Efficient neighborhood search for Just-in-Time scheduling problems*

Yann Hendel - Francis Sourd

LIP6 - Université Pierre et Marie Curie
4, place Jussieu — F-75252 Paris Cedex 05, France
{Yann.Hendel,Francis.Sourd}@lip6.fr

Abstract

This paper addresses the one-machine scheduling problem where the objective is to minimize a sum of costs such as earliness-tardiness costs. Since the sequencing problem is NP-hard, local search is very useful for finding good solutions. Unlike scheduling problems with regular cost functions, the scheduling (or timing) problem is not trivial when the sequence is fixed. Therefore, the local search approaches must deal with both job interchanges in the sequence and the timing of the sequenced jobs. We present a new approach that efficiently searches in a large neighborhood that always returns a solution for which the timing is optimal.

Keywords : Scheduling, Earliness-tardiness cost, neighborhoods.

1 Introduction

A set of jobs $\mathcal{J} = \{J_1, \cdots, J_n\}$ has to be scheduled on a single machine. Each job $J_j \in \mathcal{J}$ has a processing time p_j. We assume the jobs are scheduled without preemption. A schedule assigns an end time C_j to each job J_j such that it is processed in the time interval $[C_j - p_j, C_j]$. Each job J_j has *a cost function* $f_j(C_j)$ which depends on the job completion time. One has to schedule the jobs in order to minimize the sum of the cost of each job $\sum_j f_j(C_j)$.

In this paper, the f_j functions are supposed to be piecewise linear. We will denote by $\|f_j\|$ the number of segments of f_j. Both practitioners and researchers have been particularly interested in the special case in which each job J_j has a due date d_j and the completion cost function $f_j(C_j)$ has two segments, one for *early* completion ($C_j \leq d_j$) and one for *tardy* completion ($C_j \geq d_j$). In that case, we have $f_j(C_j) = \max\{\alpha_j(d_j - C_j), \beta_j(C_j - d_j)\}$ where α_j (resp. β_j) is the earliness (resp. tardiness) penalty per unit time. In this paper, the problem with piecewise linear cost functions will be referred to as PLSP (piecewise linear scheduling problem); when the functions are moreover convex, the problem is denoted by CPLSP (convex-PLSP) while the special case with earliness and tardiness penalties will be referred to as ETSP (earliness-tardiness scheduling problem).

Garey et al. [5] prove that even for the case where $\alpha_i = \beta_i = 1$ the problem is NP-complete in the strong sense. Hall et al.[7] prove that when the due dates are common

*A preliminary version of this work was presented in the conference MOSIM'03 (in French).

Problem	PLSP	CPLSP	ETSP
Complexity	$O(n \sum_{i=1}^{n} \|f_i\|)$	$O((\sum_{i=1}^{n} \|f_i\|) \log(\sum_{i=1}^{n} \|f_i\|))$	$O(n \log n)$
Reference	[10]	[2]	[5, 4]

Table 1: Complexity of the timing algorithms

($d_i = d$) the problem is NP-complete in the ordinary sense and that it can be solved by a pseudopolynomial algorithm. Researchers have then focused particularly on special cases. A central subproblem is the computation of the optimal schedule when the job order is given i.e. when the job sequence is fixed (see for instance Garey et al. [5], Davis and Kanet [4] or Szwarc and Mukhopadhyay [11]). Garey et al. [5] propose a direct algorithm based on the blocks of adjacent jobs in $O(n \log n)$, valid for the just-in-time problem with symmetric earliness and tardiness penalties. Chrétienne and Sourd [2] present a generalization for convex functions. Finally, Sourd [10] propose a dynamic programming algorithm for general non-convex piecewise linear cost functions (which also considers eventual costs for the idle periods) whose complexity depends on the number of segments of the cost functions given in input and is in $O(n(\sum_{i=1}^{n} \|f_i\|))$. These results are summarized in Table 1.

But the difficulty is the sequencing. Most of the practical methods to find good solutions to this NP-complete problem are based on variants of local search methods who, classicaly, rely on the definition of the neighborhood for any solution. For instance, James and Buchanan [8] and Wan and Yen [12] both use a tabu search algorithm and Lee and Choi [9] propose a genetic algorithm. Using the property that the optimal schedule can efficiently be obtained when the sequence is known, the feasible solutions are usually represented by the sequence of jobs. So the neighborhoods are based on the classical local search moves for the sequence, like the pairwise-interchanges — this neighborhood is sometimes limited to adjacent pairwise interchange. However, let us now consider the schedule instead of the sequence : if two adjacent jobs are swapped, leaving the start times of the other jobs unchanged, the resulting schedule is generally non-optimal with respect to the optimal timing for the new sequence. This issue clearly comes from the fact that the cost criterion is non-regular. In order to avoid this dead-end, the existing local search algorithms recompute *from scratch* the cost of all the sequences of the neighborhood in order to select the best neighbor — or at least an improving neighbor.

The aim of this paper is to show that there are more efficient methods to find the best solution in these interchange-based neighborhoods. More precisely, we show that the task interchanges and the timing operation can be combined in a single algorithm whose complexity is better than the naive approach. In the article, we refer to this operation as "*neighborhood computation*". As the number of schedules corresponding to a sequence is *a priori* infinite, the size of our neighborhood which contains all the schedules corresponding to the swapped sequences is also infinite. Therefore the proposed polynomial-time neighborhood computation is related to the very large neighborhood techniques surveyed by Ahuja et al. [1].

Two classical neighborhoods of sequencing problems will be considered : the *adjacent pairwise interchange* (API) neighborhood and the extension of API to non-adjacent jobs, which will be called the SWAP neighborhood.

In Section 2, we consider the neighborhood defined by the interchange of adjacent jobs. Then, in Section 3, we study the case of the SWAP neighborhood. Finally, in Section 4,

computational results are presented.

2 Adjacent pairwise job interchange

The computation of the API neighborhood for PLSP — which is formally presented in Section 2.2 — rely on the dynamic programming scheme proposed by Sourd [10]. The algorithm is able to compute the optimal schedule for a timing problem of a given job sequence where the objective function consists of general piecewise linear completion costs and idle period penalties. This approach solves the earliness-tardiness case in $O(n^2)$, which is not as efficient as the existing $O(n \log n)$ algorithm described in the introduction but it provides reusable results for the neighborhood computation. We first present the main features of the algorithm — leaving out the idle period penalties for simplicity.

2.1 Dynamic programming properties

For any $j \in \{1, \cdots, n\}$ and any $t \in \mathbb{R}^+$, let $P_j(t)$ denote the subproblem in which:

- the subsequence $(J_1, \cdots, J_j)$ has to be scheduled subject to the induced precedence constraints $J_1 \rightarrow J_2 \rightarrow \cdots \rightarrow J_j$.
- J_j completes at t, that is $C_j = t$.

and let $\Sigma_j(t)$ be the minimum cost of the solution of $P_j(t)$. Let us observe that the optimal cost of computing all the jobs in $\mathcal{J}$ is finally given by $\min_{t \in \mathbb{R}^+} \Sigma_n(t)$.

The interesting point is that the Σ_j functions are linked by a recursive relationship. First, we clearly have $\Sigma_1(t) = f_1(t)$. If J_j must complete at some time t' which is less than or equal to the start time of J_{j+1} which is $t - p_{j+1}$, the minimum cost of a schedule of $(J_1, \cdots, J_{j+1})$ such that $C_j = t'$ and $C_{j+1} = t$ is $\Sigma_j(t') + f_{j+1}(t)$. We thus have:

$$\Sigma_{j+1}(t) = \min_{t' \leq t - p_{j+1}} \Sigma_j(t') + f_{j+1}(t)$$

Symmetrically, we define $\bar{\Sigma}_j(t)$ as the minimum cost to schedule the subsequence $(J_j, \cdots, J_n)$ such that J_j completes at t. The functions $\bar{\Sigma}_j$ will be jointly used with Σ_j in the next section. The computation of these functions corresponds to the backward recursion : clearly, $\bar{\Sigma}_n(t) = f_n(t)$ for any t. Then, if J_j (with $j < n$) completes at t and J_{j+1} completes at t' with $t' - p_{j+1} \geq t$, the minimum cost to schedule $(J_j, \cdots, J_n)$ is $f_j(t) + \bar{\Sigma}_{j+1}(t')$. So :

$$\bar{\Sigma}_j(t) = f_j(t) + \min_{t' - p_{j+1} \geq t} \bar{\Sigma}_{j+1}(t')$$

Assuming that the cost functions f_j given in input are piecewise linear (and, for the simplicity of the presentation, continuous), the complexity of computing the functions Σ_j and $\bar{\Sigma}_j$ is a function in n and in the number of segments of the functions f_j, denoted by $\|f_j\|$. Indeed, each cost function is fully described by the sorted list of its segments (each segment is represented by the coordinates of its endpoint). For any j, the function Σ_j is continuous and piecewise linear, it can be computed in $O(\sum_{i=1}^{j} \|f_i\|)$ and it has at most $\sum_{i=1}^{j} \|f_i\|$ segments. The reader is refered to the original paper [10] for the proofs and a discussion about discontinuity points.

2.2 Neighborhood computation algorithm

Let us now exactly define the problem. An instance of PLSP is given as well as a sequence σ that corresponds to the sequencing of the so-called *current solution* in the terminology of local search algorithms. Without loss of generality, we can assume that $\sigma = (J_1, \cdots, J_n)$. The problem is to compute the cost of the optimal schedule when we are allowed to swap any two consecutive jobs, that is to find $k \in \{1, \cdots, n-1\}$ such that the cost of the optimal schedule corresponding to the sequence

$$\sigma_k = (J_1, J_2, \cdots, J_{k-1}, J_{k+1}, J_k, J_{k+2}, \cdots, J_n)$$

is minimal. As observed in the introduction, the problem is clearly polynomial, we propose an algorithm that is faster than computing from scratch the optimal cost of the $n-1$ sequences σ_k.

The algorithm has two phases. The first one, called *initialization*, pre-computes some informations that are then used in the second phase to compute the costs of the σ_k sequences. The initialization consists in computing, for any j, the $\Sigma_j(t)$ and $\bar{\Sigma}_j(t)$ functions, which is done with a time and space complexity $O(n(\sum_{i=1}^{n} \|f_i\|))$ by the DP algorithm presented in Section 2.1.

We now show that, for any $k < n$, the cost of the optimal schedule corresponding to the sequence σ_k can be computed in $O(\sum_{i=1}^{n} \|f_i\|)$. We first need the minimum cost to schedule the sequence $J_1, J_2, \cdots, J_{k-1}, J_{k+1}$ when J_{k+1} completes at t. It is given by :

$$g(t) = \min_{u \leq t - p_{k+1}} \Sigma_{k-1}(u) + f_{k+1}(t)$$

$g(t)$ is piecewise linear with a number of segments in $O(\|f_{k+1}\| + \sum_{i=1}^{k-1} \|f_i\|)$.

Symmetrically, the minimum cost to schedule the sequence $J_k, J_{k+2}, J_{k+3}, \cdots, J_n$ when J_k completes at t is given by :

$$h(t) = f_k(t) + \min_{v - p_{k+2} \geq t} \bar{\Sigma}_{k+2}(v)$$

$h(t)$ is piecewise linear with a number of segments in $O(\|f_k\| + \sum_{i=k+2}^{n} \|f_i\|)$.

Since the job J_k must begin after the completion of job J_{k+1}, the minimum cost to schedule σ_k is :

$$\min_u(\Sigma_{\sigma_k}(u)) = \min_{v \geq u + p_k} (g(u) + h(v)) = \min_u(g(u) + \min_{v \geq u + p_k} h(v))$$

We can compute this value in $O(\sum_{i=1}^{n} \|f_i\|)$ time. Indeed, the function g has $O(\|f_{k+1}\| + \sum_{i=1}^{k-1} \|f_i\|)$ segments, while the function $u \mapsto \min_{v \geq u+p_k} h(v)$ has $O(\|f_k\| + \sum_{i=k+2}^{n} \|f_i\|)$ segments [10]. The sum of these two functions is then a piecewise linear function with $O(\sum_{i=1}^{n} \|f_i\|)$ segments. The minimum of this function can be determined in time $O(\sum_{i=1}^{n} \|f_i\|)$.

2.3 Complexity

The neighborhood computation determines the best sequence in the API neighborhood $\{\sigma_1, \cdots, \sigma_{n-1}\}$ by computing the cost associated to each one of these sequences. We have just seen that the cost of any sequence σ_k is computed in $O(\sum_{i=1}^{n} \|f_i\|)$. So, it takes $O(n(\sum_{i=1}^{n} \|f_i\|))$ to find out the best sequence. Since the initialization phase has the same

Problem	PLSP	CPLSP	ETSP
Naive approach	$O(n^2 \sum_{i=1}^{n} \|f_i\|)$	$O(n(\sum_{i=1}^{n} \|f_i\|) \log(\sum_{i=1}^{n} \|f_i\|))$	$O(n^2 \log n)$
Proposed algorithm	$O(n \sum_{i=1}^{n} \|f_i\|)$	$O(n \sum_{i=1}^{n} \|f_i\|)$	$O(n^2)$

Table 2: API neighborhood computation times

complexity, the complexity of the neighborhood computation is in $O(n(\sum_{i=1}^{n} \|f_i\|))$ time (and space).

Table 2 compares the effectiveness of this approach to the naive approach consisting in computing the optimal time for each sequence σ_k. For the three considered problems, the theoretical computational time is improved.

This algorithm can be used in a local search : for every iteration we apply the dynamic programming algorithm on the current sequence to compute the Σ_j and $\bar{\Sigma}_j$ functions, we then find the best permutation in $O(n(\sum_{i=1}^{n} \|f_i\|)))$ and iterate on the sequence associated with this permutation. We notice that an initialization phase must be performed at each iteration.

3 Pairwise interchange

We now consider the SWAP neighborhood, which combines adjacent and non-adjacent pairwise interchanges. However, the proposed algorithm requires that the cost functions f_j are convex so that we only consider the problems CPLSP and ETSP in this section.

3.1 Algorithm overview

The algorithm is based on a data structure. At any time, the data structure represent a sequence of n jobs, this sequence called the *current sequence* of the data structure. Three procedures are associated to this data structure :

- the initialization procedure `init(`σ`)` creates the data structure from an initial sequence σ. Then, σ becomes the current sequence. This procedure is called only once at the beginning of the local search procedure.
- an update procedure `swap(`i`,`j`)` modifies the current sequence by swapping jobs in position i and j.
- a query procedure `getCost()` returns the cost of the optimal timing of the current sequence.

With the last two procedures, the best neighbor in the SWAP neighborhood of any sequence σ can be easily computed by the following piece of code. It is assumed that the current sequence of the data structure is σ.

$best \leftarrow$ `getCost()`
$(k, l) \leftarrow (0, 0)$
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow i + 1$ to n **do**

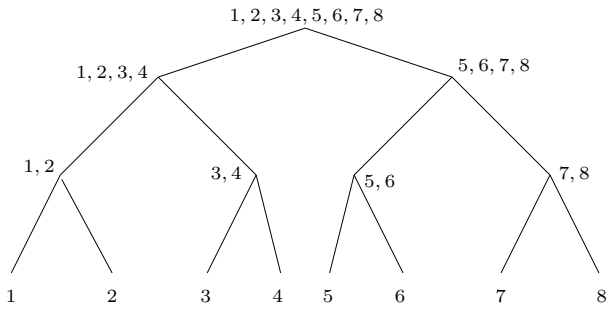

Figure 1: Data tree

```
      swap(i,j)
      if getCost() < best then
        best ← getCost()
        (k,l) ← (i,j)
      end if
      swap(i,j)
    end for
  end for
  if (k,l) ≠ (0,0) then
    swap(k,l)
  end if
```

The second `swap` instruction in the inner loop is used to reset the current sequence of the data structure to σ. At the end of this procedure, the data structure contains the best sequence in the SWAP neighborhood of σ. In view of the complexity of the algorithm, we are going to see that the critical operation is the `swap` procedure, which is called $n(n-1)$ times. A discussion of this algorithm is presented in the appendix.

The next sections present the data structure with its query and update procedures.

3.2 Definition of the data structure

The data structure stores some cost functions of some inner subsequences of the current sequence σ. Let $J_{[\alpha]}$ be the α^{th} job in the sequence σ and let $J_{\alpha\to\beta}$ denote the subsequence going from the α^{th} job to the β^{th} job of σ. We also denote by

- $\Psi_{\alpha\to\beta}(t)$ the cost function of optimally scheduling the subsequence $J_{\alpha\to\beta}$ *before* date t (i.e. with the constraint that the β^{th} job ends before t);
- $\bar{\Psi}_{\alpha\to\beta}(t)$ the cost function of optimally scheduling the subsequence $J_{\alpha\to\beta}$ *after* date t (i.e. with the constraint that the α^{th} job ends after t);
- $\hat{\Psi}_{\alpha\to\beta}$ the cost function of optimally scheduling the subsequence $J_{\alpha\to\beta}$ *as a block* (i.e. with the constraint that all the jobs are executed without idle time) when the β^{th} job completes at t.

These informations are stored in a complete binary tree, each node of the tree representing a given subsequence. In order to have a complete binary tree, if $\log_2 n$ is not an integer, we

add $2^{\lceil \log_2 n \rceil} - n$ imaginary leaves, which have null cost functions. Figure 1 represents such a tree for $n = 8$ jobs with the subsequences stored at each node.

Formally, the tree is defined as follows. Each node is indexed as a pair $\alpha \to \beta$ in the form of $\alpha = a2^{h-b} + 1$ and $\beta = (a+1)2^{h-b}$ where $0 \leq a < b$ and $h = \log_2(n)$ is the depth of the tree. The integer b represents the depth of the node in the tree, the root being at the depth 0. For example, in Figure 1, the node corresponding to the subsequence $(J_{[3]}, J_{[4]})$ is indexed by $3 \to 4$, which corresponds to $a = 1$ and $b = 2$.

The leaves correspond to the nodes such that $\alpha = \beta$. The leaf $\alpha \to \alpha$ corresponds to the job $J_{[\alpha]}$. The following data are associated to this leaf:

- the data about $J_{[\alpha]}$
- $\Psi_{\alpha\to\alpha}(t) = \min_{u \leq t} f_{[\alpha]}(u)$: the cost function to optimally schedule $J_{[\alpha]}$ before date t.
- $\bar{\Psi}_{\alpha\to\alpha}(t) = \min_{u \geq t} f_{[\alpha]}(u)$: the cost function to optimally schedule $J_{[\alpha]}$ after date t.
- $\hat{\Psi}_{\alpha\to\alpha}(t) = f_{[\alpha]}(t)$
- $p_{\alpha\to\alpha} = p_{[\alpha]}$: processing time of $J_{[\alpha]}$

For the inner nodes, we have $\alpha < \beta$. Each inner node $\alpha \to \beta$ has two child nodes, the *left* one $\alpha \to \gamma$ (with $\gamma = \lfloor (\alpha + \beta)/2 \rfloor$) and the *right* one $(\gamma + 1) \to \beta$. Each of these nodes contains the following data:

- the subsequence $J_{\alpha\to\beta}$. This subsequence is formed by the concatenation of the sequence of its left and right children.
- the function $\Psi_{\alpha\to\beta}$
- the function $\bar{\Psi}_{\alpha\to\beta}$
- the function $\hat{\Psi}_{\alpha\to\beta}$
- the value $p_{\alpha\to\beta} = \sum_{\delta=\alpha}^{\beta} p_{[\delta]}$.

We base our analyze of the space complexity of the data structure on the fact that the functions $\Psi_{\alpha\to\beta}$ and $\bar{\Psi}_{\alpha\to\beta}$ have at most $\sum_{l=\alpha}^{\beta} \|f_{[l]}\|$ segments (see Section 2.1). The number of segments of function $\hat{\Psi}_{\alpha\to\beta}$ is clearly subject to the same bound because we can show that a breakpoint of $\hat{\Psi}_{\alpha\to\beta}$ corresponds to at least one breakpoint of an individual cost function. So the size of the node $\alpha \to \beta$ is in $O(\beta - \alpha)$. Since each job is represented in exactly one node at each depth level of the data structure, the size of all the nodes at a given depth level is in $O(\sum_i \|f_i\|)$ and the size of the data structure is $O((\sum_i \|f_i\|) \log n)$.

We also have the following property of the partial cost functions.

Lemma 1. *$\Psi_{\alpha\to\beta}(t)$ (resp. $\bar{\Psi}_{\alpha\to\beta}(t)$) is a non-increasing (resp. non-decreasing) convex function. Moreover there is a time when $\Psi_{\alpha\to\beta}(t)$ (resp. $\bar{\Psi}_{\alpha\to\beta}(t)$) starts (resp. stops) to be constant. Let $t^+_{\alpha\beta}$ (resp. $t^-_{\alpha\beta}$) be the first (resp. the last) t where $\Psi_{\alpha\to\beta}(t)$ (resp. $\bar{\Psi}_{\alpha\to\beta}(t)$) reaches its minimum (see Figure 2).*

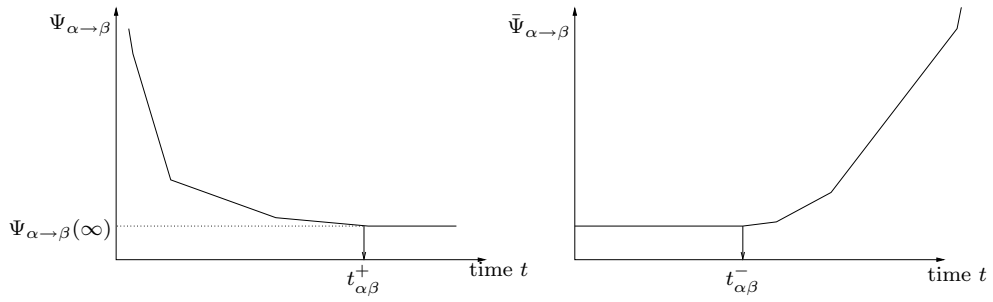

Figure 2: The functions $\Psi_{\alpha\to\beta}$ and $\bar{\Psi}_{\alpha\to\beta}$

Proof. The function $\Psi_{\alpha\to\beta}$ is non-increasing because as t increases, we relax the makespan constraint that $J_{[\beta]}$ must finish before t. Let us consider an optimal schedule of $J_{\alpha\to\beta}$. Let $C_{[\beta]}$ be the makespan of this schedule and let c be its cost. Clearly, for any $t > C_{[\beta]}$, $\Psi_{\alpha\to\beta}(t) = c$.

To prove the convexity of $\Psi_{\alpha\to\beta}$, let us consider two non-negative real values t^1 and t^2, and, for $j \in \{1, 2\}$, we consider the two schedules $C^j_{[\alpha]}, \cdots, C^j_{[\beta]}$ corresponding to $\Psi_{\alpha\to\beta}(t^j)$. For any $x \in [0, 1]$, let $t = xt^1 + (1-x)t^2$ and, for $\alpha \le \delta \le \beta$, let $C_{[\delta]} = xC^1_{[\delta]} + (1-x)C^2_{[\delta]}$. We first observe that, by the convex combinations of the inequalities $C^j_{[\delta+1]} - p_{[\delta+1]} \ge C^j_{[\delta]}$ for $j \in \{1, 2\}$, the so-defined $C_{[\alpha]} \cdots C_{[\beta]}$ schedule is feasible. Since the cost functions f_i are convex, $f_{[\delta]}(C_{[\delta]}) \le xf_{[\delta]}(C^1_{[\delta]}) + (1-x)f_{[\delta]}(C^2_{[\delta]})$. So the cost of the schedule we have build is not greater than $x\Psi_{\alpha\to\beta}(t^1) + (1-x)\Psi_{\alpha\to\beta}(t^2)$ but is greater than $\Psi_{\alpha\to\beta}(t)$. So, $\Psi_{\alpha\to\beta}(t) \le x\Psi_{\alpha\to\beta}(t^1) + (1-x)\Psi_{\alpha\to\beta}(t^2)$ □

We notice that the data structure described in this section immediately provides the minimum cost of the schedule associated to the current sequence. It is given by the value at infinity of the function $\Psi_{1\to n}$ stored at the root. So, the function `getCost()` simply reads the last segment of $\Psi_{1\to n}$ and returns the corresponding value.

3.3 Initialization and update

We present in this section the two other procedures, namely `init` and `update`. Both these procedures uses the procedure `computeNode` which computes the content of a node when the contents of its children nodes are known. First we note that the initialization and the update of the leaves are trivial, so the procedure `computeNode` is only used for the inner nodes.

Since there are n leaves in the complete binary tree described above, there are $n-1$ inner nodes. The contents of these nodes are initialized by calling Algorithm 1: `init`$(1 \to n)$.

Clearly, during the initialization phase, the procedure `computeNode` is called once for the $n-1$ inner nodes of the data structure.

The update procedure `swap`(i, j) first swap the contents of the leaves i and j and then call the recursive procedure `update`$(i, j, 1 \to n)$ that updates the inner nodes.

In this procedure, `computeNode` is called once at each node belonging to the path from the root to the leaf i or the path from the root to the leaf j so that `computeNode` is called at most $2\log_2(n)$ times. Figure 3 represents the nodes of the data structure that have to be updated when the jobs in position 2 and 5 are swapped.

Algorithm 1 `init`$(\alpha \to \beta)$

if $\alpha = \beta$ **then**
  initialize the leaf with $J_{[\alpha]}$
else
  $\gamma = \lfloor (\alpha + \beta)/2 \rfloor$
  `init`$(\alpha \to \gamma)$
  `init`$(\gamma + 1 \to \beta)$
  `computeNode`$(\alpha \to \beta)$
end if

Algorithm 2 `update`$(i, j, \alpha \to \beta)$

if $\alpha \neq \beta$ **then**
  $\gamma = \lfloor (\alpha + \beta)/2 \rfloor$
  if $\min(i, j) \leq \gamma$ **then** `update`$(i, j, \alpha \to \gamma)$
  if $\max(i, j) > \gamma$ **then** `update`$(i, j, \gamma + 1 \to \beta)$
  `computeNode`$(\alpha \to \beta)$
end if

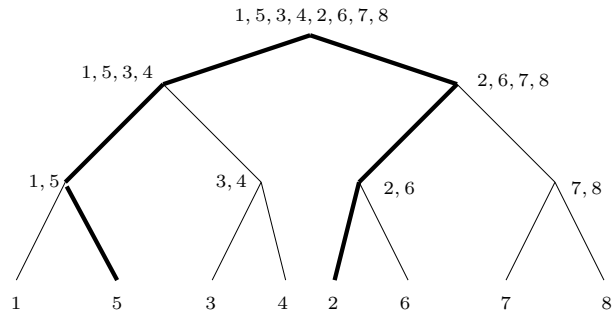

Figure 3: Swapping jobs in position 2 and 5

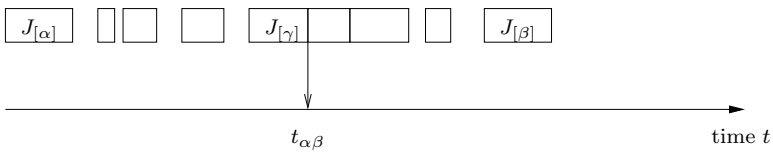

Figure 4: Schedule corresponding to $\Psi_{\alpha\to\beta}(\infty)$

We now show in detail the computation of the contents of the node $\alpha \to \beta$ by the procedure `computeNode` once the left child $\alpha \to \gamma$ and the right child $\gamma+1 \to \beta$ are both up-to-date. We will express the complexity of this procedure in function of the complexity of the two child nodes. When referring to job $J_{[l]}$, we consider the job in position l in the sequence obtained after the swap operation. First of all, we obviously have :

$$\begin{aligned} p_{\alpha\to\beta} &= p_{\alpha\to\gamma} + p_{\gamma+1\to\beta} \\ \hat{\Psi}_{\alpha\to\beta}(t) &= \hat{\Psi}_{\alpha\to\gamma}(t - p_{\gamma+1\to\beta}) + \hat{\Psi}_{\gamma+1\to\beta}(t) \end{aligned}$$

Clearly, $\hat{\Psi}_{\alpha\to\beta}$ has at most $\sum_{l=\alpha}^{\beta} \|f_{[l]}\|$ segments and this function can be computed in $O(\sum_{l=\alpha}^{\beta} \|f_{[l]}\|)$.

We now show how to compute the function $\Psi_{\alpha\to\beta}$. We build $\Psi_{\alpha\to\beta}(t)$ by computing the values of the function as t decreases. For that, we will particularly pay attention to the jobs that are located in the last block in the optimal solution[1] which give $\Psi_{\alpha\to\beta}(t)$. We will say that the jobs located in this block are t-concerned (by the makespan constraint of the schedule). Clearly, the t-concerned jobs form a terminal subsequence of $\alpha \to \beta$.

From the analysis of the algorithm of Garey et al. [5] and its generalization to convex cost functions [2], we know that, for $\epsilon > 0$ small enough, the optimal schedule corresponding to $\Psi_{\alpha\to\beta}(t-\epsilon)$ is derived from the schedule corresponding to $\Psi_{\alpha\to\beta}(t)$ by left-shifting the t-concerned jobs by ϵ and leaving unchanged the start times of the other jobs. We oberve that when t decreases, the number of t-concerned jobs increases when the block of t-concerned jobs "meets" the preceding block so that both blocks are merged.

The $\Psi_{\alpha\to\beta}$ function is going to be built in three steps. We first compute the value of $\Psi_{\alpha\to\beta}(\infty)$ when there is no restriction on the makespan of the schedule. Then, we compute the values when the only jobs which are t-concerned by the makespan of the schedule belong to the right son $\gamma+1 \to \beta$. Finally, when jobs of $\alpha \to \gamma$ are t-concerned.

In order to compute $\Psi_{\alpha\to\beta}(\infty)$, we will use $\Psi_{\alpha\to\gamma}(t)$ which is a decreasing function and $\bar{\Psi}_{\gamma+1\to\beta}(t)$ which is an increasing function. We must add the constraint that the job in position $\gamma+1$ must be processed after the γ^{th} job; then we have to minimize the expression $\Psi_{\alpha\to\gamma}(t) + \bar{\Psi}_{\gamma+1\to\beta}(t + p_{[\gamma+1]})$ when t, representing the completion time of $J_{[\gamma]}$, varies. Since the sum of two piecewise linear convex functions is a piecewise linear function, it can be done in $O(\|\Psi_{\alpha\to\gamma}\| + \|\bar{\Psi}_{\gamma+1\to\beta}\|)$. We denote by $t_{\alpha\beta}$ the earliest time when the minimum is reached (see Figure 4).

We now show that the difference between the functions $\Psi_{\alpha\to\beta}$ and $\Psi_{\gamma+1\to\beta}$ is a constant all over the time interval $[t_{\alpha\beta} + p_{\gamma+1\to\beta}, +\infty)$. Since we know the limits of these two functions at infinity, we will then have on this interval :

$$\Psi_{\alpha\to\beta}(t) = \Psi_{\gamma+1\to\beta}(t) + \Psi_{\alpha\to\beta}(\infty) - \Psi_{\gamma+1\to\beta}(\infty) \quad \forall t > t_{\alpha\beta} + p_{\gamma+1\to\beta} \tag{1}$$

[1] among all the optimal solutions, we always consider the schedule with the earliest start times.

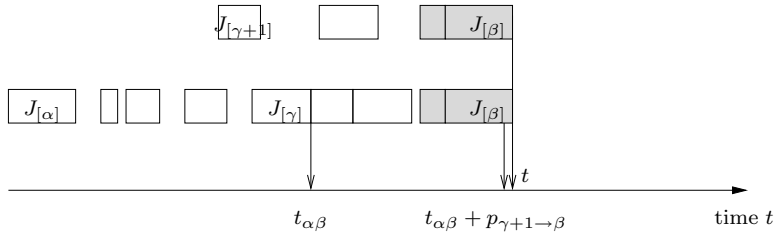

Figure 5: Schedules corresponding to $\Psi_{\gamma+1\to\beta}(t)$ and $\Psi_{\alpha\to\beta}(t)$ for $t > t_{\alpha\beta} + p_{\gamma+1\to\beta}$

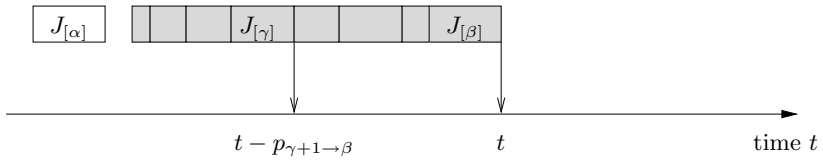

Figure 6: Schedule corresponding to $\Psi_{\gamma+1\to\beta}(t)$ for $t < t_{\alpha\beta} + p_{\gamma+1\to\beta}$

The proof comes from the fact that when t is strictly greater than $t_{\alpha\beta}+p_{\gamma+1\to\beta}$, there must be some idle periods in the interval $[t_{\alpha\beta}, t]$ in the schedule (possibly after $J_{[\beta]}$) which corresponds to $\Psi_{\alpha\to\beta}(t)$ so that $J_{[\gamma]}$ completes at $t_{\alpha\beta}$ in this schedule (see Figure 5). Therefore, the t-concerned jobs are in $J_{\gamma+1\to\beta}$ and are the same than the t-concerned jobs in the schedule corresponding to $\Psi_{\gamma+1\to\beta}(t)$. Since only the costs of the t-concerned jobs vary with t, these two functions vary at the same rate and the difference between them is indeed constant.

When t becomes less than $t_{\alpha\beta} + p_{\gamma+1\to\beta}$, all the jobs of $J_{\gamma+1\to\beta}$ are clearly processed in a single block and that they are all t-concerned. Their cost is then $\hat{\Psi}_{\gamma+1\to\beta}(t)$ (see Figure 6). The jobs of the sequence $J_{\alpha\to\gamma}$ are processed before $t - p_{\gamma+1\to\beta}$, their total cost is then

$$\Psi_{\alpha\to\beta}(t) = \hat{\Psi}_{\gamma+1\to\beta}(t) + \Psi_{\alpha\to\gamma}(t - p_{\gamma+1\to\beta}) \quad \forall t \leq t_{\alpha\beta} + p_{\gamma+1\to\beta} \tag{2}$$

The construction of the function $\Psi_{\alpha\to\beta}$ is then based on (1) and (2). These expressions show the segment of $\Psi_{\alpha\to\beta}$ can be deduced from the contents of the child nodes. The function can be computed in $O(\|\Psi_{\alpha\to\gamma}\| + \|\Psi_{\gamma+1\to\beta}\| + \|\hat{\Psi}_{\gamma+1\to\beta}\|)$, that is in $O(\sum_{l=\alpha}^{\beta} \|f_{[l]}\|)$ time.

The construction of $\bar{\Psi}_{\alpha\to\beta}(t)$ is symmetrical by reversing the time scale. This finally proves that the computation of `computeNode` for the node $\alpha \to \beta$ is done in $O(\sum_{l=\alpha}^{\beta} \|f_{[l]}\|)$ time.

3.4 Complexity

In order to compute the worst-case complexity of the `init` and `swap` procedures, we assume in this section that all the $\|f_i\|$ values are bounded by a constant K. For ETSP, we have $K = 2$. Therefore, the number of segments of each function $\bar{\Psi}_{\alpha\to\beta}$, $\hat{\Psi}_{\alpha\to\beta}$ or $\Psi_{\alpha\to\beta}$ is bounded by $K(\beta - \alpha + 1)$ so that `computeNode` for the node $\alpha \to \beta$ runs in $O(\beta - \alpha)$.

We first show that `init` runs in $O(n \log n)$ time. In the data structure, there are 2^b nodes located at depth b of the binary tree. The computation time of such a node is in $O(2^{h-b})$, so that computing all the nodes at depth b takes $O(2^{b+h-b})$, that is $O(n)$ time. As $h = \lceil \log_2 n \rceil$,

the time complexity of `init` is in $O(n \log n)$. Clearly, the initialization procedure provides us a new algorithm to compute the optimum cost for timing a sequence. The complexity of this algorithm is equivalent to the complexity of Garey et al. [5].

The procedure `swap` runs in $O(n)$ time : as at most 2 nodes at each depth of the data structure calls `computeNode`, the total complexity is in $O(\sum_{b=0}^{h} 2^{h-b})$ that is $O(n)$.

Since the size of the SWAP neighborhood is $n(n-1)/2$, the complexity of the neighborhood computation is in $O(n^3)$. By applying the algorithm of Garey *et al.* [5] for each sequence in the neighborhood, we have a complexity in $O(n^3 \log n)$. As for the API neighborhood, we have improved the theoretical complexity by a non-constant factor.

We notice that unlike the neighborhood computing method presented in the previous section, the data structure method is entirely updated after each permutation. There is no need to reinitialize when a new current solution is selected and that we thereby change our neighborhood.

4 Computational experiments

Both our algorithms were implemented in Java with Borland JBuilder on a 2GHz PC. We tested the algorithm on randomly generated instances. We have only tested the algorithms for ETSP because, as mentioned in Section 1, it is the most studied case in the literature and the generation of test instances is "standard".

4.1 Problem instance generation

The processing time were uniformly distributed in $\{1, 2, \cdots, 100\}$. The earliness α_j and tardiness β_j factors of a job J_j are uniformly distributed between 1 and 5. The due dates were uniformly distributed between 1 and $5 * PT * \rho$ where PT is the total processing time of all the jobs and ρ is the *range factor* of the due dates which took the values from 0.1, 0.5, 1, 5, 10. Table 3 compare our algorithm for the API neighborhood to the naive method consisting of computing for every neighbor the dynamic programming algorithm of Sourd (we used this algorithm because, in [10], it was shown to be faster than the algorithm of Garey et al.). Table 4 is dedicated to the SWAP neighborhood. For every pair of parameters (n, ρ), five instances were generated. The table presents the minimum, maximum and average CPU times to solve the five instances with either the naive method or our approach. The execution times are given in milliseconds.

4.2 Computational results

We observe a consequent improvement in our computations. For the first neighborhood, we have tested the two algorithms for a large number of jobs: with the naive method, the whole neighborhood is computed in 0.2 seconds for 100 jobs but in 37 seconds for 500 jobs whereas our algorithm provides results in less than a 0.4 seconds for 500 jobs. For the second neighborhood, we only observe a good improvement starting for 100 jobs: the speed ratio between the two algorithms is roughly a factor of 6.

Number of jobs	ρ	Naive approach (msecs) min	max	avg	Our approach (msecs) min	max	avg
100	0.1	231	270	244	10	20	14
100	0.5	230	250	236	10	10	10
100	1.0	230	240	234	10	20	14
100	5.0	230	241	238	10	20	12
100	10.0	230	241	238	10	20	12
200	0.1	3124	3174	3146	50	61	56
200	0.5	2663	2674	2669	40	61	56
200	1.0	2624	2684	2641	40	61	56
200	5.0	2653	2955	2742	40	60	54
200	10.0	2643	2664	2653	41	60	56
300	0.1	10315	10415	10355	110	120	118
300	0.5	7581	7631	7597	90	121	110
300	1.0	7631	8111	7756	100	121	116
300	5.0	7651	7711	7669	90	120	110
300	10.0	7641	7661	7651	90	120	108
400	0.1	20209	20379	20267	190	210	196
400	0.5	16224	16674	16327	170	200	188
400	1.0	15923	15993	15955	170	201	190
400	5.0	16293	16354	16307	180	201	194
400	10.0	16274	16343	16299	180	211	200
500	0.1	34069	34179	34141	281	290	286
500	0.5	29743	29813	29777	260	291	280
500	1.0	29893	36452	33560	330	351	346
500	5.0	36382	38445	37099	331	361	352
500	10.0	36362	37874	37045	330	401	374

Table 3: Results for the API neighborhood

Number of jobs	ρ	Naive approach (msecs)			Our approach (msecs)		
		min	max	avg	min	max	avg
40	0.1	351	401	372	390	451	406
40	0.5	360	371	368	391	401	398
40	1.0	370	430	382	401	451	428
40	5.0	351	370	360	410	411	410
40	10.0	360	370	362	400	421	410
80	0.1	5408	5418	5413	2864	2915	2878
80	0.5	5638	5659	5642	2954	2964	2960
80	1.0	5648	5689	5660	2984	2995	2990
80	5.0	5477	5498	5489	2974	3025	2990
80	10.0	5618	5638	5626	2974	2985	2982
120	0.1	37294	37474	37355	8823	8833	8826
120	0.5	37684	37744	37714	9033	9094	9055
120	1.0	38345	38415	38375	9213	9264	9225
120	5.0	38465	38916	38605	9003	9063	9017
120	10.0	37774	38866	38098	9013	9074	9035
160	0.1	155243	165178	157282	27649	30895	29139
160	0.5	151808	157397	154850	27359	27400	27371
160	1.0	149676	150806	150222	27800	28491	28088
160	5.0	155083	228309	196278	27049	27379	27139
160	10.0	149035	150115	149380	26889	27400	27080
200	0.1	395349	515351	425005	54058	76881	62928
200	0.5	383652	398132	389524	55470	55810	55660
200	1.0	359807	394978	374554	55549	68920	59806
200	5.0	370012	387557	375143	54378	55280	54861
200	10.0	370803	372205	371316	54639	55650	55181

Table 4: Results for the SWAP neighborhood

5 Conclusion

We have presented two original approaches to explore the interchanged-based neighborhoods for the one-machine problem with non-regular cost functions. We have show that our algorithms improve the theoretical complexity of this computation.

In our experimental tests, we also observed a significant improvement in computation times. Future works could consider for even larger neighborhoods based on the dynasearch neighborhood [3, 6]. The generalization of these methods for both parallel machines and shop problems is of practical and theoretical interest too.

A Exploring the SWAP neighborhood

The algorithm presented in Section 3.1 shows that the schedules of the $n(n-1)/2$ sequences of the SWAP neighborhood can be computed with $n(n-1)$ calls to the `swap` procedure of the data structure. A natural is to know whether the neighborhood could be computed with less calls to the procedure (typically $n(n-1)/2 + o(n)$ calls). In fact, all the permutation in the SWAP neighborhood of a sequence σ have the same sign as σ. As `swap` corresponds to a transposition, each call to the procedure inverses the sign of the current sequence, which shows that $n(n-1)/2$ calls to `swap` are necessary to explore SWAP. In this view, the algorithm proposed in Section 3.1 is optimal.

However, we observe that this procedure compute $n(n-1)/2$ times the cost of the sequence σ to reset the data structure, which can be considered as "wasted" CPU time. In a practical view, it is better to execute a sequence of $n(n-1)$ `swap` operations that produces all the sequences in the SWAP neighborhood but also other sequences. We propose the following algorithm :

for $i \leftarrow 1$ to n **do**
  `swap`$(i, i+1)$
  for $j \leftarrow i+1$ to $n-1$ **do**
    `swap`$(j, j+1)$
    `swap`(i, j)
  end for
  `swap`(i, n)
end for

It can be observed that this algorithm re-generates the initial sequence σ exactly n times (after `swap`(i, n)) so that $n(n-2)$ different sequences are tested in the neighborhood computation.

Finally, we remark that the neighborhood *extraction and reinsertion* (ER) can also be explored by a list of `swap` operations. This neighborhood consists in extracting a job and reinserting it somewhere else in the sequence. The sub-neighborhood EFSR (extraction and forward-shifted reinsertion) consists in reinserting the job after its initial position. The complementary of EFSR in ER is EBSR (extraction and backward-shifted reinsertion). We give an algorithm for exploring EFSR with a list of `swap` calls.

for $i \leftarrow 1$ to $\lfloor n/2 \rfloor$ **do**
  for $j \leftarrow 2i$ to n **do**
    `swap`$(j-1, j)$
  end for
  `swap`$(2i-1, n)$

for $j \leftarrow n$ downto $2i+1$ **do**
`swap`$(j-1, j)$
end for
end for

In this algorithm, the original sequence is re-computed only $\lfloor n/2 \rfloor$ times. All the neighbors in EFSR are generated once. EBSR is generated symmetrically.

References

[1] R.K. Ahuja, Ö. Ergun, J.B. Orlin and A.P. Punnen (2002). A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics* **123**, 75 – 102.

[2] Ph. Chrétienne and F. Sourd (2003). PERT scheduling with convex cost functions. *Theoretical Computer Science* **292**, 145 – 164.

[3] R.K. Congram, C.N. Potts and S.L. van de Velde (2002). An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. *INFORMS Journal on Computing* **14**, 52 – 67.

[4] J.S. Davis and J.J. Kanet (1993). Single-machine scheduling with early and tardy completion costs. *Naval Research Logistics* **40**, 85 – 101.

[5] M.R. Garey, R.E. Tarjan and G.T. Wilfong (1988). One processor scheduling with symmetric earliness and tardiness penalties. *Mathematics of Operations Research* **13**, 330 – 348.

[6] A. Grosso, F. Della Croce and R. Tadei (2004). An enhanced dynasearch neighborhood for the single-machine total weighted tardiness scheduling problem. *Operations Research Letters* **32**, 68 – 72.

[7] N.G. Hall, W. Kubiak and S.P. Sethi (1991). Earliness-tardiness scheduling problems, II: deviation of completion times about a restrictive common due date. *Operations Research* **39**, 847 – 856.

[8] RJW. James and JT. Buchanan (1997). A neighborhood scheme with a compressed solution space for the early/tardy scheduling problem. *European Journal of Operational Research* **102**, 513 – 527.

[9] CY. Lee and JY. Choy (1995). A genetic algorithm for jobs sequencing with distinct due dates and general early-tardy penalty weights. *Computers & Operations Research* **22**, 857 – 869

[10] F. Sourd (2002). Scheduling a sequence of tasks with general completion costs. *Technical Report* LIP6 2002/013. Revision submitted to the *European Journal of Operational Research*.

[11] W. Szwarc and SK. Mukhopadhyay (1995). Optimal timing schedules in earliness-tardiness single machine sequencing. *Naval Research Logistics* **42**, 1109 – 1114.

[12] G. Wan and B.P.-C. Yen (2002). Tabu search for single machine with distinct due windows and weighted earliness/tardiness penalties. *European Journal of Operational Research* **142**, 271 – 281.