Available online at www.sciencedirect.com
ScienceDirect
Computers & Operations Research 34 (2007) 2931–2938
computers & operations research
www.elsevier.com/locate/cor

An improved earliness–tardiness timing algorithm

Yann Hendel*, Francis Sourd

LIP6, UMR7606, 8, rue du Capitaine Scott, 75015 Paris, France

Available online 6 January 2006

Abstract

Earliness–tardiness criteria with distinct due dates usually induce NP-complete problems. Researchers have focused on particular cases like the timing problem, which is to look for the optimal schedule when the jobs sequence is already known. These timing algorithms are very useful since they can be used in more complex procedures. In the first part of this paper we provide the most efficient and fairly general algorithm to solve the one-machine timing problem. It is then adapted to a permutation flow shop problem.
© 2005 Elsevier Ltd. All rights reserved.

Keywords: Scheduling; Earliness–tardiness; Timing algorithm

1. Introduction

For regular criteria such as minimizing the makespan or the weighted sum of completion times, information about the execution order of the jobs on each machine generally leads to an optimal schedule. Indeed, apart from release dates, there is no idle time between the execution of each job. However, when the cost function is irregular, as in the earliness–tardiness case, finding the schedule when the job sequence is fixed on every machine is not trivial. Algorithms, used to solve these problems, are known as *timing* or *timetabling* algorithms. They are of practical interest since most earliness–tardiness problems are NP-complete. Moreover, for efficiency reasons, neighborhood search or branch-and-bound algorithms often rely on the job sequence instead of using the schedule. For instance, most neighborhoods used in the literature are based on the permutation of jobs in the sequence [1–3]. The timing procedure must be called for each iteration and, therefore, has to be done efficiently.

Timing algorithms have been proposed for different machine environments. In the case of the one-machine problem, the seminal article of Garey et al. [4] propose an algorithm in $O(n \log n)$ based on blocks of adjacent jobs. The authors prove the validity of their algorithm when the earliness and tardiness penalties are unitary. Some authors, Szwarc and Mukhopadhyay [5], Davis and Kanet [6], Bauman and Jozefowska [7], or Pan and Shi [8], have then proposed variants of this algorithm in order to solve problems with more general cost functions (weighted earliness/tardiness [5–7], convex piecewise linear [8]) or to make it run faster in practice [5,8].

In a multiple machine environment (parallel or shop problems), we observe that when the operations processed by each machine are sequenced, the resource constraints can necessarily satisfied so that they can be relaxed, which gives a project scheduling problem. For the problem with symmetrical earliness–tardiness penalties, Della Croce and Trubian [9] propose an $O(np)$ algorithm where p is the number of precedence constraints. It is based on repeated solutions

* Corresponding author. Fax: +33 144277000.
E-mail addresses: Yann.Hendel@lip6.fr (Y. Hendel), Francis.Sourd@lip6.fr (F. Sourd).

0305-0548/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2005.11.004

of maximum cardinality matching problems on a bipartite graph. Vanhoucke et al. [10] present another procedure for the weighted earliness–tardiness case, which is used as a lower bound in a branch-and-bound algorithm for the resource-constrained problem. Finally, Chrétienne and Sourd [11] generalize Garey et al. [4] approach for the problem with convex cost functions. For the special case with weighted earliness and tardiness penalties, the complexity of their algorithm is in $O(n \max(n, p))$.

One can observe that both one- and multiple-machine problems can be formulated as linear programs. Then, it is not surprising to observe that all the above-mentioned algorithms can be viewed as a specialization of the simplex algorithm. Indeed, all the iterations of these algorithms stop at extreme points of the polyhedron.

When the cost functions are non-convex the extension of the simplex-based approach is not possible any more. For the one-machine problem with general cost functions, Sourd [12] proposes a dynamic programming algorithm. When the cost functions are piecewise linear cost functions, $\|f_i\|$ denotes the number of segments of the piecewise linear function f_i and m stands for $\sum_i \|f_i\|$. In this case, the algorithm runs in $O(nm)$. In the case of weighted earliness and tardiness penalties, we have that $m = 2n$ so that the complexity of the algorithm is $O(n^2)$. However, Hendel and Sourd [3] show that combining the dynamic programming principles with a divide-and-conquer approach yield another algorithm with complexity $O(n \log n)$.

This article focuses on the problem with convex cost functions and more specially piecewise linear cost functions. Indeed, even if the paper of Chrétienne and Sourd [11] shows the properties of the optimal solutions and present an algorithm, the complexity for the one-machine problem is not given (Hendel and Sourd [3] observe it runs in $O(m \log m)$). Moreover, independently of our study, Pan and Shi [8] have very recently revisited the algorithm of Garey et al. [4] in order to propose an $O(m \log m)$ algorithm for the problem. The aim of this article is to point out that the problem can be solved in $O(m \log n)$ time. In our presentation, we are going to use the elegant reduction of the isotonic regression problem to our problem, which was proposed by Pan and Shi [8]. Finally, we mention another, also recent, algorithm by Colin and Quinino [13], which is useful when cost functions are not piecewise linear. The complexity of their algorithm depends on the time horizon.

The paper is organized as follows. We first present the problem and propose an $O(m \log n)$ algorithm for the case with convex piecewise linear cost functions. Then we extend it for the case with convex piecewise quadratic cost functions. We close our work by studying the flow-shop environment. In particular, we present two cases that reduce to the one-machine problem.

2. The one-machine problem

2.1. Problem definition

We consider the sequence of n jobs $(J_1, \ldots, J_n)$ that are to be scheduled in the order of their indices. For each job, a convex piecewise linear cost function $f_i(C_i)$ is given where C_i is the completion time of job J_i (see Fig. 1).

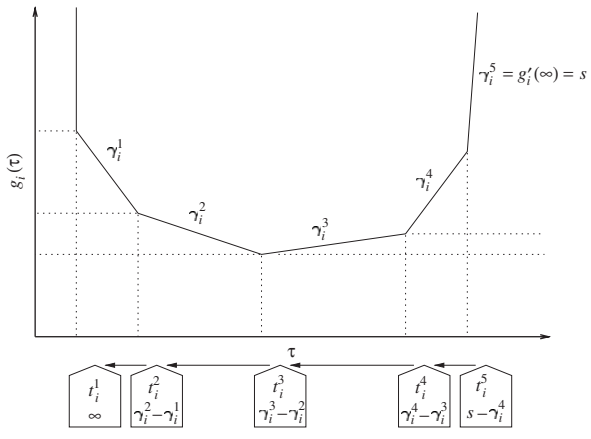

Fig. 1. Cost function of job J_i and its corresponding linked list.

The functions are encoded by the sorted list of their breakpoints and their corresponding slopes. Our objective is to schedule the jobs in order to minimize the sum of the costs of each job.

In a preliminary remark, we observe that by using convex piecewise linear cost functions, it becomes easy to model release dates and deadlines (respectively, denoted by r_i and $\tilde{d}_i$). Indeed, they appear as vertical segments, which means that the cost of the job is infinite when it is processed before the release date or after the deadline. In a practical implementation, it could be preferable to have functions defined over $[r_i + p_i, \infty) \to \mathbb{R}$. Therefore, in order to model a deadline $\tilde{d}_i$ for J_i, we use a segment whose slope is equal to $S = \sum_{i=1}^{n} \max_j(\text{slope}_{ij})$ where slope_{ij} is the slope of the jth segment of f_i (see Fig. 1). This ensures that the job is not executed out of its time windows unless there is no feasible schedule.

The problem can be stated as

$$\text{(TIMING)} : \min \quad \sum_{i=1}^{n} f_i(C_i)$$
$$\text{s.t.:} \quad p_i \leqslant C_i \leqslant C_{i+1} - p_{i+1} \quad \forall 1 \leqslant i < n.$$

Pan and Shi [8] have noticed that instead of considering the starting times or completion times of the jobs as the variables, one can use variables that model the total idle time before the job starts, i.e. variable x_i gives the amount of idle time before the starting of job J_i. Consequently, there is still an order amid the variables with $x_1 \leqslant x_2 \leqslant \cdots \leqslant x_n$. The completion times of the jobs and their associated idle time variables are linked by the relation $x_i = C_i - \sum_{j=1}^{i} p_j$. We define the cost functions $g_i(x_i)$ by $g_i(x_i) = f_i(x_i + \sum_{j=1}^{i} p_j)$, they are clearly convex piecewise linear.

This simplified problem is known in the literature as the isotonic regression problem [14,15] and can be stated as

$$\text{(IRP)} : \min \quad \sum_{i=1}^{n} g_i(x_i)$$
$$\text{s.t.:} \quad 0 \leqslant x_1 \leqslant x_2 \leqslant \cdots \leqslant x_n.$$

This problem is equivalent to (TIMING) with null processing times. We can still apply the principles of Garey et al. [4] algorithm. But, with this simpler problem, only one simple heap is needed. The major drawback of Garey et al. [4] algorithm is that it needs heaps to be merged in logarithmic time. As pointed out by Pan and Shi [8], such heaps like leftish heaps or Fibonacci heaps are very hard to implement and are practically not efficient [16,17]. Solving (IRP) avoid this drawback since, as illustrated in the next subsection, only one simple heap is required.

2.2. Convex piecewise linear costs

By applying the ideas of Garey et al. [4], Pan and Shi [8] provide an algorithm with a complexity of $O(m \log m)$. But, we can still slightly improve this complexity. Indeed, in their algorithm the size of the heap is at most m which corresponds to the number of events of the algorithm. Hence operations like inserting, searching and deleting the minimum are done in $O(\log m)$. We now show how to lower this complexity to $O(\log n)$.

Even if we consider the (IRP) formulation of (TIMING), we will use the scheduling terminology, that is x_i is the execution time of a job with null duration. A *block* is composed by jobs which are processed without idle time between them, that is they are executed at the same time. The main idea of the algorithm is, at the ith iteration, to insert in the schedule the job J_i processed at time infinity. The job is then left-shifted till either it is set at a time when its cost is minimum or it encounters a block and is merged with it. The process goes on with this block until it is set at a time where its cost is minimum or it met the preceding block. At the end of the iteration, the schedule must be optimal for the first i jobs. The point is that we can identify i possible "next events" that can occur when adjusting the schedule, each corresponding to a job. So that, only i events have to be stored in the heap. In order to have exactly i elements stored in the heap, we use a *reversed* list per cost function; the list contains the breakpoints sorted from the last one to the first one. The pointer on each list is set to the last breakpoint of the function (see Fig. 1). When an event corresponding to a function occurs, this current breakpoint is deleted from the heap, and the next one is added to the heap. Consequently, at any moment, there are at most n elements in the heap.

More formally, each cost function g_i is given as a number corresponding to the last slope of the function (i.e. $g_i'(\infty)$), and a "reversed" linked list of pairs (t_i^e, m_i^e) made of the abscissa of the breakpoint and a so-called *modifier* which is the variation between the right slope of the breakpoint to the left one (see Fig. 1, here $g_i'(\infty) = S$ because J_i had a deadline in the initial problem, the list has five elements: $[(t_i^5, S-\gamma_4)(t_i^4, \gamma_4-\gamma_3)(t_i^3, \gamma_3-\gamma_2)(t_i^2, \gamma_2-\gamma_1)(t_i^1, \infty)]$). At the beginning,

all the pointers of all the functions are set to the rightmost breakpoint of each cost function (which corresponds to the first element of the list). The last modifier of each list (which corresponds to the leftmost breakpoint) is equal to ∞. We can observe that only slopes are represented in this data structure so that a reference point $(t, g_i(t))$ is also necessary to have a complete representation of the function. However, this data structure is sufficient for our algorithm.

We use a unique heap to store all the events, which are sorted so that the event with the maximal abscissa $t = t_j^e$ is at the top of the heap. An event is encoded by a 4-tuple (t, j, k, m). The value $t = t_j^e$ is typically the abscissa of a breakpoint of a cost function whose index is given by j. The second index, k, is the index of the first job of the block (the role of this information will be apparent later) and m is the modifier associated to the breakpoint. We recall that the classic heap operations (insert an event and remove the maximal event) are all done in $O(\log h)$ time, where h is the size of the heap.

Let us consider the block that contains the jobs $J_k, J_{k+1}, \ldots, J_l$ scheduled at time t. The cost function of the block is $\sum_{i=k}^{l} g_i(t)$ and the *slope of the block* is defined as $\sum_{i=k}^{l} g_i'(t)$ and represents the marginal benefit if the block is left-shifted. We will emphasize the slope of the current block which will be denoted by the variable σ: when it is positive, the sum of the costs of the jobs in the current block is not minimal so that the block has to be left-shifted. The second variable associated to the current block is f which indicates the index of the first job in the block.

We are now ready to present Algorithm 1 which indicates how to insert J_i in the schedule, $J_1, \ldots, J_{i-1}$ are assumed to have been already inserted. This algorithm has to be called n times, for $J_1, J_2, \ldots, J_n$ in this order. In the algorithm, an event represents either

- a job, say J_j, that is scheduled at a breakpoint of its cost functions (these events are inserted in the heap at the first line of the while-loop). In order to left-shift J_j (and its block), the slope of the block has to be modified by m, or
- a block that is optimally scheduled (this event is inserted in the heap at the last line of Algorithm 1). Then m is the slope of the block. When the current block meet this optimal block, the cost functions are merged, which means that the current slope must be increased by m.

Algorithm 1. Insert J_i in the schedule

```
Initialize the current block: σ ← g_i'(∞) and f ← i
let (t^e, m^e) be the last breakpoint of g_i
j ← i
while σ ⩾ 0 do
    insert the event (t^e, j, f, m^e) in the heap
    let (t, j, k, m) be the event removed from the top of the heap
    σ ← σ − m
    f ← min(f, k)
    let (t^e, m^e) be the next event in the reversed list of g_j
end while
insert the event (t, j, f, −σ) in the heap
```

Algorithm 2 computes the execution time of each job (starting by the last one) by emptying the heap. The second index f helps to determine all the jobs belonging to the block.

Algorithm 2. Post processing

```
i ← n
while i > 0 do
    let (t, j, f, m) be the event removed from the top of the heap
    while i ⩾ f do
        x_i = t
        decrease i
    end while
end while
```

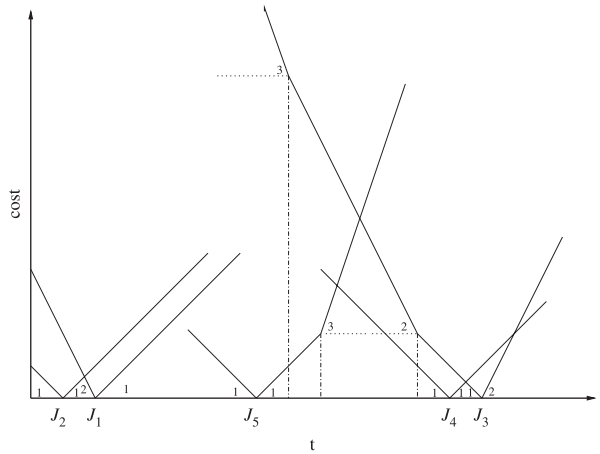

Fig. 2. The jobs and their cost functions.

Table 1
An instance with five jobs

Jobs	J_1	J_2	J_3	J_4	J_5
Slopes at infinity	1	1	2	1	3
Abscissa of the breakpoints	0, 2	0, 1	0, 8, 12, 14	0, 13	0, 7, 9
Modifiers	∞, 3	∞, 2	∞, 1, 1, 3	∞, 2	∞, 2, 2

Eventually, the completion times C_i for the initial problem are computed by $C_j = x_j + \sum_{i=1}^{j} p_i$. Correctness of the algorithm has already been proved by both Chrétienne and Sourd [11] and Pan and Shi [8]. The number of events in the algorithm is in $O(m)$ but, as announced in the beginning of the section, the heap contains exactly one event for each job inserted in the schedule, that is, it contains at most n events. Therefore, for every event, at most one element has to be deleted and one inserted into the heap, which is done in $O(\log n)$. The complexity is then in $O(m \log n)$. The post processing algorithm consists in emptying the heap which contains n element: it is done in $O(n \log n)$. The global complexity is then in $O(m \log n)$.

Example. Five jobs are considered in this instance of (IRP). The cost functions are represented in Fig. 2. The slopes and the abscissas of the functions are given in Table 1.

The five iterations of the algorithm:

1. $\sigma = 1$, the event (2, 1, 1, 3) is inserted, then is removed. At the end of the iteration $\sigma = -2$, thus (2, 1, 1, 2) is inserted.
2. $\sigma = 1$, the event (1, 2, 2, 2) is inserted, (2, 1, 1, 2), then is removed. At the end of the iteration $\sigma = -1$, thus (2, 1, 1, 1) is inserted.
3. $\sigma = 2$, the event (14, 3, 3, 3) is inserted, then is removed. At the end of the iteration $\sigma = -1$, thus (14, 3, 3, 3, 1) is inserted.
4. $\sigma = 1$, the event (13, 4, 4, 2) is inserted, (14, 3, 3, 3, 1) is removed. $\sigma = 0$, (13, 4, 4, 2) is removed, (12, 3, 3, 1) is inserted. At the end of the iteration $\sigma = -2$, thus (13, 4, 3, 2) is inserted.
5. $\sigma = 3$, the event (9, 5, 5, 2) is inserted, (13, 4, 3, 2) is removed. $\sigma = 1$, (0, 4, 4, ∞) is inserted, (12, 3, 3, 1) is removed. $\sigma = 0$, (8, 3, 3, 1) is inserted, (9, 5, 5, 2) is removed. At the end of the iteration, $\sigma = -2$, (9, 5, 3, 2) is inserted.

In Fig. 3, the state of the heap is given at the end of each iteration.

Algorithm 2 gives the execution times of each job: J_3, J_4 and J_5 are executed at $t = 9$, J_1 and J_2 at $t = 2$. Eventually, we notice that the schedule which is computed by the algorithm is the *earliest* schedule.

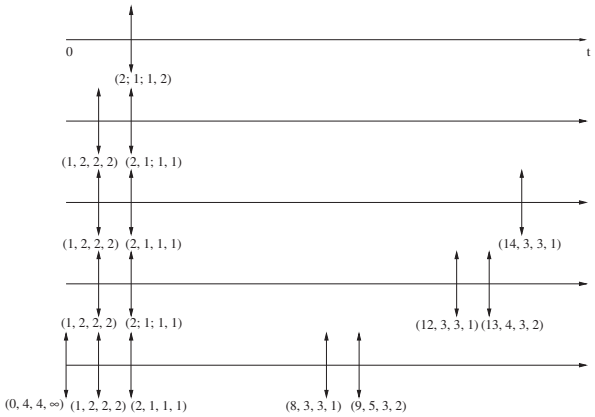

Fig. 3. State of the heap at the end of each iteration.

2.3. Convex piecewise quadratic costs

We now consider that the cost of a job is given as a convex piecewise quadratic function. In comparison with the linear case, we need to know not only the current first derivative, but also the second derivative since the minimum of these functions may not be at their breakpoints. Indeed as we add a new job to the schedule, two events can occur: either, as in the piecewise linear case, a breakpoint is reached, or the minimum of the current sum of the quadratic cost function is reached.

At any moment, the cost function of the current block is convex piecewise quadratic as a sum of convex piecewise quadratic functions: $g(t) = at^2 + bt + c$ with $a \geqslant 0$. The minimum of this function is reached for $t = -b/2a$ (which is $-g'(0)/g''$). Contrary to the piecewise linear case, the slope of the current block function depends on t.

Each cost function g_i is now given as two numbers corresponding to the value of the first derivative at time 0 and to the second derivative of the rightmost piece of the function, and a reverse linked list of triplets (t_i^e, m_i^e, q_i^e) made of the abscissa of the breakpoint, and two modifiers, m_i^e being the variation of the right first derivative at time 0 to the left and q_i^e being the variation of the right second derivative to the left.

An event is now encoded by a 5-tuple (t, j, k, m, q). t is still the abscissa of the event, j the index of the job of the breakpoint corresponding to the event, k the index of the first job of the block , m the modifier of the first derivative at time 0 and q the modifier of the second derivative.

The algorithm is an extension of the previous one. In order to find the next event, we have to compare the abscissa of the current breakpoint, the one of the next breakpoint and the minimum of the function which is $-g'(0)/g''$: let (t^e, j, k, m^e, q^e) be the current event and (t, j', k', m, q) the next event in the heap; if $t^e < -m^e/q^e$, then the iteration is stopped and the event corresponding to j in the heap is (t^e, j, k, m^e, q^e), else if $t < -m^e/q^e$, the iteration is stopped and the event corresponding to j in the heap is $(-m^e/q^e, j, k, m^e, q^e)$ (the block is therefore at its minimum), else we proceed with $(t, j', k', m - m^e, q - q^e)$. To each job corresponds at most one event, therefore the size of the heap is still in $O(n)$. The complexity remains in $O(m \log n)$.

3. Some generalizations for flow shop problem

The flow shop problem consists of n jobs. A job J_i is made of M successive operations $O_{i,1}, \ldots, O_{i,M}$ where operation $O_{i,j}$ is executed without preemption by the machine M_j and has a processing time $p_{i,j}$. In the permutation flow shop, every machine performs the jobs in the same order. We still consider an earliness–tardiness criterion: we first assign a convex piecewise linear cost function f_i on the last operation of each job. The problem being NP-complete, the rest of the paper still deals with the timing problem: the sequence of jobs is already known and we search for the optimal schedule.

Since, the first $M - 1$ operations of a job incurs no cost, they are executed as soon as possible. It comes down to introduce release dates r_i for the last operation of each job J_i. The cost function of a job is then modified: for each

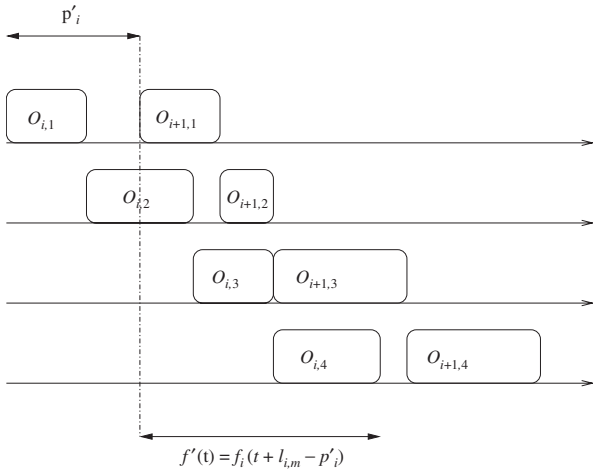

Fig. 4. The new processing time and the new cost function.

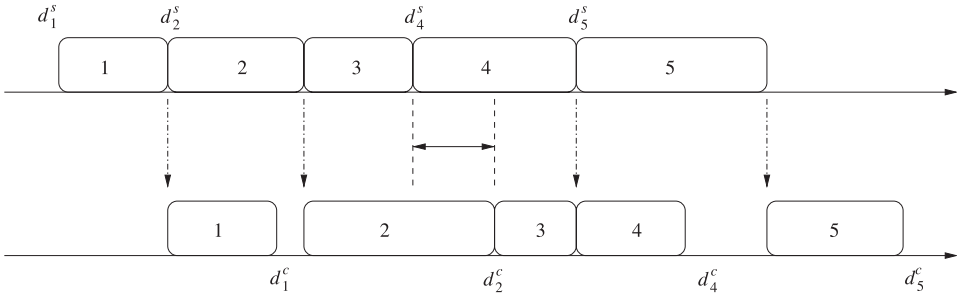

Fig. 5. A bad instance without No-Wait property.

job J_i, the cost is at infinity for $t \leqslant r_i$. The last operations $O_{1,M}, O_{2,M}, \ldots, O_{n,M}$ are then scheduled according to the algorithm presented in the previous section.

However, setting the earliness–tardiness cost on the last operation of the job may not render the will of the scheduler. Indeed, as we have started a job, we generally want to finish it as soon as possible in order to avoid storage costs between the operation executions. One way to penalize this storage cost is to ban idle time between operations of the same job. A job J_i is then said to be *No-Wait*: we denote by C_i the completion time of the job and by S_i, its starting time and we have for each job the following formula: $C_i - S_i = \sum_{j=1}^{M} p_{i,j}$. We now solve this problem by reducing it again to the one of the previous section.

For each pair of consecutive jobs (J_i, J_{i+1}), the index j of the *critical* operation (the one such that $C_{i,j} = S_{i+1,j}$) has to be found. We denote by $l_{i,j}$ the value of $\sum_{k=1}^{j} p_{ik}$. The critical operation $O_{i,j}$ has the j that maximizes $l_{i,j} - l_{i+1,j-1}$.

We can now view this problem as a one-machine timing problem. Each job J_i has a new processing time p'_i (see Fig. 4): it is the time the first operation of job J_{i+1} has to wait after S_i before starting: $p'_i = \max_j(l_{i,j} - l_{i+1,j-1})$. The new cost function f'_i is $f'_i(t) = f_i(t + l_{i,m} - p'_i)$. We are then in the conditions to apply the algorithm of the preceding section. Finding all the critical operations is done in $O(nM)$. The global complexity is then $O(\max(nM, m \log n))$.

This modelization may be insufficient too: we may want to allow idle time between operations of the same job and penalize it. A possible model is to have the earliness cost tied to the first operation of the job and the tardiness cost tied to the last operation. More formally, a preferred start time d_i^s is assigned to the first operation of J_i; to the last operation of J_i, a due date d_i^c is assigned. We redefine tardiness costs as $T_i = \max(0, C_i - d_i^c)$ where C_i is the completion time of the job and earliness costs as $E_i = \max(0, d_i^s - S_i)$ where S_i is the starting time of job J_i. We still want to minimize $\sum_{i=1}^{n}(\alpha_i E_i + \beta_i T_i)$. In this case, even with restrictions such as being limited to two machines and having unitary earliness and tardiness costs, the optimal solution may not have the No-Wait property. Indeed, we provide a very simple instance (see Fig. 5) with five jobs. There is no optimal solution where the third job has the No-Wait property.

The problem becomes significantly more difficult: a job may be simultaneously early and tardy at the same time. However, this problem is still solved by Chrétienne and Sourd algorithm [11].

4. Conclusion

In this paper we have proposed a general timing algorithm for the one-machine problem: it solves the cases where there are release dates, deadlines, convex piecewise quadratic cost functions. Furthermore, it is the most efficient to our knowledge: for the classic $\sum_{i=1}^{n}(\alpha_i E_i + \beta_i T_i)$ criterion, it runs in $O(n \log n)$ time, and for convex piecewise quadratic cost functions, it runs in $O(m \log n)$ time.

We have also taken interest in the permutation flow shop problem with three different criteria. For the first case, where the cost function is tied to the last operation of the jobs, an algorithm is trivially derived from the one of the one-machine problem. For the second criterion, with the No-Wait property, we still provide an efficient algorithm. However, we failed to find a simple algorithm to solve the problem when earliness cost is tied to the first operation.

References

[1] James RJW, Buchanan JT. A neighborhood scheme with a compressed solution space for the early/tardy scheduling problem. European Journal of Operational Research 1997;102:513–27.
[2] Wan G, Yen BPC. Tabu search for single machine with distinct due windows and weighted earliness/tardiness penalties. European Journal of Operational Research 2002;142:271–81.
[3] Hendel Y, Sourd F. Efficient neighborhood search for the one-machine earliness–tardiness scheduling problem, European Journal of Operational Research, in press.
[4] Garey MR, Tarjan RE, Wilfong GT. One processor scheduling with symmetrical earliness and tardiness penalties. Mathematics of Operations Research 1988;13:330–48.
[5] Szwarc W, Mukhopadhyay SK. Optimal timing schedules in earliness–tardiness single machine sequencing. Naval Research Logistics 1995;42:1109–14.
[6] Davis JS, Kanet JJ. Single-machine scheduling with early and tardy completion costs. Naval Research Logistics 1993;40:85–101.
[7] Bauman J, Jozefowska J. Minimizing the earliness–tardiness costs on a single machine. Computers and Operations Research 2005, in press.
[8] Pan Y, Shi L. Dual constrained single machine sequencing to minimize total weighted completion time. IEEE Transactions on Automation Science and Engineering 2005; 2(4): 344–57.
[9] Della Croce F, Trubian M. Optimal idle time insertion in early–tardy parallel machines scheduling with precedence constraints. Production Planning and Control 2002;13:133–42.
[10] Vanhoucke M, Demeulemeester E, Herroelen W. An exact procedure for the resource-constrained weighted earliness–tardiness project scheduling problem. Annals of Operations Research 2001;102:179–96.
[11] Chrétienne Ph, Sourd F. PERT scheduling with convex cost functions. Theoretical Computer Science 2002;292:145–64.
[12] Sourd F. Optimal timing of a sequence of tasks with general completion costs. European Journal of Operational Research 2002;165:82–96.
[13] Colin EC, Quinino RC. An algorithm for insertion of idle time in the single-machine scheduling problem with convex cost functions. Computers and Operations Research 2005;32:2285–96.
[14] Ahuja RK, Orlin JB. A fast scaling algorithm for minimizing separable convex functions subject to chain constraints. Operations Research 2001;49:784–9.
[15] Chakravarti N. Isotonic median regression: a linear programming approach. Mathematics of Operations Research 1989;14:303–8.
[16] Tarjan RE. Data structure and networks algorithms. Philadelphia, PA: SIAM; 1983 [chapter 3].
[17] Knuth D. The art of computer programming. Sorting and searching, vol. 3. Reading, MA: Addison-Wesley; 1973.