Contents

10
Minmax criteria with preemption

Eugene L. Lawler
University of California, Berkeley

Charles U. Martel
University of California, Davis

Good news! This chapter has no NP-hardness results to limit us, and hence no complicated approximation algorithms to consider. There are only polynomial algorithms to appreciate. In contrast to the nonpreemptive case where even $P2||C_{\max}$ is NP-hard, we will be able to solve preemptive settings even for unrelated machines with release times and due-dates.

We begin with a simple linear time algorithm for $P|pmtn|C_{\max}$ and then present the more complex algorithm of Gonzalez and Sahni that solves $Q|pmtn|C_{\max}$ in essentially $O(n)$ time. We exploit the insights gained from this algorithm to derive efficient algorithms for $Q|pmtn|L_{\max}$, $Q|pmtn, r_j|C_{\max}$, $Q|pmtn, r_j|L_{\max}$ and $Q|pmtn|C_{\max}$ when processors have memory capacities and jobs have memory requirements. We conclude by showing that the problems $R|pmtn|C_{\max}$, $R|pmtn|L_{\max}$, $R|pmtn, r_j|C_{\max}$ and $R|pmtn, r_j|L_{\max}$ can be solved by linear programming.

10.1. Minimizing Makespan

McNaughton's 1959 solution of the problem $P|pmtn|C_{\max}$ is probably the simplest and earliest instance of an approach that has been successfully applied to other preemptive scheduling problems: First provide an obvious lower bound on the cost of an optimal solution and then construct a schedule that meets this bound.

j	1	2	3	4	5	6	7	8
p_j	5	8	6	2	3	1	3	4

Table 10.1 Sample Data for McNaughton's Algorithm

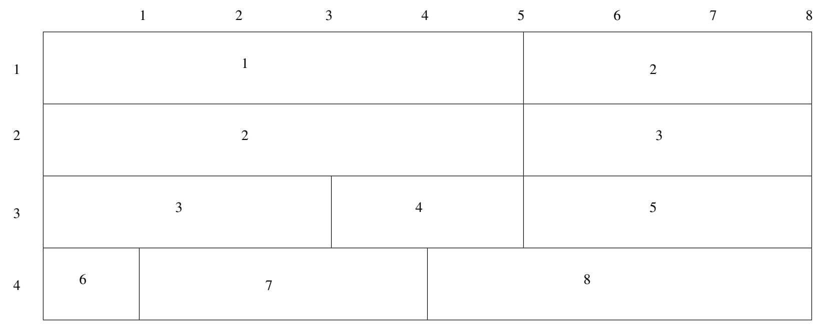

Figure 10.1. Schedule obtained by McNaughton's algorithm.

In the case of $P|pmtn|C_{\max}$, let $p_{\max}$ be the largest p_j value. We see that $C_{\max}$ must be at least

$$\max\{p_{\max}, (\sum_{j=1}^{n} p_j)/m\}. \tag{10.1}$$

A schedule meeting this bound can be constructed in $O(n)$ time: Schedule the jobs one at a time, in arbitrary order, filling up the available time on each successive machine before proceeding to the next, splitting the processing of a job whenever the above time bound on a given machine is met.

For example, consider the problem data in table 10.1:

Suppose there are four machines. Then the time bound given by (10.1) is

$C_{\max} \geq \max\{8, 32/4\} = 8$.

A schedule meeting this time bound is shown in Figure 10.1.

The number of preemptions occuring in a schedule constructed by McNaughton's algorithm is at most m-1, and it is possible to construct a class of problem instances for which an optimal schedule has at least this many preemptions. It is not hard to see that the problem of minimizing the number of preemptions is NP-hard. (Observe that our numerical example can be solved without preemptions.)

10.1.1. Uniform Machines: The Gonzalez-Sahni Algorithm

In the case of $Q|pmtn|C_{\max}$, generalizing the bound (10.1) shows that the length of an optimal schedule must be at least

$$\max\{\max(1 \leq k \leq m-1),\ \{(\sum_{j=1}^{k} p_j)/\sum_{i=1}^{k} s_i\}, (\sum_{j=1}^{n} p_j)/\sum_{i=1}^{m} s_i\}. \tag{10.2}$$

where $p_1 \geq \ldots \geq p_n$, and $s_1 \geq \ldots \geq s_m$. An algorithm of Gonzalez and Sahni enables us to construct a schedule meeting this bound. Their algorithm requires only $O(n)$ time if the jobs are given in in order of nonincreasing p_j and the machines in order of nonincreasing s_i; without this assumption, the running time is $O(n + mlogm)$. The algorithm yields an optimal schedule with at most $2(m-1)$ preemptions, which is a tight bound.

To explain the algorithm, we find it convenient to generalize the usual definition of uniform parallel machines so as to allow the speeds of the machines to be time varying. Let $s_i(t)$ denote the speed of machine i at time t and assume that $s_1(t) \geq \ldots \geq s_m(t)$, for all t. The functions $s_i(t)$ may be discontinuous, but are integrable. The *processing capacity* of machine i in the time interval $[t,t']$ is then

$$S_i(t,t') = \int_t^{t'} s_i(u)du. \tag{10.3}$$

In order for a job to be completed, it is necessary that the sum of the processing capacities in the time intervals in which the job is processed should equal its processing requirement. For example, if job j is processed on machine 1 in the interval $[t_1,t_1']$ and on machine 2 in the interval $[t_2,t_2']$, then this processing is sufficient to complete the job if $S_1(t_1,t_1') + S_2(t_2,t_2') = p_j$.

Let $S_i, i = 1,\ldots,m$, denote the processing capacity of machine i in the interval $[0,T]$. By a generalization of (10.2), we see that for there to exist a feasible preemptive schedule in the interval [0,T] it is necessary that

$$\begin{array}{ll} S_1 & \geq p_1 \\ S_1 + S_2 & \geq p_1 + p_2 \\ \\ . & . \\ \\ S_1 + S_2 + \ldots + S_{m-1} & \geq p_1 + p_2 + \ldots + p_{m-1} \\ S_1 + S_2 + \ldots + S_m & \geq p_1 + p_2 + \ldots + p_n. \end{array} \tag{10.4}$$

Let T be the smallest value for which the inequalities (10.4) are satisfied. We shall construct a feasible schedule in the interval $[0,T]$ by scheduling the jobs one at a time, in arbitrary order. For each successive job j, we first find the machine with largest index k such that its (remaining) processing capacity $S_k \geq p_j$ and then consider three cases:

Case 1 $S_k = p_j$, where $k \leq m-1$. In this case we schedule job j to be processed by machine k for the entire period $[0,T]$. We then eliminate machine k and job j from the problem, leaving a problem with $m-1$ machines and $n-1$ jobs for which the inequalities (10.4) are again satisfied.

Case 2 $S_k > p_j > S_k + 1$, where $k \leq m-1$. We assert that there exists a time $t, 0 < t < T$, such that
$p_j = \int_0^t s_k(u)du + \int_t^T s_{k+1}(u)du.$

To convince ourselves of this fact, we need only plot the curves of the continuous functions
$f(t) = \int_0^t s_k(u)du$, $g(t) = \int_t^T s_{k+1}(u)du$, and $f(t)+g(t)$.

We propose to schedule job j for processing on machine k in the interval $[0,t]$ and on machine $k+1$ in the interval $[t,T]$. We then create a *composite* machine from the remaining available time on machines k and $k+1$. The capacity of this composite machine in the interval $[0,T]$ is

$$S_k + S_{k+1} - p_j = \int_0^t s_{k+1}(u)du + \int_t^T s_k(u)du \qquad (10.5)$$

We then replace machines k and $k+1$ with this new composite machine, leaving us with a problem with $m-1$ machines and $n-1$ jobs, for which the inequalities (10.4) are again satisfied.

Case 3 $S_m \geq p_j$. In this case we schedule job j on machine m, thereby reducing its capacity to $S_m - p_j$. This leaves a problem with m machines and $n-1$ jobs for which the inequalities (10.4) are again satisfied.

It is now a simple matter to prove, by induction on the number of jobs, that inequalities (10.4) imply the existence of a feasible schedule. In other words, satisfaction of inequalities (10.4) is both necessary and sufficient for feasibility. We note that in fact the above algorithm has the strong property that after scheduling any job j *each* of the sums $S_1, S_1+S_2, \ldots, S_1+\ldots+S_m$ is as large as possible for any legal scheduling of job j on the set of processors which exist when j is scheduled.

To illustrate the algorithm, consider a problem with jobs of $p_j = 28, 26, 16, 12, 10$. The total processing time of all jobs is 92. Thus if we were to schedule these jobs on machines of speeds 10, 8, 4 and 1, the value of equation (10.2) is the maximum of $26/10, 54/18, 70/22, 92/23$ which is $92/23 = 4$. The scheduling of job one (using Case 2 of the above algorithm) is shown in Figure 10.2 and the final schedule is shown in Figure 10.3.

10.1.2. Implementation of the algorithm

The idea of machines with time-varying speeds was introduced only to make it easy to describe the processing capacities of composite machines. At the beginning of the computation, before any composite machines are formed, all machines have constant speeds. In order to compute the minimum value of $C_{\max}$ by (10.2), we need only to determine the m largest p_j values and to order them. If the p_j values are not already sorted, we can find the m largest in $O(n)$ time by applying a selection algorithm, and

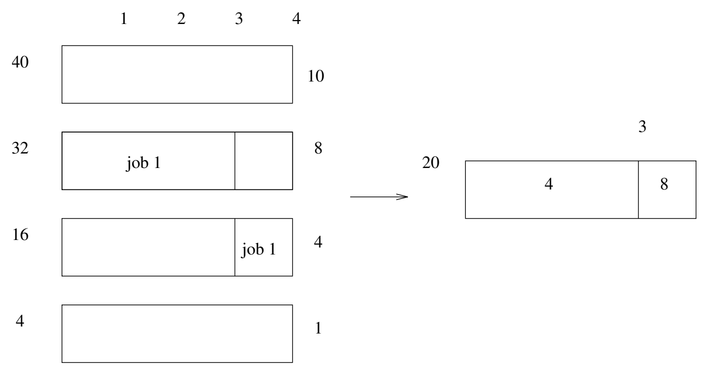

Figure 10.2. Scheduling job 1 using Gonzalex-Sahni algorithm.

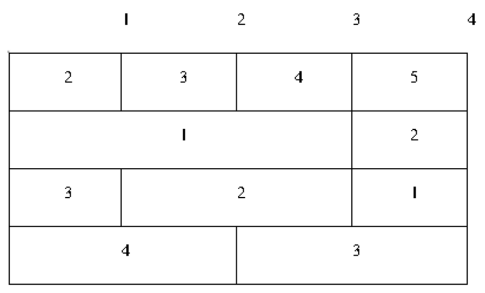

Figure 10.3. Final schedule for $p_j = 28, 26, 16, 12, 10$.

then sort the m largest values in $O(mlogm)$ time. Likewise, if the machine speeds are not already sorted, $O(m \log m)$ time suffices to sort the s_i values.

We propose to maintain the (composite) machine capacities in a linked list $(S_1, ..., S_m)$. This means that each time a job is scheduled, only constant time is required to revise the list. Suppose, instead of scheduling the jobs in arbitrary order as described above, we schedule the jobs in nonincreasing order of processing time. Let $k(j)$ denote the largest value of k such that $S_k \geq p_j$, with reference to the processing capacities existing at the time job j is scheduled. Notice that, in all cases, $k(j) \geq k(j-1)-1$. It follows that, by recording the value of $k(j-1)$ at the end of iteration $j-1$ for use in iteration j, only $O(n+m) = O(n)$ time is required to scan the list of machine capacities over the course of n iterations. Moreover, after $m-1$ jobs have been scheduled, either only one composite machine remains, or Case 3 has occured. Once Case 3 occurs, the only machine capacities that must be checked are the last two capacities in the list.

It follows from the above analysis that $O(n+mlogm)$ time suffices for all operations required by the algorithm, except for the time required to actually schedule the jobs on the composite machines.

We propose to represent each composite machine by a doubly-linked list of triples $(i, [t_i, t_i'])$, where each triple gives the index of an elementary machine of speed s_i and each time interval $[t_i, t_i']$ indicates a time interval of processing on that machine. By scanning the linked list for composite machine $k+1$ from the end and k from the front in parallel (advancing by one interval on each machine until we find a pair of intervals which overlap), it is easy to determine the time t in Case 2. We can thus assure that we take $O(r+1)$ time to find the intervals used for job j where r is the number of elementary machine intervals which are completely used up by job j. It is not hard to verify that at most $O(n+m) = O(n)$ time is required for list scanning overall by the algorithm. Thus the total time for the algorithm is $O(n+m \log m)$. The number of preemptions introduced is discussed in the exercises.

The Gonzalez-Sahni algorithm is a an important building block for algorithms on uniform machines. In the next four sections we will use this algorithm as a subroutine to solve more complex uniform machine settings.

Exercises

10.1. Construct a class of $P|pmtn|C_{\max}$ problem instances for which any optimal schedule has at least $m-1$ preemptions.

10.2. Construct a class of $Q|pmtn|C_{\max}$ problem instances for which any optimal schedule has at least $2(m-1)$ preemptions.

10.3. Suppose each machine is available for processing in only certain specified time intervals. Describe how you would apply the Gonzalez-Sahni algorithm, subject to constraints of this type.

10.4. Show that, if jobs are scheduled in arbitary order, the Gonzalez-Sahni algorithm can be implemented to run in $O(m^2+n)$ time.

10.5. Suppose that k time units of idle time must elapse between the time when a job is preempted and the time when its processing is resumed on another machine.

(a) Prove that imposing this condition on identical machines increases $C^*_{\max}$ by at most $k-1$.

(b) Show that, for $k = 1$, the $C_{\max}$ problem is solvable in polynomial time by a simple adaptation of McNaughton's rule. (For any fixed $k \geq 2$, the problem is NP-hard.)

10.6. Prove a bound on the on the maximum number of $(i,[t_i,t'_i])$ triples which will be used to represent the composite processors.
10.7. Use the result of problem 6 to show that the algorithm uses at most 2(m-1) preemptions.

10.2. Meeting Deadlines

The Gonzalez-Sahni algorithm gives us the insight necessary to develop an algorithm for solving the feasibility problem $Q|pmtn,\overline{d}_j|--$. We shall schedule the jobs in deadline order, so assume they are indexed with $\overline{d}_1 \leq \overline{d}_2 \leq \ldots \leq \overline{d}_n$. Let $S_i^{(j-1)}$ denote the capacity of (composite) machine i in the time interval $[0,\overline{d}_{j-1}]$ after jobs $1,2,\ldots j-1$ have been scheduled. The capacities of the composite machines available for the processing of job j will then be $S_i^{(j-1)} + s_i(\bar{d}_j - \overline{d}_{j-1})$, for $i = 1,...,m$. The Gonzalez-Sahni (G-S) algorithm tells us that whatever the capacities of the composite machines, it is optimal to schedule job j on the composite machines as determined by Cases 1-3 in the previous section (that is, this schedule maximizes each of the partial sums of the remaining machine capacities).

Concerning ourselves only with the composite machine capacities, and not with the actual scheduling of the jobs, the algorithm of Sahni and Cho is as follows:

```
Algorithm  *Sahni-Cho*
**for** $i = 1,...,m$;
  $S_i := 0$;
**for** $j = 1,...,n$;
  **for** $i = 1,...,m$;
    $S_i := S_i + s_i\ (\overline{d}_j - \overline{d}_{j-1})$;
Schedule job $j$ using G-S algorithm. Let $k$ be the largest index with $S_k \geq p_j$.
  Case ($S_k = p_j$):          * schedule the job on composite machine $k$ from zero to $d_j$*
    **for** $i = k,..,m-1$
      $S_i := S_{i+1}$;
      $S_m := 0$;
  Case ($S_k > p_j > S_{k+1}$, and $k < m$):  * schedule on composite machines $k$,   $k+1$ *
    $S_k := S_k + S_{k+1} - p_j$;
    **for** $i = k+1,...,m-1$
      $S_i := S_{i+1}$;
    $S_m := 0$;
  Case ($S_m \geq p_j$)                        * schedule the job on composite machine $m$ *
```

$S_m := S_m - p_j$;

The Sahni-Cho algorithm can be implemented to run in $O(mn + nlogn)$ time. The schedule it creates has at most $O(mn)$ preemptions, and there are settings which require $\Omega(mn)$ preemptions.

10.2.1. Specialization to identical machines

Interestingly, the specialization of the Sahni-Cho algorithm to $P|pmtn, \overline{d}_j| - -$ is a good deal more efficient.

At each successive deadline $\overline{d}_j$, each (elementary) machine i has been scheduled for the continuous processing of jobs in the interval $[0, a_i]$. We will now show that we can schedule each job using at most one preemption and maintaining the above property of continuous processing.

Suppose we maintain a nearly balanced tree of pairs (i, a_i), sorted by a_i value. Then, in $O(\log m)$ time, we will be able to locate the largest a_i such that $\overline{d}_j - a_i \geq p_j$. We now schedule as follows.

If $\overline{d}_j - a_i = p_j$, we schedule job j from a_i to $\overline{d}_j$ on machine i (as in case one of the G-S algorithm).

If this is the maximum a_i value (thus the 'slowest' processor) we simply schedule job j from a_i to $a_i + p_j$ (this corresponds to case 3).

Otherwise we also find the smallest a_r such that $\overline{d}_j - a_r < p_j$. Now schedule job j on machine r from a_r to $\overline{d}_j$ and on machine i for the remaining $p_j - (\overline{d}_j - a_r)$ time units starting at time a_i. This corresponds to case 2.

In each of the three cases above it is then easy to update a_i (and a_r in case 2) to its new value.

Finding a_i and a_r as well as updating their values can all be done in $O(logm)$ time, and each job is then scheduled in O(1) time. Hence the overall running time of the algorithm is $O(n \log m)$. If time to sort the $\overline{d}_j$'s is included, the overall running time becomes $O(nlogn)$.

10.2.2. Minimization of $L_{\max}$

We can apply Meggido's method to transform the Sahni-Cho feasibility algorithms to algorithms for solving $P|pmtn|L_{\max}$ and $Q|pmtn|L_{\max}$. What is needed is the smallest value of λ such that the induced deadlines $\overline{d}_j = d_j + \lambda$ admit a feasible schedule. The Sahni-Cho feasibility algorithm has n iterations. In the case of uniform machines, at each iteration a bisection search enables us to find the desired index k with $O(\log m)$ comparisons of p_j against the machine capacities, which are linear functions of λ. It follows that $L_{\max}$ can be minimized with $O(n \log m)$ calls on the feasiblility algorithm, or $O(mn^2 \log m)$ time overall. In the case of identical machines, the feasibility algorithm also requires $O(\log m)$ comparisons at each iteration. The result of each comparison can be resolved by a call to the feasibility algorithm. Hence $L_{\max}$ can be minimized with $O(n \log m)$ calls on the feasibility algorithm, or $O(n^2 \log^2 m)$ time overall.

Note: The problems $P|pmtn|L_{\max}$ and $Q|pmtn|L_{\max}$ are equivalent, by symmetry of release dates and due dates, to the problem $P|pmtn,r_j|C_{\max}$ and $Q|pmtn,r_j|C_{\max}$. Hence the algorithms described above solve these problems as well.

Exercises

10.8. Describe how to minimize *weighted* maximum lateness for identical and uniform machines. (Note: The relative order of the induced deadlines $\overline{d}_j = d_j + \lambda/w_j$ may change with λ.)

10.9. Give a family of $Q|pmtn,\overline{d}_j|--$ problems which require $\Omega(mn)$ preemptions.

10.3. The Staircase Algorithm for Release Dates

In this section we shall describe an algorithm for solving the problem $Q|pmtn,r_j|C_{\max}$ For reasons that will become apparent, we refer to this as the "staircase" algorithm. The details of the algorithm and its implementation are somewhat involved. However, the principal idea behind the algorithm is actually quite simple.

Assume the jobs are numbered in release date order, $r_1 \leq \ldots \leq r_n$. The algorithm creates a schedule by moving from one release date to the next, creating a subschedule for each successive interval $[r_k, r_{k+1}]$, $k = 1,2,\ldots n-1$. At r_k, k jobs have been released, and their remaining processing requirements in sorted order are $p_1^{(k)} \geq \ldots \geq p_k^{(k)} \geq 0$. (Included among these processing requirements is p_k, since job k is released at r_k; no correspondence between the indexing of the the $p_j^{(k)}$'s and the release dates is intended.) The algorithm has the property that the remaining processing requirements are as *evenly* distributed as possible. More specifically, there is no way that the jobs could have been processed before r_k that would have yielded a smaller value for *any* of the partial sums

$p_1^{(k)}$,
$p_1^{(k)} + p_2^{(k)}$,
...
$p_1^{(k)} + p_2^{(k)} + \ldots + p_k^{(k)}$.

At r_k, the algorithm utilizes the capacities of the m machines in the interval $[r_k, r_{k+1}]$ so as to minimize each and every one of the new partial sums

$p_1^{(k+1)}$,
$p_1^{(k+1)} + p_2^{(k+1)}$,
....
$p_1^{(k+1)} + p_2^{(k+1)} + \ldots p_k^{(k+1)}$.

where $p_1^{(k+1)} \geq \ldots \geq p_k^{(k+1)} \geq 0$. This means that when r_n is reached, the remaining processing requirements $p_j^{(n)}, j = 1,...,n$, are such that the length of the final interval $[r_n, C_{\max}]$ is minimized. (The length of this final interval is determined by the smallest value of T such that inequalities (10.4) are satisfied, with respect to the

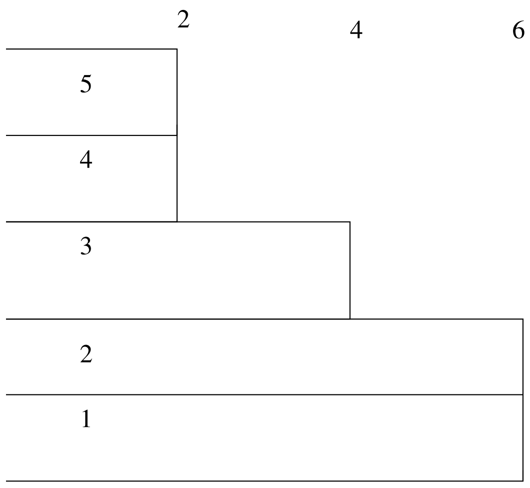

Figure 10.4. staircase of remaining processing times

remaining processing requirements.)

The reader may wish to reflect on a certain duality between the staircase algorithm and the Sahni-Cho algorithm. As we have described, the staircase algorithm schedules several jobs at each iteration in such a way as to minimize each of the partial sums of the processing requirements remaining for the next iteration. By contrast, the Sahni-Cho algorithm schedules a single job at each iteration in such a way as to maximize each of the partial sums of the processor capacities remaining for the next iteration.

The staircase algorithm has the desirable property that it is *nearly on-line*. By this we mean that at each successive release date r_k the algorithm does not require any knowledge of the jobs that are to be released at future times; all it requires is knowledge of the next release date r_{k+1}. (If it did not require knowledge of r_{k+1}, it would be truly on-line.) By contrast, the Sahni-Cho algorithm is off-line. When the Sahni-Cho algorithm is applied to an instance of the feasibility problem $Q|pmtn, r_j, \overline{d}_j = \overline{d}| - -$ (by applying it to a symmetrically equivalent instance of $Q|pmtn, \overline{d}_j| - -$), the algorithm iterates from the latest release date backward.

10.3.1. Constructing the staircase

Let us consider how the algorithm determines the amount of processing to perform on each job in the interval $[r_k, r_{k+1}]$. For ease of notation, let us drop the superscripts and denote $p_j^{(k)}$ by p_j and let q_j denote the remaining processing time of the job associated with p_j after processing in this interval. We also let $t = r_{k+1} - r_k$. For purposes of exposition, we assume for the time being that, if $m < k$, machines $m + 1, \ldots, k$, with $s_{m+1} = \ldots = s_k = 0$, are added to the model.

The p_j can be viewed as defining a staircase pattern as shown in Figure 10.4. The q_j will chosen in such a way that they form a similar pattern. Such a staircase can be characterized by grouping the remaining processing times into blocks of equal values. If there are u different values remaining, we let $q(1)$ denote the maximum remaining value, $q(2)$ the second largest remaining value, ..., and $q(u)$ the smallest remaining value (with $q(u) \geq 0$). We then let $h(i)$ represent the *height* of the first i blocks. Thus $h(1)$ is the number of jobs with $q(1)$ units of processing remaining, and in general $h(i)$ is the number of jobs with $q(i)$ or more units of processing remaining. In example 10.4 $q(1) = 6, q(2) = 4, q(3) = 2, h(1) = 2, h(2) = 3, h(3) = 5$. The staircase can be characterized by

$((h(1),q(1)),...,(h(u),q(u))$,

where $q_j = q(i)$, for each job $j, h(i-1)+1 \leq j \leq h(i)$.

(Here $i = 1,...,u$; $h(0) = 0$, $h(u) = k$). We always have:

$$q(i) > q(i+1), \quad i = 1,...,u-1. \tag{10.6}$$

The staircase is constructed in such a way that for each i, the capacities of machines $h(i-1)+1,\ldots,h(i)$ are fully utilized to decrease $p_{h(i-1)+1},\ldots,p_{h(i)}$ to $q(i)$. A second condition for feasibility is therefore that

$$\sum_{j=h(i-1)+1}^{l} q_j = (l-h(i-1))q(i) \geq \sum_{j=h(i-1)+1}^{l} p_j - t \sum_{j=h(i-1)+1}^{l} s_j \tag{10.7}$$

for $l = h(i-1)+1,\ldots,h(i); i = 1,...,u$.

The corners of the staircase, except possibly the last one, correspond to strict equalities:

$$\sum_{j=h(i-1)+1}^{h(i)} q_j = (h(i)-h(i-1))q(i) = \sum_{j=h(i-1)+1}^{h(i)} p_j - t \sum_{j=h(i-1)+1}^{h(i)} s_j \tag{10.8}$$

for $i = 1,...,u-1$.

A third condition for feasibility is of course that

$$0 \leq q_j \leq p_j, j = 1,2,...,k. \tag{10.9}$$

We tentatively construct the first step of the staircase by setting

$h(1) := 1$, $q(1) := p_1 - ts_1$.

Generally, after constructing i tentative steps, $(h(1),q(1)),...,(h(i),q(i))$, the tentative steps satisfy the feasibility conditions (10.6) to (10.9), except that possibly $q(i) < 0$, in violation of (10.9). We construct an $(i+1)st$ tentative step by setting

$h(i+1) := h(i)+1, \qquad q(i+1) := p_{h(i+1)} - ts_{h(i+1)}$.

If $q(i) > q(i+1)$ and $q(i) \geq 0$, we are finished with the construction of the new tentative step.

Suppose now that $q(i) < 0$, in violation of condition (10.9), or $q(i) \leq q(i+1)$, in violation of condition (10.6). In the first case, there is unused capacity on machines

$h(i-1)+1, \ldots h(i)$; in both cases, some of the capacity of these machines must be used to process job h(i)+1, in order to satisfy conditions (10.6), (10.8) and (10.9). We therefore reconstruct the i-th step so as to include job $h(i)+1$ in that step, setting

$h(i) := h(i)+1$,

and recalculating $q(i)$ according to (10.8),

$$q(i) := (\sum_{j=h(i-1)+1}^{h(i)} p_j - \ t \sum_{j=h(i-1)+1}^{h(i)} s_j)/(h(i)-h(i-1)) \tag{10.10}$$

Using the old $q(i), q(i+1)$ values, the formula for the total remaining processing of the jobs in the new combined step is:

$(h(i)-h(i-1)-1)q(i)+p_{h(i)}-ts_{h(i)} \ = \ (h(i)-h(i-1)-1)q(i)+q(i+1)$

Thus we can compute the new value of $q(i)$ as:

$$q(i) := [(h(i)-h(i-1)-1)q(i)+q(i+1)]/(h(i)-h(i-1) \tag{10.11}$$

As a result, it may now be that $q(i-1) \leq q(i)$ ($q(i-1) < 0$ cannot occur). In this case, we reconstruct the $(i-1)$st step so as to include the i-th step: $h(i-1)$ is set to $h(i)$ and $q(i-1)$ is recalculated using an analagous approach to (10.11). We continue in this way until condition (10.6) is satisfied. The adjusted staircase includes one more job and may have fewer steps than before.

Pseudo-code for the staircase subalgorithm is as follows, where the interval length t and processing times $p_1 \geq \ldots \geq p_k$ which remain at the start of interval k are the inputs, and the resulting $q(i), h(i)$ values are the outputs:

staircase$(t, p_1, ..., p_k; q(1), \ldots, q(i); h(1), \ldots h(i))$:

```
$h(0) := 0$;
$q(0) := \infty$
$i := 0$;
**for** $j = 1, .., k$
    $i := i+1$;
    $h(i) := j$;
    $q(i) := p_j - ts_{h(i)}$;
    **while** ($q(i-1) \leq q(i)$ or $q(i-1) < 0$)
        $q(i-1) := [(h(i-1)-h(i-2))q(i-1)+(h(i)-h(i-i))q(i)]/$
        $[h(i)-h(i-2)]$;
        $h(i-1) := h(i)$;
        $i := i-1$;
**return** $q(1), \ldots, q(i); h(1), \ldots, h(i)$;
```

We have to verify that the resulting staircase $(h(1), q(1)), ...(q(k), h(k))$ and the corresponding remaining processing requirements $q(1), ..., q(u)$ indeed satisfy the the feasibility conditions (10.6–10.9). For (10.6) and (10.8) this is obvious. To see that (10.7) must be true, note that each $q(i)$ is initially defined by an equality constraint and can only increase thereafter. To verify condition (10.9), it suffices to show that $q(i) \leq p_{h(i)}$ (since if the new processing amount for jobs in the *ith* step ($q(i)$) is equal to or less than the smallest intial processing amount ($p_{h(i)}$) then the condition holds for those jobs in the step which started with more processing). We initially set $q(i)$ to $p_{h(i)} - ts_{h(i)}$. When two steps are combined, the current value $h(i)$ becomes the height of the new step $i-1$, and the new value of $q(i-1)$ is at most the old value of $q(i)$. Thus after combining the two steps, the new value $q(i-1)$ is still at most $p_{h(i-1)}$ which implies the desired result.

10.3.2. Complexity analysis

We now analyze the running time of the subalgorithm. The number of step constructions (first three lines of *for* loop) is exactly k. Each iteration of the *while* loop combines two steps, so this is done at most $k-1$ times in total. Thus the entire *staircase* function runs in O(k) time. This presupposes that the given p_j values are ordered; but since the relative order of the remaining processing requirements does not change, we can maintain an ordered list of these values and insert the processing requirement p_k of job k that becomes available at r_k in $O(k)$ time. Hence the subalgorithm determines the values $q(i)$ for each interval in $O(k)$ time. As has been indicated above, the Gonzalez-Sahni algorithm can be applied to construct an actual schedule for each interval in $O(k)$ time as well (since the machines and processing times are assumed to have been sorted). We thus have arrived at a nearly on-line algorithm that requires $O(n^2)$ time overall.

10.3.3. Correctness of the algorithm

We note first that not only does the relative order of the remaining processing requirements remain invariant, but also the following stronger property holds: as soon as two remaining processing requirements become equal, they remain equal. To see this, suppose that $p_j = p_{j+1}$ at time r_k, and they are in different steps, so, let $h(i) = j$. We set $q(i+1) = p_{j+1} - ts_{j+1}$. But $q(i) \leq p_j - ts_j \leq p_j - ts_{j+1} = q(i+1)$, and we have to reconstruct the i-th step so as to include job $j+1$ as well.

This leads us to define the *rank* of an available job j at time r_k as the value $h(i)$ for which $h(i-1)+1 \leq j \leq h(i)$ (where these are the final $h(i)$ values computed for the interval $[r_k, r_{k+1}]$). The rank of a job at time r_n is defined analogously as its step height that would be found if the subalgorithm were to be applied in the interval $[r_n, C^*_{\max}]$. A job will be called *critical* if its rank is at most $m-1$ and it hasn't been completed. Otherwise the job is *noncritical.* The rank of a job cannot decrease; in particular, once a job becomes noncritical, it never becomes critical again. It follows from (10.8) that in any interval the fastest $h(i)$ machines are exclusively processing the longest $h(i)$ critical jobs. A critical job is processed continuously from its release date until it either is completed or becomes noncritical.

These observations suggest the following correctness proof for the algorithm. First, suppose that the schedule ends at $C^*_{\max}$ with the simultaneous completion of

l critical jobs ($l < m$). At any time when l' of these jobs are available, they are processed by the fastest l' machines. Thus the maximum possible total processing has been devoted to these l critical jobs prior to r_n, so, the schedule is clearly optimal.

Alternatively, suppose that the schedule ends with the simultaneous completion of m noncritical jobs. If there is no idle time in the schedule it is clearly optimal. Otherwise, let r_k be the last release date such that there is idle time in $[r_{k-1}, r_k]$ on some machine. All the jobs that are available but noncritical at time r_{k-1} will thus be completed by time r_k. We conclude that the portion of the schedule for the remaining jobs has the following structure. Before r_k, the available critical jobs are processed by the fastest machines. Between r_k and $C^*_{\max}$, there is no idle time. It follows that the schedule is optimal for the jobs under consideration and thus that $C^*_{\max}$ is the minimum time to complete all the jobs.

10.3.4. A more efficient off-line implementation

Let us use the terminology of the prior section to describe a more efficient implementation of the staircase subalgorithm. We will reduce the running time by dealing more carefully with the noncritical jobs, circumventing the need to introduce machines of speed zero.

The noncritical jobs of lowest rank, i.e., jobs $h(i-1)+1, \ldots h(i)$, where $h(i-1)+1 \leq m \leq h(i)$, will be called *active*. In the interval $[r_k, r_{k+1}]$, their remaining processing requirements are reduced by machines $h(i-1)+1, \ldots m$ to a common amount $q(i)$. The remaining processing requirements are not reduced at all, since they are assigned to dummy machines of speed zero.

As a first refinement, the subalgorithm does not have to deal with the active noncritical jobs individually, since their remaining processing requirements will remain equal throughout. They can easily be handled simultaneously by straightforward generalizations of (10.10) and (10.11). As a second refinement, the subalgorithm can be terminated as soon as either $h(i) = k$ or $h(i) \geq m$ and $q(i) > p_{h(i)+1}$.

Instead of maintaining an ordered list of all remaining processing requirements, we have only to do so for the largest $m-1$ of them. We simply record the number of active noncritical jobs, their common remaining processing requirement, and the lowest index of any of them. Finally, we maintain a priority queue for the remaining requirements of the inactive noncritical jobs.

At each release date the processing requirement of the job(s) that becomes inactive is, depending on its size, inserted either in the ordered list in $O(m)$ time or in the priority queue in $O(\log n)$ time. The staircase computations for the longest $m-1$ jobs and the active noncritical jobs require $O(m)$ time in each interval and $O(mn)$ time overall. The queue operations require $O(logn)$ time when a job is inserted or deleted and $O(nlogn)$ time overall, since once an inactive job becomes active and is withdrawn from the queue, it remains active throughout. Hence succssive applications of the modified subalgorithm determine the value $C^*_{\max}$ in $O(n \log n + mn)$ time. As has been indicated above, the Sahni-Cho algorithm can be applied to construct an actual schedule in the interval $[r_1, C^*_{\max}]$ in $O(n \log n + mn)$ time as well. We thus have arrived at an off-line algorithm that requires $O(n \log n + mn)$ time overall.

Exercises
10.10. Devise an example to show that there can be no on-line algorithm for $Q|pmtn, r_j|C_{\max}$.
10.11. Suppose m uniform machines are capable of *processor sharing*. By this we mean that the processing capacity of a given subset of machines can be equally shared by a given subset of jobs (with processing occuring in infinitesimally small time slices, and preemption infinitely often). Show that under the assumption of processor sharing, an on-line algorithm does exist for $Q|pmtn, r_j|C_{\max}$. Describe how this algorithm is related to the staircase algorithm.

10.4. Network Flow Computations for Release Dates and Deadlines

Let us begin with the feasibility problem $P|pmtn, r_j, \bar{d}_j| - -$. Let $\{e_1, ..., e_{2n}\}, e_1 \leq \ldots \leq e_{2n}$, be the ordered collection of release dates r_j and deadlines $\bar{d}_j$. If a release date and a deadline are equal, the smaller index is assigned to the release date. Let E_k denote the time interval $[e_k, e_{k+1}]$, for $k = 1, 2, ..., 2n-1$.

We shall construct a flow network with job nodes $j = 1, .., n$, interval nodes E_k, $k = 1, ..., 2n-1$, a source node s and a sink node t. There is an arc (j,k) of capacity $e_{k+1} - e_k$ from job node j to each of the interval nodes E_k such that $r_j \leq e_k$ and $\bar{d}_j \leq e_{k+1}$. In addition there is an arc (s, j) of capacity p_j from the source node to each job node j and an arc (k,t) of capacity $m(e_{k+1} - e_k)$ from each interval node to the sink node. We assert that there exists a feasible preemptive schedule for the given instance of the feasibility problem if and only if the flow network admits a flow which saturates all the arcs out of s (and thus the total flow equals the sum of the p_j values).

The reasoning is very simple. The flow value $f(j,k)$ for the arc (j,k) represents the amount of processing of job j that is done in interval E_k. The capacities $e_{k+1} - e_k$ assigned to the arcs (j,k) assure that no job is scheduled for more than the length of E_k, and the capacity $m(e_{k+1} - e_k)$ of the arc (k,t) assures that the total processing done by all jobs in interval E_k can be completed. Thus, the amount of processing that is done on the various jobs in the interval E_k satisfies the conditions (10.1).

If the network flow computation indicates that there exists a feasible schedule, McNaughton's algorithm can be applied to the arc flow values $f(j,k)$ to construct a feasible subschedule within a given time interval E_k in $O(n)$ time. We thus can construct a feasible schedule in $O(p(n) + n^2)$ time, where $p(n)$ is the time required for the max flow computation.

Since any legal schedule can be converted to a feasible flow which saturates all arcs out of s, if the maximum flow is less than the sum of the p_j values, we know that no legal schedule exists.

10.4.1. $Q|pmtn,r_j,\bar{d}_j|-$

It requires a bit of cleverness to construct a flow network for the problem $Q|pmtn,r_j,\bar{d}_j|-$. In order for conditions (10.4) to be satisfied for the interval E_k, we must restrict the total processing done on any single job to $s_1(e_{k+1}-e_k)$, on any pair of jobs to $(s_1+s_2)(e_{k+1}-e_k),\ldots$, on any $m-1$ jobs to $(s_1+...+s_{m-1})(e_{k+1}-e_k)$, and on all jobs to $(s_1+...+s_m)(e_{k+1}-e_k)$. What we shall do is modify the network of the prior subsection by replacing the node for E_k by m nodes $(i,E_k), i=1,\ldots m$.

The capacity of each arc $(j,(i,E_k))$ will be $(s_i-s_{i+1})(e_{k+1}-e_k)$, and the capacity of each arc $((i,E_k),t)$ will be $i(s_i-s_{i+1})(e_{k+1}-e_k)$. (Here define s_{m+1} to be zero.)

To validate the network construction we argue as follows. Consider the maximum amount of flow there can be from any set of u job nodes to the m nodes associated with interval E_k. We will now show that this total flow is at most $(s_1+...+s_u)(e_{k+1}-e_k)$, which is the maximum amount of processing we can do on u jobs in this interval.

The maximum flow that can pass through the node (i,E_k) is Min $\{u,i\}(e_{k+1}-e_k)$. Hence the total flow that can pass through all the nodes (i,E_k) to t is

$$\begin{array}{ll} & (s_1-s_2)(e_{k+1}-e_k) \\ + & 2(s_2-s_3)(e_{k+1}-e_k) \\ & \ldots \\ + & (u-1)(s_{u-1}-s_u)(e_{k+1}-e_k) \\ + & u(s_u-s_{u+1})(e_{k+1}-e_k) \\ & \ldots \\ + & u(s_m)(e_{k+1}-e_k) \\ = & (s_1+...+s_u)(e_{k+1}-e_k), \end{array}$$

giving us the desired result.

Note that the network constructed for the $Q|pmtn,r_j,\bar{d}_j|-$ problem has $O(mn)$ nodes, hence the running time of the feasibility computation is $O(p(mn))$. However, the special structure of the flow network gives faster time bounds then for general networks of this size (this is partly explored in the exercises below).

Exercises

10.12. Show that the flow network for identical machines is actually a special case of the flow network for uniform machines. (That is, when $s_1=...=s_m$, and arcs of zero capacity are removed, the flow network for identical machines is obtained.)
10.13. Show that only $O(tn)$ nodes are needed for the uniform machine model when there are only t machine speeds.
10.14. Describe how to construct a network flow model for the case of 2 uniform machines, with only one node for each time interval E_k.
10.15. Suppose each job had a list of time intervals during which it could be processed. Describe how to modify the network flow formulation to deal with this setting.

10.5. Minimizing Maximum Lateness

We now apply the results of the previous section to solve the problems $P|pmtn,r_j|L_{\max}$ and $Q|pmtn,r_j|L_{\max}$. The network flow model enables us to test any given value of $L_{\max}$ for feasibility: for any given trial value λ, induce deadlines $\bar{d_j} = d_j + \lambda$ and perform a max-flow computation to test for the existence of a feasible schedule with respect to the induced deadlines. It follows that we can minimize $L_{\max}$ by finding the smallest value of λ for which there is a feasible schedule.

Observe that the relative ordering of release dates and induced deadlines changes with λ, hence the topology of the flow network also changes with λ. There are at most n^2 *critical values* of λ that are of particular concern, namely those values such that $d_j + \lambda = r_k$, for some j and k. The node-arc structure of the flow network remains invariant for all values of λ between two successive critical values; only the arc capacities change. Our first task is to compute the n^2 critical values of λ and to carry out a bisection search over them, finding the largest *infeasible* critical value λ_0. This can be done in $O(n^2)$ time, plus the time required for $O(logn)$ max-flow computations.

Having found λ_0, we are able to fix the topology of the flow network we shall be dealing with. The task that remains is finding the smallest increment δ such that the arc capacities induced for $\lambda_0 + \delta$ permit a flow value of P (the sum of the processing times). (In a degenerate case, it may be that $\lambda_0 + \delta$ equals the critical value of λ next larger than λ_0, but this causes no problem.) We shall first consider how to minimize δ in the case of identical machines.

Let $E_k,\ k = 1,...,2n-1$, be the time intervals induced by λ_0. Each interval $E_k = [e_k, e_{k+1}]$ is of one of four types, depending upon whether e_k and e_{k+1} are release dates or induced deadlines; we shall refer to these four types as $[r,r]$, $[r,d]$, $[d,r]$, and $[d,d]$ intervals. When the trial value of $L_{\max}$ is increased from λ_0 to $\lambda_0 + \delta$, the length of an $[r,r]$ or $[d,d]$ interval remains unchanged, the length of an $[r,d]$ interval increases by δ, and the length of a $[d,r]$ interval decreases by δ. This means that when the arc capacities are expressed as a function of the parameter delta, we have the following results. Each arc (j,E_k) has a capacity:

$$(e_{k+1} - e_k + \Delta) \tag{10.12}$$

and each arc (E_k,t) has a capacity:

$$m(e_{k+1} - e_k + \Delta) \tag{10.13}$$

Where $\Delta = 0$ if E_k is an $[r,r]$ or $[d,d]$ interval; $\Delta = \delta$, if E_k is an $[r,d]$ interval; and $\Delta = -\delta$, if E_k is a $[d,r]$ interval.

All arcs (s,j) have capacity p_j.

Note that each arc e_i in the flow network induced by $\lambda_0 + \delta$ has a capacity of the form $c_i + \mu_i\delta$ where c_i is a constant and μ_i is a multiplier of value 0, 1, -1, or m.

10.5.1. First approach: apply Meggido's method

Because each arc capacity is a linear function of δ, Meggido's method can be applied to find the minimum value of δ such that a flow of value P (the sum of all p_j values) can be achieved. A straightforward application of Meggido's Theorem (Chapter 2) yields a solution to $P|pmtn,r_j|L_{\max}$ in $O(p^2(n))$ time, where $p(n)$ is the running time of the max-flow algorithm chosen.

10.5.2. Second approach: iteration on trial values

The capacity of each (s,t) cut in the flow network is also a linear function of δ. Consider the capacity of a *minimum* cut. Each of the $2n-1$ nodes E_k is either on the source side or on the sink side of the cut. If E_k is on the source side, arc (E_k,t) contributes its capacity to the capacity of the cut as determined by (10.13). If E_k is on the sink side, then at most m arcs (j,E_k) contribute their capacities to the capacity of the cut, where these capacities are determined by (10.12). (At most m arcs (j,E_k) can contribute their capacities to the capacity of a minimum cut, else the total capacity of these arcs would exceed that of the arc (E_k,t).) Thus in the flow network with arc capacities induced by λ_0, each minimum cut C' has a capacity as a function of δ, which is $P'+\mu\delta$, where P' is an integer constant less than P and μ is an integer *multiplier* obtained by summing the μ_i values associated with the arcs in C'. Thus μ is no greater than $m(2n-1)$. It follows that in order for the capacity of any given minimum cut C' to be P, we must have $\delta=(P-P')/\mu$.

An iterative procedure for finding the optimal value of λ is as follows. Find a minimum cut C_0, with capacity $P_0<P$, in the flow network with arc capacities induced by λ_0. Set $\lambda_1=\lambda_0+(P-P_0)/\mu_0$. where μ_0 is the multiplier for C_0. Find a minimum cut C_1, with capacity $P_1>P_0$ in the network with arc capacities induced by λ_1. If $P_1=P$, terminate. Otherwise iterate, setting $\lambda_i=\lambda_{i-1}+(P-P_{i-1})/\mu_{i-1}$, until a minimum cut C_i is found with $P_i=P$.

Observe that in the network with arc capacities induced by λ_i, the capacities of all cuts with multipliers greater than or equal to μ_{i-1} are at least P. Hence at each iteration i, except possibly the last, $\mu_i>\mu_{i-1}$. Since there are at most $m(2n-1)$ values for the multipliers, the procedure must terminate in $O(mn)$ iterations. We now have achieved a time bound of $O(mnp(n))$ for finding the optimal value of lambda.

10.5.3. Third approach: bisection search on δ

The desired value of δ is $(P-P')/\mu$, for some cut C' with capacity $P'+\mu\delta$. If the release times and due-dates are integers, P' is a positive integer no greater than P and μ is a positive integer no greater than $m(2n-1)$. It follows that the optimum value of δ can be found by carrying out a bisection search over $O(mnP)$ ratios, which can be accomplished with $O(\log n+\log p_{\max})$ calls to the max-flow algorithm, yielding a time bound of $O(p(n)(\log n+\log p_{\max}))$.

10.5.4. Uniform Machines

Now let us extend the above results to the case of uniform machines. Instead of the capacities (10.12), we have for each arc $(j,(i,E_k))$:
$(s_i - s_{i+1})(e_{k+1} - e_k)$, if E_k is an $[r,r]$ or $[d,d]$ interval,
$(s_i - s_{i+1})(e_{k+1} - e_k + \delta)$, if E_k is an $[r,d]$ interval,
$(s_i - s_{i+1})(e_{k+1} - e_k - \delta)$, if E_k is a $[d,r]$ interval.
And instead of (10.13) we have for each arc $((i,E_k),t)$:
$i(s_i - s_{i+1})(e_{k+1} - e_k)$, if E_k is an $[r,r]$ or $[d,d]$ interval,
$i(s_i - s_{i+1})(e_{k+1} - e_k + \delta)$, if E_k is an $[r,d]$ interval,
$i(s_i - s_{1+1})(e_{k+1} - e_k - \delta)$, if E_k is a $[d,r]$ interval.

Note that each arc e_i in the flow network induced by $\lambda_0 + \delta$ still has a capacity of the form $c_i + \mu_i\delta$ where c_i is a constant but now the multiplier μ_i is of the form $i(s_i - s_{i+1})$ where i is an integer in the range $-m,\dots,-1,0,1,\dots,m$.

The capacity of each (s,t) cut C' is a linear function of the form $P' + \mu\delta$. For a min cut, μ is no greater than $(s_1 + s_2 + ... + s_m)n$ since we get at most this value by putting all of the $(i,E_k))$ nodes on the s side of the cut.

A straightforward application of Meggido's method yields a running time of $O(p^2(mn))$. In order to apply the other two methods, we must assume that the machine speeds s_i are integers. The iterative method then has a running time of $O((s_1 + ... + s_m)np(mn))$, and the bisection search method has a running time of $O(p(mn)(\log n + \log s_1 + \log p_{\max})$.

Exercises
10.16. Extend the ideas of this section to solve the *weighted* maximum lateness problems $P|pmtn,r_j|wL_{\max}$ and $Q|pmtn,r_j|wL_{\max}$. (Note: Each trial value of λ in this setting induces deadlines $\overline{d}_j = d_j + \lambda/w_j$.)
10.17. The bound of $m(2n-1)$ on μ for a min-cut in the indentical machine setting can be improved. Show that $\mu < mn$.

10.6. Memory Constraints

We now consider a variation of $Q|pmtn|C_{\max}$ where we have the additional constraint that each processor i has a *memory capacity* c_i, and each job j has a *memory requirement* q_j. A job j can be executed on machine i if and only if

$$c_i \geq q_i \tag{10.14}$$

This is a special case of $R|pmtn|C_{\max}$ which will be discussed in the next section.

Our approach will be to first describe a method for machines of identical speeds, then generalize this to a feasibility testing algorithm for uniform machines. Finally, we use search techniques to find the minimum possible completion time.

We assume that the c_i values have been sorted so that $c_1 \geq \dots \geq c_m$. We can then partition the jobs into sets G_i where

$G_i = \{J_j \mid c_i \geq q_j > c_{i+1}\} \quad 1 \leq i \leq m-1$, and
$G_m = \{J_j \mid c_m \geq q_j\}$.

Thus G_i is the set of all jobs which can be run on processor i but not on processor $i+1$. We also define F_i to be the set of all jobs which must be run on machines one to i:

$F_i = \cup_{j=1}^{i} G_j$.

We will also define X_i to be the total processing of all jobs in F_i.

10.6.1. Identical Machines

Kafura and Shen proved that when all machines have the same speed (thus speed one) the length of an optimal schedule is:

$$max\{max\{(1 \leq i \leq m) \;\; X_i/i\}, p_{\max}\} \tag{10.15}$$

The algorithm is quite simple: we form the sets G_i and F_i and use this to compute the value of $C_{\max}$ using (10.15). We then schedule the jobs in each set G_i using McNaughton's rule.

We can form the G_i sets on $O(n \log m)$ time and all other steps take $O(n)$ time, so the total time is $O(n \log m)$. Just as in McNaughton's algorithm there are at most $m-1$ preemptions.

10.6.2. Uniform Machines

For this setting we know of no closed form expression for $C_{\max}$. Thus we start by considering the case where all jobs have a common deadline D and try to find a schedule which completes all jobs by time D. At a high level our algorithm has the same structure as for identical machines: form the sets G_i and then schedule the jobs in the sets $G_1, G_2, \ldots G_m$. The main difference is that each set is now scheduled using the Gonzalez-Sahni algorithm of Section 10.1.1. We will also have to be more careful about adding new processors.

Intuitively our goal is to schedule each set G_i while leaving as much time on the fast processors as possible for future jobs. Fortunately that is exactly what the Gonzalez-Sahni algorithm does. Recall also that the Gonzalez-Sahni algorithm is set up to work on a composite processor system where each processor has a time varying speed.

Recall that our composite processors consist of *elementary processor intervals* which are just time intervals on our original processors. For example, if $D = 4$ composite processor one might have speed 5 in $[0,2]$, speed 10 in $(2,3]$ and speed 20 in $(3,4]$. Note that the speeds increase as time increases. As we schedule jobs and add processors we will maintain the following properties: i) that within each composite processor speeds are nondecreasing as time increases; ii) no speed in processor i is greater than the slowest speed in processor $i-1$. We can view this as lining up the composite processors from left to right starting with the last one on the left. As we go from left to right, the speeds of the elementary processor intervals are nondecreasing. We can view this as a sort of super composite processor of length mD (with the early part possibly of speed zero). Let $R(T)$ denote the total processing capacity of this super processor using the last T time units (e.g. $R(D)$ is the capacity of composite processor one, and $R(1.5\,D)$ is the total capacity of processor one and the fastest $D/2$ time units of processor two).

We now consider how to update a composite processor system. After scheduling a set of jobs G_{i-1} we are left with a composite processor system *CP* which has the remaining processing capacity of processors $1,2,...,i-1$. Before scheduling set G_i we need to add processor i to *CP* to create a new composite processor system *CP'*. To add processor i, we splice it into CP as follows. First find the largest speed s_r in CP such that $s_r < s_i$ (note s_r could be zero). Let k be the smallest index of a composite processor which has an elementary processor interval of speed s_r and let $[t_1,t_2]$ be the last elementary processor interval of speed s_r on composite processor k. We change composite processor k to have speed s_i in the interval $[0,t_2]$, and create a new composite processor $(k+1)$ which has speed s_1 in the interval $(t_2,D]$ and inherits the elementary processor intervals which were on machince k in the interval $[0,t_2]$. The speeds of the other composite processors are unchanged, but those with index greater than k all increase their index by 1. This maintains the property that speeds increase as we go later in time or to earlier indexed processors.

Let CP_i be the composite processor system after our algorithm schedules the jobs in G_i. We define $R_i(T)$ as the total processing capacity using the fastest T time units of processing in CP_i. The correctness of our feasibility algorithm follows from these facts:

i) Consider any alternate schedule S for the jobs in F_i; let $R'(T)$ represent the total processing capacity of the T fastest units of time which remains idle on processors $1,2,...,i$ in S. Then $R_i(T) \geq R'(T)$ for $0 \leq T \leq mD$.

ii) After splicing in processor $i+1$ as described above to get a new composite processor system, the R function for this also dominates any alternate schedule's remaining processing capacity.

These facts are proved by showing that if these properties hold before a set of jobs is scheduled or a processor inserted, they must also hold afterwards.

Theorem 10.1 [Feasibility Algorithm]. *The feasibility algorithm will schedule all jobs by time D whenever such a schedule is possible.*

Proof. We will be able to schedule each set G_i as long as CP_i satisfies the inequalities of (10.4) for the Gonzalez-Sahni algorithm. If we ever reach a point where G_i cannot be scheduled, the fact that each of the sums $S_1, S_1+S_2, \ldots, S_1+\ldots S_m$ is as large as possible (by facts i) and ii) above), shows that no schedule can complete all the jobs in F_i.

We now analyze the complexity of our algorithm. The sets G_i can be constructed easily in $O(n\log m)$ time. To analyze the time for the Gonzalez-Sahni algorithm, note that each time we splice in a new processor we add at most two elementary processor intervals to our composite processor system. Since scheduling a job never creates additional elementary processor intervals, the total number of elementary processor intervals is at most $2m$. Each job can be scheduled using $O(\log m)$ time to find the correct composite processor(s) plus O(l) time, where l is the number of elementary processor intervals used up by this job. Thus the total time to schedule all jobs is $O(n \log m)$. The only remaining issue is adding a new processor. We can easily add a new processor in $O(m)$ time by simply scanning the doubly-linked list of elementary processor intervals which represents the appropriate composite processor. A more complex apprach stores the speeds of the elementary processor intervals in a balanced binary search tree. This allows us to insert a new processor in $O(logm)$ time. Thus the overall running time is $O(n \log m)$.

The feasibility algorithm lets us test any common deadline D. To find $C_{\max}$, the smallest feasible D, we can use Meggido's method using our feasibility routine as a subroutine. A simple application would result in a running time of $O(n^2 \log^2 m)$.

Exercises

10.18. Our description of the feasibility algorithm implicitly assumed that the memory sizes c_i were all distinct. Describe how to modify the algorithm if there are ties among the c_i values. In particular, describe how new processors are now inserted.
10.19. Describe the data structures needed to insert a new processor in $O(\log m)$ time. Also describe how to use and maintain these data structures when you schedule a new job.
10.20. Show that the feasibility algorithm introduces at most $3m-3$ preemptions. Also show that this bound is tight by giving a family of problems which require $3m-3$ preemptions to be scheduled.

10.7. Linear Programming Formulations for Unrelated Machines

Let x_{ij} denote the total amount of time that machine i processes job j. We shall now formulate some constraints that must be satisfied by any feasible schedule of length $C_{\max}$.

minimize $C_{\max}$
subject to

$$\sum_{j=1}^{n} x_{ij} \le C_{\max}, \quad i = 1, \ldots m, \tag{10.16}$$

$$\sum_{i=1}^{m} x_{ij} \le C_{\max}, \quad j = 1, \ldots n, \tag{10.17}$$

$$\sum_{i=1}^{m} (x_{ij}/p_{ij}) \ = 1, \quad j = 1, \ldots n, \tag{10.18}$$

and
$x_{ij} \geq 0$, for all $i = 1, \ldots m, j = 1, \ldots n$.

In this linear programming problem constraints (10.16) ensure that the total amount of processing done by any given machine does not exceed the time available, constraints (10.17) ensure that the total amount of processing done on any given job does not exceed the time available, and constraints (10.18) ensure that the processing done on any given job must be sufficient to complete it. Meeting these constraints is clearly necessary for any feasible schedule. But the converse is far from obvious. It is by no means clear that a feasible solution to this linear programming problem can always be transformed into a feasible schedule with the same value of $C_{\max}$.

To address this sufficiency question consider a matrix X of x_{ij} values which satisfy the LP constraints above. This matrix X has exactly the same form as the constraints of an open shop problem: for each job j, the x_{ij} values state the total amount of processing to be done for job j on machine i. Thus we can achieve a legal schedule using the solution of the LP exactly when the preemptive open shop problem described by X is feasible.

One classical theorem about open shop shows that any constraint matrix X that satisfies (10.16) and (10.17) can be scheduled to complete all jobs by time $C_{\max}$. We will briefly describe that result below.

The problem $R|pmtn|L_{\max}$ can be solved by an elaboration of the above linear programming formulation. The problem $R|pmtn, r_j|L_{\max}$ requires calling on a linear programming computation as a subroutine, in essentially the same way as the network flow computation was called on in the previous section.

10.7.1. Minimizing makespan in preemptive open shops

The *open shop* scheduling environment will be the subject of Chapter 12. That chapter will be devoted to the non-preemptive setting, but here we will outline a simple result to find the minimum makespan preemptive schedule; that is, $O|ptmn|C_{\max}$, is solvable in polynomial time. The input to an open shop instance consists of the processing time p_{ij} that job j requires for its operation on machine i, for each $i = 1, \ldots, m$, $j = 1, \ldots, n$. At each point in time, each machine can process at most one operation, and each job can have at most one of its operations being processed by any machine. Let

$$C = \max\{\max_j \textstyle\sum_i p_{ij}, \max_i \sum_j p_{ij}\}.$$

Call row i *tight* if $\sum_j p_{ij} = C$, and *slack* otherwise. Similarly, column j is *tight* if $\sum_i p_{ij} = C$, and is *slack* otherwise. We will give an algorithm that constructs a feasible schedule for which $C_{\max} = C$; hence, this schedule is optimal.

Theorem 10.2. *For any input $P = (p_{ij})$ to the open shop scheduling problem*

$O|ptmn|C_{\max}$, there is an optimal solution of with makespan,

$$C = \max\{\max_j \sum_i p_{ij}, \max_i \sum_j p_{ij}\},$$

and this can be found in polynomial time

Suppose that we can find a subset S of strictly positive elements of P, with exactly one element of S in each tight row and in each tight column, and at most one element of S in each slack row and in each slack column. We say that such a set S is a *decrementing set*, and use it to construct a *partial schedule* of length δ, for some $\delta > 0$.

By construction, for this partial schedule, we can process all of the operations in S concurrently. We first require that $\delta \leq p_{ij}$ for each element $p_{ij} \in S$; this ensures that we have sufficient work remaining for each operation in S to process it throughout the full length δ of the partial schedule. However, there are rows and columns that do not have an element in S (and hence must be slack). For each slack row i (column j) of P, the total work remaining for that machine (job) is less than the maximum C computed; however, no work is being done on that machine (job) in this partial schedule. Hence, the larger we set δ for that partial schedule, the closer the slack total work remaining becomes to matching the remaining work in the tight rows and columns.

We want to ensure that the tight rows and columns remain tight. Hence, if row i has no element in S, we need that $C - \delta \geq \sum_j p_{ij}$ (or equivalently, that $\delta \leq C - \sum_j p_{ij}$); similarly, if column j has no element in S, we need that $\delta \leq C - \sum_i p_{ij}$. Hence, we set δ to the minimum of these values (i.e., p_{ij} for elements in S, and the corresponding differences for each row or column without an element in S).

We now have constructed a partial schedule of length δ, where the maximum of row and column sums for the remaining processing times decreases from C to $C - \delta$, and yet there are either more elements of value 0, or else more tight rows or columns. We can iteratively build a schedule of length C for the input P by repeating this construction iteratively until the remaining processing times are all equal to 0. Since at least one row or column becomes tight or one element becomes 0, this must occur after $mn + m + n$ iterations.

We must still show that such a decrementing set always exists. We have that

$$\begin{aligned}
\sum_j p_{ij} &= C, && \text{for each tight row } i;\\
\sum_i p_{ij} &= C, && \text{for each tight column } j;\\
\sum_j p_{ij} &\leq C, && \text{for each slack row } i;\\
\sum_i p_{ij} &\leq C, && \text{for each slack column } j.
\end{aligned}$$

If we let $x_{ij} = p_{ij}/C$, then this is equivalent to:

$$\begin{aligned}
\sum_j x_{ij} &= 1, && \text{for each tight row } i; \\
\sum_i x_{ij} &= 1, && \text{for each tight column } j; \\
\sum_j x_{ij} &\le 1, && \text{for each slack row } i; \\
\sum_i x_{ij} &\le 1, && \text{for each slack column } j.
\end{aligned}$$

Of course, we also know that $0 \le x_{ij} \le 1$, for each $i = 1, \ldots, m,\ j = 1, \ldots, n$. If we instead view x as a variable, what we have argued is that p/C is a feasible solution to this system of linear constraints. This system defines a bounded, non-empty polytope, and hence there must be an extreme point solution x^*. But this is nothing more than the assignment polytope, and we know that all of its extreme points are integer. If we let S be the set of elements p_{ij} corresponding to those $x^*_{ij} = 1$, we have obtained a decrementing set.

Exercises

10.21. Suppose "a little birdie" told you the optimal ordering of the completion times for an instance of the problem $R|pmtn|\Sigma\, C_j$. Formulate and solve as a linear programming problem.

Notes

10.1. *Minimizing Makespan.* The algorithms of this section are adapted from McNaughton [1959] and Gonzalez & Sahni [1978B].

Horvath, Lam & Sethi [1977] proved that the bound (10.2) can be met by a preemptive variant of the *LPT* rule, which, at each point in time, assigns the jobs with the largest remaining processing requirement to the fastest available machines. The algorithm runs in $O(mn^2)$ time and creates an optimal schedule with no more than $(m-1)n^2$ preemptions. In Chapter 15 we show that the same variant of the *LPT* rule solves a stochastic version of $Q|pmtn|C_{\max}$.

Exercise 10.3 concerning constraints on machine availability, is suggested by a problem formulation of Schmidt [1983]. Exercise 10.5 is from Rayward-Smith [1987B].

10.2. *Meeting Deadlines.* The algorithm of this section is adapted from [Sahni & Cho. 1980]. The Sahni-Cho algorithm is also used to solve $Q|pmtn|\sum w_j U_j$ [Lawler, 1979A]. See also Lawler & Martel [1989].

10.3. *The Staircase Algorithm for Release Dates.* Horn [1974] gives an $O(n^2)$ procedure for $P|pmtn|L_{\max}$ and $P|pmtn, r_j|C_{\max}$. The staircase algorithm for uniform machines was developed independently by Sahni & Cho [1979B] and Labetoulle, Lawler, Lenstra & Rinnooy Kan [1979]. The more efficient implementation for

$P|pmtn,r_j|C_{\max}$ is due to Sahni [ ]. Cf also Gonzalez & Johnson [1980].

10.4. *Network Flow Computations for Release Dates and Deadlines.* Horn [1974] showed that the feasibility problem $P|pmtn,r_j,\bar{d}_j|-$ can be formulated as a network flow problem. Bruno & Gonzalez [1976] noted that
$Q2|pmtn,r_j,\bar{d}_j|-$ can be given a similar network flow formulation. Martel [1982] showed that $Q|pmtn,r_j,\bar{d}_j|-$ can be formulated as a special type of *polymatroidal* network flow problem, as described more generally by Lawler & Martel [1982]. Federgruen & Groenevelt [1986] showed that the problem can be reformulated as an ordinary network flow problem.

10.5. *Minimizing Maximum Lateness.* The algorithms of this section are adapted from Labetoulle, Lawler, Lenstra & Rinnooy Kan [1979]. Related ideas are discussed in Martel [1982] and Federgruen & Groenevelt [1986].

10.6. *Memory Constraints.* The algorithms of this section are adapted from Kafura & Shen [1977] and Martel [1985]. The second paper describes the feasibility algorithm and also shows that $C_{\max}$ can be found in $O(mn\log^2 m)$ time by looking more carefully at when the feasibility algorithm needed to be called using Megiddo's approach. Lai & Sahni [1984] showed how to minimize $L_{\max}$ for identical speed processors with memory constraints, and also considered settings with release times and due-dates [1983].

10.7. *Linear Programming Formulations for Unrelated Machines.* The linear programming formulations presented in this section are due to Lawler & Labetoulle [1978]. For fixed m, it seems possible that the linear program for $R|pmtn|C_{\max}$ can be solved in $O(n)$ time. Certainly this is true in the case of $m=2$ [Gonzalez, Lawler & Sahni, 1990]. The algorithm for the preemptive scheduling of open shops is due to Gonzalez & Sahni [1976].