Contents

5. Weighted number of late jobs **1**
Eugene L. Lawler
5.1. Some preliminaries 2
5.2. Dynamic programming solution of $1||\sum w_j U_j$ 3
5.3. A fully polynomial approximation scheme 5
5.4. The Moore-Hodgson algorithm 6
5.5. Similarly ordered release dates and due dates 13
5.6. The general problem $1|pmtn, r_j|\sum w_j U_j$ 21
5.7. Precedence constraints 26

5

Weighted number of late jobs

Eugene L. Lawler
University of California, Berkeley

Once it happened that two of the authors of this book found themselves circling an airport in a dense fog. When the fog broke, they could see many other aircraft. In fact, there were so many planes in the air it seemed unlikely that all of them could land before running out of fuel. While waiting to reach earth again, the authors amused themselves by applying their knowledge of scheduling theory to the situation. There were n planes to be scheduled for landing on a single runway, where plane j carried w_j passengers, required p_j time units to land, and would run out of fuel by time d_j. The problem of minimizing the number of crashed planes, $1||\sum U_j$, could be solved in polynomial time. However, the problem of minimizing the number of passenger fatalities, $1||\sum w_jU_j$, was NP-hard. Because time was limited, it seemed plausible that the flight controllers might choose to solve the easier problem. And since the authors were traveling in a very small plane, this observation was reassuring. It provided additional motivation to write this book.

In this chapter we first show that the NP-hard problem confronting the flight controllers, $1||\sum w_jU_j$, can be can be solved in $O(nW)$ time, where $W = \sum w_j$, with a minor modification of the knapsack algorithm described in Chapter 2. We then show that a much streamlined version of the algorithm, due to Moore and Hodgson, solves the problem $1||\sum w_jU_j$ in $O(n\log n)$ time, under the condition that the processing times and job weights are oppositely ordered. We also show that, although the problem $1|r_j|\sum U_j$ is strongly NP-hard, it can be solved in $O(n\log n)$ time, provided that the release dates and due dates are similarly ordered. Finally, we show that a considerable elaboration of the knapsack algorithm solves the general preemptive problem $1|pmtn,r_j|\sum w_jU_j$ in $O(nk^2W^2)$ time, where k is the number of distinct release dates.

All of the computational results presented in this chapter are for independent jobs, because NP-hardness results concerning precedence constraints are very limiting. In particular, we show that even the problem $1|chains,p_j = 1|\sum U_j$ is strongly NP-hard.

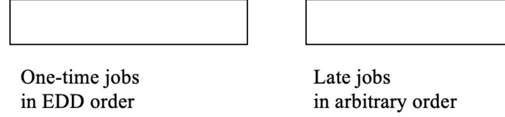


Figure 5.1. Form of an optimal schedule.

5.1. Some preliminaries

We begin with some simple but fundamental observations that hold for $1||\sum w_jU_j$, $1|pmtn,r_j|\sum w_jU_j$, and for all special cases of these problems. First, we note the following: because of the character of the $\sum w_jU_j$ objective, if a given job j is scheduled to be late, it might as well be arbitrarily late. This means that, in the absence of precedence constraints, there exists an optimal schedule in which all jobs that are on time precede all jobs that are late. Second, we observe that the jobs that are on time can be assumed to be scheduled by the following rule: always schedule the job with the earliest due date. In the presence of release dates, this is an *EDD rule* which may produce a schedule with preemptions. It follows that there must exist an optimal schedule like that shown in Figure 5.1.

From the previous observations it follows that the problems $1||\sum w_jU_j$ and $1|pmtn,r_j|\sum w_jU_j$ reduce to the problem of finding a maximum-weight feasible set of jobs, where by a *feasible* set we mean a set of jobs that are all completed on time when scheduled by the (extended) EDD rule, that is, this rule produces a feasible schedule.

Throughout this chapter we shall normally assume that jobs are numbered in EDD order, i.e.,

$$d_1 \leq d_2 \leq ... \leq d_n.$$

We shall also employ the following notation. As in Chapter 3, if $S \subseteq N = \{1,...,n\}$, let $p(S) = \sum_{j\in S} p_j$, $r(S) = \min\{r_j | j \in S\}$, and similarly define $w(S) = \sum_{j\in S} w_j$; finally, let $C(S)$ denote the completion time of the last job in an (extended) EDD schedule for the jobs indexed by S.

Exercises

5.1. Show that the following procedure computes maximum-cardinality feasible index set S for the problem $1|r_j, p_j = 1|\sum U_j$: schedule the jobs in time, starting with S empty. At each time t, from among all the unprocessed jobs j such that $r_j \leq t < d_j$, if any, add to S a job j with the earliest possible due date. Show that this algorithm can be implemented to run in $O(n \log n)$ time.

5.2. The problem $1|r_j, p_j = 1|\sum w_jU_j$ can be solved by applying the greedy matroid algorithm, as follows. Index the jobs in nonincreasing order of job weights. Then

process the jobs in order, solving the recurrence relations

$$S^{(0)} = \emptyset,$$

$$S^{(j)} = \begin{cases} S^{(j-1)} \cup \{j\}, & \text{if } S^{(j-1)} \cup \{j\} \text{ is feasible,} \\ S^{(j-1)}, & \text{otherwise.} \end{cases}$$

The set $S^{(n)}$ then indexes a maximum-weight feasible set of jobs. Show that this algorithm can be implemented to run in $O(n^2)$ time.
5.3. Show that it is possible to solve the problem $1|p_j = 1|\sum U_j$ in $O(n)$ time. (Hint: Start with an $O(n)$ bucket sort of the due dates, by noting that any due date greater than n can be reduced to n.)

5.2. Dynamic programming solution of $1||\sum w_jU_j$

The problem $1||\sum w_jU_j$ is NP-hard because the special case $1|d_j = d|\sum w_jU_j$ is equivalent to the NP-hard knapsack problem. But in Chapter 2 we showed that the knapsack problem can be solved by a dynamic programming computation within the pseudopolynomial time bound of $O(nW)$. We shall now show that a minor modification of the dynamic programming computation solves $1||\sum w_jU_j$ within the same time bound.

The problem of finding a maximum-weight feasible set for the $1||\sum w_jU_j$ problem can be formulated as an integer linear programming problem:

$$\begin{aligned} \text{maximize } & \sum w_jx_j \\ \text{subject to } & p_1x_1 & \leq d_1 \\ & p_1x_1 + p_2x_2 & \leq d_2 \\ & p_1x_1 + p_2x_2 + p_3x_3 & \leq d_3 \\ & \vdots & \vdots \\ & p_1x_1 + p_2x_2 + p_3x_3 + \ldots + p_nx_n & \leq d_n \end{aligned}$$

and

$$x_j \in \{0,1\}, \text{for} j = 1,2,\ldots,n.$$

Note that when all the d_j's are equal, the last inequality implies all the others and the problem reduces to the ordinary knapsack problem.

The triangular form of the inequality constraints enables us to employ a dynamic programming procedure essentially the same as that described for the knapsack problem in Chapter 2. Let $P^{(j)}(w)$ denote the minimum total processing time for any

feasible subset of jobs $1,\dots,j$ that has total weight exactly w. Then we have

$$P^{(0)}(w) = \begin{cases} 0, & \text{if } w = 0, \\ +\infty, & \text{otherwise.} \end{cases}$$

$$P^{(j)}(w) = \begin{cases} \min\{P^{(j-1)}(w), P^{(j-1)}(w - w_j) + p_j\}, & \text{if } P^{(j-1)}(w - w_j) + p_j \le d_j, \\ P^{(j-1)}(w), & \text{otherwise.} \end{cases} \tag{5.1}$$

As before, there are $O(W)$ equations to solve at each of n iterations, $j = 1,2,\dots,n$. Each equation requires a constant number of arithmetic operations. It follows that the values $P^{(n)}(w)$ can be computed in $O(nW)$ time; the maximum weight of a feasible set is the largest value of w such that $P^{(n)}(w)$ is finite.

As we described in Chapter 2, the dynamic programming recurrence equations can be solved by computing lists of dominant pairs $(w, P^{(j)}(w))$. At each iteration a list of candidate pairs is formed from the list existing at the end of the previous iteration. This list is merged with the existing list, with dominated pairs discarded in the course of the merge. In the case at hand, a pair $(w + w_j, P^{(j-1)}(w) + p_j)$ is discarded as infeasible at iteration j whenever $P^{(j-1)}(w) + p_j > d_j$, whereas in the the knapsack problem a pair is discarded whenever $P^{(j-1)}(w) + p_j > d$, where d is the knapsack capacity.

Recall that in Chapter 2 we also described how to compute upper bounds on the value of the solutions that can be obtained from pairs (w,P) and how to use these bounds to eliminate pairs (w,P) whose bounds do not exceed the value of a known feasible solution. We can compute comparable bounds for the problem $1||\sum w_jU_j$ by solving the linear programming relaxation of its integer programming formulation. This relaxation turns out to be an interesting scheduling problem in its own right: Let V_j denote the amount of processing of job j that is done after its due date d_j. Then the linear programming relaxation is equivalent to the problem of minimizing weighted *late work*, i.e., $1|pmtn|\sum w_jV_j$. In Exercise 5.4 we ask the reader to devise an efficient algorithm for solving this problem.

Exercises

5.4. Devise an $O(n \log n)$ algorithm for solving the weighted late work scheduling problem $1||\sum w_jV_j$. Recall that it is easy to solve the linear-programming relaxation of the knapsack problem, by including items in the knapsack in order of nonincreasing ratio w_j/p_j, until the knapsack is completely filled, using a fraction of the last item if necessary. Devise a similar algorithm for solving the linear programming relaxation of $1||\sum w_jU_j$. (Hint: Construct an optimal schedule by starting at the latest due date and working backward in time.)

5.3. A fully polynomial approximation scheme

In Chapter 2, we showed that it is possible to convert a pseudopolynomial-time algorithm for the knapsack problem into a fully polynomial approximation scheme. The $O(nW)$ algorithm for $1||\sum w_jU_j$ given in Section 5.2 is, of course, a pseudopolynomial-time algorithm. We shall now show how to construct a fully polynomial approximation scheme for the $1||\sum w_jU_j$ problem.

In the previous section, we have focused on the equivalent problem of finding a maximum-weight feasible set of jobs. These problems are not equivalent from the point of view of approximation: a ρ-approximation algorithm for finding a maximum-weight feasible set need not be a ρ-approximation algorithm for $1||\sum w_jU_j$. For example, the latter must produce an optimal schedule for instances in which all jobs can be scheduled on time, whereas this is certainly not the case for the former. In contrast to the analogous situation for $1|r_j|L_{\max}$, it is possible to obtain optimal solutions in this particular case, by using the EDD rule.

Let W^* denote the minimum weight of a set of late jobs; hence $W - W^*$ is the maximum weight of a feasible set. The dynamic programming algorithm for $1||\sum w_jU_j$ given in Section 5.2 generates $O(W - W^*)$ dominant pairs (w,P) in each of n iterations, yielding a time bound of $O(n(W - W^*))$. For our purposes, we will need a slightly different algorithm, which has an $O(nW^*)$ running time. This can be obtained by a similar dynamic programming approach that maintains pairs (w,P), where w is the weight of a set of late jobs with processing time at most P, and the notion of dominance is reversed (see Exercise 5.5).

As in the case of the knapsack problem, the principal technique used is to round the data to have fewer significant digits, in order to make the dynamic programming algorithm take less time. This rounding and rescaling technique will be augmented by one further idea. We will first need to obtain a schedule of total weight that is within a factor of n of the optimum. This will enable us to find an appropriate factor δ, by which to rescale the data. Suppose that we find a schedule that minimizes the maximum weight of a late job. Since this is the special case of $1||f_{\max}$ with $f_j = w_jU_j$, $j = 1,\ldots,n$, Theorem 3.3 implies that it can be solved using the least-cost-last rule in $O(n^2)$ time; let w denote the maximum-weight late job in this schedule. Clearly, $w \leq W^*$, and the schedule just obtained has late jobs of total weight at most nw.

We now set

$$\delta = \frac{\varepsilon w}{n},$$

and compute the rounded weights

$$\bar{w}_j = \lfloor \frac{w_j}{\delta} \rfloor.$$

The claimed dynamic programming algorithm can be applied to the rounded instance in $O(nW^*/\delta)$ time.

Let the minimum-weight set of late jobs for the original instance be indexed by S^*, and let the set of late jobs in the schedule returned by the algorithm on the rounded data be indexed by $\bar{S}$. Note that $\delta\bar{w}_j \leq w_j \leq \delta(\bar{w}_j + 1)$, and so

$$\sum_{j\in\bar{S}} w_j \leq \delta|\bar{S}| + \sum_{j\in\bar{S}} \delta\bar{w}_j.$$

Using the optimality of $\bar{S}$ with respect to $\bar{w}_j$, $j = 1,...,n$, and the inequality $|\bar{S}| \leq n$, we can further bound the right-hand side by

$$\leq \delta n + \sum_{j\in S^*} \delta\bar{w}_j \leq \varepsilon w + \sum_{j\in S^*} w_j \leq (1+\varepsilon)\bar{W}^*.$$

Since $W^*/\delta \leq n^2/\varepsilon$, the algorithm runs in $O(n^3/\varepsilon)$ time. Thus we have shown the following theorem.

Theorem 5.1. *There exists a fully polynomial approximation scheme to solve the problem* $1||\sum w_j U_j$.

Exercises

5.5. Give a dynamic programming algorithm for $1||\sum w_j U_j$ that runs in $O(nW^*)$ time.

5.6. Give a fully polynomial approximation scheme to find a maximum-weight feasible set of jobs.

5.4. The Moore-Hodgson algorithm

A simple and elegant algorithm of Moore and Hodgson solves the unweighted problem $1||\sum U_j$ in $O(n\log n)$ time. More generally, it solves the problem $1||\sum w_j U_j$ under the condition that processing times and job weights are *oppositely ordered*. By this we mean that $p_i < p_j$ implies $w_i \geq w_j$, for all i and j. Note that opposite ordering necessarily holds if either all $p_j = 1$ or all $w_j = 1$. Hence the special case of opposite ordering of processing times and job weights is a common generalization of the problems $1|p_j = 1|\sum w_j U_j$ and $1||\sum U_j$.

The Moore-Hodgson algorithm is as follows: Starting with $S^{(0)} = \emptyset$, process the jobs in EDD order, constructing feasible sets $S^{(j)}$, $j = 1,2,...,n$, by the recurrence relations

$$S^{(j)} = \begin{cases} S^{(j-1)} \cup \{j\}, & \text{if } p(S^{(j-1)} \cup \{j\}) \leq d_j, \\ S^{(j-1)} \cup \{j\} - \{l\}, & \text{otherwise,} \end{cases} \tag{5.2}$$

where

$$l = \text{argmax}\{p_i | i \in S^{(j-1)} \cup \{j\}\}$$
$$= \text{argmin}\{w_i | i \in S^{(j-1)} \cup \{j\}\}.$$

The set $S^{(n)}$ is then a maximum-weight feasible set.

The Moore-Hodgson algorithm is efficiently implemented with a priority queue S supporting the operations of *insert* and *deletemax*, as indicated below:

Algorithm 1: Moore-Hodgson algorithm

```
S := ∅;
p(S) := 0;
for j = 1,2,...,n do
    insert(S, j);
    p(S) := p(S) + p_j;
    if p(S) > d_j then
        l := deletemax(S);
        p(S) := p(S) − p_l;
    end
end
```

Each of the n insertions and each of the at most n deletions requires $O(\log n)$ time. Hence the overall running time required to generate a maximum-weight feasible set $S^{(n)}$ is bounded by $O(n \log n)$.

Although the Moore-Hodgson algorithm seems quite unrelated to the dynamic programming algorithm of Section 5.2, the two are closely related. We shall demonstrate this by deriving the Moore-Hodgson algorithm from the list-making version of the dynamic programming algorithm.

Recall that the input to iteration j of the dynamic programming algorithm is the list of dominant pairs generated at iteration $j-1$:

$$(w(0), P^{(j-1)}(w(0))), (w(1), P^{(j-1)}(w(1))), ..., (w(k), P^{(j-1)}(w(k))),$$

where

$$0 = w(0) < w(1) < ... < w(k),$$
$$0 = P^{(j-1)}(w(0)) < P^{(j-1)}(w(1)) < ... < P^{(j-1)}(w(k)) \leq d_{j-1}.$$

Each pair $(w(i), P^{(j-1)}(w(i)))$ in the list of dominant pairs is realized by a feasible

set $S^{(j-1)}(w(i)) \subseteq \{1,2,...,j-1\}$, with

$$w(S^{(j-1)}(w(i))) = w(i)$$
$$p(S^{(j-1)}(w(i))) = P^{(j-1)}(w(i)).$$

Ordinarily, we could not expect these feasible sets to be related in any special way. But when the processing times and job weights are oppositely ordered, a happy thing occurs: the feasible sets form a tower. Specifically, each set $S^{(j-1)}(w(i))$ contains exactly one job $j(i)$ that is not contained in the set $S^{(j-1)}(w(i-1))$ which is immediately below it in the tower. That is,

$$\begin{aligned} S^{(j-1)}(w(0)) = \emptyset \subseteq S^{(j-1)}(w(1)) &= \{j(1)\} \\ \subseteq S^{(j-1)}(w(2)) &= \{j(1), j(2)\} \subseteq ... \\ \subseteq S^{(j-1)}(w(k)) &= \{j(1), j(2), ..., j(k)\}. \end{aligned} \tag{5.3}$$

from which it follows that

$$P^{(j-1)}(w(i)) - P^{(j-1)}(w(i-1)) = p_{j(i)}, \text{ and } w(i) - w(i-1) = w_{j(i)}, i = 1,2,...,k.$$

Furthermore,

$$p_{j(1)} \leq p_{j(2)} \leq ... \leq p_{j(k)} \tag{5.4}$$

and

$$w_{j(1)} \geq w_{j(2)} \geq ... \geq w_{j(k)}. \tag{5.5}$$

We shall refer to relations (5.3)-(5.5) as the *tower-of-sets* property of feasible sets.

At this point we may gain some insight from an example. Consider the following problem data:

j	1	2	3	4	5
p_j	2	1	5	4	3
w_j	4	5	1	2	3

Assume that all due dates are very large, so that they are of no consequence. At the end of iteration 4 we have the following list of dominant pairs,

$$(0,0), (5,1), (9,3), (11,7), (12,12),$$

which are plotted as dots in Figure 5.2. With reference to the Gantt charts shown in Figure 5.3, we see that these pairs are realized by a tower of feasible sets:

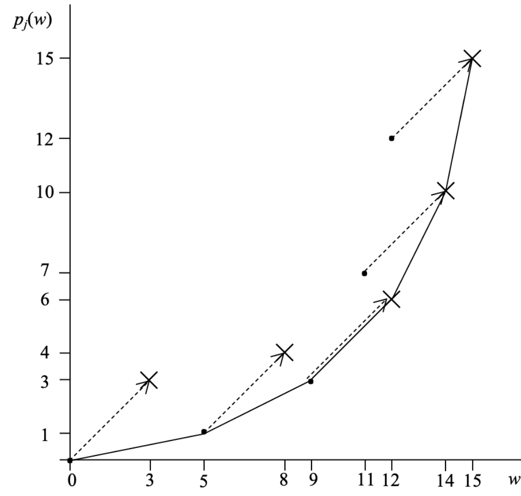


Figure 5.2. Graph of dominant pairs.

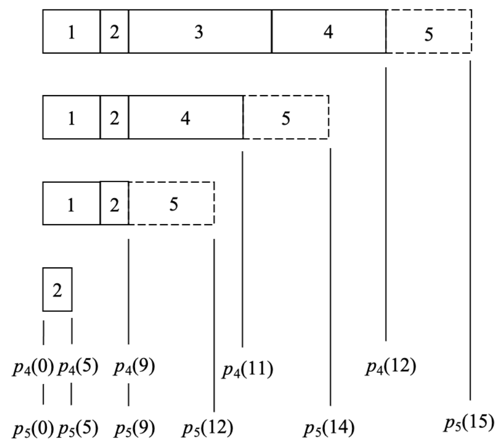


Figure 5.3. Gantt charts.

$$\begin{aligned} S^{(4)}(0) = \emptyset \subseteq S^{(4)}(5) &= \{2\} \\ \subseteq S^{(4)}(9) &= \{2,1\} \\ \subseteq S^{(4)}(11) &= \{2,1,4\} \\ \subseteq S^{(4)}(12) &= \{2,1,4,3\}. \end{aligned}$$

From the existing list of dominant pairs we form a list of candidates,

$$(3,3),(8,4),(12,6),(14,10),(15,15),$$

which are plotted as +'s in Figure 5.2. When the two lists are merged, and dominated pairs discarded, the surviving pairs are

$$(0,0),(5,1),(9,3),(12,6),(14,10),(15,15).$$

Again with reference to Figure 5.3, we find that these pairs are realized by a tower of feasible sets:

$$\begin{aligned} S^{(5)}(0) = S^{(4)}(0) = \emptyset \subseteq S^{(5)}(5) &= S^{(4)}(5) = \{2\} \\ \subseteq S^{(5)}(9) &= S^{(4)}(9) = \{2,1\} \\ \subseteq S^{(5)}(12) &= S^{(4)}(9) \cup \{5\} = \{2,1,5\} \\ \subseteq S^{(5)}(14) &= S^{(4)}(11) \cup \{5\} = \{2,1,5,4\} \\ \subseteq S^{(5)}(15) &= S^{(4)}(12) \cup \{5\} = \{2,1,5,4,3\}. \end{aligned}$$

We shall prove by induction that the tower-of-sets property holds at each iteration. For the base case of the induction, note that the property holds at the end of iteration 0 when the only dominant pair, (0,0), is realized by the empty set. Assume, by inductive hypothesis, that the property holds at the end of iteration $j-1$. At iteration j the candidate pairs that are formed are

$$\begin{aligned} &(w(0)+w_j, P^{(j-1)}(w(0))+p_j) \\ &(w(1)+w_j, P^{(j-1)}(w(1))+p_j), ..., \\ &(w(k)+w_j, P^{(j-1)}(w(k))+p_j). \end{aligned}$$

Let h be the largest index i, $1 \leq i \leq k$, if any, such that $p_{j(i)} \leq p_j$ and $w_{j(i)} \geq w_j$. (The case in which there is no such h is left as an exercise.) Recall that the existing pairs

$$(w(1), P^{(j-1)}(w(1))), ..., (w(h), P^{(j-1)}(w(h)))$$

can be rewritten as

$$(w(0)+w_{j(1)},P^{(j-1)}(w(0))+p_{j(1)}),\ldots,(w(h-1)+w_{j(h)},P^{(j-1)}(w(h-1))+p_{j(h)}).$$

Opposite ordering of processing times and job weights implies that these pairs respectively dominate the candidate pairs

$$(w(0)+w_j,P^{(j-1)}(w(0))+p_j),\ldots,(w(h-1)+w_j,P^{(j-1)}(w(h-1))+p_j).$$

Furthermore, by rewriting the other existing pairs as

$$\begin{aligned}&(w(h)+w_{j(h+1)},P^{(j-1)}(w(h))+p_{j(h+1)}),\ldots,\\&(w(k-1)+w_{j(k)},P^{(j-1)}(w(k-1))+p_{j(k)})\end{aligned}$$

we see that they are, respectively, dominated by the candidate pairs

$$(w(h)+w_j,P^{(j-1)}(w(h))+p_j),\ldots,(w(k-1)+w_j,P^{(j-1)}(w(k-1))+p_j).$$

Hence the merged list at the end of iteration j contains the pairs

$$(w(0),P^{(j-1)}(w(0))),\ldots,(w(h),P^{(j-1)}(w(h))),$$

followed by the pairs

$$(w(h)+w_j,P^{(j-1)}(w(h))+p_j),\ldots,(w(k)+w_j,P^{(j-1)}(w(k))+p_j).$$

These pairs are realized by a new tower of feasible sets

$$\begin{aligned}S^{(j)}(w(0)) &= S^{(j-1)}(w(0)) = \emptyset\\
&\subseteq S^{(j)}(w(1)) = S^{(j-1)}(w(1)) = \{j(1)\}\\
&\subseteq S^{(j)}(w(2)) = S^{(j-1)}(w(2)) = \{j(1),j(2)\}\\
&\subseteq S^{(j)}(w(h)) = S^{(j-1)}(w(h)) = \{j(1),\ldots,j(h)\}\\
&\subseteq S^{(j)}(w(h)+w_j) = S^{(j-1)}(w(h))\cup\{j\} = \{j(1),\ldots,j(h),j\}\\
&\subseteq S^{(j)}(w(k-1)+w_j) = S^{(j-1)}(w(k-1))\cup\{j\}\\
&\qquad = \{j(1),\ldots,j(h),j,j(h+1),\ldots,j(k-1)\}\\
&\subseteq S^{(j)}((w(k)+w_j) = S^{(j-1)}(w(k))\cup\{j\}\\
&\qquad = \{j(1),\ldots,j(h),j,j(h+1),\ldots,j(k-1),j(k)\}.\end{aligned}$$

Of course, this is the case only if

$$P^{(j-1)}(w(k))+p_j = p(S^{(j-1)}\cup\{j\}) \le d_j.$$

If not, the pair $(w(k)+w_j,P^{(j-1)}(w(k))+p_j)$ is discarded and the last set in the new

tower is either

$$S^{(j-1)}(w(k-1)) \cup \{j\}, \quad \text{if } h < k,$$

or

$$S^{(j-1)}(w(k)), \quad \text{if } h = k.$$

In either case, it follows that the relation (5.3) holds at the end of iteration j. Since properties (5.4) and (5.5) follow directly from the definition of h, we see that at the end of iteration j, the tower-of-sets property holds.

It is now evident that list-making and list-merging is totally unnecessary. If we know the largest set in the tower at any given iteration, then we know everything about the tower. Furthermore, it is now a small step to verify that the Moore-Hodgson recurrence relations are designed to compute the largest set $S^{(j)}$ at iteration j from the largest set $S^{(j-1)}$ at iteration $j-1$. Thus the Moore-Hodgson algorithm is, in effect, a streamlined version of the dynamic programming computation for the problem $1||\sum w_j U_j$.

As a final note, observe that the algorithm does not require that either processing times or job weights be integers. Furthermore, the maximum-weight feasible set that is computed is invariant under changes in job weights, provided the relative ordering of the weights is unchanged (see Exercises 5.10 and 5.11).

Exercises

5.7. Describe how to determine in $O(n \log n)$ time whether or not processing times and job weights are oppositely ordered.

5.8. Prove that an instance of the knapsack problem with oppositely ordered w_j's and p_j's can be solved by filling the knapsack with items in nonincreasing order of the ratios w_j/p_j, until no further item can be added. (No item is fractionalized.)

5.9. Complete the proof of the Moore-Hodgson algorithm by supplying the argument for the case in which there is no index h, as defined in the inductive proof.

5.10. The following greedy procedure is an alternative to the Moore-Hodgson algorithm (but requires $O(n^2)$ running time; cf. Exercise 5.2). Index the jobs in nonincreasing order of job weights and in nondecreasing order of processing times. Then, starting with $S^{(0)} = \emptyset$, process the jobs in this order, solving the recurrence relations

$$S^{(j)} = \begin{cases} S^{(j-1)} \cup \{j\}, & \text{if } S^{(j-1)} \cup \{j\} \text{ is feasible,} \\ S^{(j-1)}, & \text{otherwise.} \end{cases}$$

Prove that the set $S^{(n)}$ computed by this procedure is the same as the set $S^{(n)}$ computed by the Moore-Hodgson algorithm.

5.11. Prove that the algorithm in the previous exercise computes a feasible set $S^{(n)}$ satisfying the following (very strong) optimality property: The weight of the ith weightiest job in $S^{(n)}$ is at least as great as the weight of the ith weightiest job in any other feasible set.

5.12. (a) Suppose that a certain feasible subset R of jobs is required to be on time. Describe how to modify the Moore-Hodgson algorithm to accommodate this constraint. (b) Suppose that, in addition to a due date d_j, each job j has a (hard) deadline $\bar{d}_j \geq d_j$. Suppose that our objective is to minimize $\sum U_j$, subject to satisfaction of all the deadlines. Show that the problem posed in part (a) is a special case of this problem. (Note: The problem $1|\bar{d}_j|\sum U_j$ has been shown to be NP-hard.)
5.13. Describe how to adapt the Moore-Hodgson algorithm to solve the problem $1|pmtn, r_j|\sum w_j U_j$, under the condition that processing times and job weights are oppositely ordered and, in addition, release dates and due dates are oppositely ordered.
5.14. Opposite ordering of release dates and due dates is a special case of nesting of release date-due date intervals. (Intervals $[r_j, d_j]$, $j = 1, 2, ..., n$, are said to be *nested* if for all pairs j, k, either $[r_j, d_j]$ and $[r_k, d_k]$, are disjoint (except possibly at an end point) or one interval is contained in the other. Generalize the algorithm derived for Exercise 5.13 to solve $1|pmtn, r_j|\sum w_j U_j$ in the special case that processing times and job weights are oppositely ordered and release date-due date intervals are nested. (Hint: Form a rooted tree (or a forest of trees) in which the nodes are identified with intervals and the children of a node are identified with the maximal intervals contained within it. Work from the leaves of the tree upward, computing a maximum weight feasible set at each node. You will probably need a data structure supporting the operations of *insert*, *deletemax*, and *merge*, each of which can be implemented to run in $O(\log n)$ time. The algorithm should run in $O(n \log n)$ time.)

5.5. Similarly ordered release dates and due dates

Release dates make things much more difficult. The problem $1|r_j|\sum U_j$ is strongly NP-hard, as can be shown by straightforward transformation from 3-PARTITION (see Exercise 5.19). It follows that there is no polynomial-time algorithm for $1|r_j|\sum U_j$ and no pseudopolynomial-time algorithm for $1|r_j|\sum w_j U_j$, unless $P = NP$. Nevertheless, it is possible to generalize the dynamic programming algorithm of Section 5.2 to solve $1|r_j|\sum w_j U_j$ in the case that release dates and due dates are *similarly ordered*. By this we mean that $d_i < d_j$ implies $r_i \leq r_j$, for each pair of jobs i and j. Accordingly, in this section we shall hereafter assume that the jobs have been given an EDD numbering such that

$$r_1 \leq r_2 \leq ... \leq r_n, \text{and}$$
$$d_1 \leq d_2 \leq ... \leq d_n.$$

Under this condition, there is no advantage to preemption and the problems $1|r_j|\sum w_j U_j$ and $1|pmtn, r_j|\sum w_j U_j$ are equivalent. Furthermore, the extended EDD rule produces schedules with no preemptions.

Let $C^{(j)}(w)$ denote the earliest completion time of a feasible subset of jobs

$1,2,...,j$ having total weight exactly w. Generalizing the recurrences (5.1), we have

$$C^{(0)}(w) = \begin{cases} 0, & \text{if } w = 0, \\ +\infty, & \text{otherwise,} \end{cases}$$

$$C^{(j)}(w) = \begin{cases} \min\{C^{(j-1)}(w), \max\{r_j, C^{(j-1)}(w-w_j)\} + p_j\}, \\ \qquad\qquad \text{if } \max\{r_j, C^{(j-1)}(w-w_j)\} + p_j \le d_j, \\ C^{(j-1)}(w), \quad \text{otherwise.} \end{cases} \tag{5.6}$$

As in the case of (5.1), there are $O(W)$ equations to solve at each of n iterations, and each equation requires a constant number of arithmetic operations. Hence the values $C^{(n)}(w)$ can be computed in $O(nW)$ time. The maximum weight of a feasible set is given by the largest value of w such that $C^{(n)}(w)$ is finite.

It happens that when job weights are equal, a variation of the Moore-Hodgson algorithm computes a maximum-cardinality feasible set in $O(n\log n)$ time. Let $q_j \le p_j$ be an *effective* processing time that is imputed to job j, as we shall describe later, and define

$$q(S) = \sum_{j\in S} q_j,$$

for any set $S \subseteq \{1,...,n\}$. Starting with $S^{(0)} = \emptyset$, we process the jobs in EDD order, constructing feasible sets $S^{(j)}$, $j = 1,2,...,n$, by the recurrence relations

$$S^{(j)} = \begin{cases} S^{(j-1)} \cup \{j\}, & \text{if } q(S^{(j-1)} \cup \{j\}) \le d_j - r_j, \\ S^{(j-1)} \cup \{j\} - \{l\} & \text{otherwise.} \end{cases} \tag{5.7}$$

where

$$l = \text{argmax}\{q_i | i \in S^{(j-1)} \cup \{j\}\}.$$

The set $S^{(n)}$ is then a maximum-cardinality feasible set.

The inductive proof argument that we shall make to justify the above recurrence relations for $1|r_j|\sum U_j$, under the condition of similarly ordered release dates and due dates, parallels that of the previous section. Before proceeding with this argument, let us try to gain some insight from a numerical example with the following problem data:

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

Assume that all due dates are very large and hence are of no consequence. The dominant pairs existing at the end of iteration 4 are of the form $(i, C^{(4)}(i))$, $i = 0,1,...,4$. These dominant pairs, plotted as dots in Figure 5.4, are as follows:

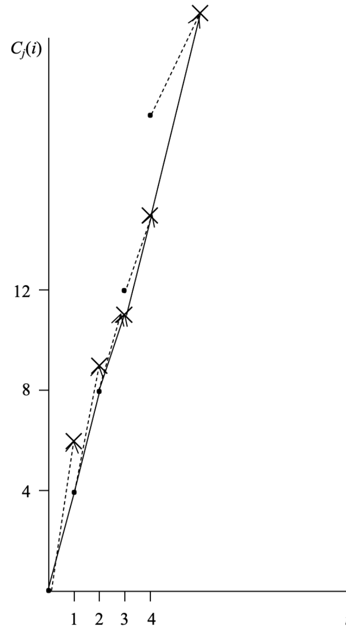


Figure 5.4.

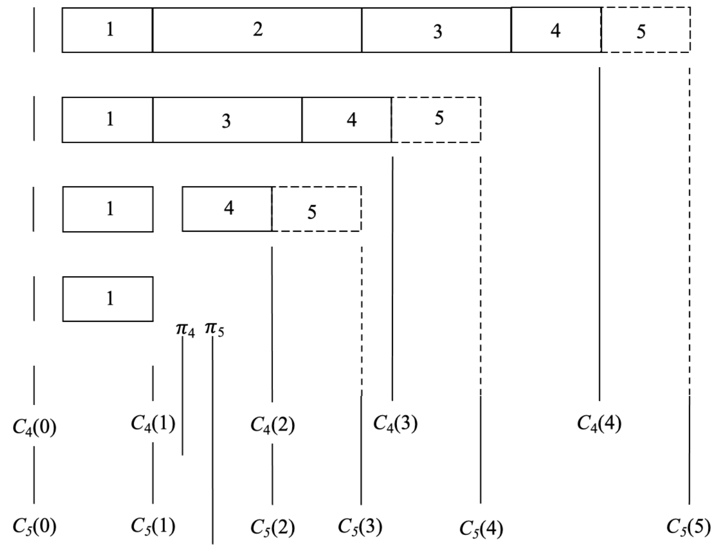


Figure 5.5.

$$(0,0),(1,4),(2,8),(3,12),(4,19).$$

With reference to Figure 5.5, we observe that these existing dominant pairs are realized by a tower of feasible sets:

$$\begin{aligned} S^{(4)}(0) = \emptyset \subseteq S^{(4)}(1) &= \{1\} \\ \subseteq S^{(4)}(2) &= \{1,4\} \\ \subseteq S^{(4)}(3) &= \{1,4,3\} \\ \subseteq S^{(4)}(4) &= \{1,4,3,2\}. \end{aligned}$$

Candidate pairs $(i+1, \max\{r_5, C^{(4)}(i)\} + p_5)$, $i = 0,1,...,4$, are formed from the existing list. These are plotted as +'s in Figure 5.4, and are as follows:

$$(1,6),(2,9),(3,11),(4,15),(5,22).$$

The candidate pairs $(1,6)$ and $(2,9)$ are each dominated by the existing pair $(2,8)$. The existing pairs $(3,12)$ and $(4,19)$ are respectively dominated by the candidate pairs $(3,11)$ and $(4,15)$. Hence when the two lists of pairs are merged, the surviving dominant pairs are

$$(0,0),(1,4),(2,8),(3,11),(4,15),(5,22).$$

Again with reference to Figure 5.5, we see that these pairs are again realized by a tower of feasible sets:

$$\begin{aligned} S^{(5)}(0) = S^{(4)}(0) = \emptyset \subseteq S^{(5)}(1) &= S^{(4)}(1) = \{1\} \\ \subseteq S^{(5)}(2) &= S^{(4)}(2) = \{1,4\} \\ \subseteq S^{(5)}(3) &= S^{(4)}(2) \cup \{5\} = \{1,4,5\} \\ \subseteq S^{(5)}(4) &= S^{(4)}(3) \cup \{5\} = \{1,4,5,3\} \\ \subseteq S^{(5)}(5) &= S^{(4)}(4) \cup \{5\} = \{1,4,5,3,2\}. \end{aligned}$$

In general, the input to iteration j is a list of dominant pairs,

$$(0, C^{(j-1)}(0)), (1, C^{(j-1)}(1)), ..., (k, C^{(j-1)}(k)),$$

where each pair $(i, C^{(j-1)}(i))$ is realized by a feasible set $S^{(j-1)}(i)$, with

$$\begin{aligned} |S^{(j-1)}(i)| &= i, \\ c(S^{(j-1)}(i)) &= C^{(j-1)}(i). \end{aligned}$$

(Recall that, by definition, $c(S)$ is the time of completion of the last job in an extended

EDD schedule for S.) We assert that the feasible sets form a tower:

$$\begin{aligned} S^{(j-1)}(0) = \emptyset &\subseteq S^{(j-1)}(1) = \{j(1)\} \\ &\subseteq S^{(j-1)}(2) = \{j(1), j(2)\} \\ &... \\ &\subseteq S^{(j-1)}(k) = \{j(1), j(2), ..., j(k)\}. \end{aligned} \tag{5.8}$$

For each $j(i) \in S^{(j-1)}(k)$, $i = 1, 2, ..., k$, define

$$q_{j(i)} = \max\{r_j, C^{(j-1)}(i)\} - \max\{r_j, C^{(j-1)}(i-1)\}. \tag{5.9}$$

We assert that

$$q_{j(1)} \leq q_{j(2)} \leq ... \leq q_{j(k)}, \tag{5.10}$$

and refer to (5.8) and (5.10) as the *tower-of-sets* property for this problem.

It is useful to point out that (5.9) is equivalent to the following less compact way to determine the effective processing time:

$$q_{j(i)} = \begin{cases} 0, & \text{if } C^{(j-1)}(i) \leq r_j, \\ C^{(j-1)}(i) - C^{(j-1)}(i-1), & \text{if } r_j \leq C^{(j-1)}(i-1), \\ C^{(j-1)}(i) - r_j, & \text{if } C^{(j-1)}(i-1) < r_j < C^{(j-1)}(i). \end{cases}$$

The third condition holds for at most one of $q_{j(i)}$, $i = 1, ..., k$. It is not hard to verify that in each of the three cases, $q_{j(i)} \leq p_{j(i)}$: the additional processing of job $j(i)$ can not delay the completion time of the extended EDD schedule by more than $p_{j(i)}$ time units beyond the maximum of the completion time for jobs $j(1), ..., j(i-1)$ and its release date.

We show next that the effective processing times play an important implicit role in determining the dominating pairs after iteration j of the dynamic programming algorithm. To perform iteration j, we form a list of candidate pairs,

$$(i+1, \max\{r_j, C^{(j-1)}(i)\} + p_j), i = 0, 1, ..., k.$$

Let h be the largest index i, $1 \leq i \leq k$, if any, such that $q_{j(i)} \leq p_j$. (As in Section 5.4, the case in which there is no such h is left as an exercise.) For $i = 1, ..., h$,

$$\begin{aligned} p_j \geq q_{j(i)} &= \max\{r_j, C^{(j-1)}(i)\} - \max\{r_j, C^{(j-1)}(i-1)\} \\ &\geq C^{(j-1)}(i) - \max\{r_j, C^{(j-1)}(i-1)\}, \end{aligned}$$

and so it follows that the existing pairs

$$(1, C^{(j-1)}(1)), ..., (h, C^{(j-1)}(h))$$

respectively dominate the candidate pairs

$$(1, \max\{r_j, C^{(j-1)}(0)\} + p_j), ..., (h, \max\{r_j, C^{(j-1)}(h-1)\} + p_j).$$

Furthermore, when $i > h$, we have that $q_{j(i)} > p_j \geq 0$, which implies that $q_{j(i)} = C^{(j-1)}(i) - \max\{r_j, C^{(j-1)}(i-1)\}$. As a result,

$$p_j < q_{j(i)} = C^{(j-1)}(i) - \max\{r_j, C^{(j-1)}(i-1)\},$$

and so the candidate pairs

$$(h+1, \max\{r_j, C^{(j-1)}(h)\} + p_j), ..., (k, \max\{r_j, C^{(j-1)}(k-1)\} + p_j).$$

respectively dominate the existing pairs

$$(h+1, C^{(j-1)}(h+1)), ..., (k, C^{(j-1)}(k)).$$

Hence the merged list at the end of iteration j contains the pairs

$$(0, C^{(j-1)}(0)), ..., (h, C^{(j-1)}(h)),$$

followed by the pairs

$$(h+1, \max\{r_j, C^{(j-1)}(h)\} + p_j), ..., (k+1, \max\{r_j, C^{(j-1)}(k)\} + p_j).$$

These pairs are realized by a new tower of feasible sets

$$\begin{aligned}
S^{(j)}(0) = S^{(j-1)}(0) = \emptyset& \\
&\subseteq S^{(j)}(1) = S^{(j-1)}(1) = \{j(1)\} \\
&\subseteq S^{(j)}(2) = S^{(j-1)}(2) = \{j(1), j(2)\} \subseteq ... \\
&\subseteq S^{(j)}(h) = S^{(j-1)}(h) = \{j(1), ..., j(h)\} \\
&\subseteq S^{(j)}(h+1) = S^{(j-1)}(h) \cup \{j\} = \{j(1), ..., j(h), j\} \subseteq ... \\
&\subseteq S^{(j)}(k) = S^{(j-1)}(k-1) \cup \{j\} = \{j(1), ..., j(h), j, j(h+1), ..., j(k-1)\} \\
&\subseteq S^{(j)}(k+1) = S^{(j-1)}(k) \cup \{j\} = \{j(1), ..., j(h), j, j(h+1), ..., j(k-1), j(k)\}.
\end{aligned}$$

That is, this is the case if

$$\max\{r_j, C^{(j-1)}(k)\} + p_j \leq d_j.$$

If not, the pair $(k+1, \max\{r_j, C^{(j-1)}(k)\} + p_j)$ is discarded and the largest set in the new tower is either

$$S^{(j-1)}(k-1) \cup \{j\}, \text{ if } h < k,$$

or

$$S^{(j-1)}(k), \text{ if } h = k.$$

This shows that property (5.8) holds at the end of iteration j. Property (5.10) holds as well, as we indicate below.

It would be inefficient to compute the $q_{j(i)}$ values at each iteration directly from the definition (5.9). Instead, we shall derive a more efficient procedure. Let $j(1), j(2), ..., j(k)$ be the elements of $S^{(j-1)}(k)$ ordered to indicate the inclusions (5.8). Suppose that h is the index computed above, so that the tower at the end of iteration j is ordered

$$j(1), j(2), ..., j(h), j, j(h+1), ..., j(k);$$

of course, $j(k)$ might be deleted, but assume that it is not. (An identical argument proves the other case.) Let $j'(i)$, $i = 1, ..., k+1$, denote the ith element of this sequence.

At the start of iteration j, each job k in this tower has effective processing time q_k. (For job j, let $q_j = p_j$.) We have already seen that

$$C^{(j)}(i) = \begin{cases} C^{(j-1)}(i), & \text{if } i = 1, ..., h, \\ \max\{r_j, C^{(j-1)}(i-1)\} + p_j, & \text{if } i = h+1, ..., k+1. \end{cases}$$

This recurrence for $C^{(j)}(i)$ can be used to show that the effective processing times used in iteration j,

$$q_{j'(i)} = \max\{r_j, C^{(j)}(i)\} - \max\{r_j, C^{(j)}(i-1)\}, \tag{5.11}$$

by applying the recurrence separately in each of the three cases: (a) $i = 1, ..., h$; (b) $i = h+1$; and (c) $i = h+2, ..., k+1$. Therefore, to update these values for the next iteration, we need only update them to reflect the new release date r_{j+1}. If the release date is increased by one unit, the effect is to decrease by one the effective processing time of the first job in the tower that currently has a positive effective processing time. The same idea can be used to update the release date from r_j to r_{j+1} in (5.11), by repeating this $r_{j+1} - r_j$ times; this can clearly be made efficient by making the maximum allowed reduction to $q_{j'(i)}$ at once, for each $i = 1, ..., k$, in that order.

We can now show that (5.10) is maintained by the algorithm. In iteration j, j is inserted into the tower between $j(h)$ and $j(h+1)$ according q_j's place in the sorted order of effective processing times. Furthermore, after the update just described, the modified effective processing times are still sorted in nondecreasing order.

In order to implement the algorithm we propose to maintain two sets, S and S', where S contains jobs whose effective processing times are strictly positive and S' contains jobs with zero effective processing times. At the end of iteration j, the maximum feasible set in the tower is then $S^{(j)} = S \cup S'$. We propose to implement the set S by a priority queue that supports the operations of *insert*, *deletemin*, and

deletemax. For the set S', we need only support the operation *insert*. At iteration j, the following subroutine is performed to modify the effective processing times.

```
modify( ):
r := r_j − r_{j−1};
while (r > 0 and S ≠ ∅) do
    i := deletemin(S);
    q(S) := q(S) − q_i;
    if (q_i ≤ r) then
        insert(S', i);
        r := r − q_i;
    else
        q_i := q_i − r;
        insert(S, i);
        q(S) := q(S) + q_i;
        r := 0;
    end
end
```

We can now implement the recurrence relations (5.7) as follows:

```
similar( ):
S := ∅;
S' := ∅;
q(S) := 0;
for j = 1 to n do
    call modify( );
    q_j := p_j;
    insert(S, j);
    q(S) := q(S) + q_j;
    if (q(S) > d_j − r_j) then
        l := deletemax(S);
        q(S) := q(S) − q_l;
    end
end
return (S ∪ S');
```

Note that $O(n)$ of each of the operations *insert*, *deletemin*, and *deletemax* are performed in the course of the computation. Each of these operations takes $O(\log n)$ time. Hence the overall running time of the algorithm is bounded by $O(n \log n)$.

Exercises

5.15. Devise an example to show that the tower-of-sets property (5.8),(5.10), does

not hold for $1|r_j|\sum w_j U_j$ with similarly ordered release dates and due dates and with oppositely ordered processing times and weights.
5.16. Show that the tower-of-sets property does hold for the problem $1|r_j, p_j = 1|\sum w_j U_j$ in the case of similarly ordered release dates and due dates. Devise an $O(n \log n)$ algorithm for solving this problem (cf. Exercise 5.2).
5.17. If V_j is defined as in Section 5.4, show that the problem $1|pmtn, r_j|\sum w_j V_j$ can be solved in $O(n \log n)$ time, under the condition that $[r_{j,} d_j]$ intervals are nested. Processing times and job weights may be arbitrary. (Hint: Modify the algorithm for Exercise 5.14, so as to remove jobs from an infeasible set $S^{(j)}$ in nonincreasing order of the ratios w_i/p_i, if necessary fractionalizing the last job removed and reinserting it, until $p(S^{(j)}) = d_j - r_j$.)
5.18. Devise an $O(nW^2)$ algorithm for $1|pmtn, r_j|\sum w_j U_j$ with nested release date-due date intervals (and arbitrary processing times and job weights).
5.19. Show that $1|r_j|\sum U_j$ is strongly NP-hard.

5.6. The general problem $1|pmtn, r_j|\sum w_j U_j$

The problem $1|pmtn, r_j|\sum w_j U_j$ can be solved by dynamic programming in $O(nk^2W^2)$ time, where k is the number of distinct release dates. When release dates and due dates are similarly ordered, the dynamic programming recurrences specialize to (5.6), which can be solved in $O(nW)$ time. And when $k = 1$, the recurrences further specialize to (5.1).

Our dynamic programming recurrences iteratively compute two types of values, $C^{(j)}(r, w)$ and $P^{(j-1)}(r, r', w)$. In the following definitions, j is an index, $1 \le j \le n$, r and r' are release dates, $r \le r'$, and w is an integer, $0 \le w \le W$. Once again, we assume the jobs are indexed in EDD order.

We define

$$C^{(j)}(r, w) = \min\{c(S)\},$$

where the minimum is taken with respect to feasible sets S such that

$$S \subseteq \{1, 2, ..., j\}, r(S) \ge r, \text{ and } w(S) \ge w.$$

If there is no such feasible set S, $C^{(j)}(r, w) = +\infty$. We also define $P^{(j-1)}(r, r', w)$ to be the minimum amount of processing *after time* r_j in an extended EDD schedule, with respect to feasible sets S such that

$$S \subseteq \{1, 2, ..., j-1\},\ r(S) \ge r,\ c(S) \le r',\ \text{and } w(S) \ge w.$$

If there is no such feasible set, $P^{(j-1)}(r, r', w) = +\infty$.

It follows from the above definitions that the maximum weight of a feasible set is given by the largest value of w such that $C^{(n)}(r_{\min}, w)$ is finite, where $r_{\min} =$

$\min_j\{r_j\}$.

Computation of $C^{(j)}(r,w)$. We shall compute the values $C^{(j)}(r,w)$ in n iterations, $j = 1,2,...,n$, starting with the initial conditions

$$C^{(0)}(r,w) = \begin{cases} r, & \text{if } w = 0, \\ +\infty, & \text{otherwise.} \end{cases}$$

Observe that j cannot be contained in a feasible set S with $r(S) > r_j$. Hence

$$\begin{aligned} C^{(j)}(r,w) &= C^{(j-1)}(r,w), \text{ if } r > r_j, \\ &\leq C^{(j-1)}(r,w), \text{ otherwise.} \end{aligned}$$

It follows that at iteration j we have only to compute the values of $C^{(j)}(r,w)$ for which $r \leq r_j$. So let $r \leq r_j$ and suppose $S \subseteq \{1,2,...,j\}$ realizes the value $C^{(j)}(r,w)$. We distinguish three cases, as follows.

Case 1. $j \notin S$. Then

$$C^{(j)}(r,w) = C^{(j-1)}(r,w).$$

Case 2. $j \in S$, and the processing of job j begins after all jobs in $S - \{j\}$ are completed in the extended EDD schedule. We then have

$$C^{(j)}(r,w) = \max\{r_j, c(S - \{j\})\} + p_j.$$

We may assume that $S - \{j\}$ is such that

$$c(S - \{j\}) = C^{(j-1)}(r, w - w_j).$$

(If not, replace $S - \{j\}$ by a feasible subset of $\{1,2,...,j-1$ for which this is so.) Then

$$C^{(j)}(r,w) = \max\{r_j, C^{(j-1)}(r, w - w_j)\} + p_j.$$

Case 3. $j \in S$, and the processing of job j begins before all jobs in $S - \{j\}$ are completed in the extended EDD schedule of S. This means that there is idle time within the interval $[r_j, c(S - \{j\})]$ in the EDD schedule for $S - \{j\}$. Recall from Chapter 4 that a block indexes a maximal subset of jobs that are processed continuously without idle time. Let S' be the *last block* in the extended EDD schedule of $S - \{j\}$, i.e.,

$$r(S') = \max\{r(B) | B \text{ a block of the extended EDD schedule of } S - \{j\}\},$$

where $r(S') > r_j$. Let $r' = r(S')$ and $w' = w(S')$. It must be the case that

$$c(S') = C^{(j-1)}(r',w'),$$

else S is not optimal. We may also assume that the total amount of processing done on jobs in $(S-\{j\})-S'$ in the interval $[r_j,r']$ does not exceed $P^{(j-1)}(r,r',w-w_j-w')$. This means that the total amount time available for the processing of job j in the interval $[r_j,r']$ is

$$(r'-r_j)-P^{(j-1)}(r,r(S'),w-w_j-w(S')),$$

and the amount of processing done on job j after time r' is

$$\max\{0,p_j-(r'-r_j)+P^{(j-1)}(r,r',w-w_j-w')\}.$$

Hence the completion time of the last job in S is

$$C^{(j-1)}(r',w')+\max\{0,p_j-(r'-r_j)+P^{(j-1)}(r,r',w-w_j-w')\}. \tag{5.12}$$

Now observe that the expression (5.12) must be minimum with respect to sets S', with $r(S') > r_j$, $w' = w(S') \leq w-w_j$. In other words,

$$C^{(j)}(r,w) = \min_{r',w'}\{C^{(j-1)}(r',w')+max\{0,p_j-r'+r_j+P^{(j-1)}(r,r',w-w_j-w')\}\}$$

Putting Cases 1, 2, and 3 together, for $r \leq r_j$ we have the recurrence relations

$$C^{(j)}(r,w) = \min\left\{\begin{array}{l} C^{(j-1)}(r,w), \\ \max\{r_j,C^{(j-1)}(r,w-w_j)\}+p_j, \\ \min_{r',w'}\{C^{(j-1)}(r',w')+\max\{0,p_j-r'+r_j+P^{(j-1)}(r,r',w-w_j-w')\}\} \end{array}\right\}, \tag{5.13}$$

where the inner minimization is taken over all distinct release dates $r' > r_j$ such that $r' \in \{r_1,r_2,...,r_{j-1}\}$ and all integers w', $0 < w' \leq w-w_j$. It is important to note that (5.13) is valid only if the right hand side does not exceed d_j ; if this is not so, set $C^{(j)}(r,w) = +\infty$.

Computation of $P^{(j-1)}(r,r',w)$. We shall now derive recurrence relations for computing $P^{(j-1)}(r,r',w)$, for all distinct release dates r,r', with $r \leq r'$.

We have as initial conditions

$$P^{(j-1)}(r,r',0) = 0.$$

If $w > 0$, then $P^{(j-1)}(r,r',w)$ is realized by a nonempty set $S \subseteq \{1,2,...,j-1\}$. We distinguish two cases.

Case 1. $r(S) > r$. Then

$$P^{(j-1)}(r,r',w) \leq P^{(j-1)}(r^+,r',w),$$

where we define r^+ to be the smallest distinct due date larger than r.

Case 2. $r(S) = r$. Let $S' \subseteq S$ be the block of S such that $r(S') = r$, and let w(S') = w'. We may assume that $c(S') = C^{(j-1)}(r,w'))$. Hence the total amount of processing done on S' in the interval $[r_j, r']$ is

$$\max\{0, C^{(j-1)}(r,w') - r_j\}.$$

Let r'' be the smallest release date greater than or equal to $C^{(j-1)}(r,w')$. It must be the case that the total amount of processing done on $S - S'$ in the interval $[r_j, r']$ is $P^{(j-1)}(r'',r',w-w')$. Hence the total amount of processing done on S in the interval $[r_j, r']$ is

$$\max\{0, C^{(j-1)}(r,w') - r_j\} + P^{(j-1)}(r'',r',w-w'). \tag{5.14}$$

Now observe that the expression (5.14) must be minimum with respect to sets S', with $r(S') = r$, $c(S') \leq r'' \leq r'$, and $w(S') \leq w$. That is,

$$P^{(j-1)}(r,r',w) = \min_{0<w'\leq w}\{\max\{0, C^{(j-1)}(r,w') - r_j\} + P^{(j-1)}(r'',r',w-w')\},$$

where r'' is the smallest release date no less than $C^{(j-1)}(r,w')$.

Putting the above two cases together, we have

$$P^{(j-1)}(r,r',w) =$$
$$\min\left\{\begin{array}{l} P^{(j-1)}(r^+,r',w), \\ \min_{0<w'\leq w}\{\max\{0, C^{(j-1)}(r,w') - r_j\} + P^{(j-1)}(r'',r',w-w')\} \end{array}\right\}, \tag{5.15}$$

giving us the recurrences we need.

We shall now analyze the time and space complexity of the dynamic programming computation. At each of n iterations, $j = 1,2,...,n$, there are $O(k^2W)$ of the $P^{(j-1)}(r,r',w)$ values to compute, one for each combination of r,r',w. By (5.15), each $P^{(j-1)}(r,r',w)$ is found by minimization over $O(W)$ choices of $w' \leq w$. Hence the time required to compute the $P^{(j-1)}(r,r',w)$ values at each iteration is bounded by $O(k^2W^2)$. There are $O(kW)$ of the $C^{(j)}(r,w)$ values to compute, one for each combination of r and w. By (5.13), each $C^{(j)}(r,w)$ is found by minimization over $O(kW)$ choices of r',w'. Hence the time required to compute the $C^{(j)}(r,w)$ values at each iteration is bounded by $O(k^2W^2)$. It follows that the overall time bound for these computations is $O(nk^2W^2)$. Space requirements are clearly bounded by $O(k^2W)$.

As we have observed, the maximum weight of a feasible subset can be obtained by finding the maximum value of w such that $C^{(n)}(r_{\min},w)$ is finite. (The $O(W)$ time required for this is dominated by the time required for other computations.) In

practice, however, one wants to be able to construct a maximum-weight feasible set, not simply to find its weight. The most straightforward way to do this is to compute an incidence vector of the set realizing each $P^{(j-1)}(r,r',w)$ and $C^{(j)}(r,w)$ value. The computation of these incidence vectors can be carried out with an expenditure of $O(n^2k^2W)$ time, which is dominated by the $O(nk^2W^2)$ time bound obtained above. However, because $O(k^2W)$ n-vectors must be stored, this approach increases space requirements to $O(nk^2W)$.

We claim that it is possible to use pointers to construct a maximum-weight feasible set, while maintaining the time and space bounds of $O(nk^2W^2)$ and $O(k^2W)$. We leave this as an exercise.

The EDD Rule creates preemptions only at release dates. Hence when the jobs in a maximum-weight feasible set are scheduled, at most $k-1$ preemptions are created, at most one at each distinct release date other than the first.

Observe that when release dates and due dates are similarly ordered, there are no release dates $r' > r_j$ over which the inner minimization in (5.14) can be, hence these recurrence relations simplify to

$$C^{(j)}(r,w) = \min\left\{ \begin{array}{l} C^{(j-1)}(r,w), \\ \max\{r_j, C^{(j-1)}(r,w-w_j)\} + p_j \end{array} \right\}.$$

Now let $C^{(j)}(w) = C^{(j)}(r_{\min},w)$ and we have simply

$$C^{(j)}(w) = \min\left\{ \begin{array}{l} C^{(j-1)}(w), \\ \max\{r_j, C^{(j-1)}(w-w_j)\} + p_j \end{array} \right\},$$

and we have the recurrence equations (5.6).

When all release dates are equal,

$$\max\{r_j, C^{(j-1)}(w-w_j)\} = C^{(j-1)}(w-w_j),$$

and the recurrence further simplifies to

$$C^{(j)}(w) = \min\{C^{(j-1)}(w), C^{(j-1)}(w-w_j) + p_j\},$$

and we have the recurrence equations (5.1).

Exercises

5.20. Show that it is possible to use pointers to construct a maximum-weight feasible set, while maintaining the time and space bounds of $O(nk^2W^2)$ and $O(k^2W)$.

5.7. Precedence constraints

When the jobs are related by precedence constraints, severely restricted versions of the $1||\sum w_j U_j$ problem become *NP*-hard. We will show below that the case of unit weights and unit processing times, $1|prec, p_j = 1|\sum U_j$, is *NP*-hard. We will then refine this result and show that it still holds for the case of chain-type precedence constraints, where each job has at most one predecessor and at most one successor.

Theorem 5.2. $1|prec, p_j = 1|\sum U_j$ *is NP-hard in the strong sense.*

Proof. We will show that the clique problem reduces to the decision version of $1|prec, p_j = 1|\sum U_j$. An instance of the clique problem is given by a graph $G = (V, E)$ and an integer k. The question is if G has a clique (i.e., a complete subgraph) on at least k vertices.

Given an instance of clique, let $l = \binom{k}{2}$ denote the number of edges in a clique of size k. The corresponding instance of the scheduling problem will have $n = |V| + |E|$ unit-time jobs. We introduce a job v for every vertex $v \in V$ and job e for every edge $e \in E$, with a precedence constraint $v \rightarrow e$ whenever v is an endpoint of e. Each "vertex job" v has a due date $d_v = n$, and each "edge job" e has a due date $d_e = k + l$. Note that no vertex job can be late in a schedule without idle time. We claim that there is a schedule with at most $|E| - l$ late jobs if and only if G has a clique of size at least k.

Suppose that a clique on k vertices exists. We first schedule the k corresponding vertex jobs in the interval $[0, k]$. In view of the precedence constraints, we can then schedule the l jobs corresponding to the clique edges in the interval $[k, k + l]$; they are on time. They are followed by the remaining vertex jobs in $[k + l, |V| + l]$ and, finally, the remaining edge jobs in $[|V| + l, n]$; these $|E| - l$ edge jobs are late. This schedule meets the claimed bound on the number of late jobs.

Conversely, suppose that there is a schedule in which at most $|E| - l$ jobs are late, or, equivalently, in which at least $l = \binom{k}{2}$ edge jobs are completed by time $k + l$. These jobs must be preceded by at least k vertex jobs. It follows that there is a set of k vertex jobs that releases l edge jobs for processing or, in other words, that a clique of size k exists. □

Theorem 5.3. $1|chain, p_j = 1|\sum U_j$ *is NP-hard in the strong sense.*

Proof. To simplify the exposition, we present the reduction in two stages. We first show that the exact 3-cover problem reduces to the decision version of $1|chain|\sum U_j$, with general processing times. We then show that this problem can be reduced to an equivalent problem with unit processing times.

An instance of exact 3-cover consists of a set $T = \{1, ..., 3t\}$ and a family $\mathcal{S} = \{S_1, ..., S_s\}$ of 3-element subsets of T. It is a yes-instance if $\mathcal{S}$ includes an exact cover, i.e., a subfamily of t subsets whose union is T (cf. Figure 5.6(a).

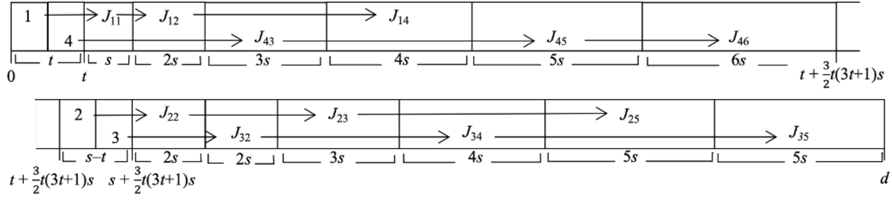
The corresponding instance of $1|chain|\sum U_j$ will have a job i for each subset S_i and a job J_{ij} for each occurrence of an element j in a subset S_i. More precisely, for each $S_i = \{j, k, l\} \in \mathcal{S}$ $(i = 1, ..., s)$, we introduce four jobs, i, J_{ij}, J_{ik}, J_{il}, which are

$t = 2, s = 4$

T	1	2	3	4	5	6
S_1	1	2		4		
S_2		2	3		5	
S_3		2		4	5	
S_4			3		5	6

$\{S_1, S_4\}$ is an exact 3-cover of T.

(a) A yes-instance of EXACT 3-COVER.



(b) The corresponding schedule.

Figure 5.6. Reduction of exact 3-cover to $1|chain|\sum U_j$.

related by chain-like precedence constraints, $i \to J_{ij} \to J_{ik} \to J_{il}$. Each 'subset job' i is of length 1, and each 'occurrence job' J_{ij} is of length js. The sum of all of the processing times (and hence the length of the schedules we will be considering) is denoted by $d = s + \sum_{i=1}^{s} \sum_{j \in S_i} js$. All subset jobs i have due date d; none of these can be late. Each occurrence job J_{ij} is assigned a due date d_{ij} defined by

$$d_{ij} = t + s + 2s + ... + js = t + j(j+1)s/2, \ j \in S_i, \ i = 1, ..., s.$$

We claim that there is a schedule with no more than $3(s-t)$ late jobs if and only if there is an exact 3-cover.

Suppose that $\mathcal{S}$ includes an exact 3-cover $\mathcal{S}'$. We then construct the following schedule (cf. Figure 5.6(b)). First, the t jobs i corresponding to the subsets $S_i \in \mathcal{S}'$ are scheduled in the interval $[0,t]$. Now note that each of these t subset jobs heads a chain of three occurrence jobs such that the $3t$ elements to which these occurrences refer are all distinct. That is, for every element $j \in T$, there is exactly one occurrence job J_{ij} for which the preceding subset job i has been scheduled; this J_{ij} is now scheduled in the interval $[d_{ij} - js, d_{ij}]$. In this way, $3t$ occurrence jobs occupy the interval $[t, t + 3t(3t+1)s/2]$; they are all on time. They are followed by the remaining $s-t$ subset jobs and, finally, the remaining $3(s-t)$ occurrence jobs; these latter jobs are late. This schedule meets the claimed bound on the number of late jobs.

Conversely, suppose that there exists a feasible schedule in which at most $3(s-t)$

jobs are late, or, equivalently, in which at least $3t$ occurrence jobs are on time. This implies that,

$$\text{for every element } j \in T, \text{ exactly one of its occurrence jobs } J_{ij} \text{ is on time.} \quad (5.16)$$

The verification of this crucial implication is left to the reader (see Exercise 5.21). Assertion (5.16), in turn, implies that the amount of time available for processing the subset jobs that release these $3t$ occurrence jobs is bounded from above by

$$\max_{j\in T} d_{ij} - \sum_{j\in T} js = t + 3t(3t+1)s/2 - 3t(3t+1)s/2 = t.$$

Hence, $\mathcal{S}$ must include a subfamily of at most t subsets whose union covers T. This subfamily constitutes an exact 3-cover.

This completes the first stage of the proof. It remains to be shown that the scheduling problem can be reduced to an equivalent problem with unit processing times. This is straightforward. We replace each occurrence job J_{ij} by a chain of js unit-time jobs, $J_{ij}^{(1)} \to ... \to J_{ij}^{(js-1)} \to J_{ij}^{(js)}$, with due dates $d_{ij}^{(1)} = ... = d_{ij}^{(js-1)} = d$, $d_{ij}^{(js)} = d_{ij}$ $(j \in S_i, i = 1,...,s)$. Given any feasible schedule in which the jobs of such a chain are not processed consecutively, we can obtain another schedule by moving the first $js-1$ jobs of the chain to the right, up to the jsth one, thereby moving some other jobs to the left. The new schedule is still feasible, since no precedence constraints are violated, and it has no more late jobs, since the jobs that are moved to the right cannot be late. Hence, each chain $J_{ij}^{(1)} \to ... \to J_{ij}^{(js)}$ can be considered as a single job J_{ij} with processing time js and due date d_{ij}. As for the size of the final $1|chain, p_j = 1|\sum U_j$ instance, note that it has only $d < 9s^2t$ jobs; the entire transformation is polynomial because sufficiently small processing times have been chosen at the first stage. □

Exercises

5.21. (a) Consider the $1||\sum U_j$ problem. Suppose that there are t nonempty job sets, $\text{FSJ}_1,...,\text{FSJ}_t$, such that all jobs in FSJ_j have a processing time j and a due date $1+2+...+j$ $(j = 1,...,t)$. Prove that in each optimal schedule exactly one job from FSJ_j finishes at its due date, while the other jobs from FSJ_j are late $(j = 1,...,t)$. (b) Use (a) to verify assertion (5.16) in the proof of Theorem 5.3.

Notes

5.1. *Some preliminaries.* Each feasible set for an instance of the problem $1|r_j, p_j = 1|\sum w_j U_j$ is an independent set of a transversal matroid, hence the algorithm described in Exercise 5.2 is indeed an instance of the matroid greedy algorithm. The observation that $1|p_j = 1|\sum U_j$ can be solved in $O(n)$ time is due to Monma (1982).

5.2. *Dynamic programming solution of* $1||\sum w_j U_j$. Karp (1972) gave the reduction of the knapsack problem to $1|d_j = d|\sum w_j U_j$. The integer linear programming formulation of $1||\sum w_j U_j$ and the dynamic programming algorithm for solving this

generalization of the knapsack problem are adapted from (Lawler & Moore, 1969). Exercise 5.4 is due to Potts & Van Wassenhove (1988), who use linear relaxations as lower bounds within a branch and bound procedure for $1||\sum w_j U_j$.

5.3. *A fully polynomial approximation scheme.* Theorem 5.1 is due to Gens & Levner (1978). By obtaining a preliminary upper bound on the optimum that is within a factor of 2, Gens & Levner (1981) improved the running time to $O(n^2 \log n + n^2 k)$. Exercise 5.6 is due to Sahni (1976). Ibarra & Kim (1978) gave a polynomial approximation scheme to find a maximum feasible set for $1|d_j = d, tree|\sum w_j U_j$.

5.4. *The Moore-Hodgson algorithm.* Moore (1968) suggested a less elegant algorithm for the unweighted problem $1||\sum U_j$ but specifically credited the recurrences (5.2) to a suggestion by his colleague Hodgson, hence the appellation *Moore-Hodgson*. Maxwell (1970) provided an alternative derivation of the Moore-Hodgson algorithm based on ideas from linear and integer programming. The generalization of the algorithm to oppositely ordered processing times and job weights was noted by Lawler (1976A). Sidney (1973) observed that the algorithm could be adapted to the case considered in Exercise 5.12. The NP-hardness of the problem $1|\bar{d}_j|\sum U_j$ was proved by Lawler (1982B). The Moore-Hodgson algorithm and the algorithm of Lawler (1976A) were used to obtain lower bounds in the branch and bound procedure of Villareal & Bulfin (1983) to solve $1||\sum w_j U_j$ with arbitrary weights and processing times.

5.5. *Similarly ordered release dates and due dates.* The strong NP-hardness proof of $1|r_j|\sum U_j$ is due to Lenstra (–). An $O(n^2)$ dynamic programming solution of $1|r_j|\sum U_j$, under the condition of similar ordering of release dates and due dates, is described by Kise, Ibaraki & Mine (1978). The $O(n \log n)$ algorithm for this problem is due to Lawler (1982B). The tower-of-sets property is known to hold for a number of special cases of $1|pmtn, r_j|\sum w_j U_j$, aside from those mentioned in the exercises, for example, as in the case that release dates and processing times are similarly ordered, and in opposite order to job weights, which is due to Lawler (–).

5.6. *The general problem* $1|pmtn, r_j|\sum w_j U_j$. The dynamic programming algorithm of this section is adapted from Lawler (1990), which improves on a less efficient algorithm given earlier by Lawler (1982B).

5.7. *Precedence constraints.* Theorem 6.2 is due to Garey and Johnson (1976), Theorem 6.3 to Lenstra and Rinnooy Kan (1980). Ibaraki, Kise, and Mine (1976) proved that $1|chain, r_j, p_j = 1|\sum U_j$ is NP-hard.