Contents

3

Minmax criteria

Eugene L. Lawler
University of California, Berkeley

Jan Karel Lenstra
Centrum Wiskunde & Informatica

David B. Shmoys
Cornell University

Pity poor Bob Cratchit, his in-basket piled high and the holidays upon him. The in-basket demands the completion of n jobs, each with its own due date. Cratchit has no idea how long any of the jobs will take. But he knows full well that he must complete all the jobs on time, else his employment at Scrooge, Ltd. will be terminated. In what order should he do the jobs?

J. R. Jackson supplied the answer in 1955, too late for Cratchit, but early enough to qualify as one of the first optimization algorithms in scheduling theory. According to Jackson's Earliest Due Date (EDD) rule, if there is any way to complete all of the jobs on time, then it can be accomplished by performing the jobs in order of nondecreasing due dates. Furthermore, such an EDD order minimizes maximum lateness, whether or not it meets all of the due dates. Thus, Cratchit's $1||L_{\max}$ problem is solved by simply putting the jobs in EDD order.

The EDD rule, as given by Jackson, is extremely simple. But it has many implications that are quite nonobvious. With prior modification of due dates, the EDD rule can be applied to solve $1|prec|L_{\max}$. A preemptive extension of the rule solves $1|pmtn,r_j|L_{\max}$ and $1|pmtn,prec,r_j|L_{\max}$. Furthermore, a generalization of the EDD rule, the Least Cost Last rule, solves the general problems $1|prec|f_{\max}$ and $1|pmtn,prec,r_j|f_{\max}$ in $O(n^2)$ time.

Given these successes, one might suppose that other problems can be solved efficiently by some sort of elaboration of the EDD rule as well. Indeed, a complicated algorithm that involves an iterative application of the EDD rule solves the case of equal processing times, $1|r_j, p_j = p|L_{\max}$, in polynomial time. This is probably as far as we can get, however: the unrestricted problem $1|r_j|L_{\max}$ is strongly NP-hard. In later chapters, we shall have many occasions to reflect on the fact that preemptive scheduling problems are often easier than their nonpreemptive counterparts, and never known to be harder. In the case of single-machine minmax problems, nonuniform release dates are innocuous when preemption is permitted, but calamatous when it is not.

When faced with an NP-hard problem, there are various things we can do. First, we can try to find interesting and useful special cases that are solvable in polynomial time, such as the ones mentioned above. Second, we can devise approximation algorithms with performance guarantees, as we will do for the related 'head-body-tail problem'. Third, we can avail ourselves of enumerative methods.

3.1. Earliest Due Date and Least Cost Last rules

The *Earliest Due Date (EDD) rule* schedules the jobs without idle time in order of nondecreasing due dates; ties are broken arbitrarily. The resulting schedule rule is called an EDD schedule.

Theorem 3.1 [Jackson's EDD rule]. *Any EDD schedule is optimal for the problem* $1||L_{\max}$.

Proof. Let π be the ordering of the jobs in an EDD schedule, i.e., job $\pi(j)$ is the jth job scheduled, and let π^* be an optimal ordering. Suppose that $\pi \neq \pi^*$. Then there exist jobs j and k such that job k immediately precedes job j in π^*, but job j precedes job k in π, with $d_j \leq d_k$. Interchanging the positions of jobs k and j in π^* does not increase $L_{\max}$, and decreases the number of pairs of consecutive jobs in π^* that are in the opposite order in π. It follows that a finite number of such interchanges transforms π^* into π, so that π is optimal.

Corollary 3.2. *For any instance of* $1||L_{\max}$*, all due dates can be met if and only if they are met in any EDD schedule.*

An EDD schedule can be found by sorting the due dates, which requires $O(\log n)$ time. Sometimes $O(n)$ time suffices, as in the special case of equal processing times (see Exercises 3.3 and 3.4).

It is perhaps surprising that the EDD rule also solves the problem $1|prec|L_{\max}$, provided that the due dates are first modified to reflect the precedence constraints. Similar modifications schemes will be encountered in Chapter 11 as well, but they will not be as simple as this one.

If $d_j < d_k$ whenever $j \to k$, then any EDD schedule is consistent with the given precedence constraints and thereby optimal for $1|prec|L_{\max}$. Observe that, if $j \to k$, we may set

$$d_j := \min\{d_j, d_k - p_k\} \tag{3.1}$$

without increasing the value of $L_{\max}$ in any feasible schedule, since

$$L_k = C_k - d_k \geq C_j + p_k - d_k = C_j - (d_k - p_k).$$

Let $G = (\{1,\ldots,n\}, A)$ be an acyclic digraph that represents the precedence relation. Consider updating the due dates in the following systematic manner: in each iteration, select a vertex $k \in V$ of outdegree 0 (that is, a *sink*) and, for each arc (j,k), update d_j using (3.1); when all such arcs have been processed, delete them and vertex k from G.

We claim that, after executing this algorithm, $d_j < d_k$ whenever $(j,k) \in A$. At some stage, vertex k becomes a sink and is subsequently selected in a particular iteration. Immediately after that iteration, the update (3.1) implies that $d_j < d_k$; since d_k remains unchanged from this point on and d_j can only decrease, our claim is valid. Hence, by blindly applying the EDD rule to the modified due dates, we automatically obtain a feasible schedule, which must therefore be optimal.

The due date modification algorithm takes $O(n + |A|)$ time. Since the EDD rule takes $O(n \log n)$ time, we can solve $1|prec|L_{\max}$ in $O(n \log n + |A|)$ time. Moreover, the problem can be solved without any knowledge of the processing times of the jobs (see Exercise 3.5).

There is an important symmetry between the problems $1||L_{\max}$ and $1|r_j|C_{\max}$. The former problem can be viewed as minimizing the amount by which the due dates must be uniformly increased so that there exists a schedule in which each job meets its (modified) due date. For the latter problem, we may also restrict attention to schedules in which the jobs are processed without idle time between them, since in each feasible schedule of length $C_{\max}$ the start of processing can be delayed until $C_{\max} - \sum_j p_j$. The problem can then be viewed as minimizing the amount by which the release dates must be uniformly decreased so that there exists a schedule of length $\sum_j p_j$ in which no job starts before its (modified) release date. These two problems are mirror images of each other; one problem can be transformed into the other by letting time run in reverse.

More precisely, an instance of $1|r_j|C_{\max}$ can be transformed into an instance of $1||L_{\max}$ by defining due dates $d_j = K - r_j$ for some integer K. (One may choose $K \geq \max_j r_j$ in order to obtain nonnegative due dates.) An optimal schedule for the former problem is found by reversing the job order in an optimal schedule for the latter. If we wish to solve $1|prec, r_j|C_{\max}$, then we must also reverse the precedence constraints, i.e., make $j \rightarrow k$ in the $L_{\max}$ problem if and only if $k \rightarrow j$ in the $C_{\max}$ problem. It follows that the algorithm for $1|prec|L_{\max}$ also solves $1|prec, r_j|C_{\max}$, in $O(n \log n + |A|)$ time. In Section 3.5 we will make the symmetry between release dates and due dates more explicit by introducing the 'head-body-tail' formulation of $1|r_j|L_{\max}$.

While $1|prec|L_{\max}$ can be solved without knowledge of the processing times, this is definitely not true for the more general problem $1|prec|f_{\max}$. Nevertheless, the latter problem can be solved in $O(n^2)$ time, by a generalization of the EDD rule that we shall call the *Least Cost Last rule*.

Let $N = \{1,2,\ldots,n\}$ be the set of jobs, and let $L \subseteq N$ be the set of jobs without successors. For any subset $S \subseteq N$, let $p(S) = \sum_{j\in S} p_j$, and let $f^*_{\max}(S)$ denote the cost of an optimal schedule for the subset of jobs indexed by S. We may assume that the machine completes processing by time $p(N)$. Since one of the jobs in L must be scheduled last, we have

$$f^*_{\max}(N) \geq \min_{j\in L} f_j(p(N)).$$

Since omitting one of the jobs cannot increase the optimal cost, we also have

$$f^*_{\max}(N) \geq f^*_{\max}(N-\{j\}) \text{ for all } j \in N.$$

Now let job l with $l \in L$ be such that

$$f_l(p(N)) = \min_{j\in L} f_j(p(N)).$$

We have

$$f^*_{\max}(N) \geq \max\{f_l(p(N)), f^*_{\max}(N-\{l\})\}.$$

But the right-hand side of this inequality is precisely the cost of an optimal schedule subject to the condition that job l is processed last. It follows that there exists an optimal schedule in which job l is in the last position. Since job l is found in $O(n)$ time, repeated application of this Least Cost Last rule yields an optimal schedule in $O(n^2)$ time.

Theorem 3.3. *The Least Cost Last rule solves the problem* $1|prec|f_{\max}$ *in* $O(n^2)$ *time.* □

Theorem 3.3 can also be proved by a straightforward interchange argument, but the method used here will be applied to the problem $1|pmtn, prec, r_j|f_{\max}$ in Section 3.2.

Exercises

3.1. Define the *earliness* of job j as $E_j = d_j - C_j$. Use an interchange argument similar to the one in the proof of Theorem 3.1 to show that the maximum earliness $E_{\max}$ is minimized by scheduling the jobs j in order of nonincreasing $p_j - d_j$.
3.2. Prove Corollary 3.2
3.3. Show that $1|p_j \leq p|L_{\max}$ can be solved in $O(np)$ time by appropriate sorting techniques.
3.4. Show that $1|p_j = p|L_{\max}$ can be solved in $O(n)$ time. (Hint: Find $d_{\min} = \min_j d_j$. For each interval $[d_{\min} + (k-1)p, d_{\min} + kp)$ $(k = 1,\ldots,n)$, count the number of jobs with due dates in the interval and record the largest due date in the interval. Use these values to compute the maximum lateness of an EDD schedule. Then find a schedule that meets this bound.)

3.5. Show that $1|prec|L_{\max}$ can be solved without any knowledge of the p_j values. Devise a two-job example to show that this is not the case for the maximum weighted lateness problem $1||wL_{\max}$, where each job j has a weight w_j and $wL_{\max} = \max_j w_j L_j$.
3.6. Prove Theorem 3.3 using an interchange argument.
3.7. Suppose that processing times are allowed to be negative. (This may seem less unnatural if one views p_j as the amount of a resource that is either consumed, if $p_j > 0$, or produced, if $p_j < 0$.) Processing begins at time 0, and each completion time C_j is the total processing time of the jobs scheduled up to (and including) job j. Extend the Least Cost Last rule to solve this generalization of $1||f_{\max}$ in $O(n^2)$ time. (Hint: It is possible to partition an instance of the generalized problem into two instances of the ordinary kind.)
3.8. Prove that the precedence-constrained version of the problem posed in Exercise 3.7 is NP-hard.

3.2. Preemptive EDD and Least Cost Last rules

We wish to extend the EDD rule to deal with nonuniform release dates. In this section we will consider such an extension for the case in which preemption is permitted. A nonpreemptive extension of the EDD rule is introduced in the next section.

The *preemptive EDD rule* solves the problem $1|pmtn,r_j|L_{\max}$ by scheduling the jobs in time, with a decision point at each release date and at each job completion time. A job j is said to be *available* at time t if $r_j \leq t$ and it has not yet completed processing. At each decision point, from among all available jobs, choose to process a job with the earliest due date. If no jobs are available at a decision point, schedule idle time until the next release date.

Observe that the preemptive EDD rule creates preemptions only at release dates, but not at the first one. That is, if there are k distinct release dates, the rule introduces at most $k-1$ preemptions. If all release dates are equal, then there are no preemptions and the rule generates an ordinary EDD schedule, as described in the previous section.

We shall prove that the preemptive EDD rule produces an optimal schedule. For $S \subseteq N$, let $r(S) = \min_{j\in S} r_j$, $p(S) = \sum_{j\in S} p_j$, and $d(S) = \max_{j\in S} d_j$.

Lemma 3.4. *For any instance of* $1|pmtn,r_j|L_{\max}$ *or* $1|r_j|L_{\max}$, $L^*_{\max} \geq r(S)+p(S)-d(S)$ *for each* $S \subseteq N$.

Proof. Consider an optimal schedule, and let job j be the last job in S to finish. Since none of the jobs in S can start processing earlier than $r(S)$, $C_j \geq r(S)+p(S)$. Furthermore, $d_j \leq d(S)$, and so $L^*_{\max} \geq L_j = C_j - d_j \geq r(S)+p(S)-d(S)$. □

Theorem 3.5. *The preemptive EDD rule solves the problem* $1|pmtn,r_j|L_{\max}$, *with* $L^*_{\max} = \max_{S\subseteq N} r(S)+p(S)-d(S)$.

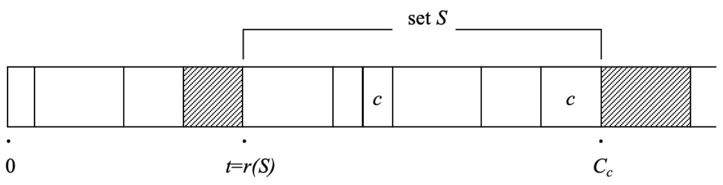


Figure 3.1. Schedule obtained by the preemptive EDD rule.

Proof. We will show that the preemptive EDD rule produces a schedule that meets the lower bound of Lemma 3.4 for an appropriately chosen set S.

Consider the schedule produced by the preemptive EDD rule (cf. Figure 3.1). Let job c be a *critical* job, that is, $L_c = L_{\max}$. Let t be the latest time such that each job j processed in the interval $[t, C_c]$ has $r_j \geq t$, and let S be the subset of jobs processed in this interval. The interval contains no idle time: if there were an idle period, then its end satisfies the criterion used to choose t and is later than t; hence, $p(S) \geq C_c - t$. Further, $r_j \geq t$ for each $j \in S$ and, since the machine is never idle in $[t, C_c]$, some job starts at time t; hence, $r(S) = t$. Finally, we show that $d(S) = d_c$. Suppose that this is not true, and let t' denote the latest time within $[t, C_c]$ at which some job j with $d_j > d_c$ is processed. Hence, each job k processed in $[t', C_c]$ has $d_k \leq d_c < d_j$; since the preemptive EDD rule did not preempt job j to start job k, we have $r_k \geq t'$. But this implies that t' should have been selected instead of t, which is a contradiction. Therefore, $L_{\max} = L_c = C_c - d_c \leq r(S) + p(S) - d(S)$, which proves the theorem. □

The preemptive EDD rule is easily implemented to run in $O(n \log n)$ time with the use of two priority queues. Jobs are extracted from one queue in order of nondecreasing release dates. The second queue contains the jobs that are currently available. The available job with smallest due date is extracted from this queue, and reinserted upon preemption.

It is particularly important to note that the preemptive EDD rule is *on-line*, in the sense that it only considers available jobs for the next position in the schedule, without requiring any knowledge of their processing times and of the release dates of future jobs. This makes it an ideal rule for scheduling tasks with due dates on a real-time computer system.

After appropriate modification of both release dates and due dates, the preemptive EDD rule also solves the problem $1|\,pmtn, prec, r_j|L_{\max}$. The condition that must be satisfied is that $r_j < r_k$ and $d_j < d_k$ whenever $j \rightarrow k$. In Section 3.1 we have already given a procedure for modifying the due dates. For the release dates, observe that, if $j \rightarrow k$, we may set

$$r_k := \max\{r_k, r_j + p_j\}$$

without changing the set of feasible schedules. We leave it to the reader to verify that $1|\,pmtn, prec, r_j|L_{\max}$ can be solved in $O(n \log n + |A|)$ time, where A is the arc set of the precedence digraph.

In some cases, the preemptive EDD rule produces a schedule without preemptions, which is therefore optimal for the nonpreemptive variant as well. This observation applies to the cases of equal release dates ($1||L_{\max}$), equal due dates ($1|r_j|C_{\max}$), and unit processing times ($1|r_j, p_j = 1|L_{\max}$); in the last case, our assumption that all release dates are integral is essential. The preemptive EDD rule also solves a common generalization of the cases of equal release dates and equal due dates, which occurs when $d_j \leq d_k$ whenever $r_j < r_k$; such release and due dates are said to be *similarly ordered* (see Exercise 3.9).

The preemptive EDD rule produces an optimal schedule with no *unforced idle time*; idle time is unforced if there is an available job. A significant consequence of this observation is the following.

Theorem 3.6. *For any instance of* $1|\,pmtn, prec, r_j|f_{\max}$ *and* $1|\,pmtn, prec, r_j|\sum f_j$, *there exists an optimal schedule with no unforced idle time and with at most* $n-1$ *preemptions.*

Proof. Given an optimal schedule in which each job j has a completion time C_j^*, create an instance of $1|\,pmtn, prec, r_j|L_{\max}$ by setting $d_j = C_j^*$. These due dates satisfy the property that $d_j < d_k$ whenever $j \rightarrow k$. Modify the release dates so that they also conform to the precedence constraints, and apply the preemptive EDD rule. In the resulting schedule, each job j completes no later than C_j^*. Hence, the new schedule is also optimal and the theorem is proved. □

We shall now obtain a preemptive extension of the Least Cost Last rule that will enable us to solve $1|\,pmtn, r_j|f_{\max}$ and even $1|\,pmtn, prec, r_j|f_{\max}$ in $O(n^2)$ time.

Taking our cue from Theorem 3.6 which states that there is an optimal schedule with no unforced idle time, we first notice that we must work with a certain block structure. Consider running the following algorithm on an instance of $1|\,pmtn, r_j|f_{\max}$: as long as jobs are available, choose one of them and schedule it to completion; otherwise, find the minimum release date of an unscheduled job, create an idle period until then, and continue from there. This algorithm partitions N into *blocks*, where a block consists of a maximal subset of jobs processed continuously without idle time. Call this algorithm *Find Blocks*(N). Recall that, for any subset $S \subseteq N$, $r(S) = \min_{j \in S} r_j$ and $p(S) = \sum_{j \in S} p_j$, and define $t(S) = r(S) + p(S)$. It follows that, if we find the blocks $B_1, \ldots, B_k$ in that order, then B_i starts processing at time $r(B_i)$ and completes processing at time $t(B_i) < r(B_{i+1})$. Using this information, we obtain the following corollary of Theorem 3.6.

Corollary 3.7. *If* $B_1, \ldots, B_k$ *are the blocks of an instance of* $1|\,pmtn, r_j|f_{\max}$, *then there exists an optimal schedule in which the jobs of* B_i $(i = 1, \ldots, k)$ *are scheduled from* $r(B_i)$ *to* $t(B_i)$ *without idle time.* □

Corollary 3.7 implies that we can find an optimal schedule for each block, and then combine them to obtain an optimal schedule for the entire instance. Furthermore, the algorithm to solve $1|\,pmtn, r_j|f_{\max}$ for a block can now follow a plan nearly identical to the one used for $1||f_{\max}$. Let $f^*_{\max}(B)$ denote the cost of an optimal schedule for the subset of jobs indexed by B. As above, we may assume that the machine completes processing block B at time $t(B)$. Since some job in B must finish last, we have

$$f^*_{\max}(B) \geq \min_{j\in B} f_j(t(B)).$$

Again, it is clear that

$$f^*_{\max}(B) \geq f^*_{\max}(B-\{j\}) \quad \text{for all } j \in B.$$

Now choose job l with $l \in B$ such that

$$f_l(t(B)) = \min_{j\in B} f_j(t(B)).$$

We have

$$f^*_{\max}(B) \geq \max\{f_l(t(B)), f^*_{\max}(B-\{l\})\}. \tag{3.2}$$

We will give a recursive algorithm that produces a schedule whose value $f_{\max}(B)$ meets this lower bound.

```
Algorithm  Schedule Block($B$)
if $B = \emptyset$  then return the empty schedule;
$l :=$ argmin$_{j\in B} f_j(t(B))$;
call  Find Blocks($B - \{l\}$);
schedule job $l$ in the idle time generated between $r(B)$ and $t(B)$;
for each block $B_i$ found  call  Schedule Block($B_i$).
```

Note that, in the resulting schedule, job l is processed only if no other job is available.

We shall prove by induction on the number of jobs in $|B|$ that *Schedule Block* constructs an optimal schedule. Clearly, this is true if $|B| = 0$. Otherwise, consider a *critical* job, i.e., one whose completion cost equals $f_{\max}(B)$. If job l is critical, then

$$f_{\max}(B) \leq f_l(t(B)),$$

since job l is certainly completed by $t(B)$. If one of the other jobs in B is critical, then the correctness of the algorithm for smaller blocks, along with Corollary 3.7, implies that

$$f_{\max}(B) \leq f^*_{\max}(B-\{l\}).$$

It follows that the algorithm produces a schedule whose value matches the lower bound (3.2) on the optimal value. Hence, the schedule is optimal.

To analyze the running time of this algorithm, we must be a bit more careful about a few implementation details. Let $T(n)$ be the worst-case running time of *Schedule Block* on an input with n jobs. We start by reindexing the jobs so that they are ordered by nondecreasing release dates. It then takes linear time to identify job l, and also linear time to run *Find Blocks*, since the jobs are nicely ordered. In order to call *Schedule Block* recursively on each block found, we must make sure that the jobs in each block are numbered appropriately. However, we can separate the original sorted list of n jobs into sorted lists for each block in linear time. Thus, if $n_1, \ldots, n_k$ are the sizes of the blocks found, then

$$T(n) \leq T(n_1) + \cdots + T(n_k) + O(n),$$

where $n_1 + \cdots + n_k = n - 1$, and $T(1) = O(1)$. This implies that $T(n) = O(n^2)$, as is easily verified by induction.

It is also important to analyze the number of preemptions generated by this algorithm. In the schedule constructed, job l can be preempted only at times $r(B')$ for the blocks B' of $B - \{l\}$. This implies that the schedule contains at most $n - 1$ preemptions, and it is not hard to construct instances for which every optimal schedule has that many preemptions (see Exercise 3.10).

Theorem 3.8. *The problem* $1|\,pmtn, r_j|f_{\max}$ *can be solved in* $O(n^2)$ *time in such a way that the optimal schedule has at most* $n - 1$ *preemptions.*

Exercises

3.9. Show that the preemptive EDD rule solves $1|r_j|L_{\max}$ in $O(n \log n)$ time in case the release and due dates are similarly ordered, i.e., $d_j \leq d_k$ whenever $r_j < r_k$.

3.10. Give a class of instances of $1|\,pmtn, r_j|f_{\max}$ for which every optimal schedule has $n - 1$ preemptions.

3.11. Show how to extend Theorem 3.8 to apply to $1|\,pmtn, prec, r_j|f_{\max}$. (Hint: After modifying the release dates in the usual way, the main difficulty is in identifying the correct job l, which is now restricted to be a job that has no successors within its block.)

3.12. Apply the algorithm of Theorem 3.8 to find an optimal schedule for the instance of $1|\,pmtn, prec, r_j|f_{\max}$ given in Figure 3.2.

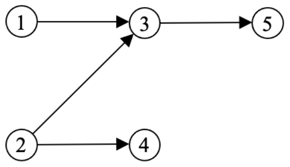
3.13. Show that the preemptive Least Cost Last rule solves $1|\,prec, r_j, p_j = 1|f_{\max}$ in $O(n^2)$ time.

3.14. Devise and prove a minmax theorem that generalizes Theorem 3.5 to $1|\,pmtn, r_j|f_{\max}$.

3.3. A polynomial-time algorithm for jobs of equal length

We have seen that the preemptive EDD rule solves $1|r_j|L_{\max}$ in $O(n \log n)$ time when all r_j are equal, when all d_j are equal, when all $p_j = 1$, and even when the r_j and d_j are similarly ordered. It is not easy to find additional special cases for which the preemptive EDD rule creates no preemptions.

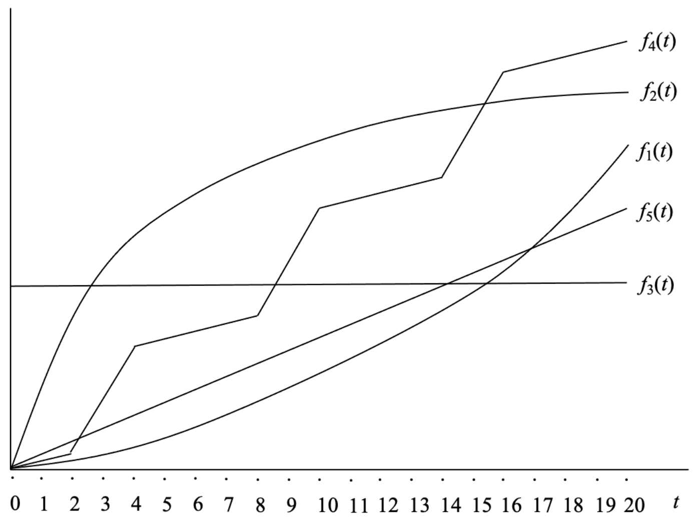
We can modify the preemptive rule so that it is forced to produce a nonpreemptive schedule. With luck, some new special cases can be solved by the *nonpreemptive EDD rule*: schedule the jobs in time, with a decision point at the beginning of each block and at each job completion time. At each decision point, choose to process an



(a) Precedence constraints.

j	1	2	3	4	5
r_j	0	2	0	8	14
p_j	4	2	4	2	4

(b) Release dates and processing times.



(c) Cost functions.

Figure 3.2. Five-job instance of $1|pmtn, prec, r_j|f_{\max}$ for Exercise 3.12.

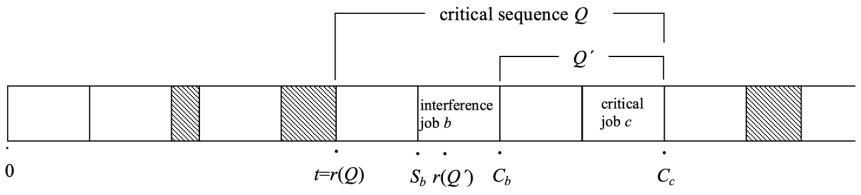


Figure 3.3. Schedule obtained by the nonpreemptive EDD rule.

available job with the earliest due date. If no jobs are available at a decision point, schedule idle time until the next release date.

Unfortunately, it is also not easy to find new cases for which the direct application of the nonpreemptive EDD rule can be guaranteed to yield an optimal schedule. However, the rule is invaluable as a component of more elaborate algorithms, for solving special cases, for obtaining near-optimal solutions, and for computing optimal schedules by enumeration. In this section, we shall develop an algorithm for the special case of equal processing times, $1|r_j, p_j = p|L_{\max}$. This algorithm, unlike the simple EDD rule, is off-line, and unlike the special cases we have seen thus far, $1|r_j, p_j = p|L_{\max}$ is not solvable by an on-line algorithm (see Exercise 3.15).

The reader should observe that the case in which all p_j are equal to an arbitrary positive integer p is very different from the case in which all $p_j = 1$. It is possible to rescale time so that $p = 1$. But then the release dates become rational numbers, contrary to the ground rule under which we asserted that the preemptive EDD rule solves $1|r_j, p_j = 1|L_{\max}$.

There *are* instances of the general $1|r_j|L_{\max}$ problem for which the solution delivered by the nonpreemptive EDD rule can be proved to be optimal. Recall that, for $S \subseteq N$, $r(S) = \min_{j\in S} r_j$, $p(S) = \sum_{j\in S} p_j$, and $d(S) = \max_{j\in S} d_j$. Consider a schedule delivered by the nonpreemptive EDD rule (cf. Figure 3.3). Let job c be a *critical* job, that is, $L_c = \max_j L_j$, and let t be the earliest time such that the machine is not idle in the interval $[t, C_c]$. The sequence of jobs Q processed in this interval is called a *critical sequence*. None of the jobs in Q has a release date smaller than t; that is, $r(Q) = t$. Suppose that, in addition, none of the jobs in Q has a due date larger than d_c; that is, $d(Q) = d_c$. In that case, the schedule must be optimal: its maximum lateness is equal to $r(Q) + p(Q) - d(Q)$, which, by Lemma 3.4, is a lower bound on the optimum.

This proof of optimality, however, is a happy turn of events, and many instances will not have the property that $d(Q) = d_c$. For those instances, there will be a job b with $b \in Q$ and $d_b > d_c$. Any such job interferes with the optimality proof, and the one scheduled last in Q is called an *interference* job. This notion will be useful in several aspects of algorithm design for $1|r_j|L_{\max}$. We shall now apply it to derive a polynomial-time algorithm for $1|r_j, p_j = p|L_{\max}$.

We will first consider the decision version of this problem and give a polynomial-time algorithm to decide whether, for a given integer l, there exists a schedule with value $L_{\max} \leq l$. This is equivalent to the problem of deciding if every job can meet its due date, since a schedule satisfies $L_{\max} \leq l$ if and only it satisfies $L_{\max} \leq 0$ with respect to modified due dates $d_j + l$ $(j = 1, \ldots, n)$. Thus, we view the due dates as *deadlines* and call a schedule in which every job is on time a *feasible* schedule.

Suppose that the nonpreemptive EDD rule constructs an infeasible schedule. Find a critical job c and its critical sequence Q. If there is no interference job in Q, then Lemma 3.4 implies that no feasible schedule exists. Suppose that there is an interference job b, and let S_b be its starting time. Focus on that part of Q that follows job b; call this part Q'. The definition of an interference job implies that $d(Q') = d_c < d_b$. In spite of this, the nonpreemptive EDD rule selected job b for processing at time S_b. It follows that $r(Q') > S_b$. Consider any schedule in which a job starts in the interval $[S_b, r(Q'))$. Clearly, this cannot be a job in Q'. Since all processing times are equal, the jobs in Q' are delayed at least as much as in the schedule produced by the nonpreemptive EDD rule. Hence, some job in Q' must be late, and the schedule is infeasible. We conclude that no feasible schedule has any job starting in $[S_b, r(Q'))$, and therefore call that interval a *forbidden region*.

The main idea of the algorithm is to repeat this approach. We apply the nonpreemptive EDD rule, never starting jobs in any forbidden region that has been identified before. There are three possible outcomes: either a feasible schedule is found, or the instance is proved infeasible, or a new forbidden region is identified. In the first two cases, we are done; in the last case, we repeat. The running time analysis will rely on the fact that the number of forbidden regions is limited.

Before stating the algorithm in detail, we have to modify our definition of a critical sequence, since idle time that is caused by a forbidden region does not count. Given a critical job c, its critical sequence Q consists of the jobs processed during the maximal interval $[t, C_c)$ such that all idle time within the interval occurs within forbidden regions. The decision algorithm is now as follows.

```
$\mathcal{F} := \emptyset$;                                * initialize the set of forbidden regions *
**until** a feasible schedule is found                           * produce the next schedule *
  $N := \{1, \ldots, n\}$;                                 * $N$ is the set of unscheduled jobs *
  $t := 0$;                                * $t$ is the time at which the next job may start *
  **while** $N \neq \emptyset$                                * some job is not yet scheduled *
    **while** there exists $F \in \mathcal{F}$ such that $t \in F = [t_1, t_2)$
      $t := t_2$;                               * advance $t$ beyond forbidden regions *
    $A := \{j \in N | r_j \leq t\}$;                        * $A$ is the set of available jobs at $t$ *
    select $j \in A$ for which $d_j$ is minimum;                           * choose next job *
    $S_j := t$;  $C_j := t + p$;                                         * schedule job $j$ *
    $N := N - \{j\}$;                                        * mark job $j$ as scheduled *
    $t := t + p$;                                                     * advance $t$ to $C_j$ *
  **if** some job is late
  **then**                         * prove infeasibility or find new forbidden region *
    find a critical job $c$;
```

```
**if** there is no interference job **then** output 'infeasible' and **halt**;
$b :=$ the interference job in the critical sequence ending with $c$;
$Q' := \{j | S_b < S_j \le S_c\}$;
$\mathcal{F} := \mathcal{F} \cup \{[S_b, r(Q'))\}$.
```

The reader may wonder why this algorithm must stop. Observe that the upper endpoint of a forbidden region is always a release date. Hence, each job has a starting time of the form $r_j + sp$, where s is an integer between 0 and $n-1$. Since the lower endpoints of the forbidden regions are starting times, there are $O(n^2)$ possible lower endpoints, and after that many iterations we cannot possibly generate a new forbidden region. The algorithm must terminate within $O(n^2)$ iterations.

In order to prove that the algorithm is correct, we need a lemma. Suppose that the jobs have identical release dates and identical processing times, and that we just wish to minimize schedule length; however, no job is allowed to start in any of a number of given forbidden regions. This problem can be solved in a simple way.

Lemma 3.9 [Simple algorithm]. *Given a set $\mathcal{F}$ of forbidden regions, the problem $1|r_j = r, p_j = p|C_{\max}$ is solved by repeatedly choosing a job and starting it at the earliest possible time that is not contained in any interval in $\mathcal{F}$.*

Proof. Suppose that the lemma is incorrect. Choose a counterexample with the smallest number of jobs. Since the jobs are identical, we may assume that they are ordered $1, \ldots, n$ in both the schedule produced by the simple algorithm and the optimal schedule. Let C_j and C_j^*, respectively, denote the completion time of job j in each of these schedules. By the choice of counterexample, $C_j \le C_j^*$ for $j = 1, \ldots, n-1$. In particular, since $C_{n-1} \le C_{n-1}^*$, any time at which the optimal schedule may start job n is also available for the simple algorithm. Hence, $C_n \le C_n^*$, which is a contradiction. □

We claim that the decision algorithm has the following invariant property: *for any feasible schedule, no job starts at a time contained in a forbidden region $F \in \mathcal{F}$.* We shall first use this property to show that the algorithm is correct, and then establish its validity.

Assuming that the invariant property holds, it will be sufficient for us to show that no feasible schedule exists whenever the algorithm concludes that this is so. Suppose the algorithm halts without having produced a feasible schedule. In that case, the final schedule found has a critical sequence Q with no interference job. By definition, the jobs in Q are the only ones scheduled between $r(Q)$ and their maximum completion time, $\max_{j \in Q} C_j > d(Q)$. We shall invoke Lemma 3.9 to show that this set of jobs cannot be completed earlier than in the schedule found. Observe that the schedule of the critical sequence contains no idle time outside of the forbidden regions. Hence, this schedule could have been produced by the simple algorithm on a modified instance in which each release date is set equal to $r(Q)$. By Lemma 3.9, the maximum completion time of this schedule is optimal. However, it exceeds $d(Q)$. It follows that, in any schedule, the job in Q that finishes last must be late.

We still must verify the invariant property. It certainly holds initially, when there are no forbidden regions, and we have already seen that it holds when the first forbidden region has been found. Assume that the invariant property holds for the first k forbidden regions found. Suppose that, in iteration $k+1$, an infeasible schedule is found, with a corresponding forbidden region $[S_b, r(Q'))$. Extract the schedule for the jobs in $Q' \cup \{b\}$. There is no idle time outside of a forbidden region in this schedule from S_b until all jobs in $Q' \cup \{b\}$ are completed. Thus, if the release dates of these jobs were to be modified to S_b, then this schedule is a possible output of the simple algorithm. Its completion time exceeds $d(Q')$ and, by Lemma 3.9, no earlier maximum completion time is possible for this subinstance.

Suppose that there exists a feasible schedule with $S_j \in [S_b, r(Q'))$ for some job j. Of course, $j \notin Q'$. From this feasible schedule, extract the schedule for the jobs in $Q' \cup \{j\}$. By the invariant property, no job starts during any of the k forbidden regions already identified. The schedule remains feasible if we change each release date to S_b. This makes job j identical to job b. But it now follows that the completion time of the schedule must exceed $d(Q')$, which contradicts its feasibility. This completes the correctness proof of the algorithm.

As for the running time of the algorithm, we already know that there are $O(n^2)$ iterations. We still must figure out how to implement each iteration. The jobs are initially stored in a priority queue, ordered by their release dates. Whenever the available set A is updated, we repeatedly test whether the minimum element in the queue is no more than the current time parameter t and, if so, extract the minimum element from the queue. The available set is stored in another priority queue, ordered by due date. With these data structures, each iteration takes $O(n \log n)$ time. Hence, the decision algorithm runs in $O(n^3 \log n)$ time.

We now have solved the decision problem. In order to solve the optimization problem, we can use a naive bisection search to find $L^*_{\max}$ but there is a better way. Observe that any lateness is the difference between a starting time and a due date. Since there are only $O(n^2)$ possible starting times and $O(n)$ due dates, there are $O(n^3)$ possible values of $L_{\max}$. Even using brute force, we can calculate all of these, sort them, and then do a bisection search to find $L^*_{\max}$. As a result, $O(\log n)$ iterations suffice to find the optimal solution.

Theorem 3.10. *The problem $1|r_j, p_j = p|L_{\max}$ can be solved in $O(n^3 \log^2 n)$ time by an iterated version of the nonpreemptive EDD rule.*

We should make a final comment about implementing this algorithm. While we have stipulated that each forbidden region is derived from a critical job, the proof uses nothing more than that the job is late. Hence, whenever the algorithm schedules a job, it should check if that job is late. If it is, the current iteration ends, either by identifying an interference job relative to that job, or by concluding that the instance is infeasible.

The algorithm is easily extended to solve $1|prec, r_j, p_j = p|L_{\max}$ as well. As we have indicated in the previous sections, the release and due dates of a problem instance can be modified in such a way that $r_j < r_k$ and $d_j < d_k$ whenever $j \rightarrow k$, without changing the set of feasible schedules and their objective values. The

$n = 6, p = 3$

j	1	2	3	4	5	6
r_j	0	1	2	6	13	15
$\bar{d}_j$	12	22	5	10	20	19

Figure 3.4. Instance of $1|r_j, p_j = p, \bar{d}_j|-$ for Exercise 3.16.

nonpreemptive EDD rule, when run on the modified instance, will automatically respect the precedence constraints. Hence, any schedule generated by the algorithm will satisfy the precedence constraints.

Exercises

3.15. Show that there is no on-line algorithm to solve $1|r_j, p_j = p|L_{\max}$.

3.16. Apply the algorithm of Theorem 3.10 to find a feasible schedule for the instance of $1|r_j, p_j = p, \bar{d}_j|-$ given in Figure 3.4. What happens if $\bar{d}_6$ is changed to 18?

3.17. Consider the special case of $1|r_j|L_{\max}$ in which $r_j + p_j \geq r_k$ for each pair (j,k).
(a) Show that there exists an optimal schedule in which at least $n-1$ of the jobs are in EDD order.
(b) Describe how to find an optimal schedule in $O(n^2)$ time.
(c) Describe how to find an optimal schedule in $O(n \log n)$ time.
(d) Suppose that, instead of $r_j + p_j \geq r_k$, we have $d_j - p_j \leq d_k$ for each pair (j,k). What can be said about this special case?

3.4. NP-hardness results

We will now show that one cannot expect to obtain a polynomial-time algorithm for $1|r_j|L_{\max}$ in its full generality. More precisely, $1|r_j|L_{\max}$ is strongly NP-hard.

Theorem 3.11. *$1|r_j|L_{\max}$ is NP-hard in the strong sense.*

Proof. We shall show that the problem is NP-hard in the ordinary sense, by proving that the partition problem reduces to the decision version of $1|r_j|L_{\max}$. To establish strong NP-hardness, one can give an analogous reduction from 3-partition, but this is left to the reader (see Exercise 3.18).

An instance of partition consists of integers $a_1, \ldots, a_t, b$ with $\sum_{j=1}^{t} a_j = 2b$. The

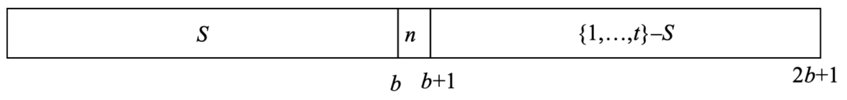


Figure 3.5. The reduction of partition to $1|r_j|L_{\max}$.

corresponding instance of $1|r_j|L_{\max}$ has $n = t+1$ jobs, defined by

$$r_j = 0,\;\; p_j = a_j,\;\; d_j = 2b+1,\quad j = 1,\ldots,t;$$

$$r_n = b,\;\; p_n = 1,\;\; d_n = b+1.$$

We claim that there exists a schedule of value $L_{\max} \leq 0$ if and only if the instance of partition is a yes-instance.

Suppose we have a yes-instance of partition; that is, there exists a subset $S \subset \{1,\ldots,t\}$ with $\sum_{j\in S} a_j = b$. We then schedule the jobs j with $j \in S$ in the interval $[0,b]$, job n in $[b, b+1]$, and the remaining jobs in $[b+1, 2b+1]$ (cf. Figure 3.5). Note that no job starts before its release date, and that no job is late.

Conversely, consider a feasible schedule of value $L_{\max} \leq 0$. In such a schedule, job n must start at time b and there cannot be any idle time. Hence, the jobs that precede job n have a total processing time of b. Since $p_j = a_j$ $(j = 1,\ldots,n)$, we conclude that we have a yes-instance of partition. □

Exercises

3.18. Prove that $1|r_j|L_{\max}$ is NP-hard in the strong sense.
3.19. Show that $1||max\{E_{\max}, L_{\max}\}$ is strongly NP-hard (cf. Exercise 3.1).

3.5. Approximation algorithms

Since $1|r_j|L_{\max}$ is NP-hard, it is natural to ask whether there are polynomial-time algorithms that find solutions guaranteed to be close to the optimum. Before answering this question, we must make sure that it is well posed. As discussed in Chapter 3, we are primarily concerned with the performance of an approximation algorithm in terms of its worst-case relative error. We would like to give an algorithm A such that, for example, $L_{\max}(\mathrm{A}) \leq 2L^*_{\max}$ for every instance. This is a reasonable goal when $L^*_{\max}$ is positive. However, when $L^*_{\max} = 0$, we must find the optimum, and when $L^*_{\max} < 0$, we must find a solution that is better than optimal!

This difficulty is not avoided if we switch to the $T_{\max}$ objective. The maximum tardiness of a schedule is always nonnegative, but it can be zero. This has the interesting consequence that, for the problem $1|r_j|T_{\max}$, we cannot hope to find a polynomial-time algorithm with any finite performance bound.

Theorem 3.12. *If there exist a polynomial-time algorithm* A *and a constant* $c > 1$ *such that, for any instance of* $1|r_j|T_{\max}$, $T_{\max}(\mathrm{A}) < cT^*_{\max}$, *then* $P = NP$. □

The proof follows directly from the reduction given in the proof of Theorem 3.11 (see Exercise 3.20).

For $L_{\max}$ problems, however, the whole notion of a c-approximation algorithm makes no sense. We can get around this difficulty by realizing that, unlike release dates and processing times, due dates have never been assumed to be nonnegative. By requiring that all due dates are *negative*, we ensure that each lateness is positive. And there do exist good approximation algorithms and, in fact, a polynomial approximation scheme for the problem $1|prec,r_j,d_j < 0|L_{\max}$.

A few words about the significance of this problem are in order. Any instance of $1|prec,r_j|L_{\max}$ can be transformed into an instance with negative due dates by subtracting a constant K from each d_j, where $K > \max_j d_j$. Both instances are equivalent to the extent that a schedule is optimal for one if and only if it is optimal for the other. Still, a good approximation for the transformed instance may be a very poor approximation for the original, since the transformation adds K to the maximum lateness of each schedule.

However, the problem $1|prec,r_j,d_j < 0|L_{\max}$ is not something we concocted for the sole purpose of being able to obtain good performance guarantees for approximation algorithms. The problem is equivalent to the *head-body-tail* problem, in which each job j is to be processed on the machine during a period of length p_j, which must be preceded by a *release time* of length r_j and followed by a *delivery time* of length q_j. A job j can only start at time $S_j \geq r_j$; it then finishes at time $C_j = S_j + p_j$ and is delivered at time $L_j = C_j + q_j$. If $j \to k$, then it is required, as usual, that $C_j \leq S_k$. The objective is to minimize the maximum job delivery time. Obviously, the two problems correspond to each other via $d_j = -q_j$. A schedule is feasible for one problem if and only if it feasible for the other, and it has the same criterion values in both formulations.

We shall encounter the head-body-tail problem in Chapters 12 and 13, where it arises quite naturally as a relaxation of flow shop and job shop scheduling problems. In that context, it is convenient to view it as a three-machine problem, with each job j having to be processed by machines 1, 2, 3 in that order, for r_j, p_j, q_j time units, respectively. Contrary to our usual assumptions, machines 1 and 3 are *non-bottleneck* machines, in that they can process any number of jobs simultaneously; machine 2 can handle at most one job at a time and corresponds to the original single machine. Precedence constraints have to be respected on machine 2 only. The objective is to minimize the maximum job completion time on machine 3.

An appealing property of the head-body-tail problem is its symmetry. An instance defined by n triples (r_j,p_j,q_j) and a precedence relation $\to$ is equivalent to an *inverse* instance defined by n triples (q_j,p_j,r_j) and a precedence relation $\to'$, where $j \to' k$ if and only if $k \to j$. A feasible schedule for one problem instance is converted into a feasible schedule for its inverse by reversing the order in which the jobs are processed. The equivalence of $1|prec|L_{\max}$ and $1|prec,r_j|C_{\max}$ is a manifestation of this symmetry.

We shall use the language and notation of the head-body-tail problem in the rest of this chapter. Since $q_j = -d_j$ for each job j, each variant of the EDD rule for $1|prec, r_j, d_j < 0|L_{\max}$ can be translated into its analogue for the head-body-tail problem by selecting the jobs in order of nonincreasing delivery times. Similarly, if $q(S) = \min_{j \in S} q_j$ for $S \subseteq N$, then we can restate Lemma 3.4 in the following way.

Corollary 3.13. *For any instance of* $1|r_j, d_j < 0|L_{\max}$*,* $L^*_{\max} \geq r(S) + p(S) + q(S)$ *for any* $S \subseteq N$*.* □

The nonpreemptive EDD rule (NEDD)

In Section 3.3 we have studied the structure of schedules produced by this rule. We now use that information to investigate its performance for instances with arbitrary processing requirements. We first derive two data-dependent bounds.

Lemma 3.14. *For any instance of* $1|r_j|L_{\max}$*, let job c be a critical job and let job b be an interference job in a critical sequence in a schedule produced by the nonpreemptive EDD rule; we then have*
(i) $L_{\max}(\text{NEDD}) - L^*_{\max} < q_c$*;*
(ii) $L_{\max}(\text{NEDD}) - L^*_{\max} < p_b$*.*

Proof. Consider a schedule produced by the nonpreemptive EDD rule. Let Q be the critical sequence corresponding to job c. By the definition of Q, the jobs in Q start processing at $r(Q)$ and are processed without idle time for $p(Q)$ time units. Hence, job c completes processing at $r(Q) + p(Q)$, and

$$L_{\max}(\text{NEDD}) = r(Q) + p(Q) + q_c.$$

Corollary 3.13 implies

$$L^*_{\max} \geq r(Q) + p(Q) + q(Q) > r(Q) + p(Q).$$

By combining these inequalities, we obtain part (i) of the lemma.

Let job b be the interference job in Q, and let Q' be the part of Q that follows job b. By the definition of b, $q_b < q_c = q(Q')$. Since job b was selected to start processing at time S_b, we have $S_b < r(Q')$, and

$$L_{\max}(\text{NEDD}) = S_b + p_b + p(Q') + q_c < r(Q') + p_b + p(Q') + q(Q').$$

Corollary 3.13 implies

$$L^*_{\max} \geq r(Q') + p(Q') + q(Q').$$

These inequalities together prove part (ii) of the lemma. □

Since $L^*_{\max} \geq \max_j q_j$, part (i) of Lemma 3.14 implies that the nonpreemptive EDD rule is a 2-approximation algorithm for $1|r_j, d_j < 0|L_{\max}$. Alternatively, since $L^*_{\max} \geq p(N)$, part (ii) also implies this result.

In the precedence-constrained case, we first modify the data so that $r_j < r_k$ and $q_j > q_k$ whenever $j \to k$; each feasible schedule for the modified instance is feasible for the original one and has the same value. When the nonpreemptive EDD rule is applied to the modified instance without precedence constraints, the resulting schedule will still respect these constraints, and has a value strictly less than twice the optimum for the modified instance without precedence constraints. Clearly, this feasible solution for the original instance of $1|\,prec, r_j, d_j < 0|L_{\max}$ is also within a factor of 2 of the optimum for the more constrained problem.

Theorem 3.15. *For any instance of* $1|\,prec, r_j, d_j < 0|L_{\max}$, $L_{\max}(\text{NEDD})/L^*_{\max} < 2$. □

It is not hard to give a family of two-job examples that show that the bounds of Lemma 3.14 and Theorem 3.15 are tight (see Exercise 3.21).

The iterated nonpreemptive EDD rule (INEDD)

In Section 3.3, an iterated version of the nonpreemptive EDD rule did succeed where just the rule itself was not sufficient. We will adopt a similar strategy here.

When applied to an instance of $1|\,prec, r_j, d_j < 0|L_{\max}$, we view the nonpreemptive EDD rule as the procedure that first preprocesses the data to ensure that $r_j < r_k$ and $q_j > q_k$ whenever $j \to k$, and then builds a feasible schedule using the proper rule.

The iterated nonpreemptive EDD rule applies this nonpreemptive EDD rule to a series of at most n instances, starting with the original instance. If the schedule obtained has a critical sequence with no interference job, then the algorithm terminates. Otherwise, there is a critical sequence Q, ending with the critical job c and containing the interference job b. Since the small delivery time q_b of job b interferes with the proof of optimality, we force job b to be processed after job c by increasing its release time r_b to r_c and reapplying the nonpreemptive EDD rule. The algorithm continues in this way, until either no interference job exists or n schedules have been generated. From among the schedules generated, we select one with minimum $L_{\max}$ value. We claim that this is a 3/2-approximation algorithm.

Note that any schedule found is feasible for the original data. The original release dates are respected, since release dates are only increased throughout the algorithm. The precedence constraints are respected, since the data are modified in each iteration.

The following two lemmas deal with complementary cases.

Lemma 3.16. *For any instance of* $1|\,prec, r_j, d_j < 0|L_{\max}$, *if there does* not *exist an optimal schedule that is feasible with respect to the modified data used in the final iteration of the iterated nonpreemptive EDD rule, then* $L_{\max}(\text{INEDD})/L^*_{\max} < 3/2$.

Proof. Consider an optimal schedule σ^* for the original instance. Focus on the last iteration in which σ^* is still feasible with respect to the modified data. (These are the data *after* any preprocessing by the nonpreemptive EDD rule in that iteration.) Let $L^*_{\max}$ denote the optimum value for both the original instance and the modified instance considered in this iteration. In the schedule obtained in this iteration, let job c be a critical job, Q its critical sequence, and job b the interference job, and let $L_{\max}$ denote its value. We have

$$L_{\max} = r(Q) + p(Q) + q_c \leq r_b + p_b + p(Q - \{b\}) + q_c.$$

As a result of this iteration, r_b is increased to r_c, and σ^* is no longer feasible. Thus, job b must precede job c in σ^*, so that

$$L^*_{\max} \geq r_b + p_b + q_c.$$

It follows that

$$L_{\max} - L^*_{\max} \leq p(Q - \{b\}).$$

Lemma 3.14(ii) implies that

$$L_{\max} - L^*_{\max} < p_b.$$

Since $L^*_{\max} \geq p(N)$, we have

$$\frac{L_{\max}}{L^*_{\max}} < 1 + \frac{\min\{p_b, p(Q - \{b\})\}}{L^*_{\max}} \leq 1 + \frac{p(N)/2}{p(N)} = \frac{3}{2}. \quad \Box$$

Lemma 3.17. *For any instance of* $1|prec, r_j, d_j < 0|L_{\max}$, *if there* does *exist an optimal schedule that is feasible with respect to the modified data used in the final iteration of the iterated nonpreemptive EDD rule, then* $L_{\max}(\text{INEDD})/L^*_{\max} < 3/2$.

Proof. The algorithm terminates for one of two reasons: either there is no interference job, or n schedules have been generated.

In the former case, the algorithm produces an optimal schedule for the final modified instance. This schedule must also be optimal for the original instance, since both instances have the same optimum value.

In the latter case, we will show that one of the schedules produced must meet the claimed bound. Observe that, in spite of the data modifications, the optimum value remains unchanged throughout the algorithm. Lemma 3.4 (ii) now implies that, in each iteration, $L_{\max} - L^*_{\max} < p_b$, where job b is the interference job. The proof would be complete if we could show that, in some iteration, $p_b \leq p(N)/2$, since $p(N) \leq L^*_{\max}$. However, this is true for all but at most one job! Could the same job be the interference job for all n iterations? No: in each iteration, another job (the critical one) is forced to precede it, and so after $n-1$ iterations it would be the last job, which cannot be an interference job. □

We combine Lemmas 3.16 and 3.17 to obtain the following theorem.

Theorem 3.18. *For any instance of* $1|prec, r_j, d_j < 0|L_{\max}$, $L_{\max}(\text{INEDD})/L^*_{\max} < 3/2$.

It is not hard to show that this bound is tight (see Exercise 3.22).

A polynomial approximation scheme

The crucial fact in the proof of Theorem 3.18 is that there is at most one job with processing time greater than $p(N)/2$. Can we obtain better performance guarantees by using the more general observation that there are at most $k-1$ jobs with a processing time greater than $p(N)/k$? Yes, this is possible indeed, at least if we are willing to restrict attention to the problem without precedence constraints.

Let k be a fixed positive integer. Call a job l *long* if $p_l > p(N)/k$. Suppose that, somehow, we know the starting time S_l^* of each long job l in an optimal schedule σ^*, as well as the value $L_{\max}^*$. We can use this information to produce a near-optimal schedule. Modify the release date and delivery time of each long job l as follows:

$$r_l := S_l^*, \;\; q_l := L_{\max}^* - (S_l^* + p_l).$$

The modified instance has the same optimal value $L_{\max}^*$ as the original instance, since σ^* is still feasible and its value is unchanged. Also, for each long job l we now have that $q_l \geq q_j$ for all of its successors j in σ^*.

Now apply the nonpreemptive EDD rule to the modified instance to obtain a schedule σ. We claim that σ has a value $L_{\max} < (1+1/k)L_{\max}^*$. Let job c be a critical job in σ, and let Q be its critical sequence. If Q does not have an interference job, then σ is optimal. Otherwise, we will show that the interference job b is not a long job; hence, $p_b \leq p(N)/k \leq L_{\max}^*/k$, and Lemma 3.14(ii) implies the claimed performance bound. Suppose, for sake of contradiction, that job b is a long job. We have $r_b < r_c$ and $q_b < q_c$. The first inequality in combination with $S_b^* = r_b$ implies that job b precedes job c in σ^*. The second inequality in combination with the above observation regarding the successors of long jobs in σ^* implies that c precedes b in σ^*. This is a contradiction.

Of course, we have no way of knowing when the long jobs start in an optimal schedule. We avoid this difficulty as follows. Suppose that, rather than the starting times of the long jobs in σ^*, we just know the positions at which they occur in σ. This knowledge is sufficient for us to reconstruct σ. All we need to do is to apply the nonpreemptive EDD rule to the n jobs, subject to the condition that, when a long job occupies the next position in σ, then that job is scheduled next.

Once again, we have no way of knowing the positions of the long jobs in σ. However, there are at most $k-1$ long jobs, and hence there are $O(n^{k-1})$ ways to assign the long jobs to positions in a sequence of n jobs. Our algorithm A_k tests each of these possibilities and chooses the best schedule. The resulting schedule is guaranteed to be at least as good as σ. Since each application of the nonpreemptive EDD rule takes $O(n \log n)$ time, A_k runs in $O(n^k \log n)$ time, which is polynomial for any positive constant k.

Theorem 3.19. *The family of algorithms* $\{\mathrm{A}_k\}$ *is a polynomial approximation scheme for* $1|r_j, d_j < 0|L_{\max}$. $\square$

This is the best result that we can reasonably expect to obtain for a strongly NP-hard problem, since the existence of a *fully* polynomial approximation scheme would imply that P=NP (cf. Chapter 2). It is not clear how the scheme should be extended to handle precedence constraints. A polynomial approximation scheme for $1|prec, r_j, d_j < 0|L_{\max}$ does exist, but is beyond the scope of this book.

Exercises

3.20. Prove Theorem 3.12. (Hint: How does algorithm A perform on the type of instance constructed in the proof of Theorem 3.11?)

3.21. Give a family of two-job instances showing that the performance bounds of the nonpreemptive EDD rule given in Lemma 3.14 and Theorem 3.15 are tight.

3.22. Prove that the performance bound of the iterated nonpreemptive EDD rule given in Theorem 3.18 is tight.

3.23. Give a tight performance analysis of the following algorithm for $1|r_j, d_j < 0|L_{\max}$: Run the nonpreemptive EDD rule on both the original instance and its inverse, and choose the better schedule.

3.24. Show that the following procedure is a 3/2-approximation algorithm for $1|r_j, d_j < 0|L_{\max}$: Run the nonpreemptive EDD rule. If there is no interference job b, output this schedule. Otherwise, let $A = \{j : r_j \leq q_j, j \neq b\}$, $B = \{j : r_j > q_j, j \neq b\}$, and construct a second schedule by first scheduling the jobs indexed by A in order of nondecreasing release dates, then job b, and finally the jobs indexed by B in order of nonincreasing delivery times; output the better schedule.

3.6. Enumerative methods

Although $1|prec, r_j|L_{\max}$ is strongly NP-hard, computational experience has shown that it is not such a hard problem. There exist clever enumerative algorithms that perform remarkably well. In fact, this success has motivated the use of these algorithms for the computation of lower bounds for flow shop and job shop problems, which appear to pose greater computational challenges (see Chapters 12 and 13).

We will present two branch-and-bound algorithms to solve $1|r_j|L_{\max}$. Once again, it will be convenient to view the problem in its head-body-tail formulation.

First branch-and-bound algorithm

This is a relatively simple method. The *branching rule* generates all feasible schedules by making all possible choices for the first position in the schedule; for each of these choices it considers all possible choices for the second position, and so on. It is possible to exclude certain choices as being dominated by others. For example, it would be unfortunate to select as the next job one whose release date is so large that another job can be scheduled prior to it. More precisely, let S denote the set of jobs assigned to the first $l-1$ positions, and let t be the completion time of the last job in S; for the lth position, we need consider a job k only if

$$r_k < \min_{j \notin S}\{\max\{t, r_j\} + p_j\}.$$

If this inequality does not hold, then the job minimizing the right-hand side could be made to precede job k without delaying its processing. Thus, we are sure that there exists an optimal schedule that satisfies this restriction.

We next consider the way in which a *lower bound* is computed for each node in the search tree. An attractive possibility is to schedule the remaining jobs from time t onwards while allowing preemption. The preemptive EDD solves this problem in $O(n \log n)$ time (cf. Section 3.2).

Finally, we must specify a *search strategy*, which selects the next node of the tree for further exploration. A common rule is to select a node with *minimum lower bound* value. Whereas this rule helps to limit the number of nodes examined, the overhead in implementing the approach may overwhelm its advantages. A simple alternative is to do a *depth-first search* of the tree: the next node is a child of the current node (perhaps the one with minimum lower bound value); when all children of a node have been examined, the path towards the root is retraced until a node with an unexplored child is found.

A nice aspect of this approach is that it can be applied to solve other NP-hard $1|r_j|f_{\max}$ problems in an analogous way.

Second branch-and-bound algorithm

The second method makes a more extensive use of the mathematical structure that was developed in this chapter. The main idea is that each node in the search tree will correspond to a restricted instance of the problem, on which we will run the nonpreemptive EDD rule. The restrictions imposed on the instance are nothing more than precedence constraints between certain pairs of jobs. These constraints will be incorporated by modifying the release dates and the delivery times, as we have done throughout this chapter.

Consider a particular node in the tree and apply the nonpreemptive EDD rule to the corresponding instance. Suppose that the schedule obtained has a critical sequence with no interference job. This means that the schedule is optimal for the modified data or, in other words, optimal subject to the precedence constraints specified in that node.

On the other hand, suppose that there is an interference job b. Let Q' be the set of the jobs in the critical sequence that follow job b, and let $L_{\max}$ be the value of the schedule. We know that

$$L_{\max} = S_b + p_b + p(Q') + q(Q') < r(Q') + p_b + p(Q') + q(Q').$$

Consider another schedule in which some job in Q' precedes job b and another in Q' follows it. The proof of Lemma 3.4 implies that the value of this schedule must be at least $r(Q') + p_b + p(Q') + q(Q')$, and so it is worse than the schedule just obtained. Hence, we may further restrict attention to those schedules in which job b either precedes all of the jobs in Q' or follows all of them. This gives us our *branching rule*. Each node will have two children, each corresponding to one of these two additional constraints. Since we use the nonpreemptive EDD rule, we can enforce the first constraint by setting

$$q_b := \max_{j \in Q'} q_j, \tag{3.3}$$

and the second one by

$$r_b := \max_{j \in Q'} r_j. \tag{3.4}$$

A simple way to compute a *lower bound* for a node is to take the maximum of $r(Q') + p(Q') + q(Q')$, $r(Q' \cup \{b\}) + p(Q' \cup \{b\}) + q(Q' \cup \{b\})$, and the lower bound of its parent. Note that, although job b and Q' are determined by running the nonpreemptive EDD rule on the parent, the release dates and delivery times have been updated afterwards.

As in any branch-and-bound algorithm, a node is discarded if its lower bound is no smaller than the global *upper bound*, i.e., the value of the best schedule found thus far. An advantage of using the nonpreemptive EDD rule at each node is that each time we obtain a new feasible solution, which may improve the upper bound. For the same purpose, we also evaluate the schedule in which job b follows the jobs in Q'. The *search strategy* always selects a node with minimum lower bound.

The following trick can help to restrict the instance for a new node in the tree even further. Suppose there is a job k, $k \notin Q' \cup \{b\}$, for which $r(Q') + p_k + p(Q') + q(Q')$ exceeds the upper bound. By the same reasoning as above, we conclude that in any better schedule job k either precedes or follows all of the jobs in Q'. If also $r(Q') + p(Q') + p_k + q_k$ exceeds the upper bound, then job k must precede Q', and we set $q_k := \max_{j \in Q'} q_j$. Similarly, if $r_k + p_k + p(Q') + q(Q')$ exceeds the upper bound, then job k follows Q', and we set $r_k := \max_{j \in Q'} r_j$.

In comparing the two branch-and-bound methods, one may wonder how the two lower bound procedures relate. In this respect, Theorem 3.5 implies that the preemptive bound dominates the simple bound of the second algorithm. Why does the second algorithm neglect to run a superior bounding procedure? This is merely a question of balancing the sorts of work done by an enumerative method. It takes more time to obtain a better bound, and it is not clear *a priori* whether the improved lower bound justifies the additional work. There are procedures that in some cases even improve on the preemptive bound, but from an empirical point of view it appears to be preferable to use the simpler lower bound in case of the second branching strategy. Indeed, computational experiments suggest that this algorithm is among the current champions for solving $1|r_j|L_{\max}$.

Exercises

3.25. Prove that (3.3) and (3.4) enforce the desired precedence constraints not only in each of the two child nodes generated, but also in all of their descendants.

3.26. Construct an instance of $1|\mathit{prec}, r_j|L_{\max}$ which satisfies the property that $r_j < r_k$ and $d_j < d_k$ whenever $j \to k$, while its $L^*_{\max}$ value would decrease if the precedence constraints would be ignored.

3.27. How can the two branch-and-bound algorithms be adapted to solve $1|\mathit{prec}, r_j|L_{\max}$?

Notes

3.1. *Earliest Due Date and Least Cost Last rules.* These rules are due to Jackson (1955) and Lawler (1973).

Exercises 3.7 and 3.8 are from Monma (1980). Hochbaum and Shamir (1989) gave an $O(n\log^2 n)$ algorithm for the maximum weighted tardiness problem, $1||wT_{\max}$. Fields and Frederickson (1990) gave an $O(n\log n + |A|)$ algorithm for $1|prec|wT_{\max}$, where A is the arc set of the precedence digraph.

3.2. *Preemptive EDD and Least Cost Last rules.* Horn (1974) observed that $1|pmtn,r_j|L_{\max}$ and $1|r_j, p_j = 1|L_{\max}$ are solved by the preemptive EDD rule. Frederickson (1983) gave an $O(n)$ algorithm for $1|r_j, p_j = 1, \bar{d}_j|-$. Theorem 3.5 is due to Carlier (1982); Nowicki and Zdrzalka (1986) observed that its proof is somewhat more elusive than originally believed.

The procedure for modifying release and due dates so that the several variants of the EDD rule automatically satisfy given precedence constraints was described by Lageweg, Lenstra, and Rinnooy Kan (1976). Monma (1982) gave a linear-time algorithm for $1|prec, p_j = 1|L_{\max}$.

The generalization of the Least Cost Last rule for solving $1|pmtn, r_j|f_{\max}$ is due to Baker, Lawler, Lenstra, and Rinnooy Kan (1983). Exercises 3.10–3.12 are also from their paper.

3.3. *A polynomial-time algorithm for jobs of equal length.* The algorithm for equal-length jobs is due to Simons (1978). An alternative algorithm was proposed by Carlier (1979). Garey, Johnson, Simons, and Tarjan (1981) gave an improved implementation of the decision algorithm, which runs in $O(n\log n)$ time.

3.4. *NP-hardness results.* Theorem 3.11 is due to Lenstra, Rinnooy Kan, and Brucker (1977).

3.5. *Approximation algorithms.* Schrage (1971) proposed the nonpreemptive EDD rule as a heuristic for $1|r_j|L_{\max}$, with the addition that ties on due date should be broken by selecting a job with maximum processing time. For the head-body-tail formulation, Kise, Ibaraki, and Mine (1979) showed that every left-justified schedule is shorter than three times the optimum. They considered six approximation algorithms, including the nonpreemptive EDD rule, and proved that all of them have a performance ratio of 2. Potts (1980B) proposed the iterated nonpreemptive EDD rule, which was the first method to achieve a better performance bound. Hall and Shmoys (1992) showed that the procedure which applies the iterated nonpreemptive EDD rule to an instance and its inverse and selects the better schedule is a 4/3-approximation algorithm. They observed that all approximation algorithms proposed thus far could easily be extended to handle precedence constraints. The polynomial approximation scheme presented is due to Lawler (–). Hall and Shmoys (1992) gave a more efficient scheme: more precisely, they developed a family of algorithms $\{\mathrm{A}'_k\}$ that guarantees $L_{\max}(\mathrm{A}'_k)/L^*_{\max} \leq 1 + 1/k$, where A'_k runs in

$O(n \log n + nk^{16k^2+8k})$ time. In a later paper, Hall and Shmoys (1990) extended their scheme to the precedence-constrained problem.

Exercises 3.23 and 3.24 are from Kise, Ibaraki, and Mine (1979) and Nowicki and Smutnicki (1994), respectively.

3.6. *Enumerative methods.* The first branch-and-bound algorithm is due to Baker and Su (1974), and the second one to Carlier (1982). Woerlee (1991) showed that the release date modification rule originally proposed by Carlier does not necessarily enforce the desired precedence constraints. Exercise 3.25 is from Verkooijen (1991). Carlier's method is able to solve problem instances with up to 10,000 jobs, often without branching. Vazacopoulos (1991) gave an instance for which the method generates at least $2^{n/2}$ nodes.

Earlier branch-and-bound algorithms were given by Dessouky and Margenthaler (1972) and by Bratley, Florian, and Robillard (1973). McMahon and Florian (1975) proposed a lower bound that is not dominated by the preemptive bound. Lageweg, Lenstra, and Rinnooy Kan (1976) extended the algorithms of Baker and Su and of McMahon and Florian to the precedence-constrained problem, and demonstrated that, if for a given problem instance the range of the release times is smaller than the range of the delivery times, then it is computationally advantageous to apply the McMahon-Florian algorithm to the inverse instance. Larson, Dessouky, and Devor (1985) employed this idea in their branch-and-bound algorithm. Exercise 3.26 is from Lageweg, Lenstra, and Rinnooy Kan (1976). Nowicki & Smutnicki (1987) discussed the relations between various lower bounds. Zdrzalka and Grabowski (1989) considered extensions of these enumerative methods to $1|\,prec, r_j|f_{\max}$.

Pan and Shi (2006) proposed new rules for bounding, problem reduction, branching and problem reversal, and tested these extensively on classes of hard instances. Della Croce and T'kindt (2010) gave improved lower bounds.

Dominance results among the schedules may be used in the obvious way to speed up enumerative procedures. Erschler, Fontan, Merce, and Roubellat (1982, 1983) considered the decision problem $1|r_j, \bar{d}_j|-$, and introduced dominance based on the $(r_j, \bar{d}_j)$ intervals.