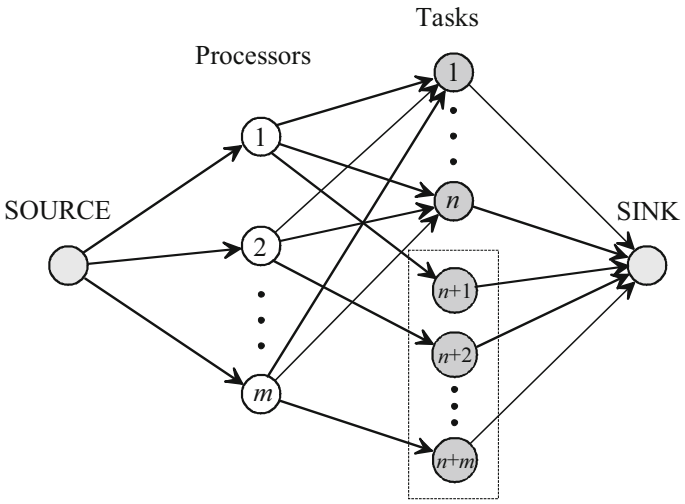


Figure 5.1.17 *Finding set $\mathcal{U}$ by the network flow approach.*

Example 5.1.21 Suppose that for a certain scheduling problem a linear programming solution of the two phase method has the form given in Figure 5.1.18(a). An optimal schedule is then constructed in the following way. First, matrix $\boldsymbol{V}$ is calculated.

$$\boldsymbol{V} = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccc} T_1 & T_2 & T_3 & T_4 & T_5 \end{array} \\ \left[\begin{array}{ccccc} \underline{3} & 2 & 1 & 4 & 0 \\ 2 & 2 & 0 & \underline{2} & 2 \\ 2 & 1 & \underline{4} & 0 & 1 \end{array}\right] \\ \begin{array}{ccccc} 7 & 5 & 5 & 6 & 3 \end{array} \end{array} \begin{array}{c} \begin{array}{ccc} T_6 & T_7 & T_8 \end{array} \\ \left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{array}\right] \\ \begin{array}{ccc} 0 & 2 & 2 \end{array} \end{array}$$

Then elements constituting set $\mathcal{U}$ are chosen according to Algorithm 5.1.20, as depicted above. The value of a partial schedule length is $\delta = 2$. Next, the while-loop of Algorithm 5.1.20 is repeated yielding the following sequence of matrices $\boldsymbol{V_i}$.

$$\boldsymbol{V_1} = \left[\begin{array}{ccccc} 1 & 2 & 1 & \underline{4} & 0 \\ 2 & \underline{2} & 0 & 0 & 2 \\ \underline{2} & 1 & 2 & 0 & 1 \end{array}\right] \left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{array}\right]$$

$$V_2 = \begin{bmatrix} 1 & \underline{2} & 1 & 2 & 0 \\ \underline{2} & 0 & 0 & 0 & 2 \\ 0 & 1 & \underline{2} & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

$$V_3 = \begin{bmatrix} 1 & 0 & 1 & \underline{2} & 0 \\ 0 & 0 & 0 & 0 & \underline{2} \\ 0 & \underline{1} & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

$$V_4 = \begin{bmatrix} \underline{1} & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & \underline{1} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & \underline{2} \end{bmatrix}$$

$$V_5 = \begin{bmatrix} 0 & 0 & \underline{1} & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \underline{1} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & \underline{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$V_6 = \begin{bmatrix} 0 & 0 & 0 & \underline{1} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & \underline{1} & 0 \\ 0 & 0 & \underline{1} \end{bmatrix}.$$

A corresponding optimal schedule is presented in Figure 5.1.18(b). □

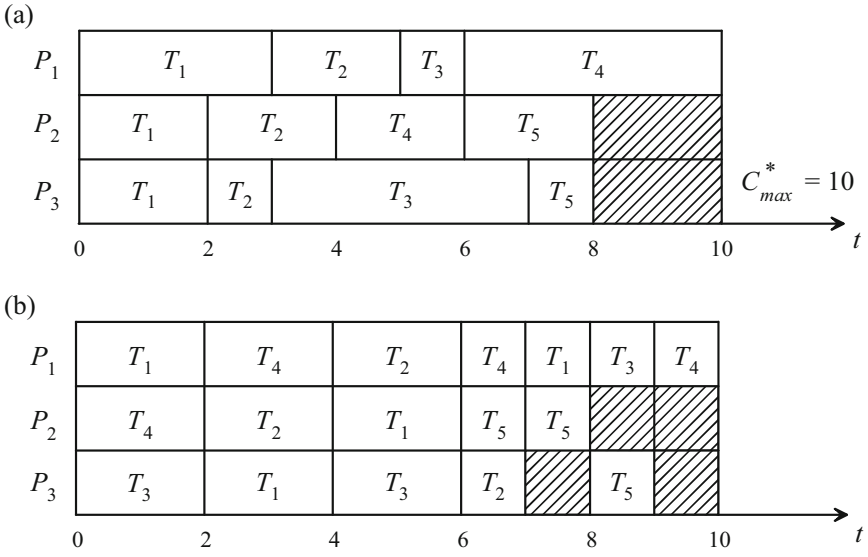


Figure 5.1.18 **(a)** *A linear programming solution for an instance of* $R \mid pmtn \mid C_{max}$, **(b)** *an optimal schedule.*

The overall complexity of the above approach is bounded from above by a polynomial in the input length. This is because the transformation to the *LP* problem is polynomial, and the *LP* problem may be solved in polynomial time using Khachiyan's algorithm [Kha79]; the loop in Algorithm 5.1.20 is repeated at most $O(mn)$ times and solving the network flow problem requires $O(z^3)$ time, where z is the number of network nodes [Kar74].

Problem $R|pmtn,prec|C_{max}$

If dependent tasks are considered, i.e. in the case $R|pmtn,prec|C_{max}$, linear programming problems similar to those discussed in (5.1.14)-(5.1.15) or (5.1.23)-(5.1.26) and based on the activity network presentation, can be formulated. For example, in the latter formulation one defines x_{ijk} as a part of task T_j processed on processor P_i in the main set S_k. Solving the *LP* problem for x_{ijk}, one then applies Algorithm 5.1.20 for each main set. If the activity network is uniconnected (a corresponding task-on-node graph represents an interval order), an optimal schedule is constructed in this way, otherwise only an approximate schedule is obtained. Notice that in [JMR+04] a two-phase method has been proposed for problem $P|pmtn, uan|C_{max}$ with the McNaughton algorithm applied for each main set. This reduces the complexity of the second phase to $O(n^2)$. In this paper also several heuristics for ordering network nodes have been proposed and tested experimentally, leading finally to an almost optimal algorithm for problem $P|pmtn,prec|C_{max}$.

We complete this chapter by remarking that introduction of ready times into the model considered so far is equivalent to the problem of minimizing maximum lateness. We will consider this type of problems in Section 5.3.

5.2 Minimizing Mean Flow Time

5.2.1 Identical Processors

Problem $P||\Sigma C_j$

In the case of identical processors and equal ready times preemptions are not profitable from the viewpoint of the value of the mean flow time [McN59]. Thus, we can limit ourselves to considering non-preemptive schedules only.

When analyzing the nature of criterion ΣC_j, one might expect that, as in the case of one processor (cf. Section 4.2), by assigning tasks in non-decreasing order of their processing times the mean flow time will be minimized. In fact, a

proper generalization of this simple rule leads to an optimization algorithm for $P||\Sigma C_j$ (Conway et al. [CMM67]). It is as follows.

Algorithm 5.2.1 *SPT rule for problem* $P||\Sigma C_j$ [CMM67].
begin
Order tasks on list L in non-decreasing order of their processing times;
while $L \neq \emptyset$ **do**
  begin
  Take the m first tasks from the list (if any) and assign these tasks arbitrarily to the m different processors;
  Remove the assigned tasks from list L;
  end;
Process tasks assigned to each processor in *SPT* order;
end;

The complexity of the algorithm is obviously $O(n\log n)$.

In this context let us also mention that introducing different ready times makes the problem strongly NP-hard even for the case of one processor (see Section 4.2 and [LRKB77]). Also, if we introduce different weights, then the 2-processor problem without release times, $P2||\Sigma w_j C_j$, is already NP-hard [BCS74].

Problem $P|prec|\Sigma C_j$

Let us now pass to the case of dependent tasks. Here, $P|out\text{-}tree, p_j = 1|\Sigma C_j$ is solved by an adaptation of Algorithm 5.1.11 (Hu's algorithm) to the out-tree case [Ros–], and $P2|prec, p_j = 1|\Sigma C_j$ is strongly NP-hard [LRK78]. In the case of arbitrary processing times results by Du et al. [DLY91] indicate that even simplest precedence constraints result in computational hardness of the problem. That is problem $P2|chains|\Sigma C_j$ is already NP-hard in the strong sense. On the other hand, it was also proved in [DLY91] that preemptions cannot reduce the mean weighted flow time for a set of chains. Together with the last result this implies that problem $P2|chains, pmtn|\Sigma C_j$ is also NP-hard in the strong sense. Unfortunately, no approximation algorithms for these problems are evaluated from their worst-case behavior point of view.

5.2.2 Uniform and Unrelated Processors

The results of Section 5.2.1 also indicate that scheduling dependent tasks on uniform or unrelated processors is an NP-hard problem in general. No approximation algorithms have been investigated either. Thus, we will not consider this subject. On the other hand, in the case of independent tasks, preemptions may be

worthwhile, thus we have to treat non-preemptive and preemptive scheduling separately.

Problem $Q||\Sigma C_j$

Let us start with uniform processors and non-preemptive schedules. In this case the flow time has to take into account processor speed; so the flow time of task $T_{i[k]}$ processed in the k^{th} position on processor P_i is defined as $F_{i[k]} = \frac{1}{b_i}\sum_{j=1}^{k} p_{i[j]}$. Let us denote by n_i the number of tasks processed on processor P_i. Thus, $n = \sum_{i=1}^{m} n_i$. The mean flow time is then given by

$$\overline{F} = \frac{\sum_{i=1}^{m} \frac{1}{b_i} \sum_{k=1}^{n_i} (n_i - k + 1) p_{i[k]}}{n} . \tag{5.2.1}$$

It is easy to see that the numerator in the above formula is the sum of n terms each of which is the product of a processing time and one of the following coefficients:

$$\frac{1}{b_1}n_1, \frac{1}{b_1}(n_1-1), \cdots, \frac{1}{b_1}, \frac{1}{b_2}n_2, \frac{1}{b_2}(n_2-1), \cdots, \frac{1}{b_2}, \cdots, \frac{1}{b_m}n_m, \frac{1}{b_m}(n_m-1), \cdots, \frac{1}{b_m}.$$

It is known from [CMM67] that such a sum is minimized by matching n smallest coefficients in non-decreasing order with processing times in non-increasing order. An $O(n\log n)$ implementation of this rule has been given by Horowitz and Sahni [HS76].

Problem $Q|pmtn|\Sigma C_j$

In the case of preemptive scheduling, it is possible to show that there exists an optimal schedule for $Q|pmtn|\Sigma C_j$ in which $C_j \le C_k$ if $p_j < p_k$. On the basis of this observation, the following algorithm has been proposed by Gonzalez [Gon77].

Algorithm 5.2.2 *Algorithm by Gonzalez for* $Q|pmtn|\Sigma C_j$ [Gon77].

```
begin
Order processors in non-increasing order of their processing speed factors;
Order tasks in non-decreasing order of their standard processing times;
for j = 1 to n do
  begin
  Schedule task T_j to be completed as early as possible, preempting when
    necessary;
      -- tasks will create a staircase pattern "jumping" to a faster processor
      -- whenever a shorter task has been finished
```

```
    end;
end;
```

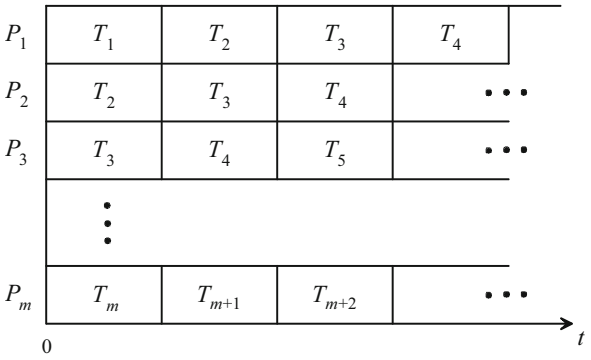


Figure 5.2.1 *An example of the application of Algorithm* 5.2.2.

The complexity of this algorithm is $O(n\log n+mn)$. An example of its application is given in Figure 5.2.1.

Problem $R||\Sigma C_j$

Let us now turn to the case of unrelated processors and consider problem $R||\Sigma C_j$. An approach to its solution is based on the observation that task $T_j \in \{T_1,\cdots,T_n\}$ processed on processor $P_i \in \{P_1,\cdots,P_m\}$ as the last task contributes its processing time p_{ij} to $\overline{F}$. The same task processed in the last but one position contributes $2p_{ij}$, and so on [BCS74]. This reasoning allows one to construct an $(m\,n)\times n$ matrix $\boldsymbol{Q}$ presenting contributions of particular tasks processed in different positions on different processors to the value of $\overline{F}$:

$$\boldsymbol{Q}=\begin{bmatrix}[p_{ij}]\\ 2[p_{ij}]\\ \cdot\\ \cdot\\ \cdot\\ n[p_{ij}]\end{bmatrix}$$

The problem is now to choose n elements from matrix $\boldsymbol{Q}$ such that
– exactly one element is taken from each column,
– at most one element is taken from each row,
– the sum of selected elements is minimum.

We see that the above problem is a variant of the assignment problem (cf. [Law76]), which may be solved in a natural way via the transportation problem. The corresponding transportation network is shown in Figure 5.2.2.

Careful analysis of the problem shows that it can be solved in $O(n^3)$ time [BCS74]. The following example illustrates this technique.

Example 5.2.3 Let us consider the following instance of problem $R||\Sigma C_j$: $n = 5$, $m = 3$, and matrix $\boldsymbol{p}$ of processing times

$$\boldsymbol{p} = \begin{bmatrix} 3 & 2 & 4 & 3 & 1 \\ 4 & 3 & 1 & 2 & 1 \\ 2 & 4 & 5 & 3 & 4 \end{bmatrix}.$$

Using this data the matrix $\boldsymbol{Q}$ is constructed as follows:

$$\boldsymbol{Q} = \begin{bmatrix} 3 & \underline{2} & 4 & 3 & 1 \\ 4 & 3 & 1 & \underline{2} & 1 \\ \underline{2} & 4 & 5 & 3 & 4 \\ 6 & 4 & 8 & 6 & \underline{2} \\ 8 & 6 & \underline{2} & 4 & 2 \\ 4 & 8 & 10 & 6 & 8 \\ 9 & 6 & 12 & 9 & 3 \\ 12 & 9 & 3 & 6 & 3 \\ 6 & 12 & 15 & 9 & 12 \\ 12 & 8 & 16 & 12 & 4 \\ 16 & 12 & 4 & 8 & 4 \\ 8 & 16 & 20 & 12 & 16 \\ 15 & 10 & 20 & 15 & 5 \\ 20 & 15 & 5 & 10 & 5 \\ 10 & 20 & 25 & 15 & 20 \end{bmatrix}.$$

On the basis of this matrix a network as shown in Figure 5.2.2 is constructed.

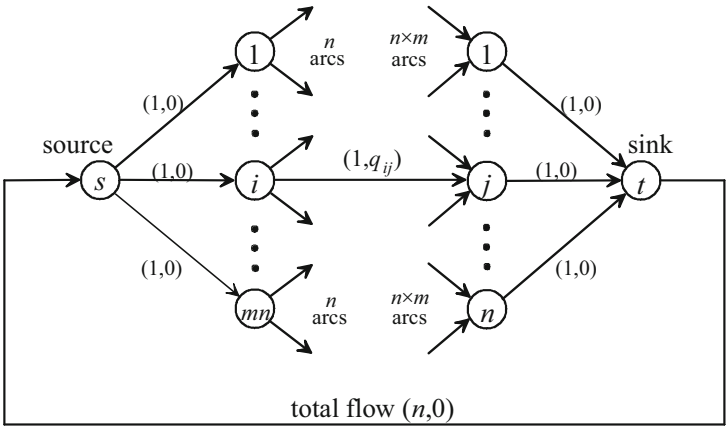


Figure 5.2.2 *The transportation network for problem $R||\Sigma C_j$: arcs are denoted by (c, y), where c is the capacity and y is the cost of unit flow.*

Solving the transportation problem results in the selection of the underlined elements of matrix $\boldsymbol{Q}$. They correspond to the schedule shown in Figure 5.2.3. □

A very surprising result has been recently obtained by Sitters. Problem $R|pmtn|\Sigma C_j$ has been proved to be strongly NP-hard [Sit05].

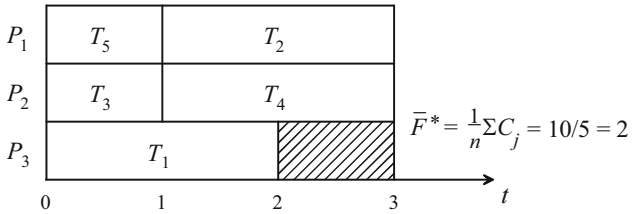


Figure 5.2.3 *An optimal schedule for Example* 5.2.3.

5.3 Minimizing Due Date Involving Criteria

5.3.1 Identical Processors

In Section 4.3 we have seen that single processor problems with due date optimization criteria involving due dates are NP-hard in most cases. In the following we will concentrate on minimization of L_{max} criterion. It seems to be quite natural that in this case the general rule should be to schedule tasks according to their earliest due dates (*EDD*-rule, cf. Section 4.3.1). However, this simple rule of Jackson [Jac55] produces optimal schedules under very restricted assumptions only. In other cases more sophisticated algorithms are necessary, or the problems are NP-hard.

Problem $P||L_{max}$

Let us start with non-preemptive scheduling of independent tasks. Taking into account simple transformations between scheduling problems (cf. Section 3.4) and the relationship between the C_{max} and L_{max} criteria, we see that all the problems that are NP-hard under the C_{max} criterion remain NP-hard under the L_{max} criterion. Hence, for example, $P2||L_{max}$ is NP-hard. On the other hand, unit processing times of tasks make the problem easy, and $P|p_j=1, r_j|L_{max}$ can be solved by an obvious application of the *EDD* rule [Bla77]. Moreover, problem $P|p_j=p, r_j|L_{max}$ can be solved in polynomial time by an extension of the single processor algorithm (see Section 4.3.1 and [GJST81]). Unfortunately very little is known about the worst-case behavior of approximation algorithms for the NP-hard problems in question.

Problem $P|pmtn, r_j|L_{max}$

The preemptive mode of processing makes the solution of the scheduling problem much easier. The fundamental approach in that area is testing feasibility of problem $P|pmtn, r_j, \tilde{d}_j|-$ via the network flow approach [Hor74]. Using this approach repetitively, one can then solve the original problem $P|pmtn, r_j|L_{max}$ by changing due dates (deadlines) according to a binary search procedure.

Let us now describe Horn's approach for testing feasibility of problem $P|pmtn, r_j, \tilde{d}_j|-$, i.e. deciding whether or not for a given set of ready times and deadlines there exists a schedule with no late task. Let the values of ready times and deadlines of an instance of $P|pmtn, r_j, \tilde{d}_j|-$ be ordered on a list in such a way that $e_0 < e_1 < \cdots < e_k$, $k < 2n$, where e_i stands for some r_j or $\tilde{d}_j$. We construct a network that has two sets of nodes, besides source and sink (cf. Figure 5.3.1). The first set corresponds to time intervals in a schedule, i.e. node w_i corresponds to interval $[e_{i-1}, e_i]$, $i = 1, 2, \cdots, k$. The second set corresponds to the task set. The capacity of an arc joining the source of the network to node w_i is equal to $m(e_i - e_{i-1})$ and thus corresponds to the total processing capacity of m processors in this interval. If task T_j could be processed in interval $[e_{i-1}, e_i]$ (because of its ready time and deadline) then w_i is joined to T_j by an arc of capacity $e_i - e_{i-1}$. Node T_j is joined to the sink of the network by an arc with capacity equal to p_j and with a lower bound on arc flow which is also equal to p_j. We see that finding a feasible flow pattern corresponds to constructing a feasible schedule and this test can be made in $O(n^3)$ time (cf. Section 2.3.3). A schedule is constructed on the basis of flow values on arcs between interval and task nodes. Let us consider the following example.

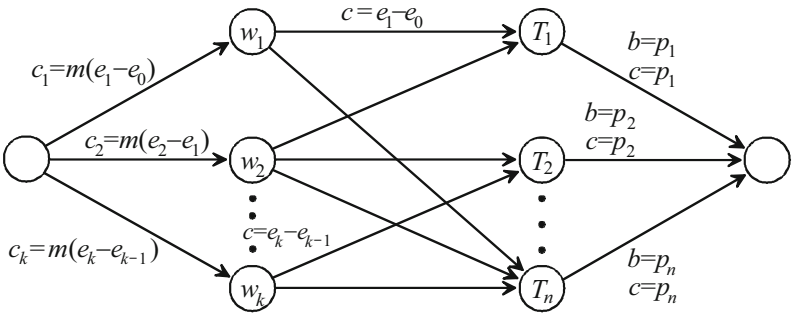


Figure 5.3.1 *Network corresponding to problem $P|pmtn, r_j, \tilde{d}_j|-$.*

Example 5.3.1 Let $n = 5$, $m = 2$, $\boldsymbol{p} = [5, 2, 3, 3, 1]$, $\boldsymbol{r} = [2, 0, 1, 0, 2]$, and $\boldsymbol{d} = [8, 2, 4, 5, 8]$. The corresponding network is shown in Figure 5.3.2(a), and a feasible

flow pattern is depicted in Figure 5.3.2(b). On the basis of this flow the feasible schedule shown in Figure 5.3.2(c) is constructed. □

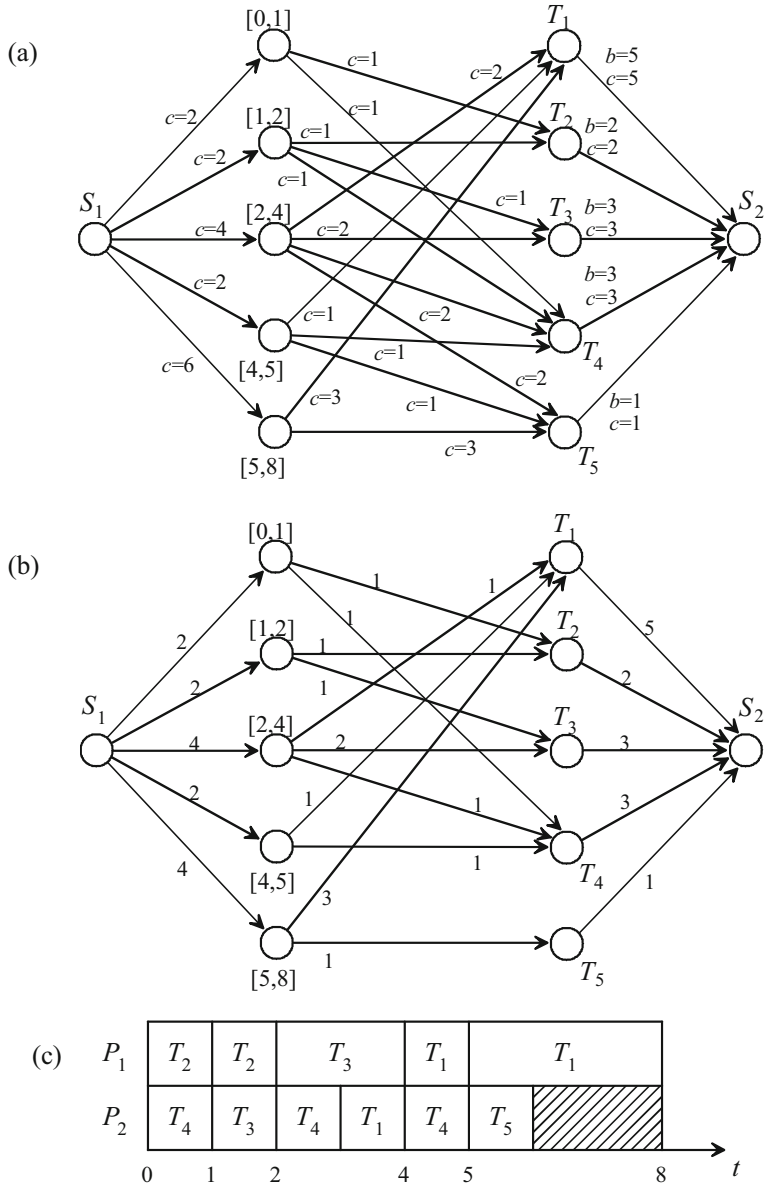


Figure 5.3.2 *Finding a feasible schedule via network flow approach* (*Example* 5.3.1)
(a) *a corresponding network,*

(b) *a feasible flow pattern,*
(c) *a schedule.*

In the next step a binary search can be conducted on the optimal value of L_{max}, with each trial value of L_{max} inducing deadlines which are checked for feasibility by means of the above network flow computation. This procedure can be implemented to solve problem $P \mid pmtn, r_j \mid L_{max}$ in $O(n^3 \min\{n^2, \log n + \log max\{p_j\}\})$ time [LLL+84].

Problem $P \mid prec, p_j = 1 \mid L_{max}$

Let us now pass to dependent tasks. A general approach in this case consists in assigning modified due dates to tasks, depending on the number and due dates of their successors. Of course, the way in which modified due dates are calculated depends on the parameters of the problem in question. If scheduling non-preemptable tasks on a multiple processor system only unit processing times can result in polynomial time scheduling algorithms. Let us start with in-tree precedence constraints and assume that if $T_i \prec T_j$ then $i > j$. The following algorithm minimizes L_{max} (isucc(j) denotes the immediate successor of T_j) [Bru76b].

Algorithm 5.3.2 *Algorithm by Brucker for $P \mid in\text{-}tree, p_j = 1 \mid L_{max}$* [Bru76b].
begin
$d_1^* := 1 - d_1$**;** -- the due date of the root node is modified
for $k = 2$ **to** n **do**
  begin
  Calculate modified due date of T_k according to the formula
    $d_k^* := \max\{1 + d_{\text{isucc}(k)}^*, 1 - d_k\}$**;**
  end;
Schedule tasks in non-increasing order of their modified due dates subject to precedence constraints**;**
end;

This algorithm can be implemented to run in $O(n \log n)$ time. An example of its application is given in Figure 5.3.3. Surprisingly *out-tree precedence constraints* result in the NP-hardness of the problem [BGJ77].

However, when we limit ourselves to two processors, a different way of computing modified due dates can be proposed which allows one to solve the problem in $O(n^2)$ time [GJ76]. In the algorithm below $g(k, d_i^*)$ is the number of successors of T_k having modified due dates not greater than d_i^*.

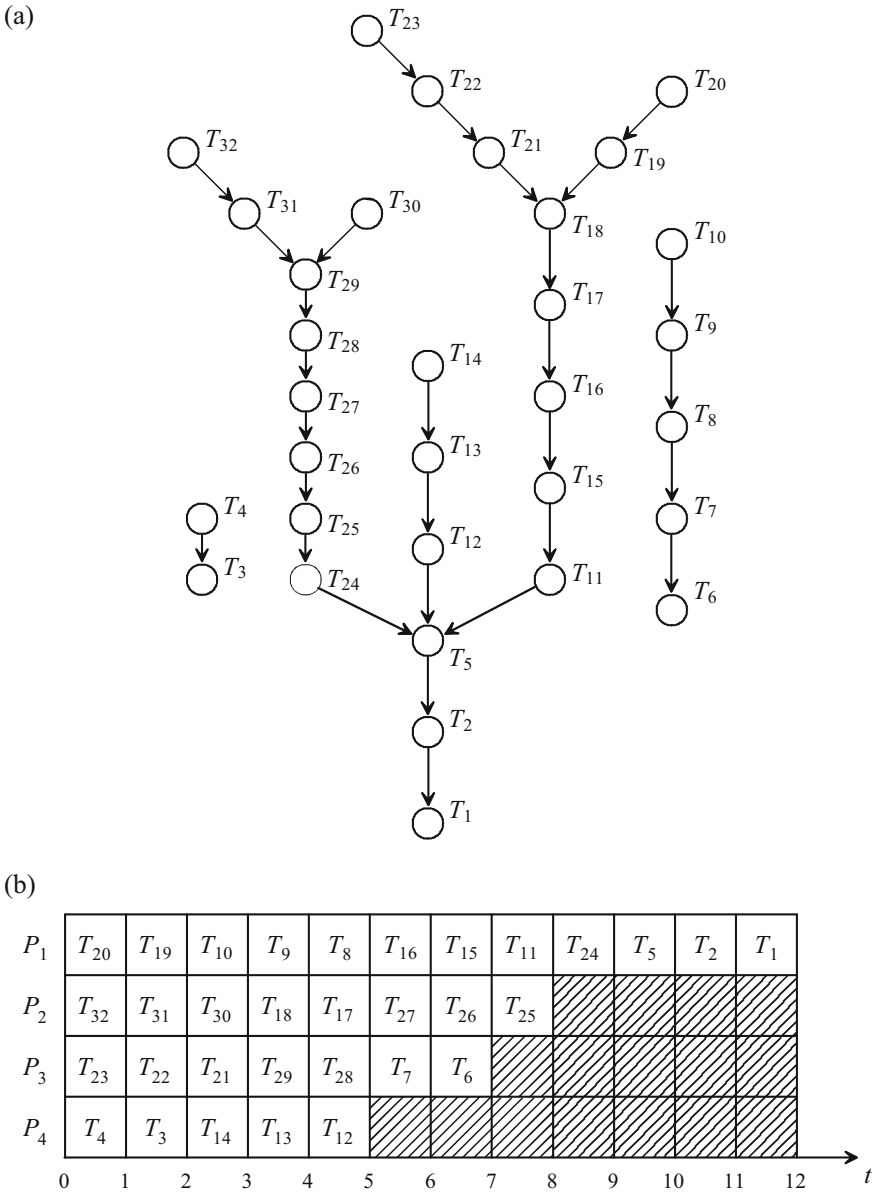


Figure 5.3.3 *An example of the application of Algorithm* 5.3.2;
$n = 32$, $m = 4$, $\boldsymbol{d} = [16,20,4,3,15,14,17,6,6,4,10,8,9,7,10,9,10,8,2,3,6,5,4,11,12,9,10,8,7,5,3,5]$
(a) *the task set,*
(b) *an optimal schedule.*

Algorithm 5.3.3 *Algorithm by Garey and Johnson for problem* $P2\,|\,prec,\ p_j = 1\,|\,L_{max}$ [GJ76].

begin
$\mathcal{Z} := \mathcal{T}$;
while $\mathcal{Z} \neq \emptyset$ **do**
  begin
  Choose $T_k \in \mathcal{Z}$ which is not yet assigned a modified due date and all of whose successors have been assigned modified due dates;
  Calculate a modified due date of T_k as:

$$d_k^* := \min\{d_k, \min\{(d_i^* - \lceil \tfrac{1}{2} g(k, d_i^*) \rceil) \mid T_i \in \mathrm{succ}(T_k)\}\};$$

  $\mathcal{Z} := \mathcal{Z} - \{T_k\}$;
  end;
Schedule tasks in non-decreasing order of their modified due dates subject to precedence constraints;
end;

For $m > 2$ this algorithm may not lead to optimal schedules, as demonstrated in the example in Figure 5.3.4. However, the algorithm can be generalized to cover the case of different ready times too, but the running time is then $O(n^3)$ [GJ77] and this is as much as we can get in non-preemptive scheduling.

Problem $P\,|\,pmtn, prec\,|\,L_{max}$

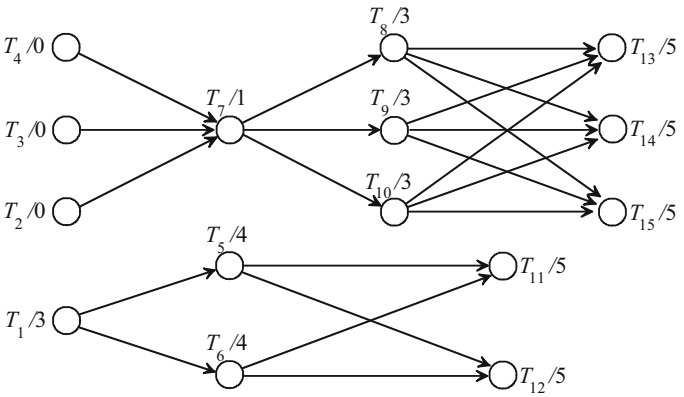
Preemptions allow one to solve problems with arbitrary processing times. In [Law82b] algorithms have been presented that are preemptive counterparts of Algorithms 5.3.2 and 5.3.3 and the one presented by Garey and Johnson [GJ77] for non-preemptive scheduling and unit-length tasks. Hence problems $P\,|\,pmtn,\ in\text{-}tree\,|\,L_{max}$, $P2\,|\,pmtn,\ prec\,|\,L_{max}$ and $P2\,|\,pmtn,\ prec,\ r_j\,|\,L_{max}$ are solvable in polynomial time. Algorithms for these problems employ essentially the same techniques for dealing with precedence constraints as the corresponding algorithms for unit-length tasks. However, the algorithms are more complex and will not be presented here.

5.3.2 Uniform and Unrelated Processors

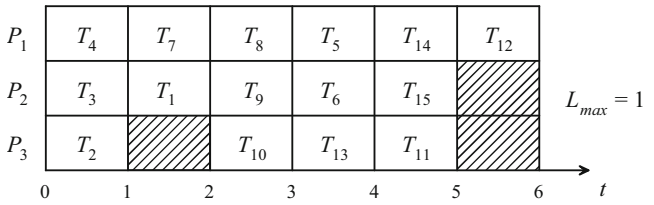
Problem $Q\,|\,|\,L_{max}$

From the considerations of Section 5.3.1 we see that non-preemptive scheduling to minimize L_{max} is in general a hard problem. Only for the problem $Q\,|\,p_j = 1\,|\,L_{max}$ a polynomial time optimization algorithm is known. This problem can be solved via a transportation problem formulation as in (5.1.16) - (5.1.19), where now $c_{ijk} = k/b_i - d_j$. Thus, from now on we will concentrate on preemptive scheduling.

(a)



(b)



(c)

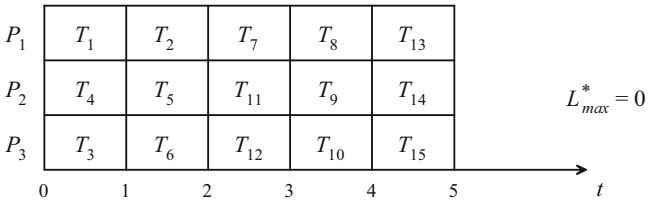


Figure 5.3.4 *Non-optimal schedules generated by Algorithm* 5.3.3 *for* m=3, n=15, *and all due dates* $d_j = 5$
(a) *a task set* (*all tasks are denoted by* T_j/d_j^*),
(b) *a schedule constructed by Algorithm* 5.3.3,
(c) *an optimal schedule.*

Problem $Q|pmtn|L_{max}$

One of the most interesting algorithms in that area has been presented for problem $Q|pmtn, r_j|L_{max}$ by Federgruen and Groenevelt [FG86]. It is a generalization of the network flow approach to the feasibility testing of problem $P|pmtn, r_j, \widetilde{d}_j|-$ described above. The feasibility testing procedure for problem $Q|pmtn, r_j, \widetilde{d}_j|-$ uses tripartite network formulation of the scheduling problem, where the first set of nodes corresponds to tasks, the second corresponds to processor-interval (period) combination and the third corresponds to interval nodes. The source is connected to each task node, the arc to the j^{th} node having capacity p_j, $j = 1, 2, \cdots, n$. A task node is connected to all processor-interval nodes for all intervals during which the task is available. All arcs leading to a processor-interval node that corresponds to a processor of type r (processors of the same speed may be represented by one node only) and an interval of length τ, have capacity $(b_r - b_{r+1})\tau$, with the convention $b_{m+1} = 0$. Every node (w_i, r) corresponding to processor type r and interval w_i of length τ_i, $i = 1, 2, \cdots, k$, is connected to interval node w_i and has capacity $\sum_{j=1}^{r} m_j(b_r - b_{r+1})\tau_i$, where m_j denotes the number of processors of the j^{th} type (cf. Figure 5.3.5). Finally, all interval nodes are connected to the sink with incapacitated arcs. Finding a feasible flow with value $\sum_{j=1}^{n} p_j$ in such a network corresponds to a construction of a feasible schedule for $Q|pmtn, r_j, \widetilde{d}_j|-$. This can be done in $O(mn^3)$ time.

Problem $Q|pmtn, prec|L_{max}$

In case of precedence constraints, $Q2|pmtn, prec|L_{max}$ and $Q2|pmtn, prec, r_j|L_{max}$ can be solved in $O(n^2)$ and $O(n^6)$ time, respectively, by the algorithms already mentioned [Law82b].

Problem $R|pmtn|L_{max}$

As far as unrelated processors are concerned, problem $R|pmtn|L_{max}$ can be solved by a linear programming formulation similar to (5.1.23) - (5.1.26) [LL78], where x_{ij}^{k} denotes the amount of T_j processed on P_i in time interval $[d_{k-1} + L_{max}, d_k + L_{max}]$, and where due dates are assumed to be ordered, $d_1 < d_2 < \cdots < d_n$. Thus, we have the following formulation:

$$\text{Minimize} \quad L_{max} \tag{5.3.1}$$

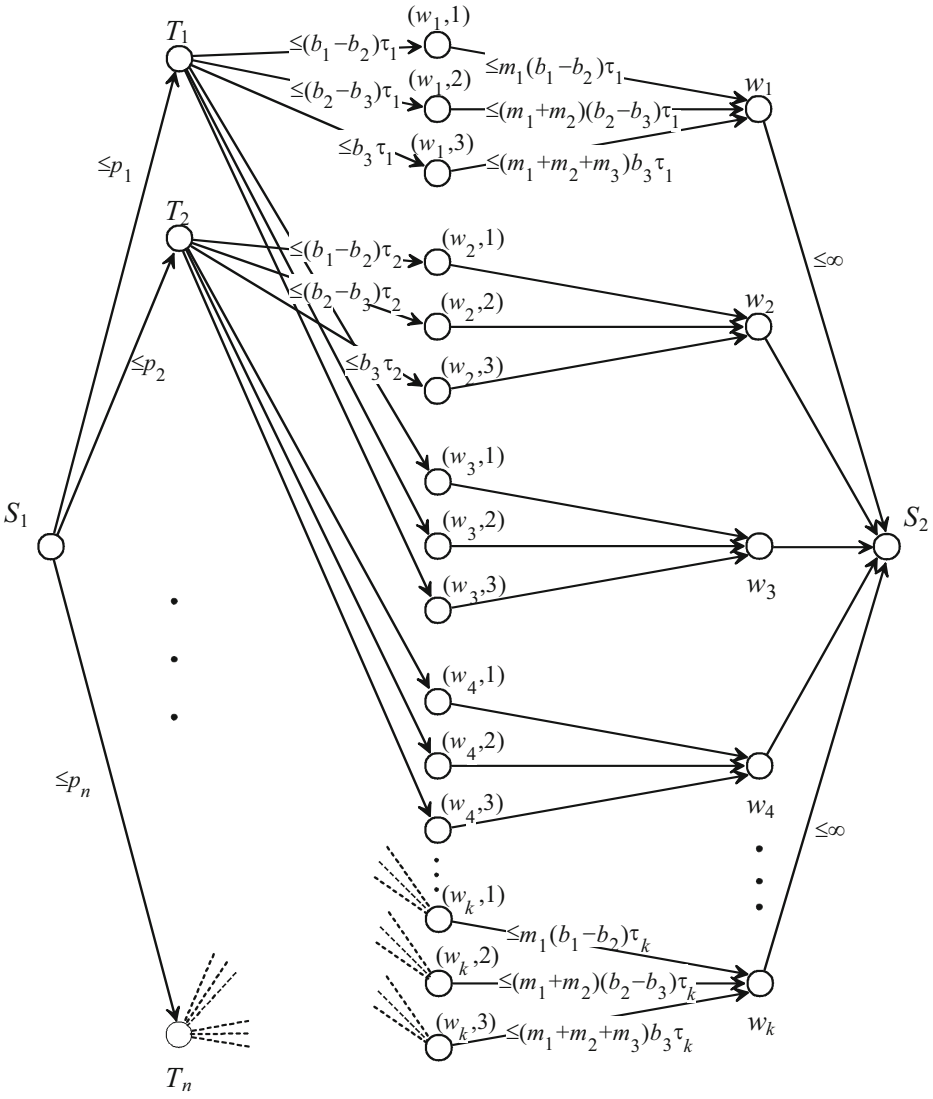


Figure 5.3.5 *A network corresponding to scheduling problem* $Q\,|\,pmtn, r_j, \widetilde{d}_j\,|-$ *for three processor types.*

$$\textit{subject to} \quad \sum_{i=1}^{m} p_{ij} x_{ij}^{(1)} \le d_1 + L_{max}, \quad j = 1, 2, \cdots, n \tag{5.3.2}$$

$$\sum_{i=1}^{m} p_{ij} x_{ij}^{(k)} \le d_k - d_{k-1}, \quad j = k, k+1, \cdots, n;\ k = 2, 3, \cdots, n \tag{5.3.3}$$

$$\sum_{j=1}^{n} p_{ij} x_{ij}^{(1)} \le d_1 + L_{max}, \quad i = 1, 2, \cdots, m \tag{5.3.4}$$

$$\sum_{j=k}^{n} p_{ij} x_{ij}^{(k)} \le d_k - d_{k-1}, \quad i = 1, 2, \cdots, m;\ k = 2, 3, \cdots, n \tag{5.3.5}$$

$$\sum_{i=1}^{m} \sum_{k=1}^{j} x_{ij}^{(k)} = 1 \qquad j = 1, 2, \cdots, n. \tag{5.3.6}$$

Solving the *LP* problem we obtain n matrices $\boldsymbol{T}^{(k)} = [t_{ij}^{(k)*}]$, $k = 1, \cdots, n$; then an optimal solution is constructed by an application of Algorithm 5.1.20 to each matrix separately.

In this context let us also mention that the case when precedence constraints form a uniconnected activity network (or interval order in a different presentation), can also be solved via the same modification of the *LP* problem as described for the C_{max} criterion [Slo81].

5.4 Lot Size Scheduling

In this section the more advanced model of lot size scheduling on parallel processors is presented. Consider the same problem as discussed in Section 4.4.2 but now instead of one processor there are m processors available for processing all tasks of all job types. Recall that the lot size scheduling problem can be solved in $O(H)$ time for one processor and two job types only, where H is the sum of tasks of the two given jobs. In the following we want to investigate the problem instance with two job types again but now allowing multiple identical processors. First we introduce some basic notation. Then the algorithm is presented without considering inventory restriction; later we show how to take these limitations into account.

Assume that m identical processors P_i, $i = 1$, ..., m are available for processing the set of jobs $\mathcal{J}$ which consist of two types only; due to capacity restrictions we want to assume that the final schedule is tight. Considering a number $m > 1$ of processors we must determine to which unit time interval (UTI) on which processors a job has to be assigned. Because of continuous production requirements we might also assume an assignment of UTI $h = 0$ to some job type; this can be interpreted as an assignment of some job type to the last UTI of the preceding schedule.

The idea of the algorithm is to assign task after task of the two job types, now denoted by q and r, to empty UTI such that all deadlines are met and no other assignment can reduce change-over cost. In order to do this we have to classify UTIs appropriately. Based on this classification we will present the algorithm. With respect to each deadline d_k we define a "sequence of empty UTI" (SEU) as a processing interval $[h^*, h^*+u-1]$ on some processor consisting of u consecutive and empty UTI. UTI h^*-1 is assigned to some job; UTI h^*+u is either also assigned to some job or it is the first UTI after the occurrence of the deadline. Each SEU can be described by a 3-tuple (i, h^*, u) where i is the number of the processor on which the SEU exists, h^* the first empty UTI and u the number of the UTI in this SEU.

We differentiate between "classes" of SEU by considering the job types assigned to neighboring UTI h^*-1 and h^*+u of each SEU. In case h^*+u has no assignment we denote this by "E"; all other assignments of UTI are denoted by the number of the corresponding job type. Now a "class" is denoted by a pair $[x, y]$ where $x, y \in \{q, r, E\}$. This leads to nine possible classes of SEU from which only classes $[q,r]$, $[q,E]$, $[r,q]$, and $[r,E]$, have to be considered.

Figure 5.4.1 illustrates these definitions using an example with an assignment for UTI $h = 0$. For $d_1 = 6$ we have a SEU (2,6,1) of class [1, E]; for $d_2 = 11$ we have (1, 9, 3) of class [1, E], (2, 6, 2) of class [1, 2], (2, 10, 2) of class [2, E].

For each d_k we have to schedule $n_{qk} \geq 0$ and $n_{rk} \geq 0$ tasks. We schedule the corresponding jobs according to non-decreasing deadlines with positive time orientation starting with $k = 1$ up to $k = K$ by applying the following algorithm.

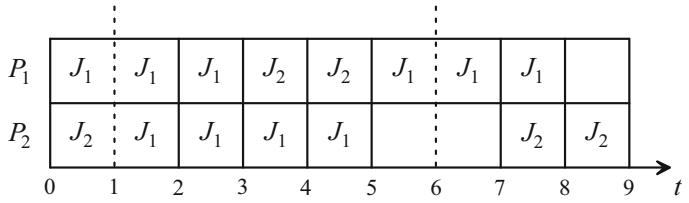


Figure 5.4.1 *Example schedule showing different SEU.*

Algorithm 5.4.1 *Lot size scheduling of two job types on identical processors* (*LIM*) [PS96].

begin
for $k := 1$ **to** K **do**
  while tasks required at $\tilde{d}_k$ are not finished **do**
    begin
    if class $[j,E]$ is not empty
    then Assign job type j to UTI h^* of a SEU (i, h^*, u) of class $[j,E]$ with minimum u

```
    else
      if classes [q,r] or [r,q] are not empty
      then Assign job type q(r) to UTI h* of a SEU (i,h*,u) of class
             [q,r] ([r,q]) or if this class is empty to UTI h*+u−1 of a
             SEU (i,h*,u) of class [r,q] ([q,r])
      else Assign job type q(r) to UTI h*+u−1 of a SEU (i,h*,u) of
             class [r,E] ([q,E]) with maximum u;
    Use new task assignment to calculate SEU of classes [r, E], [r, q], [q, r],
      and [q,E];
    end;
end;
```

In case the "**while**"-loop cannot be carried out no feasible schedule for the problem under consideration exists. It is necessary to update the classes after each iteration because after a task assignment the number u of consecutive and empty UTI of the concerned SEU decreases by one and thus the SEU might even disappear. Furthermore an assignment of UTI h^* or h^*+u-1 might force the SEU to change the class.

Let us demonstrate the approach by the following example. Let $m = 3$, $\mathcal{J} = \{J_1, J_2\}$, $\tilde{d}_1 = 4$, $\tilde{d}_2 = 8$, $\tilde{d}_3 = 11$, $n_{11} = 3$, $n_{12} = 7$, $n_{13} = 5$, $n_{21} = 5$, $n_{22} = 6$, $n_{23} = 7$ and zero initial inventory. Let us assume that there is a pre-assignment for $h = 0$ such that J_1 is processed by P_1 and J_2 is processed by P_2 and P_3. In Figure 5.4.2 the optimal schedule generated by Algorithm 5.4.1 is given.

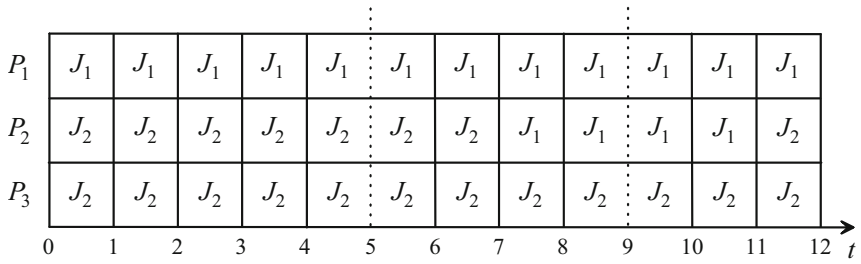


Figure 5.4.2 *Optimal schedule for the example problem.*

It can be shown that Algorithm 5.4.1 generates an optimal schedule if one exists. Feasibility of the algorithm is guaranteed by scheduling the job types according to earliest deadlines using only free UTI of the interval $[0, d_k]$. To prove optimality of the algorithm one has to show that the selection of the UTI for assigning the task under consideration is best possible. These facts have been proved in the following lemmas [PS96] which are formulated and proved for job type q, but they also hold in case of job type r.

Lemma 5.4.2 *There exists an optimal solution that can be built such that job type q is assigned to UTI h^* on processor P_i in case the selected SEU belongs to classes $[q, E]$ or $[q, r]$. If the SEU belongs to class $[r, E]$ or $[r, q]$ then q is assigned to UTI h^*+u-1 on processor P_i.* □

Lemma 5.4.3 *Algorithm* 5.4.1 *generates schedules with a minimum number of change-overs for two types of jobs.* □

The complexity of Algorithm 5.4.1 is $O(Hm)$.

Let us now investigate how we can consider inventory restrictions for both job types, i.e. for each job type an upper bound B_j on in-process inventory is given. If there are only two job types, limited in-process storage capacity can be translated to updated demands of unit time tasks referring to given deadlines d_k. If processing of some job type has to be stopped because of storage limitations, processing of the other job has to be started as $Hm = \sum_{j=1,\ldots,n} n_j$. This can be achieved by increasing the demand of the other job type, appropriately.

Assume that a demand and inventory feasible and tight schedule exists for the problem instance. Let N_{jk} be the updated demand after some preprocessing step now used as input for the algorithm. To define this input more precisely let us first consider how many unit time tasks of some job type, e.g. q, have to be processed up to some deadline d_k:

- at most the number of tasks of job type q which does not exceed storage limit, i.e. $L_q = B_q - \sum_{i=1,\ldots,k-1} (N_{qi} - n_{qi})$;
- at least the number of required tasks of job type q, i.e. $D_q = n_{qk} - \sum_{i=1,\ldots,k-1} (N_{qi} - n_{qi})$;
- at least the remaining processing capacity reduced by the number of tasks of job type r which can be processed feasibly. From this we get $R_q = c_k - \sum_{i=1,\ldots,k-1} (N_q^{\,i} + n_{qi}) - (B_r - \sum_{i=1,\ldots,k-1} (N_{ri} + n_{ri}))$, where $c_k = md_k$ is the total processing capacity in the intervals $[0, d_k]$ on m processors.

The same considerations hold respectively for the other job type r.

With the following lemmas we show how the demand has to be updated such that not only feasibility (Lemma 5.4.4) but also optimality (Lemma 5.4.6) concerning change-overs is retained. We start with showing that L_j can be omitted if we calculate N_{jk}.

Lemma 5.4.4 *In case that a feasible and tight schedule exists, $L_j = B_j - \sum_{i=1,\ldots,k-1} (N_{ji} - n_{ji})$ can be neglected.* □

From the result of Lemma 5.4.4 we can define N_{jk} more precisely by

$$N_{qk} := \max\{\ n_{qk} - \sum_{i=1,\ldots,k-1}(N_{qi} - n_{qi}),$$
$$c_k - \sum_{i=1,\ldots,k-1}(N_{qi} + N_{ri}) - (B_r - \sum_{i=1,\ldots,k-1}(N_{ri} - n_{ri}))\} \qquad (5.4.1)$$

$$N_{rk} := \max\{\ n_{rk} - \sum_{i=1,\ldots,k-1}(N_{ri} - n_{ri}),$$
$$c_k - \sum_{i=1,\ldots,k-1}(N_{ri} + N_q^{\,i}) - (B_q - \sum_{i=1,\ldots,k-1}(N_{qi} - n_{qi}))\} \qquad (5.4.2)$$

One may show [PS96] that after updating all demands of unit time jobs of type q according to (5.4.1) the new problem instance is equivalent to the original one. We omit the case of job type r and (5.4.2), which directly follows in an analogous way. Notice that the demand will only be updated, if inventory restrictions limit assignment possibilities up to a certain deadline d_k. Only in this case the k^{th} interval will be completely filled with jobs. If no inventory restrictions have to be considered equations (5.4.1) and (5.4.2) result in the original demand pattern.

Lemma 5.4.5 *After adapting N_{qk} according to* (5.4.1) *the feasibility of the solution according to the inventory constraints on r is guaranteed.* □

Lemma 5.4.6 *If*

(*i*) $n_{qk} - \sum_{i=1,\ldots,k-1}(N_{qi} - n_{qi}) \geq$
$c_k - \sum_{i=1,\ldots,k-1}(N_{qi} + N_{ri}) - (B_r - \sum_{i=1,\ldots,k-1}(N_{ri} - n_{ri})$

or

(*ii*) $n_{qk} - \sum_{i=1,\ldots,k-1}(N_{qi} - n_{qi}) <$
$c_k - \sum_{i=1,\ldots,k-1}(N_{qi} + N_{ri}) - (B_r - \sum_{i=1,\ldots,k-1}(N_{ri} - n_{ri})$

for some deadline d_k then a demand feasible and optimal schedule can be constructed. □

The presented algorithm also solves the corresponding problem instance with arbitrary positive change-over cost because for two job types only, minimizing the number of change-overs is equivalent to minimizing the sum of their positive change-over cost. In order to solve the practical gear-box manufacturing problem where more than two job types have to be considered a heuristic has been implemented which uses the ideas of the presented approach. The corresponding scheduling rule is considered to be that no unforced change-overs should occur. The resulting algorithm is part of a scheduling system, which incorporates a graphical representation scheme using Gantt-charts and further devices to give the manufacturing staff an effective tool for decision support. For more results on the implementation of scheduling systems on the shop floor we refer to Chapter 18.

References

AH73 D. Adolphson, T. C. Hu, Optimal linear ordering, *SIAM J. Appl. Math.* 25, 1973, 403-423.

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

Ash72 S. Ashour, *Sequencing Theory*, Springer, Berlin, 1972.

Bak74 K. Baker, *Introduction to Sequencing and Scheduling*, J. Wiley, New York, 1974.

BCS74 J. Bruno, E. G. Coffman, Jr., R. Sethi, Scheduling independent tasks to reduce mean finishing time, *Commun. ACM* 17, 1974, 382-387.

BCSW76a J. Błażewicz, W. Cellary, R. Słowiński, J. Węglarz, Deterministic problems of scheduling on parallel processors, Part I. Sets of independent jobs, *Podstawy Sterowania* 6, 1976, 155-178 (in Polish).

BCSW76b J. Błażewicz, W. Cellary, R. Słowiński, J. Węglarz, Deterministic problems of scheduling on parallel processors, Part II. Sets of dependent jobs, *Podstawy Sterowania* 6, 1976, 297-320 (in Polish).

BCW77 J. Błażewicz, W. Cellary, J. Węglarz, A strategy for scheduling splittable tasks to reduce schedule length, *Acta Cybernetica* 3, 1977, 99-106.

BGJ77 P. Brucker, M. R. Garey, D. S. Johnson, Scheduling equal-length tasks under treelike precedence constraints to minimize maximum lateness, *Math. Oper. Res.* 2, 1977, 275-284.

BK00 J. Błażewicz, D. Kobler, On the ties between different graph representation for scheduling problems, Report, Poznan University of Technology, Poznan, 2000.

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

Bla77 J. Błażewicz, Simple algorithms for multiprocessor scheduling to meet deadlines, *Inf. Process. Lett.* 6, 1977, 162-164.

Bru76a J. Bruno, Scheduling algorithms for minimizing the mean weighted flow-time, in: E. G. Coffman, Jr. (ed.), *Computer and Job-Shop Scheduling Theory*, J. Wiley, New York, 1976.

Bru76b P. J. Brucker, Sequencing unit-time jobs with treelike precedence on m processors to minimize maximum lateness, *Proceedings of the IX. International Symposium on Mathematical Programming*, Budapest, 1976.

BT94 B. Braschi, D. Trystram, A new insight into the Coffman-Graham algorithm, *SIAM J. Comput*. 23, 1994, 662-669.

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

CFL83 E. G. Coffman, Jr., G. N. Frederickson, G. S. Lueker, Probabilistic analysis of the LPT processor scheduling heuristic, unpublished paper, 1983.

CFL84 E. G. Coffman, Jr., G. N. Frederickson, G. S. Lueker, A note on expected makespans for largest-first sequences of independent task on two processors, *Math. Oper. Res.* 9, 1984, 260-266.

CG72 E. G. Coffman, Jr., R. L. Graham, Optimal scheduling for two-processor systems, *Acta Inform.* 1, 1972, 200-213.

CG91 E. G. Coffman, Jr., M. R. Garey, Proof of the 4/3 conjecture for preemptive versus nonpreemptive two-processor scheduling, Report, Bell Laboratories, Murray Hill, 1991.

CGJ78 E. G. Coffman, Jr., M. R. Garey, D. S. Johnson, An application of bin-packing to multiprocessor scheduling, *SIAM J. Comput.* 7, 1978, 1-17.

CGJ84 E. G. Coffman, Jr., M. R. Garey, D. S. Johnson, Approximation algorithms for bin packing - an updated survey, in: G. Ausiello, M. Lucertini, P. Serafini (eds.), *Algorithm Design for Computer System Design*, Springer, Vienna, 1984, 49-106.

CL75 N.-F. Chen, C. L. Liu, On a class of scheduling algorithms for multiprocessor computing systems, in: T.-Y. Feng (ed.), Parallel Processing, *Lect. Notes Comput. Sc.* 24, 1975, 1-16.

CMM67 R. W. Conway, W. L. Maxwell, L. W. Miller, *Theory of Scheduling*, Addison-Wesley, Reading, Mass., 1967.

Cof73 E. G. Coffman, Jr., A survey of mathematical results in flow-time scheduling for computer systems, *GI - 3. Jahrestagung, Hamburg*, Springer, Berlin, 1973, 25-46.

Cof76 E. G. Coffman, Jr. (ed.), *Scheduling in Computer and Job Shop Systems*, J. Wiley, New York, 1976.

CS76 E. G. Coffman, Jr., R. Sethi, A generalized bound on LPT sequencing, *RAIRO-Informatique* 10, 1976, 17-25.

DL88 J. Du, J. Y-T. Leung, Scheduling tree-structured tasks with restricted execution times, *Inf. Process. Lett.* 28, 1988, 183-188.

DL89 J. Du, J. Y-T. Leung, Scheduling tree-structured tasks on two processors to minimize schedule length, *SIAM Discret Math.* 2, 1989, 176-196.

DLY91 J. Du, J. Y-T. Leung, G. H. Young, Scheduling chain structured tasks to minimize makespan and mean flow time, *Inform. Comput.* 92, 1991, 219-236.

DW85 D. Dolev, M. K. Warmuth, Scheduling flat graphs, *SIAM J. Comput.* 14, 1985, 638-657.

EH93 K. H. Ecker, R. Hirschberg, Task scheduling with restricted preemptions, in: A. Bode, M. Reeve, G. Wolf (eds.), Proceedings of PARLE93 - Parallel Architectures and Languages, *Lect. Notes Comput. Sc.* 694, 1993, 464-475.

FB73 E. B. Fernandez, B. Bussel, Bounds on the number of processors and time for multiprocessor optimal schedules, *IEEE Trans. Comput.* 22, 1973, 745-751.

FG86 A. Federgruen, H. Groenevelt, Preemptive scheduling of uniform processors by ordinary network flow techniques, *Manage. Sci.* 32, 1986, 341-349.

FKN69 M. Fujii, T. Kasami, K. Ninomiya, Optimal sequencing of two equivalent processors, *SIAM J. Appl. Math.* 17, 1969, 784-789 (Erratum: *SIAM J. Appl. Math.* 20, 1971, 141).

Fre82 S. French, *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Horwood, Chichester, 1982.

FRK86 J. B. G. Frenk, A. H. G. Rinnooy Kan, The rate of convergence to optimality of the LPT rule, *Discret Appl. Math.* 14, 1986, 187-197.

FRK87 J. B. G. Frenk, A. H. G. Rinnooy Kan, The asymptotic optimality of the LPT rule, *Math. Oper. Res.* 12, 1987, 241-254.

Gab82 H. N. Gabow, An almost linear algorithm for two-processor scheduling, *J. ACM* 29, 1982, 766-780.

Gar - M. R. Garey, Unpublished result.

Gar73 M. R. Garey, Optimal task sequencing with precedence constraints, *Discrete Math.* 4, 1973, 37-56.

GG73 M. R. Garey, R. L. Graham, Bounds on scheduling with limited resources, *ACM SIGOPS Operating Systems Review,* 1973, 104-111.

GG75 M. R. Garey, R. L. Graham, Bounds for multiprocessor scheduling with resource constraints, *SIAM J. Comput.* 4, 1975, 187-200.

GIS77 T. Gonzalez, O. H. Ibarra, S. Sahni, Bounds for LPT schedules on uniform processors, *SIAM J. Comput.* 6, 1977, 155-166.

GJ76 M. R. Garey, D. S. Johnson, Scheduling tasks with nonuniform deadlines on two processors, *J. ACM* 23, 1976, 461-467.

GJ77 M. R. Garey, D. S. Johnson, Two-processor scheduling with start-times and deadlines, *SIAM J. Comput.* 6, 1977, 416-426.

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

GJST81 M. R. Garey, D. S. Johnson, B. B. Simons, R. E. Tarjan, Scheduling unit time tasks with arbitrary release times and deadlines, *SIAM J. Comput.* 10, 1981, 256-269.

GJTY83 M. R. Garey, D. S. Johnson, R. E. Tarjan, M. Yannakakis, Scheduling opposing forests, *SIAM J. Algebra. Discr.* 4, 1983, 72-93.

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.

Gon77 T. Gonzalez, Optimal mean finish time preemptive schedules, Technical report 220, Computer Science Department, Pennsylvania State University, 1977.

Gra66 R. L. Graham, Bounds for certain multiprocessing anomalies, *Bell Labs Tech. J.* 45, 1966, 1563-1581.

Gra69 R. L. Graham, Bounds on multiprocessing timing anomalies, *SIAM J. Appl. Math.* 17, 1969, 416-429.

Gra76 R. L. Graham, Bounds on performance of scheduling algorithms, Chapter 5 in: E. G. Coffman, Jr. (ed.), *Scheduling in Computer and Job Shop Systems*, J. Wiley, New York, 1976.

GS78 T. Gonzalez, S. Sahni, Preemptive scheduling of uniform processor systems, *J. ACM* 25, 1978, 92-101.

HLS77 E. G. Horvath, S. Lam, R. Sethi, A level algorithm for preemptive scheduling, *J. ACM* 24, 1977, 32-43.

Hor73 W. A. Horn, Minimizing average flow time with parallel processors, *Oper. Res.* 21, 1973, 846-847.

Hor74 W. A. Horn, Some simple scheduling algorithms, *Nav. Res. Logist. Quart.* 21, 1974, 177-185.

HS76 E. Horowitz, S. Sahni, Exact and approximate algorithms for scheduling non-identical processors, *J. ACM* 23, 1976, 317-327.

HS87 D. S. Hochbaum, D. B. Shmoys, Using dual approximation algorithms for scheduling problems: theoretical and practical results, *J. ACM* 34, 1987, 144-162.

Hu61 T. C. Hu, Parallel sequencing and assembly line problems, *Oper. Res.* 9, 1961, 841-848.

IK77 O. H. Ibarra, C. E. Kim, Heuristic algorithms for scheduling independent tasks on nonidentical processors, *J. ACM* 24, 1977, 280-289.

Jac55 J. R. Jackson, Scheduling a production line to minimize maximum tardiness, Research report 43, Management Research Project, University of California, Los Angeles, 1955.

JMR+04 J. Jozefowska, M. Mika, R. Rozycki, G. Waligora, J. Weglarz, An almost optimal heurisitc for preemptive C_{max} scheduling of dependent tasks on parallel identical machines, *Ann. Oper. Res.* 129, 2004, 205-216.

Joh83 D. S. Johnson, The NP-completeness column: an ongoing guide, *J. Algorithms* 4, 1983, 189-203.

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.

Kar74 A. W. Karzanov, Determining the maximal flow in a network by the method of preflows, *Dokl. Akad. Nauk. SSSR* 215, 1974, 434-437 (in Russian).

Kar84 N. Karmarkar, A new polynomial-time algorithm for linear programming, *Combinatorica* 4, 1984, 373-395.

KE75 O. Kariv, S. Even. An $O(n^{2.5})$ algorithm for maximum matching in general graphs, *Proceedings of the 16th Annual Symposium on Foundations of Computer Science,* 1975, 100-112.

Ked70 S. K. Kedia, A job scheduling problem with parallel processors, Unpublished report, Department of Industrial Engineering, University of Michigan, Ann Arbor, 1970.

Kha79 L. G. Khachiyan, A polynomial algorithm for linear programming, *Dokl. Akad. Nauk SSSR*, 244, 1979, 1093-1096 (in Russian).

KK82 N. Karmarkar, R. M. Karp, The differencing method of set partitioning, Report UCB/CSD 82/113, Computer Science Division, University of California, Berkeley, 1982.

Kun76 M. Kunde, Beste Schranke beim LP-Scheduling, Bericht 7603, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1976.

Law73 E. L. Lawler, Optimal sequencing of a single processor subject to precedence constraints, *Manage. Sci.* 19, 1973, 544-546.

Law76 E. L. Lawler, *Combinatorial optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

Law82a E. L. Lawler, Recent results in the theory of processor scheduling, in: A. Bachem, M. Grötschel, B. Korte (eds.) *Mathematical Programming: The State of Art*, Springer, Berlin, 1982, 202-234.

Law82b E. L. Lawler, Preemptive scheduling in precedence-constrained jobs on parallel processors, in: M. A. H. Dempster, J. K. Lenstra, A. H. G. Rinnooy Kan (eds.), *Deterministic and Stochastic Scheduling*, Reidel, Dordrecht, 1982, 101-123.

Lee91 C.-Y. Lee, Parallel processor scheduling with nonsimultaneous processor available time, *Discret Appl. Math.* 30, 1991, 53-61.

Len77 J. K. Lenstra, *Sequencing by Enumerative Methods*, Mathematical Centre Tract 69, Mathematisch Centrum, Amsterdam, 1977.

LL74a J. W. S. Liu, C. L. Liu, Performance analysis of heterogeneous multiprocessor computing systems, in: E. Gelenbe, R. Mahl (eds.), *Computer Architecture and Networks*, North Holland, Amsterdam, 1974, 331-343.

LL74b J. W. S. Liu, C. L. Liu, Bounds on scheduling algorithms for heterogeneous computing systems, Technical report UIUCDCS-R-74-632, Department of Computer Science, University of Illinois at Urbana-Champaign, 1974.

LL78 E. L. Lawler, J. Labetoulle, Preemptive scheduling of unrelated parallel processors by linear programming, *J. ACM* 25, 1978, 612-619.

LLL+84 J. Labetoulle, E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, Preemptive scheduling of uniform processors subject to release dates, in: W. R. Pulleyblank (ed.), *Progress in Combinatorial Optimization*, Academic Press, New York, 1984, 245-261.

LLRK82 E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, Recent developments in deterministic sequencing and scheduling: a survey, in: M. A. H. Dempster, J. K. Lenstra, A. H. G. Rinnooy Kan (eds.), *Deterministic and Stochastic Scheduling*, Reidel, Dordrecht, 1982, 35-73.

LLR+93 E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, D. B. Shmoys, Sequencing and scheduling: Algorithms and complexity, in: S. C. Graves, A. H. G. Rinnooy Kan, P. H. Zipkin (eds.), *Handbook in Operations Research and Management Science, Vol. 4: Logistics of Production and Inventory*, Elsevier, Amsterdam, 1993.

LRK78 J. K. Lenstra, A. H. G. Rinnooy Kan, Complexity of scheduling under precedence constraints, *Oper. Res.* 26, 1978, 22-35.

LRK84 J. K. Lenstra, A. H. G. Rinnooy Kan, Scheduling theory since 1981: an annotated bibliography, in: M. O'h Eigearthaigh, J. K. Lenstra, A. H. G. Rinnooy Kan (eds.), *Combinatorial Optimization: Annotated Bibliographies,* J. Wiley, Chichester, 1984.

LRKB77 J. K. Lenstra, A. H. G. Rinnooy Kan, P. Brucker, Complexity of processor scheduling problems, *Annals of Discrete Mathematics* 1, 1977, 343-362.

LS77 S. Lam, R. Sethi, Worst case analysis of two scheduling algorithms, *SIAM J. Comput.* 6, 1977, 518-536.

MC69 R. Muntz, E. G. Coffman, Jr., Optimal preemptive scheduling on two-processor systems, *IEEE Trans. Comput.* 18, 1969, 1014-1029.

MC70 R. Muntz, E. G. Coffman, Jr., Preemptive scheduling of real time tasks on multiprocessor systems, *J. ACM* 17, 1970, 324-338.

McN59 R. McNaughton, Scheduling with deadlines and loss functions, *Manage. Sci.* 6, 1959, 1-12.

NLH81 K. Nakajima, J. Y-T. Leung, S. L. Hakimi, Optimal two processor scheduling of tree precedence constrained tasks with two execution times, *Performance Evaluation* 1, 1981, 320-330.

PS96 M. Pattloch, G. Schmidt, Lotsize scheduling of two job types on identical processors, *Discret Appl. Math*., 1996, 409-419.

Rin78 A. H. G. Rinnooy Kan, *Processor Scheduling Problems: Classification, Complexity and Computations*, Nijhoff, The Hague, 1978.

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.

Ros– P. Rosenfeld, unpublished result.

Rot66 M. H. Rothkopf, Scheduling independent tasks on parallel processors, *Manage. Sci.* 12, 1966, 347-447.

RS83 H. Röck, G. Schmidt, Processor aggregation heuristics in shop scheduling, *Methods of Operations Research* 45, 1983, 303-314.

Sah79 S. Sahni, Preemptive scheduling with due dates, *Oper. Res.* 5, 1979, 925-934.

SC80 S. Sahni, Y. Cho, Scheduling independent tasks with due times on a uniform processor system, *J. ACM* 27, 1980, 550-563.

Sch84 G. Schmidt, Scheduling on semi-identical processors, *Zeitschrift für OR* A28, 1984, 153-162.

Sch88 G. Schmidt, Scheduling independent tasks with deadlines on semi-identical processors, *J. Oper. Res. Soc.* 39, 1988, 271-277.

Set76 R. Sethi, Algorithms for minimal-length schedules, Chapter 2 in: E. G. Coffman, Jr. (ed.), *Scheduling in Computer and Job Shop Systems*, J. Wiley, New York, 1976.

Set77 R. Sethi, On the complexity of mean flow time scheduling, *Math. Oper. Res.* 2, 1977, 320-330.

Sev91 S. V. Sevastjanov, Private communication, 1991.

Sit05 R. Sitters, Complexity of preemptive minsum scheduling on unrelated parallel machines, *J. Algorithms* 57, 2005, 37-48.

Slo78 R. Słowiński, Scheduling preemptible tasks on unrelated processors with additional resources to minimise schedule length, in: G. Bracci, R. C. Lockemann (eds.), Information Systems Methodology, *Lect. Notes Comput. Sc.* 65, 1978, 536-547.

SW77 R. Słowiński, J. Węglarz, Time-minimal network model with different modes of the execution of activities, *Przeglad Statystyczny* 24, 1977, 409-416 (in Polish).

Ull76 J. D. Ullman, Complexity of sequencing problems, Chapter 4 in: E. G. Coffman, Jr. (ed.), *Scheduling in Computer and Job Shop Systems*, J. Wiley, New York, 1976.

WBCS77 J. Węglarz, J. Błażewicz, W. Cellary, R. Słowiński, An automatic revised simplex method for constrained resource network scheduling, *ACM Trans. Math. Softw.* 3, 1977, 295-300.

Wer84 D. de Werra, Preemptive scheduling, linear programming and network flows, *SIAM J. Algebra. Discr.* 5, 1984, 11-20.