Annals of Operations Research 69(1997)135–156

A dynamic programming algorithm for single machine scheduling with ready times

Sylvie Gélinas and François Soumis
GERAD and École Polytechnique de Montréal, Université McGill, 5255, avenue Decelles, Montréal, Québec, Canada H3T 1V6

We propose a dynamic programming algorithm for the single machine scheduling problem with ready times and deadlines to minimize total weighted completion time. Weights may be positive or negative and the cost function may be non-regular. This problem appears as a subproblem in the Dantzig–Wolfe decomposition of the job-shop scheduling problem. We show that the algorithm is polynomial if time window length is bounded by a constant and times are integer-valued. We present computational results for problems with up to 200 jobs.

1. Introduction

The single machine scheduling problem with ready times and deadlines to minimize total weighted completion time, denoted $1/r_i, d_i/\sum w_i C_i$, is defined as follows. Each job in a set $N = \{1, 2, \ldots, n\}$ is to be processed without interruption on a single machine. Let job i have process time p_i, ready time r_i, deadline d_i and weight w_i. The completion time for job i is denoted by C_i. A positive weight w_i indicates that job i should terminate as quickly as possible; a negative weight indicates that it should terminate as late as possible. We wish to sequence all jobs in such a way that the weighted sum of their completion times is minimized.

This problem with non-regular cost functions on operation completion time arises frequently in practical scheduling problems. In the following example, these functions are linear. Consider n jobs that must be carried out on a single machine. Each job must terminate during a client-specified time interval. Some have been requested by external clients and will generate revenue for the firm; others have been requested by internal clients and will cost the firm money. It is clearly preferable to terminate revenue-generating jobs as soon as possible and cost-generating jobs as late as possible. This kind of situation occurs when a firm must schedule production and maintenance operations on a machine. The proposed algorithm can also be used to handle piecewise linear cost functions, as in the following example. Consider a set of jobs, each of which should ideally terminate on a date set by the client. Costs are

incurred if a job terminates too soon (e.g., storage of product until client is ready to receive it, insurance costs, product deterioration, low capital productivity) or too late (client dissatisfaction, penalty for breach of contract, loss of sale). The problem examined in this article can also be applied to vehicle routing. For example, consider a truck that delivers goods to clients and picks up goods from suppliers. Clients and suppliers must be visited during specified time intervals. To free truck space and reduce freight-handling costs, it is preferable to visit clients as early as possible and suppliers as late as possible.

The problem $1/r_i, d_i/\sum w_i C_i$ appears also as a subproblem in the Dantzig–Wolfe decomposition of the job-shop scheduling problem, in which n jobs are scheduled on m machines [12]. Precedence constraints define the machine sequence for each job, and machine constraints prevent two jobs from being processed simultaneously on the same machine. The problem is to determine the sequence of jobs on each machine such that the last job to terminate does so as soon as possible. In the formulation considered here, the precedence constraints are left in the master problem and the machine constraints are transferred to the subproblem. Each subproblem sequences jobs on a single machine such that the weighted sum of the completion times is minimized. Weights w_i of arbitrary sign are defined using the dual variables in the master problem. The problem $1/r_i, d_i/\sum w_i C_i$ also appears as a subproblem in the formulation of the job-shop scheduling problem by Fisher et al. [11] using surrogate constraints. These authors noted at the time that no efficient algorithm existed to solve the subproblem.

The problem without time constraints, $1//\sum w_i C_i$, can be solved in polynomial time using Smith's rule, according to which it suffices to list jobs in non-decreasing order by p_i/w_i. Lenstra et al. [17] show that $1/r_i/\sum w_i C_i$ and $1/d_i/\sum w_i C_i$ are strongly NP-hard. Bianco and Ricciardelli [7], Hariri and Potts [13] and Belouadah et al. [6] propose algorithms to solve the problem with ready times ($1/r_i/\sum w_i C_i$). The optimal solution is bounded if all weights w_i are nonnegative. Bansal [5], Potts and Van Wassenhove [19] and Posner [18] study the problem with deadlines and nonnegative weights ($1/d_i/\sum w_i C_i$, $w_i \geq 0$). These authors propose branch-and-bound algorithms with dominance rules capable of solving problems with up to 50 jobs. A lower bound is obtained by Lagrangian relaxation on the time constraints or by allowing splitting. Dyer and Wolsey [9] propose and discuss various mixed-integer programming approaches to obtaining lower bounds for the problem with ready times. Bagchi and Ahmadi [3] improve on the lower bound obtained by Posner [18] for the problem with deadlines; like Posner, they use the splitting relaxation to obtain the lower bound.

Few authors have written about the more general problem involving ready times, deadlines and weights of arbitrary sign ($1/r_i, d_i/\sum w_i C_i$). Sousa and Wolsey [22] define binary variables for each job and each time period, and solve the problem using cuts and a branch and bound scheme. Computational tests are carried out using weights $w_i \geq 0$ on 20-job problems with ready times and 30-job problems with deadlines. Values for p_i and w_i are chosen to be integer and falling within [1, 5] and [1, 10],

respectively. While the authors note that this algorithm can also be used to solve the problem $1/r_i, d_i/\sum w_i C_i$ with weights of arbitrary sign, they do not provide computational results. Erschler et al. [10], in considering the problem with ready times and deadlines, establish a dominance relationship between job sequences. While the number of sequences can be reduced using this relationship if certain criteria are minimized, it is not valid for minimizing total weighted completion time.

The effectiveness of branch-and-bound methods that relax time constraints diminishes when such constraints are tight. Dynamic programming approaches, on the other hand, can take advantages of tighter time constraints to reduce the state space. Dynamic programming approaches have been used primarily to minimize regular functions, that is, non-decreasing functions of completion time. Baker and Schrage [4], Schrage and Baker [21] and Lawler [16] propose dynamic programming algorithms for the problem $1//\sum g_i(C_i)$, where g_i is a regular function. They take advantage of precedence constraints, which are generated from dominance relations. Kao and Queyranne [15] modify the Schrage–Baker algorithm to reduce the amount of computer memory required to solve the problem. Potts and Van Wassenhove [20] add a dominance relationship to Lawler's algorithm to reduce the number of states.

Comparisons of the Schrage–Baker, Lawler and modified algorithms have been carried out by Kao and Queyranne [15] for assembly line balancing problems; by Potts and Van Wassenhove [20] for the single machine scheduling problem to minimize total tardiness; and by Abdul-Razaq et al. [2] for problems to minimize total weighted tardiness. Computer memory limitations pose more of a problem for these algorithms than solution time. In general, problems of up to 50 jobs can be solved. The Schrage–Baker algorithm has the advantage of being easy to program; furthermore, requirements for its execution are known in advance. Lawler's algorithm uses less memory.

Abdul-Razaq and Potts [1], in studying the problem $1//\sum g_i(C_i)$, make no assumptions about the monotonicity of the function g_i. They do not allow idle time, however. A lower bound is obtained by dynamic programming. The state graph is mapped onto a smaller graph. The solution consists of a sequence of jobs; a job can be missing from the sequence or can appear several times. The lower bound is improved by penalizing certain jobs and by eliminating sequences in which a job appears in two adjacent positions. Penalties are adjusted using a subgradient algorithm. Abdul-Razaq and Potts use the bound obtained by dynamic programming in a branch-and-bound procedure. They conduct computational tests on problems with up to 25 jobs and with early and tardy completion costs. Abdul-Razaq et al. [2] compare this algorithm to the Schrage–Baker and Lawler algorithms. The Abdul-Razaq–Potts algorithm requires a large amount of CPU time to calculate the lower bound for a branch node.

Cheng [8] has published recursion equations for the scheduling problem in which the sum of deviations between a job's completion time and its due date is minimized. No idle time is permitted. Computational results are not provided.

This paper applies dynamic programming to problems of scheduling on one machine with a non-regular cost function. A different problem with some similarities

is discussed in Ioachim et al. [14], who use dynamic programming to solve the shortest path problem with linear schedule costs. Both problems consider positively or negatively sloped linear cost functions on schedule variables. A job sequence in the scheduling problem corresponds to a path in the shortest path problem. The scheduling problem considered here, however, is more difficult than the shortest path problem. In the scheduling problem, a sequence must contain all jobs (like the path of a travelling salesman must visit all nodes), while the shortest path problem does not require this. Some of the results developed in Ioachim et al. will be taken up again in this paper.

The following sections describe the dynamic programming approach and algorithm, discuss its computer implementation and complexity, and present some computational results.

2. Dynamic programming approach

The states in the dynamic programming approach are subsets of jobs. For each state $S \subseteq N$, let $F(S, t)$ be the minimum cost of a schedule that processes all jobs in S exactly once and terminates at time t; similarly, let $G(S, t)$ be the minimum cost of schedule that processes all jobs in S exactly once and terminates at time $t' \leq t$. The functions F and G can be calculated recursively, using the following relationships:

$$F(S,t) = \begin{cases} \min\limits_{i \in S} \{G(S - \{i\}, t - p_i) + w_i t\}, & \text{if } t \in [r_i + p_i, d_i], \\ \infty, & \text{otherwise.} \end{cases}$$

$$G(S,t) = \min_{t' \leq t} F(S,t').$$

Initially, we set

$$F(\{i\},t) = \begin{cases} w_i t, & \text{if } t \in [r_i + p_i, d_i], \\ \infty, & \text{otherwise.} \end{cases}$$

The value of the solution to the scheduling problem is given by $G(N, T)$, where $T = \max_{i \in N} d_i$.

Property 1. $F(S, t)$ is a piecewise linear function of t.

Proof. Let C be a sequence of jobs and let $f(C, t)$ be the minimum cost of a schedule for C that terminates at time t. Ioachim et al. [14] proved that $f(C, t)$ is a piecewise linear function of t. $F(S, t)$ is the minimum of $f(C, t)$ with respect to C for sequences C that contain all jobs in S exactly once. F is therefore the minimum of piecewise linear functions. Since the number of sequences is finite, F is piecewise linear as well. □

Property 2. $G(S, t)$ is a non-increasing, piecewise linear function of t.

Proof. This result follows immediately from the definition of $G(S, t)$. □

3. Dynamic programming algorithm

The dynamic programming algorithm requires (1) a procedure for enumerating sets of jobs; (2) a state management procedure capable of easily locating sets of jobs to be considered in calculating the cost function; and (3) efficient structures for representing and manipulating cost functions.

State enumeration and management procedures developed by Schrage and Baker [21] and Lawler [16] can be used for the problem $1/r_i, d_i/\sum w_i C_i$. In section 3.1, we describe these procedures and propose a procedure similar to Lawler's for solving the present problem. Cost calculations for states are more difficult, however. Schrage, Baker and Lawler consider problems without time constraints, with the objective of minimizing a regular function. The recursion equations simplify in this case and can be solved when only a single value for each state S is known, namely the minimum cost for performing all jobs in the set S. The function $G(S, t)$ must be known for the problem $1/r_i, d_i/\sum w_i C_i$. Section 3.2 presents two approaches to calculating these functions. While Schrage, Baker and Lawler consider precedence constraints, they only study feasible sets, that is, sets that contain a job only if they contain all of its predecessors. The approach presented here places constraints on time rather than precedence; these constraints induce precedence relations between jobs. This aspect of the approach is discussed in section 4, as are additional tests that can be applied to reduce the state space.

3.1. State generation and management

A set S is represented by a binary vector with dimension equal to the number of jobs. This vector is stored using a sequence of integers in which the number of bits is greater than or equal to the number of jobs. If a job belongs to a set S, its corresponding bit is set to 1.

Schrage and Baker [21] proposed an approach in which a state S is processed by considering all states $S - \{i\}$ that can lead to S. Sets are enumerated in lexicographic order, ensuring that states $S - \{i\}$ are enumerated before state S. Schrage and Baker use a compact labelling procedure in which labels are selected so that a unique label is assigned to each feasible set. The cost of a feasible set is saved in a table at an address given by the label. Some table addresses, however, are not used.

Lawler [16] proposed an algorithm in which sets are generated in order of cardinality. At iteration k of the algorithm, all feasible k-element sets S are enumerated and saved in a list in lexicographical order. Feasible $(k + 1)$-element sets are then generated using the following procedure. A list of new sets of the form $S \cup \{1\}$ is formed by adding job 1 to all S such that $1 \notin S$ and the predecessors of job 1 belong to S. A second list is constructed in the same way to obtain sets of the form $S \cup \{2\}$. The two list are merged into a single list; if two sets are identical, the one with the lower cost is retained. The list obtained in this way remains in lexicographical order.

All potential successor jobs for states S are considered in this way: each time, a new list is obtained and then merged with the first. The second list (of $(k+1)$-element sets) does not have to be constructed explicitly; the procedure works with a list of k-element sets and another for $(k+1)$-element sets. When all jobs have been examined, the list of k-element sets is destroyed and the procedure moves on to the next iteration. Computer memory required is proportional to the maximum number of sets with a given number of elements.

In the problem $1/r_i, d_i/\sum w_i C_i$, all information necessary to construct $G(S, t)$ must be saved, not just a cost. We have chosen Lawler's method to use computer resources effectively, adapted as follows to apply tests presented in section 4.2. The procedure considers each k-element set S, one at a time. For a set S, all new sets of the form $S \cup \{i\}$ are created using successors i that can be added to S. This procedure eliminates states S as they are considered. Three methods for managing new sets and recognizing identical sets have been tested; these are now described. The complexity of these methods will be discussed in more detail in section 5.

The first method is similar to Lawler's method. A list of new sets of the form $S_1 \cup \{i\}$ is formed by adding potential successor jobs i to set S_1. A second list of the form $S_2 \cup \{i\}$ is constructed in the same way and merged with the first. A third list is obtained with S_3 and merged with the first. The process continues for each k-element set S. As in Lawler's method, only the first list is constructed explicitly. New $(k+1)$-element sets are inserted in the first list as they are obtained. This first method will be called "single list update". Let A denote a bound on the number of sets for one iteration. At each iteration, the method constructs $O(A)$ lists containing $O(n)$ elements and requires $O(A)$ merges; by comparison, Lawler's method constructs $O(n)$ lists containing $O(A)$ elements and requires $O(n)$ merges. Because A may be very large compared to n, this first procedure is not very efficient.

A second method uses a tree merge. First, for each k-element set S, a list of sets of the form $S \cup \{i\}$ is constructed. Next, the lists are recursively merged pairwise, to produce a single list. This second method involves only $O(\log A)$ merges. Not all lists need to be constructed before the merging process begins; lists can be merged as they are constructed. Thus, a maximum of $O(\log A)$ lists need to be stored. This method requires somewhat more memory than the preceding method, since identical sets may be stored in different lists, but it is faster.

A third procedure uses hashing to identify identical sets. This procedure sets up a large table and a hashing function defined on the binary vector associated with a state S. One element of the table contains a pointer to a set S and the number of elements in S. New $(k+1)$-element sets are stored in a list, not necessarily in lexicographic order. When a set $S \cup \{i\}$ is encountered for the first time, data regarding it are stored at its calculated address in the hash table if the space is free. In case of a collision (the table element points to another $(k+1)$-element set $S' \cup \{j\}$ different from $S \cup \{i\}$ but having the same address) the new set $S \cup \{i\}$ is placed after set $S' \cup \{j\}$ in the list of $(k+1)$-element sets. To check whether a set $S \cup \{i\}$ has already

been enumerated, the set pointed to at the address of $S \cup \{i\}$ is examined, as are all succeeding sets in the list of sets occupying the same address. This procedure requires additional memory for the hash table, many of whose elements are not used. Any one element of the table (a pointer to a set S and an integer for the number of elements in S) takes up little space. This procedure is fast if the hashing function distributes sets evenly throughout the table and if there are not too many collisions. The final section of this paper gives a proposed hashing function and reports some computational results.

3.2. *Cost functions*

The enumeration procedure considers states S such that $|S| = k$ and creates states of the form $S \cup \{i\}$ by adding potential successors. Cost function calculation includes two critical steps: addition of a successor i to a set S; and creation of two identical sets by adding a successor i to S and a successor j to S', where $S' = S \cup \{i\} - \{j\}$.

The function $G(S, t)$ gives the minimum cost of schedules covering all jobs in S. This function is used to calculate the minimum cost of schedules obtained by adding job i to set S as its last job. We define $F(S, \{i\}, t)$ and $G(S, \{i\}, t)$ for schedules that begin with jobs set S and end with job i. We have

$$F(S, \{i\}, t) = \begin{cases} G(S, t - p_i) + w_i t, & \text{if } t \in [r_i + p_i, d_i], \\ \infty, & \text{otherwise} \end{cases}$$

and

$$G(S, \{i\}, t) = \min_{t' \le t} F(S, \{i\}, t').$$

Similarly, the minimum cost $G(S', \{j\}, t)$ of schedules obtained by adding job j to set S' as the final operation can be calculated. If $S' = S \cup \{i\} - \{j\}$, two identical sets are obtained; the minimum with respect to t of the functions $G(S, \{i\}, t)$ and $G(S', \{j\}, t)$ is then calculated. This function gives the minimum cost of schedules covering all jobs in $S \cup \{i\}$ such that job i or j is the last operation. At the end of the iteration when all j are considered, the function $G(S \cup \{i\}, t)$ is obtained, giving the minimum cost of schedules covering all jobs in $S \cup \{i\}$ and ending with an arbitrary job in $S \cup \{i\}$.

Two approaches to representing the function $G(S, t)$ associated with the set S have been investigated. This function is non-increasing and piecewise linear in t. The first approach considers time as discrete and calculates a single value of G for each (integer) value of t (a single value of t is stored, however, for each constant segment of $G(S, t)$). Labels with two components are used, as follows:

$$(t_1, G(S, t_1)), (t_2, G(S, t_2)), \ldots, (t_K, G(S, t_K)),$$

where $t_1 < t_2 < \ldots < t_K$ and $G(S, t_1) > G(S, t_2) > \ldots > G(S, t_K)$. This approach gives the optimal solution if the data p_i, r_i and d_i are integer, because an integer optimal completion time is then known to exist as well [14].

A second approach, called the segmentation approach, stores information on each segment of the function. Labels with three components are used, as follows:

$$(t_1, G(S, t_1), m_1), (t_2, G(S, t_2), m_2), \ldots, (t_L, G(S, t_L), m_L),$$

where $t_1 < t_2 < \ldots < t_L$ and $G(S, t_1) > G(S, t_2) > \ldots > G(S, t_L)$. A label $(t_l, G(S, t_l), m_l)$ is associated with the segment falling within the interval $[t_l, t_{l+1})$ for $l = 1,\ldots,L-1$ and with the segment falling within the interval $[t_l, T]$ for $l = L$, where $T = \max_{i \in N} d_i$. The component $G(S, t_l)$ gives the value of the function of the beginning of the segment and m_l gives the slope of the segment. This information is sufficient for calculating the value of G for arbitrary t. This approach is more complex than the first in many respects, for example in the calculation of slopes and the search for the point at which two segments intersect. It can be applied, however, even when time windows are large and data are real-valued. Both methods have been programmed and tested.

4. Implementation of an efficient algorithm

This section discusses several options for improving the efficiency of the dynamic programming algorithm. A preprocessing step to reduce time windows for job execution is described first. Tests are then proposed for reducing the size of the state space and restricting the domain of evaluation for cost functions.

4.1. Preprocessing

Let $Pred(i)$ and $Succ(i)$ denote the sets of jobs that must be performed before and after job i, respectively.

$$Pred(i) = \{j \in N \mid r_i + p_i + p_j > d_j\},$$

$$Succ(i) = \{j \in N \mid r_j + p_j + p_i > d_i\}.$$

The time windows $[r_i, d_i]$ associated with jobs $i \in N$ can be tightened by applying the following rules:

- Job i may not begin before the end of a job $j \in Pred(i)$:

$$r_i = \max\left\{r_i, \max_{j \in Pred(i)} \{r_j + p_j\}\right\}.$$

- Job i must end before the start of a job $j \in Succ(i)$:

$$d_i = \min\left\{d_i, \min_{j \in Succ(i)} \{d_j - p_j\}\right\}.$$

These rules are applied iteratively until time windows can be tightened no further.

4.2. Tests

Four tests – to identify and reject sets corresponding to infeasible solutions or to eliminate jobs as possible successors for a state – are proposed. We first define some terminology. A schedule is $(S, \bar{S})$-partially ordered if it begins with jobs from set S, in any order, and ends with jobs from set $\bar{S}$. A schedule is $(S, \{i\}, \bar{S} - \{i\})$-partially ordered, for $i \notin S$, if it begins with jobs in S, continues with job i and terminates with jobs in $\bar{S} - \{i\}$.

The first test, which was applied by Schrage and Baker [21] and Lawler [16] using precedence relations obtained from the time constraints, can be used to reduce the number of possible successors for a state S.

Test 1. $(S, \{i\}, \bar{S} - \{i\})$-partially ordered schedules lead to infeasible solutions if $Pred(i) \not\subseteq S$.

The next two tests make explicit use of time constraints to eliminate states. Test 3, used by Posner [18] to solve $1/d_i/\sum w_i C_i$, is more effective than test 2 but requires more computational effort. Computational results indicate, however, that this additional effort is justified. In addition, $C_{\max}(S)$ denotes the earliest time at which all jobs in S can be finished.

Test 2. $(S, \bar{S})$-partially ordered schedules lead to infeasible solutions if

$$C_{\max}(S) > \min_{i \in \bar{S}} \{d_i - p_i\}.$$

Test 3. Let $i_1, i_2, \ldots, i_{n-|S|}$ denote jobs not in S arranged in non-decreasing order by deadline. $(S, \bar{S})$-partially ordered schedules lead to infeasible solutions if

$$C_{\max}(S) > \min_{k=1,\ldots,n-|S|} \{d_{i_k} - p_{i_k} - p_{i_{k-1}} - \cdots - p_{i_1}\}.$$

The next test eliminates the same states as test 3, but does so as successors are added. It is also capable of eliminating jobs as possible successors to a state that has not been eliminated. This test yields substantial savings in calculating cost functions without imposing much additional computational effort. Section 4 demonstrates how test 4 applied to all successors of a state S has the same complexity as test 3 applied to state S. Let $C_{\max}(S, \{i\})$ denote the earliest time by which all jobs in S can be completed, finishing with job i, $i \notin S$. It follows that $C_{\max}(S, \{i\}) = \max\{C_{\max}(S), r_i\} + p_i$.

Test 4. Let $i_1, i_2, \ldots, i_{n-|S|-1}$ be jobs not in $S \cup \{i\}$ sorted in non-decreasing order by deadline. $(S, \{i\}, \bar{S} - \{i\})$-partially ordered schedules lead to infeasible solutions if

$$C_{\max}(S, \{i\}) > \min_{k=1,\ldots,n-|S|-1} \{d_{i_k} - p_{i_k} - p_{i_{k-1}} - \cdots - p_{i_1}\}.$$

4.3. Domain of evaluation for cost functions

An important part of the problem $1/r_i, d_i/\sum w_i C_i$ is cost function evaluation. Test 4 can be used to restrict the domain of evaluation for $F(S, \{i\}, t)$ and $G(S, \{i\}, t)$ when a successor i is added to S. This test provides an upper bound $B(S, \{i\})$ on the latest time at which jobs in $S \cup \{i\}$ can be executed in an $(S, \{i\}, \bar{S} - \{i\})$-partially ordered schedule.

$$B(S, \{i\}) = \min_{k=1,\ldots,n-|S|-1} \{d_{i_k} - p_{i_k} - p_{i_{k-1}} - \cdots - p_{i_1}\}.$$

The domain of evaluation for $F(S, \{i\}, t)$ and $G(S, \{i\}, t)$ is restricted to the time interval $[C_{\max}(S, \{i\}), \min\{d_i, B(S, \{i\})\}]$.

5. Complexity

The complexity of the algorithm depends on the number of jobs, the number of states constructed in one iteration, the number of labels for a cost function and the tests applied. We show that the algorithm is polynomial if the time window length is bounded by a constant and the data p_i, r_i and d_i are integer.

5.1. Number of states

The number of states obtained in one iteration is limited by the time windows specified for executing jobs. In any such iteration, some jobs must be included in the set S, while others must be excluded. We demonstrate in this section that the number of states does not grow exponentially with the number of jobs when the time window length is bounded by a constant, but depends on this constant.

Let $I \geq \max_{i \in N}\{d_i - r_i\}$, a bound on maximum length of a time window for executing a job. Let $p = \min_{i \in N} p_i$ be the minimum processing time for a job, and $q = \lfloor I/p \rfloor$ a bound on the maximum number of jobs that can be carried out within a time window. Note that for integer-valued processing times, $p \geq 1$ and $q \leq I$. Let $i_1, i_2, \ldots, i_n$ denote jobs sorted in non-decreasing order by deadline.

Property 3. $\forall k \geq q$, jobs $i_1, i_2, \ldots, i_{k-q+1}$ belong to set S in any $(S, \bar{S})$-partially ordered schedule for which $|S| = k$.

Proof. Let i_l denote the job with the smallest index that is not in S and assume that $l \leq k - q + 1$. S contains at least q jobs with deadlines greater than or equal to d_{i_l}. These jobs may not begin before $d_{i_l} - I$ and require total execution time greater than or equal to pq. It follows that

$$\begin{aligned} C_{\max}(S) &\geq (d_{i_l} - I) + pq \\ &> (d_{i_l} - (pq + p)) + pq \\ &= d_{i_l} - p \\ &\geq d_{i_l} - p_{i_l}, \end{aligned}$$

where $C_{\max}(S) > d_{i_l} - p_{i_l}$ and state S leads to infeasible solutions. □

Property 4. $\forall k \leq n - q$, jobs $i_{k+q}, i_{k+q+1}, \ldots, i_n$ are not in S for any $(S, \bar{S})$-partially ordered schedule for which $|S| = k$.

Proof. Let i_l denote the last job in set S and assume that $l \geq k + q$. At least q jobs before job i_l do not belong to S. They require an execution time greater than or equal to pq and their execution must begin no later than $d_{i_l} - pq$. It follows that

$$\begin{aligned} C_{\max}(S) &\leq d_{i_l} - pq \\ &< d_{i_l} - I + p \\ &\leq r_{i_l} + p \\ &\leq r_{i_l} + p_{i_l}, \end{aligned}$$

where $C_{\max}(S) < r_{i_l} + p_{i_l}$. Since job i_l belongs to S, this gives a contradiction. □

Property 5. For problems with time window lengths bounded by a constant and integer-valued times, the number of states at each iteration of the dynamic programming algorithm is bounded by a number that does not depend on the number of jobs.

Proof. To construct a set S with k elements, at most $q - 1$ jobs are selected from among $2(q - 1)$ jobs from properties 3 and 4. An upper bound on the number of states is given by $A = C_{2(q-1)}^{q-1}$. □

The following property demonstrates that bound A can be improved because many combinations given by the bound A are not feasible. Jobs $i_{k-q+2}, i_{k-q+3}, \ldots, i_{k+q-1}$ are re-indexed as $j_1, j_2, \ldots, j_{2(q-1)}$.

Property 6. At least l jobs among jobs $j_1, j_2, \ldots, j_{2l}$ must belong to set S, for $l = 1, 2, \ldots, q - 1$.

Proof. Assume that x jobs, where $x \leq q - 1$, are selected from jobs $j_1, j_2, \ldots, j_{2l}$. At least $(q - 1) - x$ jobs must be selected with deadlines greater than or equal to $d_{j_{2l}}$:

$$\begin{aligned} C_{\max}(S) &\geq (d_{j_{2l}} - I) + (q - 1 - x)p \\ &> (d_{j_{2l}} - pq - p) + (q - 1 - x)p \\ &= d_{j_{2l}} - (x + 2)p. \end{aligned}$$

On the other hand, $(2l - x)$ jobs are not selected; these jobs have deadlines less than or equal to $d_{j_{2l}}$:

$$C_{\max}(S) \leq d_{j_{2l}} - (2l - x)p.$$

Therefore, $x > l - 1$ or $x \geq l$. □

Property 6 implies that one job must be selected from among j_1, j_2, that two jobs must be selected from among j_1, j_2, j_3, j_4, and so forth. It is clear from this property that the number of states obtained is well below A, so that a tighter bound can be found. For $q = 10$, the bound based on property 6 is 16,796 compared to $C_{18}^9 = 48{,}620$.

5.2. Number of labels

Let L be an upper bound on the maximum number of labels used to represent a function F or a function G. We have already calculated $C_{\max}(S)$, the earliest time by which all jobs in S can be terminated. Ready times for jobs in S must be less than or equal to $C_{\max}(S) - p$, and deadlines must be less than or equal to $C_{\max}(S) - p + I$. A feasible schedule for jobs in S must terminate in the interval $[C_{\max}(S), C_{\max}(S) - p + I]$; it suffices to know the functions $F(S, t)$ and $G(S, t)$ for this interval. If the data r_i, d_i and p_i are integer, then we can set $L = I - p + 1$. If not, the number of labels may grow exponentially in the number of jobs.

5.3. State management and calculation of cost functions

Possible candidates for the final job in a feasible schedule in a set are the q jobs with the latest deadlines. There are q k-element sets S capable of yielding a $(k + 1)$-element set S' and at most qA sets with $(k + 1)$ elements can be constructed by adding a successor to a k-element set. In one iteration, computation of $F(S, \{i\}, t)$, $G(S, \{i\}, t)$ and minimum functions for identical sets requires $O(qAL)$ operations.

A set S is represented by $O(n)$ integers. Comparison of two sets calls for logical operations on integers. If jobs are arranged in non-decreasing order by deadline, at most $2(q - 1)$ bits must be compared (by properties 3 and 4). We have proposed three methods for managing k-element sets and identifying identical sets. Updating a single list requires $O(qA^2)$ set comparisons, that is, $O(q^2A^2)$ operations. In a tree merge, there are $O(A)$ lists to be merged involving $O(\log A)$ merge levels, $O(qA \log A)$ set comparisons and $O(q^2A \log A)$ operations. The hashing procedure requires a constant number of set comparisons for each new set, hence $O(qA)$ set comparisons and $O(q^2A)$ operations.

5.4. Tests and domain of evaluation

If test 4 is used, there is no point in using any of the others. This test is applied using the bounds $B(S, \{i\})$ (defined in section 4.3) which also yield the interval over which the cost functions must be evaluated. Bounds $B(S, \{i\})$ for all successors i of a state S are obtained in a two-stage procedure. First, an upper bound $B(S)$ is calculated for the latest time by which all jobs in S can be completed, in an $(S, \bar{S})$-partially ordered schedule. Let $i_1, i_2, \ldots, i_{n-|S|}$ denote the jobs not in S, arranged in non-decreasing order by deadline. Then

$$B(S) = \min_{k=1,\ldots,n-|S|} \{d_{i_k} - p_{i_k} - p_{i_{k-1}} - \cdots p_{i_1}\},$$

where the index of the job producing the minimum is stored as

$$k^*(S) = \arg\left(\min_{k=1,\ldots,n-|S|} \{d_{i_k} - p_{i_k} - p_{i_{k-1}} - \cdots - p_{i_1}\} \right).$$

$B(S)$ and $k^*(S)$ can be calculated in one pass once over the sorted list of jobs, by identifying those that do not belong to S. By property 3, however, the first jobs on the list belong to S; since these jobs do not figure in the calculations, the entire list does not have to be examined. Similarly, by property 4 we know that the last jobs do not belong to the list. Calculations for these jobs can be carried out in a preliminary step and do not need to be repeated during the course of the algorithm. Only $2(q - 1)$ jobs need to be verified to identify those not in S.

In a second stage, $B(S, \{i_l\})$ is calculated for each job i_l, $l = 1,\ldots, n - |S|$. If $l \leq k^*(S)$, two cases arise. Either the minimum is reached at index $k^*(S)$, in which case $B(S, \{i_l\}) = B(S) + p_{i_l}$; or it is reached at an index preceding l, in which case $B(S, \{i_l\}) = \min_{k=1,\ldots,l-1}\{d_{i_k} - p_{i_k} - \cdots - p_{i_1}\}$. All values of $B(S, \{i_l\})$, $l = 1,\ldots, k^*(S)$ may be obtained in one pass over the list of these jobs. The values of $\min_{k=1,\ldots,l-1}\{d_{i_k} - p_{i_k} - \cdots - p_{i_1}\}$ are obtained recursively and are compared with $B(S) + p_{i_l}$. If $l > k^*(S)$, the minimum is reached at index $k^*(S)$ and $B(S, \{i_l\}) = B(S)$.

In summary, $O(q)$ jobs are examined to identify those not in S and to calculate the bound $B(S)$ and index $k^*(S)$. In a second pass, the bound $B(S, \{i_l\})$ is calculated for each job i_l in constant time. Test 4 and calculation of the domain of evaluation for the cost functions is $O(q)$ for all successors of a state S.

5.5. Complexity of the algorithm

For each iteration of the dynamic programming algorithm, the complexity of the three parts is as follows. For the state management, the complexity is presented for the three versions of the algorithm.

- Calculation of cost functions: $O(qAL)$.

- State management
  1. single list update: $O(q^2A^2)$;
  2. tree merge: $O(q^2A \log A)$;
  3. hashing: $O(q^2A)$.
- Tests: $O(qA)$.

The algorithm terminates in n iterations; the optimal solution can be found in $O(n)$ time through a backtracking procedure. The complexity of the algorithm is $O(nqA \max\{L, qA\})$ for single list update, $O(nqA \max\{L, q \log A\})$ for tree merge and $O(nqA \max\{L, q\})$ for hashing. If the time window length is bounded by a constant I and times are integer-valued, then the data q, L and A are bounded and the dynamic programming algorithm is polynomial.

6. Computational results

Computational results for the single-machine scheduling problem with ready times and deadlines to minimize total weighted completion time are presented for problems with up to 200 jobs. The behavior of the algorithm is examined for several test problems. Furthermore, its efficiency is demonstrated by comparison with a less sophisticated algorithm. A number of job-shop scheduling problems involving sub-problems similar to the test problems presented here are discussed as well.

Several alternative dynamic programming algorithms have been proposed. The first series of experiments uses tree merge to manage new states; test 4 to enumerate possible successors for a state; segmentation to represent cost functions; and bounds $B(S, \{i\})$ to restrict the domain of evaluation for the cost functions. This algorithm is used in a second series of experiments as a reference algorithm for comparing other proposed alternatives.

The algorithm has been coded in C and the experiments were carried out on a SPARC Station 2. A set S is represented using 16-bit integers. Two integers suffice for the 25-job problems, while thirteen are required for the 200-job problems. Memory requirements for the algorithm depend on the number of states and labels associated with the cost functions. If more than 100,000 labels are required for states of any given cardinality, the problem is abandoned.

6.1. Test problems

Random problems with 25, 50, 100 and 200 jobs have been generated. For any job, a processing time is selected as an integer between 1 and 50 and a real valued weight w_i is selected in the interval $(-1, 1)$. A schedule is constructed by selecting jobs i at random and then scheduling them at the earliest possible time t_i. This schedule is used to generate problems with four average widths for time windows $[r_i, d_i]$, for either 150, 200, 250 and 300. Problems with average time window width equal to 150

are obtained by setting $r_i = t_i - \mathrm{U}[1, 125]$ and $d_i = t_i + p_i + \mathrm{U}[1, 125]$, where random numbers are integer-valued. Adjustments are made to avoid negative ready times for the first jobs by increasing all schedule times t_i. A total of 16 problem types are obtained in this way; furthermore, ten problems are generated for each type. Table 1 gives the characteristics of these problems: the number of jobs, the mean width of the time windows and the time horizon. Some alternative problems were generated as well, to test various aspects of the procedure.

Table 1

Computational results.

No. jobs	Mean width	Horizon	Prob. solved	Mean states per iter.	Max states per iter.	Mean labels per state	Max labels per state	CPU (sec.)
25	150	750	10	16.0	48.1	4.2	16.9	0.1
	200	825	10	59.0	155.1	4.9	25.2	0.6
	250	900	10	258.2	780.7	5.4	34.5	3.4
	300	975	10	1237.1	4058.4	5.3	42.1	21.2
50	150	1425	10	12.9	67.4	5.3	20.3	0.2
	200	1500	10	58.8	317.5	6.3	32.8	1.6
	250	1575	10	279.9	1521.5	7.3	48.2	11.2
	300	1650	9	878.1	4571.9	7.8	62.6	41.0
100	150	2725	10	12.2	60.7	6.4	19.9	0.5
	200	2800	10	58.9	385.5	8.6	34.5	4.4
	250	2875	10	287.9	2235.9	9.5	56.6	30.4
	300	2950	7	643.9	3003.6	11.1	71.6	78.3
200	150	5125	10	13.2	112.2	7.1	19.9	1.6
	200	5200	10	82.6	881.6	9.8	39.6	16.1
	250	5275	8	310.4	2559.1	12.6	60.4	84.7
	300	5350	4	919.3	6092.0	11.7	62.0	267.7

To get a better sense of the realism of the generated problems, assume that time is measured in minutes. In 25-job problems, one job must be carried out in a period of from 2.5 to 5 hours over a horizon of from 12.5 to 16 hours. The horizon increases for problems with 50, 100 and 200 jobs, since the processing time does not change and more time is required to execute all jobs.

6.2. Behavior of the algorithm

Table 1 presents computational results from the test problems. This table contains the number of problems solved, the mean and maximum number of states per iteration, the mean and maximum number of labels used for a function and the CPU time in seconds. The means have been calculated with respect to the problems solved.

The number of states increases rapidly with the time window length, because larger windows offer more possibilities for sets with the same number of elements. The results do not show a clear relationship between the number of jobs and the mean number of states obtained in an iteration. To explain this quasi-independence, note that regardless of the total number of jobs, the time windows prevent more than a few jobs from being considered during one iteration. The number of labels grows slowly with the length of the time windows and the number of jobs. Since the number of possible orderings associated with a state increases rapidly, such slow growth is fortunate. Problems involving larger numbers of jobs take longer to solve mainly because the number of iterations in the dynamic programming algorithm is equal to the number of jobs. The time per iteration grows very slowly with the number of jobs. Most problems with a mean time window length of 250 or less could be solved without exceeding the maximum number of labels allowed for one iteration.

Other experiments were carried out to determine the effect of varying job weights on the number of labels required to represent a cost function. The problems were initially solved with the objective of minimizing makespan (the total duration of operations). We set $w_i = 0$, $i = 1,\ldots,n$ and add a fictitious job $n+1$ as the final operation with $w_{n+1} = 1$. Other experiments were constructed using only positive weights ($w_i \in (0, 1)$). Results for six problem types are given in table 2 and compared with results obtained for problems with arbitrary weights ($w_i \in (-1, 1)$).

Table 2

Number of labels and CPU time as a function of weights.

		Makespan		Positive weights		Arbitrary weights	
No. jobs	Mean width	mean labels per state	CPU (sec.)	mean labels per state	CPU (sec.)	mean labels per state	CPU (sec.)
25	200	1.0	0.2	1.2	0.2	4.9	0.6
	250	1.0	1.2	1.2	1.3	5.4	3.4
50	200	1.0	0.5	1.4	0.6	6.3	1.6
	250	1.0	3.1	1.3	3.5	7.3	11.2
100	200	1.0	1.1	1.5	1.3	8.6	4.4
	250	1.0	7.3	1.4	8.3	9.5	30.4

Time is the only factor in problems where the objective is to minimize makespan. In this case, the function $G(S, t)$ associated with a state is constant, and only one label is required. If different weights are associated with each job, then two dimensions (time and cost) must be considered for any schedule. If the weights are positive, $G(S, t)$ is a step function, that is, all segments have zero slope. A sequence of jobs

generates at most one segment or one label for $G(S, t)$. With arbitrary weights, a sequence may generate several segments. While more labels must be processed in this case, solution times remain moderate.

Problems with only negative weights are not more difficult to solve than problems with only positive weights. We have just to reverse the time axis and the weights become positive.

6.3. *Efficiency of the algorithm*

This section assesses the strategies proposed for the dynamic programming algorithm by comparing results obtained with different versions. The time windows were reduced by a factor of about 1% by applying the preprocessing step; this reduction is too small to improve solution times.

Table 3 compares results obtained by evaluating cost functions on the time interval $[Cmax(S, \{i\}), d_i]$ and the restricted interval $[Cmax(S, \{i\}), \min\{B(S, \{i\}), d_i\}]$. Restricting the domain of evaluation reduces the number of labels by about half. The other labels lead to infeasible solutions and are therefore of no use. A substantial

Table 3

Number of labels and CPU time as a function of the domain of evaluation of cost functions.

		Unrestricted intervals			Restricted intervals		
No. jobs	Mean width	mean labels per state	max labels per state	CPU* (sec.)	mean labels per state	max labels per state	CPU (sec.)
25	200	8.5	29.2	0.8	4.9	25.2	0.6
	250	11.4	41.2	6.0	5.4	34.5	3.4
50	200	11.4	37.7	2.4	6.3	32.8	1.6
	250	14.4	50.8	13.1^{1}	7.3	48.2	11.2
100	200	15.5	46.8	7.0	8.6	34.5	4.4
	250	20.4	69.0	28.7^{2}	9.5	56.6	30.4

*The superscript indicates the number of unsolved problems.

improvement in solution time and memory is obtained from use of the bounds $B(S, \{i\})$. Note that one of the times given for unrestricted intervals in the last problem type (28.7 seconds) is less than the corresponding time for restricted intervals (30.4 seconds); all problems (even the two difficult ones) were solved in the latter case, however.

Empirical trials were carried out to assess the efficiency of the tests proposed in section 4.2. These trials evaluate the cost functions on the time interval $[Cmax(S, \{i\}), d_i]$. When no tests are applied, none of the problems can be solved within the limit of 100,000 labels per iteration. The number of successors can be reduced

Table 4

Performance as a function of tests used.

No. jobs	Mean width	Test 1		Test 2		Test 3		Test 4	
		mean states per iter.	CPU* (sec.)	mean states per iter.	CPU* (sec.)	mean states per iter.	CPU* (sec.)	mean states per iter.	CPU* (sec.)
25	200	95.8	1.9	93.7	1.7	90.3	1.3	59.0	0.8
	250	471.4	18.0	461.0	15.4	399.7	9.7	258.2	6.0
50	200	99.4	5.8	95.9	4.5	90.8	3.7	58.8	2.4
	250	384.3	31.6^{1}	371.7	26.4^{1}	351.0	19.5^{1}	219.8	13.1^{1}
100	200	120.3	17.8	115.2	14.8	99.5	9.9	58.9	7.0
	250	301.8	59.0^{3}	287.2	48.6^{3}	254.7	37.3^{3}	176.5	28.7^{2}

*The superscript indicates the number of unsolved problems.

substantially by applying test 1. Table 4 presents results obtained using test 1 alone; using either test 2 or test 3 in conjunction with test 1; and using test 4. The purpose of tests 2 and 3 is to eliminate states. While test 3 requires more computational effort than test 2, it gives better results. Test 4 eliminates the same states as test 3, but does so during the preceding iteration as successors are added. This test yields a clear improvement in solution time, since a large number of infeasible states are created and cost calculation for these states can be avoided. In addition, test 4 eliminates a number of successors that cannot be added to a state. When test 4 is used, CPU time is reduced by a factor of about 60% compared with test 1.

Table 5 presents results obtained using the three state management methods proposed in section 3.1. A 20,000-element table was defined for the hashing method. Jobs are processed according to deadline; by properties 3 and 4, the only jobs that differ from one k-element set to another are in the vicinity of the kth job. The hashing function is defined using bits in the vicinity of the kth bit. A set is represented by a sequence of integers. The address associated with a set is obtained by applying an "exclusive or" operation to the integer containing the kth bit (and if they exist, the integers that precede and follow it). If the address exceeds the dimension of the table, its modulus is calculated. The tree merge method is much faster than the single list update method. The hashing procedure produces a slight improvement. Some problems show few collisions, while others show a large number. A different hashing function may give better results.

The last set of experiments compared the proposed approaches (discretization and segmentation) to calculating cost functions. Two additional groups of problems were created by varying the range of the data. The weight w_i of a job was selected in the interval $(-1, 1)$. Three groups of problems were obtained in this way:

Table 5

Evaluation of state management procedures.

No. jobs	Mean width	Prob. solved	Single list update	Tree merge	Hashing
			CPU (sec.)	CPU (sec.)	CPU (sec.)
25	200	10	0.6	0.6	0.6
	250	10	5.8	3.4	3.3
50	200	10	1.9	1.6	1.5
	250	10	30.4	11.2	10.2
100	200	10	5.9	4.4	4.1
	250	10	110.3	30.4	27.3
200	200	10	39.5	16.1	14.6
	250	8	236.9	84.7	75.6

Group I	Group II	Group III
$p_i \in [1, 10]$	$p_i \in [1, 20]$	$p_i \in [1, 50]$
mean width: 40, 50	mean width: 80, 100	mean width: 200, 250

In the segmentation approach, all problems were solved using fewer than 100,000 labels per iteration. The discretization approach requires more labels, but each label carries less information (the slope of the segment). To compare the approaches, no limit is placed on the number of labels obtained in one iteration. The results are presented in table 6. The number of labels varies with the range of the data, which is different for the three problem groups. When time windows are discrete, the number of labels increases linearly with the range of the data. This increase is far less rapid with the segmentation method. While solution times are similar for problems in group 1, the segmentation approach is clearly superior to the discretization approach in both of the other problem groups.

6.4. Relevance of the test problems

The single-machine scheduling problem with ready times and deadlines to minimize total weighted completion time appears as a subproblem in a Dantzig–Wolfe decomposition of the job-shop scheduling problem. Moreover, this subproblem must be solved several times. This paper has developed an algorithm for solving this problem and has tested it using simulated test problems. This section discusses the size of job-shop scheduling problems that can be handled using the algorithm developed for the subproblem.

Table 6

Discretization versus segmentation.

			Discretization			Segmentation		
	No. jobs	Mean width	mean labels per state	max labels per state	CPU (sec.)	mean labels per state	max labels per iter	CPU (sec.)
	25	40	6.4	34.4	0.5	3.7	15.8	0.4
		50	6.6	47.3	3.1	3.9	20.9	2.5
Group I	50	40	6.0	31.6	1.2	4.0	17.5	1.1
		50	6.5	42.1	8.5	4.5	23.4	7.9
	100	40	5.4	28.7	2.3	4.3	16.0	2.3
		50	6.4	38.8	17.3	5.1	22.9	18.1
	25	80	12.0	68.5	1.0	4.3	20.2	0.5
		100	12.2	93.3	5.6	4.5	28.8	2.9
Group II	50	80	11.1	62.4	2.0	5.2	25.0	1.3
		100	12.4	83.4	13.7	5.9	34.5	9.1
	100	80	10.1	55.9	4.1	6.4	23.8	3.4
		100	12.0	76.0	30.0	7.0	37.0	24.3
	25	200	28.4	168.8	2.2	4.9	25.2	0.6
		250	29.2	231.8	12.7	5.4	34.5	3.4
Group III	50	200	26.7	154.7	4.6	6.3	32.8	1.6
		250	29.6	206.9	32.7	7.3	48.2	11.2
	100	200	23.8	136.5	9.2	8.6	34.5	4.4
		250	28.7	189.2	68.2	9.5	56.6	30.4

Consider a job-shop scheduling problem with 50 jobs, 5 operations per job and 10 machines. Assume that one job requires an average of two hours work and must be executed in a four-hour period over a horizon of two working days (16 hours). The subproblem for one machine has the same characteristics as a simulated problem with 25 operations and time windows of 150 minutes (assuming that time is measured in minutes). In all, 250 operations must be processed, for an average of 25 operations per machine. Processing time for one operation is about 24 minutes, since one job contains five operations and requires two hours work. Since the time window specified for one job allows two hours flexibility, 144 minutes are available for one operation. Such a subproblem can be solved in 0.1 seconds. A thousand such subproblems, sufficient to give very good results for the job-shop scheduling problem, can be solved in two minutes.

If a job can be executed in 5.5 to 6 hours, the width of the time windows increases to 250 minutes and the required solution time increases to 3.5 seconds. One hundred such problems, sufficient to give good results for the underlying job-shop scheduling problem, can be solved in less than six minutes.

Now consider a problem with 100 jobs (rather than 50) and a time horizon of 5 working days (48 hours). If four hours are allowed for a job, the subproblem can be solved in 0.2 seconds. Good results for job-shop scheduling problems with 100 jobs thus become possible.

7. Conclusion

This paper has presented a dynamic programming algorithm for the single-machine scheduling problem with ready times and deadlines to minimize total weighted completion time. This problem involves a non-regular function of completion times. An approach similar to that of Lawler [16] is used to enumerate and manage dynamic programming states. The procedure was modified to permit efficient application of an improved version of Posner's test to eliminate successors and reduce the size of the state space. A piecewise linear cost function, non-increasing in time, is associated with each state. Information associated with function segments is stored in memory. This approach gives better results than one using discrete time windows. We demonstrate that the algorithm is polynomial when the time window length is bounded by a constant and times are integer-valued. Computational results are presented for problems with up to 200 jobs.

This problem crops up in the Dantzig–Wolfe decomposition of the job-shop scheduling problem. Results indicate that such problems with as many as 50 to 100 jobs may be solvable.

While the objective has been to minimize total weighted completion time, the algorithm may be used with other objectives such as: minimizing the total or the maximum weighted tardiness; minimizing the sum of early and tardy completion costs; or any other piecewise linear function of completion time, even if it is a non-regular function.

References

[1] T.S. Abdul-Razaq and C.N. Potts, Dynamic programming state-space relaxation for single-machine scheduling, Journal of the Operational Research Society 39(1988)141–152.

[2] T.S. Abdul-Razaq, C.N. Potts and L.N. Van Wassenhove, A survey of algorithms for the single machine total weighted tardiness scheduling problem, Discrete Applied Mathematics 26(1990)235–253.

[3] U. Bagchi and R.H. Ahmadi, An improved lower bound for minimizing weighted completion times with deadlines, Operations Research 35(1987)311–313.

[4] K.R. Baker and L.E. Schrage, Finding an optimal sequence by dynamic programming: An extension to precedence-related tasks, Operations Research 26(1978)111–121.

[5] S.P. Bansal, Single machine scheduling to minimize weighted sum of completion times with secondary criterion – A branch and bound approach, European Journal of Operational Research 5(1980)177–181.
[6] H. Belouadah, M.E. Posner and C.N. Potts, Scheduling with release dates on a single machine to minimize total weighted completion time, Discrete Applied Mathematics 36(1992)213–231.
[7] L. Bianco and S. Ricciardelli, Scheduling of a single machine to minimize total weighted completion time subject to release dates, Naval Research Logistics Quarterly 29(1982)151–167.
[8] T.C.E. Cheng, Dynamic programming approach to the single-machine sequencing problem with different due dates, Computers and Mathematics with Applications 19(1990)1–7.
[9] M.E. Dyer and L.A. Wolsey, Formulating the single-machine sequencing problem with release dates as a mixed integer program, Discrete Applied Mathematics 26(1990)255–270.
[10] J. Erschler, G. Fontan, C. Merce and F. Roubellat, A new dominance concept in scheduling *n* jobs on a single machine with ready times and due dates, Operations Research 31(1983)114–127.
[11] M.L. Fisher, B.J. Lageweg, J.K. Lenstra and A.H.G. Rinnooy Kan, Surrogate duality relaxation for job shop scheduling, Discrete Applied Mathematics 5(1983)65–75.
[12] S. French, *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Wiley, New York, 1982.
[13] A.M.A. Hariri and C.N. Potts, An algorithm for single machine sequencing with release dates to minimize total weigthed completion time, Discrete Applied Mathematics 5(1983)99–109.
[14] I. Ioachim, S. Gélinas, J. Desrosiers and F. Soumis, Shortest path problem with linear inconvenience costs on the time windows, Cahiers du Gérad G-92-42, École des Hautes Études Commerciales, Montréal, Canada, H3T 1V6, 1994.
[15] E.P.C. Kao and M. Queyranne, On dynamic programming methods for assembly line balancing, Operations Research 30(1982)375–390.
[16] E.L. Lawler, Efficient implementation of dynamic programming algorithms for sequencing problems, Report BW 106, Mathematisch Centrum, Amsterdam, 1979.
[17] J.K. Lenstra, A.H.G. Rinnooy Kan and P. Brucker, Complexity of machine scheduling problems, Annals of Discrete Mathematics 1(1977)343–362.
[18] M.E. Posner, Minimizing weighted completion times with deadlines, Operations Research 33(1985) 562–574.
[19] C.N. Potts and L.N. Van Wassenhove, An algorithm for a single sequencing with deadlines to minimize total weighted completion time, European Journal of Operational Research 12(1983) 379–387.
[20] C.N. Potts and L.N. Van Wassenhove, Dynamic programming and decomposition approaches for the single machine total tardiness problem, European Journal of Operational Research 32(1987) 405–414.
[21] L.E. Schrage and K.R. Baker, Dynamic programming solution of sequencing problems with precedence constraints, Operations Research 26(1978)444–449.
[22] J.P. Sousa and L.A. Wolsey, A time indexed formulation of non preemptive single machine scheduling problems, Mathematical Programming 54(1992)353–367.