Available online at www.sciencedirect.com
SCIENCE DIRECT®
European Journal of Operational Research 165 (2005) 82–96
EUROPEAN JOURNAL OF OPERATIONAL RESEARCH
www.elsevier.com/locate/dsw

Discrete Optimization

Optimal timing of a sequence of tasks with general completion costs

Francis Sourd *

Université Pierre et Marie Curie, CNRS-LIP6, 4, place Jussieu, 75252 Paris Cedex 05, France

Received 25 November 2002; accepted 8 January 2004
Available online 16 March 2004

Abstract

Scheduling a sequence of tasks—in the acceptation of finding the execution times—is not a trivial problem when the optimization criterion is irregular as for instance in earliness–tardiness problems. This paper presents an efficient dynamic programming algorithm to solve the problem with general cost functions depending on the end time of the tasks, idle time costs and variable durations also depending on the execution time of the tasks. The algorithm is also valid when the precedence graph is a tree and it can be adapted to determine the possible execution windows for each task not exceeding a maximum fixed cost.

Keywords: Scheduling; Earliness–tardiness costs; Piecewise linear cost functions; Idle time penalties; Dynamic programming

1. Introduction

Just-in-time scheduling has interested both practitioners and researchers for over a decade. A very common idea is to recognize that a job that completes either tardily or early in a schedule incurs extra costs. Therefore, a usual model consists in introducing earliness and tardiness penalties per unit time for each task and the objective is to minimize the sum of all the earliness and tardiness costs.

However, such a model may be insufficient. Very often, the earliness and tardiness costs are not linear on the whole time horizon. For example, practitioners sometimes want to model several *good* time periods during which a task would preferably be processed, but with *bad* time periods in between the good periods. Moreover, they also have to deal with idle periods: in schedules minimizing the earliness–tardiness costs, periods of inactivity are generally inserted but in practice, these periods when no work is done have an extra cost (e.g. for intermediate storage) that cannot be ignored in the model and must be penalized.

In this paper, the single-machine problem with general completion costs and idle period penalties is studied. More precisely, we will mainly consider the key problem where the tasks are already sequenced. Even if the main difficulty of the problem is in the sequencing, this subproblem is very important because most scheduling algorithms first rank the tasks by the mean of either a

* Tel.: +33-1-44-27-53-95; fax: +33-1-44-27-70-00.
E-mail address: francis.sourd@lip6.fr (F. Sourd).

0377-2217/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.ejor.2004.01.025

(meta)heuristic or an enumeration scheme and next determine the optimal—if possible—timing for the sequenced tasks. For example, both the branch-and-bound algorithms by Hoogeveen and van de Velde [10] or by Sourd and Kedad-Sidhoum [14] and the tabu search by Wan and Yen [16] are based upon this approach to solve the single machine problem with earliness/tardiness penalties. Section 5 of this paper presents how our results on this subproblem can be used in a constraint-programming based approach when solving the general sequencing problem.

When the completion costs are non-decreasing—criteria such as the flow time and the total tardiness—and when the cost of idle period is non-decreasing with the length of the period, the problem is obvious: each task is scheduled as early as possible when its predecessor in the sequence is completed.

The pure earliness–tardiness case (without idle time penalties) can be formulated as a linear program [7] but this problem can be more efficiently solved in $O(n \log n)$ time by a direct algorithm based on the blocks of adjacent tasks [6,8,15]. When the minimization criterion is the maximum cost instead of the sum of all the costs, the problem of finding optimum start times for a sequence of tasks can be efficiently solved with general cost functions [12].

Our problem is also related to the project scheduling problem with irregular starting time costs [13]. However, the approach, based on network flows, adopted by Mörhing et al. [13], requires to explicitly define the cost of each task at any time point so that the time horizon of the schedule appears in the complexity of the algorithm. Chrétienne and Sourd [4] present an algorithm for the special case where the cost functions are convex. Another recent approach to schedule tasks in parallel with earliness and tardiness is due to Della Croce and Trubian [5].

The algorithm presented in this paper is based on dynamic programming (DP) [3]. The scheme is fairly simple and, which is interesting, it can deal with additional features that cannot be dealt by the existing algorithms. For example, the problem can be formulated as a continuous linear program when the cost functions are convex but integer variables must be used to represent non-convex cost functions. Our DP approach is able to solve the problem with both convex and non-convex cost functions, but also idleness costs and durations depending on the execution time of the task, which can be very useful to model breaks or transportation activities. Moreover, the method can also be adapted to solve the problem in which the precedence graph is a tree instead of a complete sequence.

In a methodological view, this paper focuses on piecewise linear functions. We show the influence of the input encoding on the resulting complexity of the algorithm, which is expressed in function of the number of segments of the cost functions. To the best of our knowledge, there is no previously published similar analysis for a dynamic programming algorithm. Moreover, this analysis can be generalized to several classes of problems in which a function is given in input. In practice, this analysis is very useful because piecewise linear functions are very practical to represent time-varying parameters. For example, if we consider a problem that can be solved either by our approach or by the algorithm of Möhring et al. in which the cost functions are piecewise linear, our approach is polynomial whereas the algorithm of Möhring et al. is pseudo-polynomial because its complexity depends on the number of time points.

Section 2 presents with more mathematical details the problem studied in the paper. It also considers modelization questions. Section 3 is devoted to the solution of the problem by dynamic programming; the computational complexity is studied when the cost functions are piecewise linear. Some polynomial special cases are studied in Section 4. Finally, in Section 5, we adapt the dynamic programming approach to compute the possible start times of all the activities such that a fixed maximum total cost is not exceeded.

2. Problem description and complexity

2.1. Problem definition

The problem is to find the execution times of n sequenced tasks denoted by $T_1, T_2, \ldots, T_n$ that is T_i

can start only after T_{i-1} is completed. In a feasible schedule, S_i and C_i respectively denote the start time and the end time of T_i. The relationship between S_i and C_i is assumed to be known in advance, $C_i - S_i$ being the duration of T_i. More precisely, it is assumed that S_i is a continuous non-decreasing function of C_i, which is denoted by $S_i = S_i(C_i)$. In other words, the later a task starts, the later it completes and there is only one possible start time for a given end time. The fact that the processing time of a task is not constrained to be a constant over the scheduling horizon is very useful when the tasks represents road journeys: because of traffic intensity, the journey may be longer in the morning or in the evening than in the afternoon. In a shop context, the operations may be longer when a part of the workers have a break for lunch.

Note that the function $S_i(C_i)$ is not required to be strictly increasing, which is of great importance to deal with breaks. However, for simplicity of the proof, it is required to be continuous but usual non-continuous functions (with a finite number of discontinuities) can be seen as the limit of a sequence of continuous functions. We will give an example in Section 2.2.

For each task T_i, a cost function f_i depending on the completion time C_i is given. In order to avoid situation where a task would have to be scheduled at $\pm\infty$, f_i is required to be non-increasing on some interval $(-\infty, a_i]$ and non-decreasing on some interval $[b_i, \infty)$ (a_i and b_i are two constants that depends only on f_i).

If T_{i+1} does not start just at the completion time of T_i, there is an *idle period* of length $S_{i+1} - C_i$ between the two tasks. The cost of the idle period is measured by the value $w_i(S_{i+1} - C_i)$, where w_i is a function defined on $\mathbb{R}^+$. w_i need not be non-decreasing on $\mathbb{R}^+$ but as for functions f_i, it is required to be non-decreasing on some interval $[c_i, +\infty)$ (where c_i is a non-negative constant dependent only on w_i) to avoid to have some tasks scheduled at $\pm\infty$. So the total cost is

$$\sum_{1 \leqslant i \leqslant n} f_i(C_i) + \sum_{1 \leqslant i < n} w_i(S_{i+1}(C_{i+1}) - C_i) \tag{1}$$

and the aim of the problem is to minimize this cost subject to the precedence constraints $C_i \leqslant S_{i+1}$ for all $1 \leqslant i < n$.

For example, in the pure earliness–tardiness case, the cost function of T_i is defined as

$$f_i(C_i) = \max(\alpha_i(d_i - C_i), \beta_i(C_i - d_i)),$$

where α_i and β_i are respectively the earliness and tardiness penalties per unit time.

In order to have a more general model, we do not assume that the cost functions f_i and w_i are continuous but we assume that they have a left and a right limit at any point. Note that the left and right continuity implies that the cost functions have a lower bound on any interval. We do not care about the value of the cost function at a discontinuity time point. If time C_i is a discontinuity point for f_i, the cost of task T_i is $\min(\lim_{t \to C_i^-} f_i(t), \lim_{t \to C_i^+} f_i(t))$. We manage the idleness costs in the same way.

2.2. Time windows and breaks

Time window and break constraints often appear in practical models. They can be integrated in our model by setting the costs to ∞ (i.e. any upper bound on the solution) on the time intervals when the execution of the task is forbidden. For example, if task T_k is constrained to be entirely executed in the time window $[s_{\min}, e_{\max}]$, its minimum end time $e_{\min}$ is equal to $\min(S_k^{-1}(s_{\min}))$ so we set the cost function f_k to ∞ on intervals $(-\infty, e_{\min})$ and $(e_{\max}, +\infty)$. Clearly, the task can have several possible time windows (while being constrained to be entirely executed in only one time window).

A *break* is a time interval $(b_{\min}, b_{\max})$ during which the execution of a task can be suspended at the break start $b_{\min}$ and restarted at the break end $b_{\max}$. For a task T_k whose processing time is p_k (note that, for this instance, the processing time is fixed but not the duration $C_k - S_k$), the start function S_k is (see Fig. 1):

$$S_k(C) = \begin{cases} C - p_k & \text{if } C < b_{\min}, \\ b_{\min} - p_k & \text{if } b_{\min} \leqslant C < b_{\max}, \\ C - p_k - (b_{\max} - b_{\min}) & \text{if } b_{\max} \leqslant C < b_{\max} + p_k, \\ C - p_k & \text{if } b_{\max} + p_k \leqslant C. \end{cases}$$

Moreover, since T_k cannot complete during the break, its cost function f_k must be set to ∞ on the

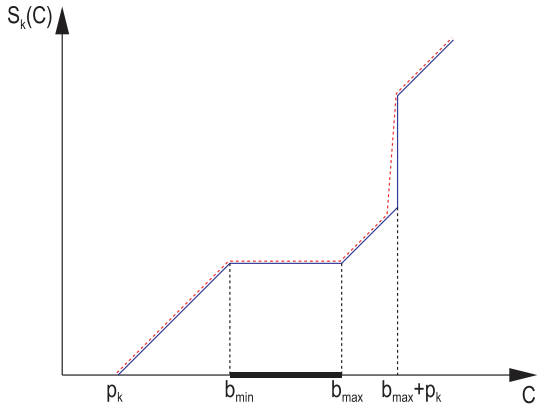

Fig. 1. Start function $S_k(C_k)$ in the presence of a break $(b_{\min}, b_{\max})$ and its transformation into a continuous function.

interval $(b_{\min}, b_{\max})$. However, we note that the above-defined function S_k is non-continuous at $C = b_{\max} + p_k$ while our mathematical model requires it to be continuous. But we can remark that if T_k ends at $b_{\max} + p_k - \epsilon$ (for some small $\epsilon > 0$), the fraction ϵ/p_k of T_k is processed just before the break, which is not desirable in practice. We can prevent such a situation by fixing the minimum fraction $\rho_k \in (0,1)$ of T_k that must be processed before the task may be interrupted by the break. T_k can be prevented from completing in $(b_{\max} + (1-\rho_k)p_k, b_{\max} + p_k]$ by setting $f_k(C_k)$ infinite on this interval and S_k can be transformed into a continuous function by modifying it only on this interval, as shown by Fig. 1. If T_k cannot be interrupted by the break—that is it must be entirely processed before or after it—we have to set the cost $f_k(C_k)$ to infinity on the interval $[b_{\max}, b_{\max} + p_k)$.

Clearly, problems with several breaks can similarly be modeled—attention must be paid to the case where the interval length between two consecutive breaks is less than p_k.

2.3. Complexity

In this section, we show that the problem is NP-complete even if the cost functions are represented by a procedure with a constant size. We define the following problem that is a special case of the problem defined in Section 2.1, in which all the tasks have a null processing time.

Scheduling sequenced tasks (SST)

INSTANCE: n integer piecewise linear cost functions $f_1, \ldots, f_n$, $n-1$ integer piecewise linear idleness cost functions $w_1, \ldots, w_{n-1}$ and a positive integer K. Each cost function is represented by a procedure that runs in constant time.

QUESTION: Is there a schedule $S_1 \leqslant S_2 \leqslant \cdots \leqslant S_n$ such that the total cost $\sum_{1 \leqslant i \leqslant n} f_i(S_i) + \sum_{1 \leqslant i < n} w_i(S_{i+1} - S_i)$ is K or less?

Theorem 1. *SST is NP-complete in the ordinary sense.*

Proof. SST is clearly in NP. We transform PARTITION to SST. Let an arbitrary instance of PARTITION given by the finite set A and the size $s(a) \in \mathbb{Z}^+$ for each $a \in A$. We are going to construct an instance of SST with positive cost functions whose total cost is null if and only if there is a subset $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$.

The size n of this instance of SST is $n = |A| + 1$. Let $a_1, \ldots, a_{n-1}$ be the items of A numbered in any order. The main idea of the transformation is to associate the time period $[S_i, S_{i+1})$ to the item a_i and to put a_i in A' or not whether $S_i = S_{i+1}$ or not.

The idleness cost functions w_i are defined by

$$w_i(t) = \begin{cases} 2t & \text{if } 0 \leqslant t < 1/2, \\ 2|t - s(a_i)| & \text{if } t \in [s(a_i) - 1/2, s(a_i) + 1/2), \\ 1 & \text{otherwise.} \end{cases}$$

The punctuality cost functions $f_2, \ldots, f_{n-1}$ are the null functions and the functions f_1 and f_n are

$$f_1(t) = \begin{cases} |t| & \text{if } -1 < t < 1, \\ 1 & \text{otherwise,} \end{cases} \quad \text{and}$$

$$f_n(t) = \begin{cases} |t - B/2| & \text{if } t \in (B/2 - 1, B/2 + 1), \\ 1 & \text{otherwise,} \end{cases}$$

where $B = \sum_{a \in A} s(a)$. We observe that all these cost functions are continuous and are represented by a bounded number of segments so that it is executed in constant time.

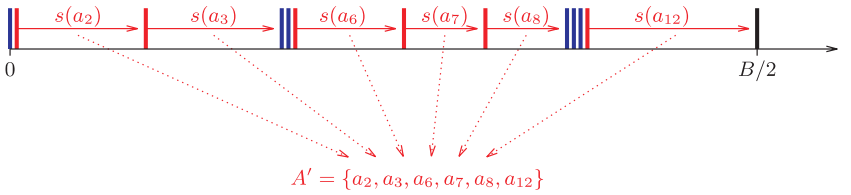

Fig. 2. Transforming SST to PARTITION.

In a schedule whose total cost is null ($K = 0$), as in Fig. 2, all the individual costs in the sum are null so that for any i, $S_{i+1} - S_i \in \{0, s(a_i)\}$, $S_1 = 0$ and $S_n = B/2$. Therefore, with $A' = \{a_i | S_{i+1} \neq S_i\}$, we have

$$\sum_{a \in A'} s(a) = \sum_{a_i \in A'} S_{i+1} - S_i = \sum_{a_i \in A} S_{i+1} - S_i = S_n - S_1 = B/2.$$

So $\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$.

Conversely, if, for the instance of PARTITION, there is a set A' such that $\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$, the schedule given by $S_i = \sum_{j<i; a_j \in A'} s(a_j)$ has clearly a null cost. Since the size of the instance of SST is in $O(n)$, SST is NP-complete. □

3. Solving the problem by dynamic programming

We are going to show that this problem can be solved by dynamic programming. For any $k \in \{1, \ldots, n\}$ and any $t \in \mathbb{R}$, $P_k(t)$ denotes the subproblem in which:

- the sequence of tasks is the subsequence $T_1, \ldots, T_k$ and
- we add the additional constraint that T_k completes at t, that is $C_k = t$.

$\Sigma_k(t)$ is the minimum cost of the solutions of $P_k(t)$. Clearly, the optimal cost of the whole problem is $\min_{t \in \mathbb{R}} \Sigma_n(t)$.

3.1. Recursive relationship

We obviously have that $\Sigma_1(t) = f_1(t)$. For any $k \geqslant 1$, $\Sigma_{k+1}(t)$ can be expressed in function of Σ_k. Indeed, if T_{k+1} is assumed to complete at time t then T_k must complete at some time t' that is less than or equal to the start time of T_{k+1} that is $S_{k+1}(t)$. The minimum cost of the schedule such that $C_k = t'$ and $C_{k+1} = t$ is $\Sigma_k(t') + w_k(S_{k+1}(t) - t') + f_{k+1}(t)$. $\Sigma_{k+1}(t)$ is then given by minimizing upon all the possible values for t':

$$\Sigma_{k+1}(t) = \min_{t' \leqslant S_{k+1}(t)} \left(\Sigma_k(t') + w_k(S_{k+1}(t) - t')\right) + f_{k+1}(t). \quad (2)$$

From the assumptions made in Section 2.1 on the cost functions f_i and w_i (and the resultant definition of a_i, b_i and c_i, see also Table 1), the minimum of Σ_n is in the interval $[\min_i a_i - \max_i c_i, \max_i b_i + \max_i c_i]$. By construction, Σ_n has only a finite number of discontinuity points on this interval and it is left and right continuous. As a consequence, if for each of the discontinuity point t_{disc} we (re)define $\Sigma_n(t_{\text{disc}})$ as the minimum between its left and right limits, [1] then there exists some time point C_n such that $\Sigma_n(C_n) = \min_{t \in \mathbb{R}} \Sigma_n(t)$. This time point C_n is the completion time of T_n. To compute C_{n-1}, we must add the constraint that $C_{n-1} \leqslant S_n = S_n(C_n)$: so C_{n-1} is a value minimizing Σ_{n-1} on $(-\infty, S_n]$. $C_{n-2}, \ldots, C_1$ are computed by iterating this process.

Finally, we can remark that this dynamic programming approach is also valid when the order between the tasks in not given by a complete chain but by a "tree" precedence graph, which means that the non-oriented underlying graph has no cycle (it is a forest). In this problem, several tasks can eventually be processed at the same time, that is there is no resource constraint (project scheduling problem). Clearly, each connected component of the precedence graph forms an independent sub-problem.

[1] That is $\Sigma_n(t_{\text{disc}}) = \min(\lim_{t \to t_{\text{disc}}^-} \Sigma_n(t), \lim_{t \to t_{\text{disc}}^+} \Sigma_n(t))$.

Table 1
Summary of the main notations

n	Number of tasks
T_i	Task
S_i, C_i	Start and completion time of T_i
$S_i = S_i(C_i)$	Function between start and completion times
$f_i(C_i)$	Cost of T_i
$(a_i, +\infty)$	Interval on which f_i is non-decreasing
$(-\infty, b_i)$	Interval on which f_i is non-increasing
$w_i(S_{i+1} - C_i)$	Idleness cost of the time interval between T_i and T_{i+1}
$(c_i, +\infty)$	Interval on which w_i is non-decreasing
$\|\|f\|\|$	Number of segments of a piecewise linear function f

Since a task may have several successors, we have to change our notation about the idleness cost functions. In the following lines, the idleness cost function between the end of T_i and the start of T_j is denoted by w_{ij} for any precedence constraint between T_i and T_j (denoted by the arc (i,j) for short). To solve the problem, we iteratively remove a task at each step. When a task is removed, a subproblem is created for each broken precedence arc. When T_j is removed, for all the predecessor $i_1, i_2, \ldots$ and each successor $k_1, k_2, \ldots$ of j in the precedence graph, we define the subproblems:

1. $P^-_{i_1 j}(t), P^-_{i_2 j}(t), \ldots$ where $P^-_{lj}(t)$ is the subproblem of scheduling the tasks included in the remaining subtree that precedes (l,j) subject to T_j starts at time t. In this subproblem, the idleness cost between T_l and T_j is considered but not the cost of scheduling T_j at time t.
2. $P^+_{jk_1}(t), P^+_{jk_2}(t), \ldots$ where $P^+_{jl}(t)$ is the subproblem of scheduling the tasks included in the remaining subtree that succeeds (j,l) subject to T_j starts at time t. In this subproblem, the idleness cost between T_j and T_l is considered but not the cost of scheduling T_j at time t.

The optimal costs of the subproblems are respectively denoted by $\Sigma^-_{ij}(t)$ and $\Sigma^+_{jk}(t)$. Fig. 3 represents the decomposition process to solve $P^-_{jk}(t)$ by removing the task T_j. At the previous step of the decomposition, k had been removed, which had been the cause of the creation of the subproblem $P^-_{jk}(t)$.

We clearly have the following recurrence equations, in which $l|(lj) \in A$ is a short notation to specify the indices of the tasks T_l such that there is a precedence constraint between T_l and T_j (A denotes the set of the precedence arcs):

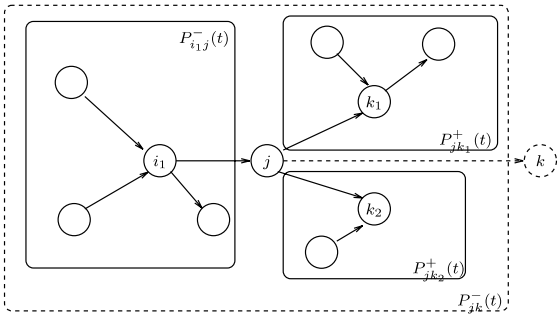

Fig. 3. Decomposition to solve the problem with a tree precedence graph.

$$\Sigma^-_{jk}(t) = \min_{t' \leqslant S_k(t)} \left(w_{jk}(S_k(t) - t') + f_j(t') + \sum_{l|(lj)\in A} \Sigma^-_{lj}(t') + \sum_{l|(jl)\in A-\{k\}} \Sigma^+_{jl}(t') \right),$$

$$\Sigma^+_{ij}(t) = \min_{S_j(t') \geqslant t} \left(w_{ij}(t - S_j(t')) + f_j(t') + \sum_{l|(lj)\in A-\{i\}} \Sigma^-_{lj}(t') + \sum_{l|(jl)\in A} \Sigma^+_{jl}(t') \right).$$

The algorithm starts by randomly selecting a task T_k. Then all the subproblems $P^-_{lk}(t)$ for $(l,k) \in A$ and $P^+_{kl}(t)$ for $(k,l) \in A$ are solved by the above recursive decomposition. A recursive decomposition stops when the subgraph only contains one

node. After all the subproblems are solved, the optimal cost is finally given by

$$\min_t \left(f_k(t) + \sum_{l/(lk)\in A} \Sigma^-_{lk}(t) + \sum_{l/(kl)\in A} \Sigma^+_{kl}(t) \right).$$

3.2. Piecewise linear functions

The computability and complexity of this algorithm strongly depend on the cost functions given in input. Hereafter, we will study the—somewhat general—case where all the functions are piecewise linear. Each piecewise linear function is supposed to be given as an ordered segment list. For each segment in the list, five values are given corresponding to its definition interval, its slope and the coordinates of one point in the segment. We of course assume that the definition intervals of the segments do not overlap. So the number of segments is a good indicator of the amount of information given in input to describe a piecewise linear function. In this paper, the number of segments of a piecewise linear function f is denoted by $\|f\|$.

For the sake of simpler notations, if a piecewise linear function f is not defined on $\mathbb{R}$, we are going to say that for any real value t outside the definition domain, $f(t) = \infty$. In other words, we transform the initial function into a piecewise linear function defined on $\mathbb{R}$ with constant segments equal to ∞. Clearly, if f has n segments, the transformed function has $O(n)$ segments. Moreover, the transformed function has the property that the rightmost point of a segment—if not infinite—has the same abscissa than the leftmost point of the next segment. Such a point will be hereafter called a *breakpoint* of the function.

We now present a series of—easy—lemmas about basic operations involving piecewise linear functions and their computation. Our complexity analysis in Section 3.3 will of course relies on all these lemmas. In these lemmas, f_1 and f_2 denote two arbitrary piecewise linear functions with respectively $n_1 = \|f_1\|$ and $n_2 = \|f_2\|$ segments. To avoid any possible confusion, note that in the text of the following lemmas, f_1 and f_2 are not the cost functions of tasks T_1 and T_2 defined in Section 2.

All the presented operators OP return a piecewise linear function denoted by $f = \mathrm{OP}(f_1, f_2)$ or $f = \mathrm{OP}(f_1)$. For each operator, we give an upper bound of the number of segments of f. This upper bound is very important to analyze the computational complexity of the algorithms: if $\|\mathrm{OP}(f_1)\| \leqslant n_1 + c$ (for some constant c) then $\|\mathrm{OP}^n(f_1)\| \leqslant n_1 + cn$ but if $\|\mathrm{OP}(f_1)\| \leqslant 2n_1$, we may have $\|\mathrm{OP}(f_1)\| \in \Theta(2^n n_1)$. Therefore, even if $\|\mathrm{OP}(f_1)\| \in \mathrm{O}(n_1)$ in both cases, the computation of $\mathrm{OP}^n(f_1)$ is polynomial in the first case, but it is exponential in the second case.

Lemma 2. *The function $f(t) = f_1(t) + f_2(t)$ is a piecewise linear function with at most $n_1 + n_2$ segments. It can be computed in $O(n_1 + n_2)$ time.*

Lemma 3. *The function $f(t) = \min(f_1(t), f_2(t))$ is a piecewise linear function with at most $2(n_1 + n_2)$ segments. It can be computed in $O(n_1 + n_2)$ time.*

Proof. Clearly, t is a breakpoint for f only if t is a breakpoint for f_1 or f_2 or if $f_1(t) = f_2(t)$ (and $f'_1(t) \neq f'_2(t)$). There are at most $n_1 + n_2 - 2$ breakpoints of f that correspond to the breakpoints of f_1 and f_2. Between two such breakpoints, both f_1 and f_2 are linear so either these two functions are identical (and there is no breakpoint) or there is zero or one intersection point so that f has at most one breakpoint on this open interval. In conclusion, f has at most $2(n_1 + n_2)$ segments. It can obviously be computed in $O(n_1 + n_2)$. □

From the previous proof, we observe that the function $f(t) = \min(f_1(t), \ldots, f_k(t))$ has at most $k \sum_{i=1}^{k} \|f_i\|$ segments.

Lemma 4. *If f_2 is continuous and non-decreasing, the function $f(t) = f_1(f_2(t))$ is a piecewise linear function with at most $n_1 + n_2$ segments. It can be computed in $O(n_1 + n_2)$ time.*

Proof. Since f_2 is continuous and non-decreasing, for each breakpoint t of f_1, the set of real values $S_t = \{t' \in \mathbb{R} | f_2(t') = t\}$ is either a single point or a closed interval if the slope of f_2 at t' is null. Let us consider the sorted list L that contains the breakpoints of f_2 and the values S_t, for all the break-

points t of f_1 such that S_t is a single point. Indeed if S_t is not a single point, both its endpoints are breakpoints of f_2 so they are already in L. So, this list contains at most $n_1 + n_2$ points. In the open interval between two consecutive points of L, the function f is clearly linear as the composition of two linear functions. So f is piecewise linear with at most $n_1 + n_2$ segments. The following algorithm, very similar to the algorithm to merge sorted arrays, computes the sorted list in $O(n_1 + n_2)$ time:

```
let t1 be the first breakpoint of f1
let t2 be the first breakpoint of f2
let L ← ∅
while t1 or t2 exists do
  if f2(t2) ⩽ t1 then
    add t2 to L
    let t2 be the next breakpoint of f2
    if f2(t2) = t1 then
      let t1 be the next breakpoint of f1
  else
    let t be smallest value such that f2(t) = t1
    add t to L
    let t1 be the next breakpoint of f1
```

So the function f can be computed in $O(n_1 + n_2)$ time. □

The following operation between f_1 and f_2 can be viewed as the composition between f_1 and f_2^{-1}. It will be used in Section 5.

Lemma 5. *If f_2 is continuous and non-decreasing, the function $f(t) = \min_{t=f_2(tT')} f_1(t')$ is a piecewise linear function with at most $n_1 + n_2$ segments. It can be computed in $O(n_1 + n_2)$ time.*

Proof. (sketch). The proof that f has at most $n_1 + n_2$ segments is similar to the proof of Lemma 4 but here, $S_{t'}$ is defined for each breakpoint t' of f_1 as $S_{t'} = \{t \in \mathbb{R} | f_2(t') = t\}$. The breakpoints of f correspond to the endpoints of these segments (or single points) $S_{t'}$ and to the image by f_2 of the breakpoints of f_2. □

Lemma 6. *Let δ be a non-negative real value. The function $f(t) = \min_{t-\delta \leqslant t' \leqslant t} f_1(t')$ is a piecewise linear function with at most $2n_1$ segments. It can be computed in $O(n_1)$ time.*

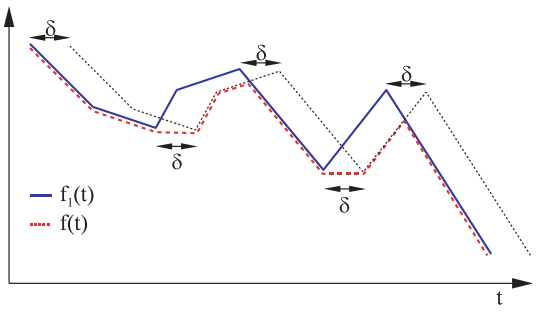

Fig. 4. $f(t) = \min_{t-\delta \leqslant t' \leqslant t} f_1(t')$.

Proof. The construction of f is illustrated by Fig. 4. For each breakpoint t_i $(1 \leqslant i < n_1)$ of the function f_1, let h_i be the constant function defined on the interval $[t_i, t_i + \delta)$ equal to the value $\lim_{\theta \to t_i^-} f_1(\theta)$. For any real value t, the global minimum of the function f_1 on the interval $[t - \delta, t]$ is reached either at t or at $t - \delta$ or at a local minimum of f_1 in the interval. Such a local minimum can only be a breakpoint of f_1 so that we have

$$f(t) = \min(f_1(t), f_1(t - \delta), h_1(t), h_2(t), \ldots, h_{n_1 - 1}(t)).$$

Let $h(t)$ be the stepwise function defined by $\min_{1 \leqslant i < n_1} h_i(t)$. Since the breakpoints of f_1 are given ordered in input and since the definition intervals of all h_i functions have the same length (δ), h can be computed in $O(n_1)$ time and has $O(n_1)$ steps. Therefore, from Lemma 3, the piecewise linear function defined by $f(t) = \min(f_1(t), f_1(t - \delta), h(t))$ can be computed in $O(n_1)$ time and has $O(n_1)$ segments.

Clearly, segments of f with a non-positive slope correspond to the segments of f_1 while segments of f with a positive slope corresponds to the segments of $t \mapsto f_1(t - \delta)$. So there are at most n_1 segments in f which corresponds to f_1 or to its translations. The number of horizontal segments corresponding to the functions h_i is bounded by the number of local minima, which is less than n_1. □

When δ is greater than $t_{n_1} - t_1$, we clearly have that $f(t) = \min_{t' \leqslant t} f_1(t')$ which gives the following corollary illustrated by Fig. 5.

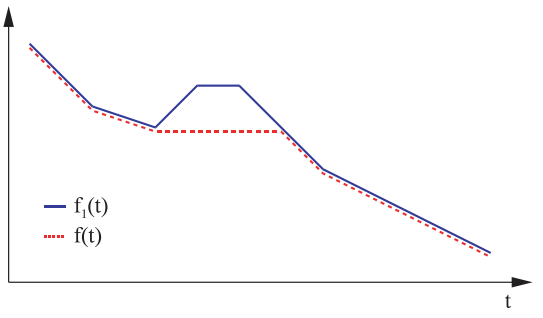

Fig. 5. $f(t) = \min_{t' \leqslant t} f_1(t')$.

Corollary 7. *The function $f(t) = \min_{t' \leqslant t} f_1(t')$ is a piecewise linear function with at most n_1 segments. It can be computed in $O(n_1)$ time.*

Proof. The function f has no segment with positive slope. A segment s of f which does not correspond to a segment of f_1 has necessary a null slope and the left endpoint of s is a local minimum of f_1. So s can be associated to the segment of f_1 that is at the right of the local minimum (the slope of which is positive). So, there is a one-to-one mapping between the segments of f_1 and f so that $\|f\| \leqslant n_1$. □

The following lemma is similar to Corollary 7 but with two functions.

Lemma 8. *The function $f(t) = \min_{t_1+t_2 \geqslant t}(f_1(t_1) + f_2(t_2))$ is a piecewise linear function. It can be computed in polynomial time in n_1 and n_2.*

Proof. We are going to prove that $\hat{f}(t) = \min_{t_1+t_2=t} f_1(t_1) + f_2(t_2)$ is a piecewise linear function with $O(n_1n_2)$ segments that can be computed in $O(n_1n_2)$ time. Indeed, we clearly have that $f(t) = \min_{t' \geqslant t} \hat{f}(t')$ and so f can be derived from $\hat{f}$ thanks to Corollary 7 by reversing the abscissa axis.

Let $I_1, I_2, \ldots, I_{n_1}$ be the definition intervals of each segment of the piecewise linear function f_1. On each interval I_k, f_1 is linear so there are α_k and β_k such that for all $t \in I_k$, $f_1(t) = \alpha_k t + \beta_k$. We can then write $\hat{f}(t) = \min_{1 \leqslant k \leqslant n_1} g_k(t)$ with $g_k(t) = \min_{t_1 \in I_k} \alpha_k t_1 + \beta_k + f_2(t - t_1)$. So $g_k(t) = \alpha_k t + \beta_k + \min_{t_1 \in I_k} f_2(t - t_1) - \alpha_k(t - t_1)$. From Lemmas 4 and 6, the last term of function g_k has at most n_2 segments and can be computed in $O(n_2)$ time. So each function g_k has at most n_2 segments and can be computed in $O(n_2)$ time, which shows that f has $O(n_1^2 n_2)$ segments and can be computed in $O(n_1^2 n_2)$ time. □

We can give a simple example in which the function $\hat{f}(t)$ defined in Lemma 8 has $\Theta(n_1n_2)$ segments. f_1 is defined on the interval $[0, n_1n_2)$ and $f_1(t) = (n_1 + 1)\lceil t/n_1 \rceil$. f_2 is defined on the interval $[0, n_1)$ and $f_2(t) = \lceil t \rceil$. f_1 and f_2 are continuous from the minimum. Since they are both non-decreasing, we have $\hat{f}(t) = \min_{t_1+t_2=t} f_1(t_1) + f_2(t_2)$. As $t_2 \in [0, n_1)$, we have that $t_1 \in (t - n_1, t]$. Let $\alpha = \lfloor t/n_1 \rfloor$ and $\beta = t - \alpha n_1$ that is a real value in $[0, n_1)$. $f_1(\alpha n_1) + f_2(\beta) = \alpha(n_1 + 1) + \lceil \beta \rceil$. If $t_1 > \alpha n_1$ then $f_1(t_1) \geqslant (\alpha + 1)(n_1 + 1) \geqslant f_1(\alpha n_1) + f_2(\beta)$. If $t - n_1 < t_1 \leqslant \alpha n_1$, $f_1(t_1) = \alpha(n_1 + 1)$ and, since f_2 is non-decreasing, $f_2(t - t_1) \geqslant f_2(\beta)$. So the minimum is reached for $t_1 = \alpha n_1$ and $t_2 = \beta$ which means that $f(t) = f_1(\lfloor t/n_1 \rfloor) + f_2(t - \lfloor t/n_1 \rfloor)$ and this function has n_1n_2 steps of length 1 in $[0, n_1n_2)$.

3.3. Complexity analysis

Lemma 8 shows that, while the functions Σ_k are iteratively computed, an exponential number of segments may be created. This result is not surprising, since the problem is NP-complete according Theorem 1.

However, we observe that when the abscissas of all the breakpoints of all the cost functions are integer and if $S_i(C_i)$ is integer when C_i is integer, there exists an optimal solution in which all the start and completion times are integer. Clearly, we observe that the endpoints of the functions Σ_k generated by the DP algorithm have integer endpoints so that their number of segments is bounded by an upper bound of the makespan of the schedule, for example $h = \max_{1 \leqslant i \leqslant n} b_i + \sum_{1 \leqslant i \leqslant n} p_i + \sum_{1 \leqslant i < n} c_i$. The complexity of the algorithm is in $O(nh^2)$, which is pseudo-polynomial in the size of the input of the instance.

For the complexity analysis of the problem, the special case where the idle time costs are linear is interesting. Indeed, the operation presented in Lemma 8 is no more necessary due to the simpli-

fication of the recursive relationship (2). The operation of Corollary 7 can be used instead so we have that the number of segments of each function Σ_k is at most $\sum_{i=1}^{k} \|f_i\| + \sum_{i=2}^{k} \|S_i\|$. Therefore, the complexity of the algorithm is $O(n(\sum_{i=1}^{n} \|f_i\| + \sum_{i=2}^{n} \|S_i\|))$. So, the problem is polynomial in the size of the input of the problem, which is the main result of this paper.

In the pure earliness–tardiness problem, each cost function f_i has two segments so that the complexity of the algorithm is $O(n^2)$. We recall this problem can be solved in $O(n \log n)$.

4. Special cases

In this section, we present some special cases in which there are some additional properties that make the problem simpler. In this section, we assume that the durations of the tasks are time-independent, that is for any i there is a constant p_i such that $S_i = C_i - p_i$.

4.1. Uniform idle time costs

When the idle time costs are linear and task-independent, that is there is some α such that for any $i \in \{1, \ldots, n\}$, $w_i(t) = \alpha t$, the problem can be reduced to a problem without idle time cost. Indeed, the total cost given by Eq. (1) is now

$$\sum_{1 \leqslant i \leqslant n} f_i(C_i) + \alpha \sum_{1 \leqslant i < n} (C_{i+1} - p_{i+1} - C_i),$$

that is

$$\left(\sum_{1 \leqslant i \leqslant n} f_i(C_i) \right) + \alpha C_n - \alpha C_1 - \alpha \sum_{1 < i \leqslant n} p_i.$$

Since the last term is a constant, the minimization problem is equivalent to the problem with null idle time costs in which the costs functions are:

$$\widehat{f_i}(C) = \begin{cases} f_1(C) - \alpha C & \text{for } i = 1, \\ f_i(C) & \text{for } 1 < i < n, \\ f_n(C) + \alpha C & \text{for } i = n. \end{cases}$$

Even if this simplification does not improve the theoretical complexity of the algorithm, it simplifies the recursive relationship so that each step of the recurrence can be more easily computed.

4.2. Weighted flowtime

When each function f_k is defined by $f_k(C_k) = \alpha_k C_k$ (with $\alpha_k > 0$) and all tasks are released at 0, the problem can be reduced to a problem with null punctuality costs by integrating the punctuality costs into the idleness costs. More formally, let us define:

$$\hat{f}_k(C_k) = 0,$$

$$\hat{w}_k(t) = w_k(t) + \sum_{k < i \leqslant n} \alpha_i t.$$

We can easily check that, for any feasible schedule, the total cost is

$$\sum_{1 \leqslant k \leqslant n} f_k(C_k) + \sum_{1 \leqslant k < n} w_k(S_{k+1} - C_k)$$

$$= K + \sum_{1 \leqslant k < n} \hat{w}_k(S_{k+1} - C_k) + \left(\sum_{1 < i \leqslant n} \alpha_i \right) C_1,$$

where K is the constant $\sum_{1 \leqslant k < n} \sum_{k < i \leqslant n} \alpha_i p_{k+1}$. As a consequence, the problem is now easy because we simply have to minimize each term of the right member of the equality. Therefore, the solution is to schedule the first task at the origin and to compute the idle time i_k between any task T_k and T_{k+1} such that $\hat{w}_k(i_k)$ is minimum, which can be done in $O(\|\hat{w}_k\|)$, that is in $O(\|w_k\|)$. Then, the complexity of the algorithm is $O(\sum \|w_k\|)$.

4.3. Earliness–tardiness around a common due date

We consider that all the tasks have a common due date d and there is no release date, that is the tasks are scheduled at $-\infty$, which corresponds to the *unrestricted* common due date case of the literature [9]. The cost function of each task T_i is $f_i(C_i) = \max(\alpha_i(d - C_i), \beta_i(C_i - d))$. In this expression, α_i and β_i are respectively the earliness and tardiness penalty per unit time.

Lemma 9. *There is an optimal schedule in which a task completes at time d.*

Proof. Let $S_1, \ldots, S_n$ be an optimal schedule. We consider the function $h(\delta)$ that returns the cost of

the schedule $S_1+\delta,\ldots,S_n+\delta$. When δ varies, each idle period stays constant so that the total idleness cost is a constant I. Therefore, $h(\delta)=I+\sum_i \max(\alpha_i(d-S_i-p_i-\delta),\ \beta_i(S_i+p_i+\delta-d))$, which means that h is convex and piecewise linear. The minimum of h is reached at a breakpoint $\delta^\star$, so we have, for some $i\in\{1,\ldots,n\}$, $d=S_i+p_i+\delta^\star$. So, in the schedule $S_1+\delta^\star,\ldots,S_n+\delta^\star$ whose cost is not greater than the optimal schedule, T_i completes at time t. □

This lemma shows that the problem can be solved by computing, for each T_i $(1\leqslant i\leqslant n)$, the optimal schedule subject to the constraint that T_i completes at d (that is $C_i=d$). For a given i, when T_i must complete at d, we have two independent subproblems that are to optimally schedule all the jobs in $\{1,\ldots,i-1\}$ and then all the jobs in $\{i+1,\ldots,n\}$. In the first subproblem, all the tasks are early and in the second one they are all late. So, each subproblem is obviously reduced to the case in which the total punctuality cost is the weighted flowtime criterion—note that the time direction has to be reversed for the first subproblem. So they are solved by the algorithm presented in the previous subsection.

However, if all the tasks are released at time 0, we can easily show that the problem is NP-complete but it becomes easy when the idleness costs are non-decreasing because there is a solution with no idle time.

4.4. Convex cost functions

When cost functions w_i and f_i are all convex, the problem can be formulated as a linear program, which shows the problem is polynomial. Here we show that the recurrence of the dynamic program is also computable in polynomial time. The following lemma improves the result of Lemma 8, which is the cause of the exponential factor in the complexity given in Section 3.3 for the general case.

Lemma 10. *If f_1 and f_2 are continuous and convex piecewise linear functions on the domain $\mathbb{R}^+$, the function $f(t)=\min_{t_1+t_2=t}(f_1(t_1)+f_2(t_2))$, defined on $\mathbb{R}^+$, is a convex piecewise linear function with at most n_1+n_2 segments. It can be computed in $O(n_1+n_2)$ time.*

Proof. For any t, we have that $f(t)=f_1(\theta_1(t))+f_2(\theta_2(t))$ with $\theta_1(t)+\theta_2(t)=t$. Our proof relies on the fact that θ_1 and θ_2 can be chosen to be non-decreasing so that f can be constructed by traversing the segments of f_1 and f_2 from the left to the right. The algorithm that constructs f, constructs θ_1 and θ_2 starting at $t=0$ with $\theta_1(t)=\theta_2(t)=0$.

Let us consider the parametrized functions ϕ_t defined on the domain $[0,t]$ such that $\phi_t(t_1)=f_1(t_1)+f_2(t-t_1)$. For any $t\geqslant 0$, ϕ_t is convex, as a sum of two convex functions so that the set on which ϕ_t is at its minimum is a closed interval of $[0,t]$—possibly a single point. Let $\theta_1(t)$ be the left endpoint of this interval and $\theta_2(t)=t-\theta_1(t)$. Clearly $f(t)=f_1(\theta_1(t))+f_2(\theta_2(t))$. The function ϕ_t is piecewise linear so that $\theta_1(t)$ is a breakpoint of ϕ_t.

ϕ_t has of course no derivative at breakpoint but two different derivatives from the left and the right. We define $\phi'_t(t_1)$ as the slope from the right at t_1 of ϕ_t (Fig. 6). The function ϕ'_t is stepwise, continuous from the right and non-decreasing. According to our definition of $\phi'_t(t_1)$, $\theta_1(t)$ is the smallest t_1 such that $\phi'_t(t_1)\geqslant 0$. We have $\phi'_t(t_1)=f'_1(t_1)-f'_2(t-t_1)$ where f'_1 (resp. f'_2) is the derivate from the right (resp. left) of f_1 (resp. f_2). Since f'_2 is non-decreasing, $\phi'_t(t_1)$ is non-increasing when t increases and t_1 is fixed. So we have that $\theta_1(t)$ is non-decreasing. We prove θ_1 is continuous by

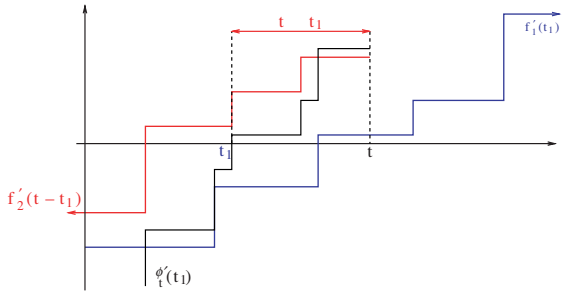

Fig. 6. The function $\phi'_t(t_1)$ in the proof of Lemma 10.

observing that $\phi'_{t+\epsilon}(t_1+\epsilon) = f'_1(t_1+\epsilon) - f'_2(t- t_1)$. So, for any $\epsilon \geqslant 0$, we have that $\phi'_{t+\epsilon}(\theta_1(t)+\epsilon) \geqslant \phi'_t(\theta_1(t)) \geqslant 0$ so that $\theta_1(t+\epsilon) \leqslant \theta_1(t)+\epsilon$. Since $\theta_1(t) \leqslant \theta_1(t+\epsilon) \leqslant \theta_1(t)+\epsilon$, θ_1 is continuous. With the help of these inequalities, we prove that for any $\epsilon \geqslant 0$,

$$\begin{aligned}\theta_2(t+\epsilon) &= t+\epsilon-\theta_1(t+\epsilon)\\ &\geqslant t+\epsilon-(\theta_1(t)+\epsilon) = \theta_2(t),\end{aligned}$$

which means that θ_2, that is clearly continuous, is also non-decreasing.

We have already said that $\theta_1(t)$ is a breakpoint of ϕ_t. This means that either $\theta_1(t)$ is a breakpoint of f_1 or $\theta_2(t)$ is a breakpoint of f_2. If there was an open interval on which both θ_1 and θ_2 are strictly increasing, there would be some t such that $\theta_1(t)$ is not a breakpoint of f_1 and $\theta_2(t)$ is not a breakpoint of f_2, which is a contradiction. So, for any t either $\theta_1(t)$ is constant and θ_2 increases with t or vice versa. When $\theta_1(t)$ and $\theta_2(t)$ are simultaneously breakpoints of f_1 and f_2 respectively, we clearly know whether $\theta_1(t)$ must increase or stay constant by comparing the slopes of f_1 and f_2.

We complete the proof by giving the description of the algorithm that constructs f by computing its breakpoints from the left to the right.

$t \leftarrow 0$; $t_1 \leftarrow 0$; $t_2 \leftarrow 0$
repeat
  $f(t) \leftarrow f_1(t_1) + f_2(t_2)$
  let t'_1 be the next breakpoint of f_1 after t_1
  let t'_2 be the next breakpoint of f_2 after t_2
  let s_1 be the right-slope of f_1 at t_1
  let s_2 be the right-slope of f_2 at t_2
  if $s_1 < s_2$
  then $t \leftarrow t'_1 + t_2$; $t_1 \leftarrow t'_1$
  else $t \leftarrow t_1 + t'_2$; $t_1 \leftarrow t'_2$
until t_1 and t_2 have reached the end of their domain

At each step of the algorithm t_1 and t_2 respectively corresponds to a breakpoint of f_1 and f_2. Since t_1 and t_2 are non-decreasing and the pair (t_1, t_2) changes at each step, we have proved that there are at most $n_1 + n_2$ steps, which proves the time complexity of the algorithm and the space complexity of f. The slopes s_1 and s_2 are also non-decreasing so that the slopes of f that are the successive values $s_1 + s_2$ are non-decreasing. Therefore f is convex. □

In corollary of this lemma, the complexity of the problem with convex cost functions is $O(n(\sum_{i=1}^{n-1} \|w_i\| + \sum_{i=1}^{n} \|f_i\| + \sum_{i=2}^{n} \|S_i\|))$.

5. Optimal filtering algorithm

Filtering algorithms are of key importance in Constraint Programming because their role is to remove from the domain of the variables, the values that cannot lead to a *feasible* solution. Sometimes, the removal of values that cannot lead to an *optimal* solution can be implemented. In constraint-based scheduling [2], a pair of variables is usually devoted to the start and end times of each activity. In this section, we give an algorithm that determines the possible end times for each activity so that the total cost of the schedule is not greater than a given maximum cost F. The possible start time can then be directly determined by the functions S_k. This filtering algorithm is said to be *optimal* because any possible end time rendered by the algorithm corresponds to at least one feasible schedule with a cost not greater than F. Conversely, an end time that is not rendered by the algorithm would necessarily lead to a schedule with a cost greater than F.

The algorithm described in this section is of prime importance to solve the general sequencing problem with general cost functions. In particular:

- only a subset of tasks are sequenced on a machine and we have a lower bound on the cost for scheduling all the non-sequenced tasks. The filtering algorithm can be used to reduce the time windows of the sequenced tasks so that we get a better description of the partial schedule. Of course, in this case the filtering algorithm is not optimal.
- there are several machines, only some machines are sequenced. Once again, the filtering algorithm can reduce the time windows for the tasks on the sequenced machines. These informations

are useful both for a branch-and-bound algorithm or a decomposition approach like the shifting-bottleneck procedure [1].

5.1. Problem description

We keep the notations introduced in Section 2 and we add the notation F that represents the maximum possible cost. Therefore, the objective criterion given by (1) is replaced by the following hard constraint:

$$\sum_{1 \leqslant i \leqslant n} f_i(C_i) + \sum_{1 \leqslant i < n} w_i(S_{i+1}(C_{i+1}) - C_i) \leqslant F. \qquad (3)$$

The problem is to compute, for each task T_k ($1 \leqslant k \leqslant n$), the set $\mathscr{C}_k(F)$ of all the possible completion times t such that there exists at least one feasible schedule satisfying the precedence constraints given in Section 2, the cost constraint (3) and the constraint $C_k = t$.

5.2. Algorithm and properties

The function $\Sigma_k(t)$ defined by relationship (2) gives the minimal cost to schedule the *initial* subsequence $T_1, \ldots, T_k$ such that $C_k = t$. Symmetrically, we define $\overline{\Sigma}_k(t)$ as the minimal cost to schedule the *terminal* subsequence $T_k, \ldots, T_n$ such that T_k ends at t, that is $C_k = t$. We set $\overline{\Sigma}_n(t) = f_n(t)$ for any t. If T_k (with $k < n$) ends at t and T_{k+1} ends at t' with $S_{k+1}(t') \geqslant t$, the minimum cost to schedule $T_k, \ldots, T_n$ is $f_k(t) + w_{k+1}(S_{k+1}(t') - t) + \overline{\Sigma}_{k+1}(t')$. So $\overline{\Sigma}_k(t) = f_k(t) + \min_{S_{k+1}(t') \geqslant t}(w_{k+1}(S_{k+1}(t') - t) + \overline{\Sigma}_{k+1}(t'))$.

For each task T_k, we can then define the function $\Sigma_k^\star(t)$ that is equal to the minimum cost to schedule all the tasks $T_1, \ldots, T_n$ such that $C_k = t$. Clearly, we have:

$$\Sigma_k^\star(t) = \Sigma_k(t) + \overline{\Sigma}_k(t) - f_k(t). \qquad (4)$$

So, $\mathscr{C}_k(F) = \{t | \Sigma_k^\star(t) \leqslant F\} = \Sigma_k^{\star -1}((-\infty, F])$. As for Σ_k, the function $\overline{\Sigma}_k$ is continuous from the minimum. $\Sigma_k^\star$ is also continuous from the minimum since $\overline{\Sigma}_k(t) - f_k(t)$ is continuous from the minimum.

Theorem 11. *$\mathscr{C}_k(F)$ is a closed set.*

Proof. If $\mathscr{C}_k(F)$ is not empty, let us consider an infinite sequence (t_i) such that for each $i \in \mathbb{Z}^+$, $t_i \in \mathscr{C}_k(F)$ and $\lim_{i \to \infty} t_i = t$. Since $\Sigma_k^\star$ is either continuous from the right or from the left, we can extract from this sequence an infinite sequence $(\hat{t}_i)$ such that $(\Sigma_k^\star(\hat{t}_i))$ has a limit ℓ. Since for each i, $\Sigma_k^\star(\hat{t}_i) \leqslant F$, we have that $\ell \leqslant F$. Since $\Sigma_k^\star$ is continuous from the minimum, $\Sigma_k^\star(t) \leqslant \ell$. Therefore, $\Sigma_k^\star(t) \leqslant F$ and $t \in \mathscr{C}_k(F)$. □

The following theorem is useful to speed up the computation of $\mathscr{C}_k(F)$ in the earliness–tardiness case without idle time costs—or with a constant idle time cost, which is equivalent (see Section 4.1).

Theorem 12. *When all the cost functions are convex and when there is no idle time costs, $\mathscr{C}_k(F)$ is an interval.*

Proof. We first remark that if a function g is convex, the function $f(t) = \min_{t' \leqslant t} g(t')$ is also a convex function because f and g are equal before the minimum of g and f is constant after the minimum of g (see Fig. 7). Therefore, Σ_k is convex as a sum of convex function. So the functions $\overline{\Sigma}_k - f_k$ and $\Sigma_k^\star$ are also convex. The convexity of $\Sigma_k^\star$ proves that $\mathscr{C}_k(F)$ is an interval. □

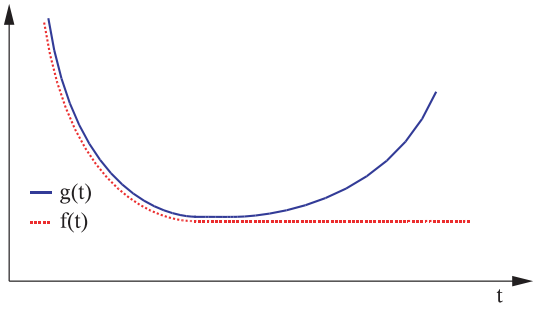

Fig. 7. $f(t) = \min_{t' \leqslant t} g(t')$ is convex if g is convex.

The study of the computation of $\mathscr{C}_k(F)$ is very similar to what was done in Section 2.3. However, we must rewrite the definition of $\overline{\Sigma}_k$ as

$$\overline{\Sigma}_k(t) = f_k(t) + \min_{t'' \geqslant t} \left(w_{k+1}(t'' - t) + \min_{t' = S_{k+1}(t')} \overline{\Sigma}_{k+1}(t') \right),$$

to show with Lemma 5 that the function can be computed with the same time and space complexity than Σ_{k+1} in Section 3.3. Therefore, $\Sigma_k^\star$ can also be computed with the same complexity. In particular, the conditions for the problem to be polynomial or pseudo-polynomial are the same.

6. Conclusion

We presented a Dynamic Programming algorithm to schedule—i.e. to time—a sequence of tasks in order to minimize the total cost, including idle time costs. The algorithm is polynomial when idle time costs are linear, which is not restrictive for practical problems. Moreover, the algorithm is still valid for the project scheduling problem without resource constraint and with a tree precedence graph.

An interesting point of this algorithm is that it can be very easily implemented. For example, the problem with piecewise linear cost functions was implemented in a few lines of code using the piecewise linear function facilities of ILOG Scheduler 5.2 [11]. In our C++ implementation, the CPU time to solve pure earliness–tardiness instances with 1500 randomly sequenced tasks ranges between 0.11 and 0.14 seconds on a 1 MHz PC. Instances with 2000 tasks are solved in about 0.17 seconds (±0.03 seconds). In comparison, running times for our implementation of the algorithm of Garey et al. [8] is about 0.39 seconds for problems with 1500 tasks (0.60 seconds for 2000 tasks).

This algorithm can be adapted—with no extra computational cost—to get information on the possible execution time window for each task so that a maximum fixed cost is not exceeded.

Both information on the minimum cost of a sequence and on the possible time windows should be very useful to determine an optimal sequence by a branch-and-bound algorithm. Further research will focus on such an algorithm and its use in solving shop problems.

Acknowledgements

Part of the work was done while the author was working for ILOG.

References

[1] J. Adams, E. Balas, D. Zawack, The shifting bottleneck procedure for job shop scheduling, Management Science 34 (1988) 391–401.
[2] Ph. Baptiste, C. Le Pape, W. Nuijten, Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems, Kluwer Academic Publishers, Norwell, MA, 2001.
[3] R. Bellman, Dynamic Programming, Princeton University Press, Princeton, NJ, 1957.
[4] Ph. Chrétienne, F. Sourd, Scheduling with convex cost functions, Theoretical Computer Science 292 (2003) 145–164.
[5] F. Della Croce, M. Trubian, Optimal idle time insertion in early–tardy parallel machines scheduling with precedence constraints, Production Planning and Control 13 (2002) 133–142.
[6] J.S. Davis, J.J. Kanet, Single machine scheduling with early and tardy completion costs, Naval Research Logistics 40 (1993) 85–101.
[7] T.D. Fry, R.D. Armstrong, J.H. Blackstone, Minimize weighted absolute deviation in single machine scheduling, IIE Transactions 19 (1987) 445–450.
[8] M.R. Garey, R.E. Tarjan, G.T. Wilfong, One-processor scheduling with symmetric earliness and tardiness penalties, Mathematics of Operations Research 13 (1988) 330–348.
[9] V. Gordon, J.M. Proth, C. Chu, A survey of the state-of-the-art of common due date assignment and scheduling research, European Journal of Operational Research 139 (2002) 125.
[10] J.A. Hoogeveen, S.L. van de Velde, A branch-and-bound algorithm for single-machine earliness–tardiness scheduling with idle time, INFORMS Journal on Computing 8 (1996) 402–412.
[11] ILOG Inc., ILOG Scheduler 5.2 User's Manual and Reference Manual, December 2001.
[12] S. Lakshminarayan, R. Lakshmanan, R.L. Papineau, R. Rochete, Optimal single-machine scheduling with earliness and tardiness penalties, Operations Research 26 (6) (1978) 1079–1082.
[13] R.H. Möhring, A.S. Schulz, F. Stork, M. Uetz, On project scheduling with irregular starting time costs, Operations Research Letters 28 (2001) 149–154.

[14] F. Sourd, S. Kedad-Sidhoum, The one machine problem with earliness and tardiness penalties, Journal of Scheduling 6 (2003) 533–549.
[15] W. Szwarc, S.K. Mukhopadhyay, Optimal timing schedules in earliness–tardiness single machine sequencing, Naval Research Logistics 42 (1995) 1109–1114.
[16] G. Wan, B.P.C. Yen, Tabu search for single machine scheduling with distinct due windows and weighted earliness/tardiness penalties, European Journal of Operational Research 142 (2002) 271–281.