HAL open science

An Exact Exponential Branch-and-Merge Algorithm for the Single Machine Total Tardiness Problem

Michele Garraffa, Lei Shang, Federico Della Croce, Vincent t'Kindt

▶ To cite this version:

Michele Garraffa, Lei Shang, Federico Della Croce, Vincent t'Kindt. An Exact Exponential Branch-and-Merge Algorithm for the Single Machine Total Tardiness Problem. 2017. hal-01477835

HAL Id: hal-01477835

https://hal.science/hal-01477835

Preprint submitted on 27 Feb 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Noname manuscript No.
(will be inserted by the editor)

An Exact Exponential Branch-and-Merge Algorithm for the Single Machine Total Tardiness Problem

Michele Garraffa · Lei Shang · Federico Della Croce · Vincent T'kindt

the date of receipt and acceptance should be inserted later

Abstract This paper proposes an exact exponential algorithm for the single machine total tardiness problem. It exploits the structure of a basic branch-and-reduce framework based on the well known Lawler's decomposition property that solves the problem with worst-case complexity $\mathcal{O}^*(3^n)$ in time and polynomial space. The proposed algorithm, called branch-and-merge, is an improvement of the branch-and-reduce technique with the embedding of a node merging operation. Its time complexity converges to $\mathcal{O}^*(2^n)$ keeping the space complexity polynomial. This improves upon the best-known complexity result for this problem provided by dynamic programming across the subsets with $\mathcal{O}^*(2^n)$ worst-case time and space complexity. The branch-and-merge technique is likely to be generalized to other sequencing problems with similar decomposition properties.

Keywords Exact exponential algorithm · Single machine total tardiness · Branch and merge

1 Introduction

Since the beginning of this century, the design of exact exponential algorithms for NP-hard problems has been attracting more and more researchers. Although the research in this area dates back to early 60s, the discovery of new design and analysis techniques has led to many new developments. The main motivation behind the rise of interest in this area is the study of the intrinsic

M. Garraffa, F. Della Croce
Politecnico di Torino, DAUIN, Corso Duca degli Abruzzi 24, 10129 Torino, Italy
E-mail: michele.garraffa, federico.dellacroce@polito.it

L. Shang, V. T'Kindt
Université François-Rabelais de Tours, Laboratoire d'Informatique (EA 6300), ERL CNRS OC 6305, 64 avenue Jean Portalis, 37200 Tours, France
E-mail: shang, tkindt@univ-tours.fr

complexity of NP-hard problems. In fact, since the dawn of computer science, some of these problems appeared to be solvable with a lower exponential complexity than others belonging to the same complexity class. For a survey on the most effective techniques in designing exact exponential algorithms, readers are kindly referred to Woeginger's paper [19] and to the book by Fomin et al. [2].

In spite of the growing interest on exact exponential algorithms, few results are yet known on scheduling problems, see the survey of Lenté et al. [11]. In [10], Lenté et al. introduced the so-called class of multiple constraint problems and showed that all problems fitting into that class could be tackled by means of the Sort & Search technique. Further, they showed that several known scheduling problems are part of that class. However, all these problems required assignment decisions only and none of them required the solution of a sequencing problem.

This paper focuses on a pure sequencing problem, the single machine total tardiness problem, denoted by $1||\sum T_j$. In this problem, a jobset $N = \{1, 2, \ldots, n\}$ of n jobs must be scheduled on a single machine. For each job j, a processing time p_j and a due date d_j are defined. The problem asks for arranging the jobset in a sequence $S = (1, 2, \ldots, n)$ so as to minimize $T(N, S) = \sum_{j=1}^{n} T_j = \sum_{j=1}^{n} \max\{C_j - d_j, 0\}$, where $C_j = \sum_{i=1}^{j} p_i$. The $1||\sum T_j$ problem is NP-hard in the ordinary sense [4]. It has been extensively studied in the literature and many exact procedures ([3,9,12,15]) have been proposed. The current state-of-the-art exact method of [15] dates back to 2001 and solves to optimality problems with up to 500 jobs. All these procedures are search tree approaches, but dynamic programming algorithms were also considered. On the one hand, in [9] a pseudo-polynomial dynamic programming algorithm was proposed running with complexity $\mathcal{O}(n^4 \sum p_i)$. On the other hand, the standard technique of doing dynamic programming across the subsets (see, for instance, [2]) applies and runs with complexity $\mathcal{O}(n^2 2^n)$ both in time and in space. Latest theoretical developments for the problem, including both exact and heuristic approaches can be found in the recent survey of Koulamas [8].

In the rest of the paper, the $\mathcal{O}^*(\cdot)$ notation [19], commonly used in the context of exact exponential algorithms, is used to measure worst-case complexities. Let $T(\cdot)$ be a super-polynomial and $p(\cdot)$ be a polynomial, both on integers. In what follows, for an integer n, we express running-time bounds of the form $\mathcal{O}(p(n) \cdot T(n)))$ as $\mathcal{O}^*(T(n))$. We denote by $T(n)$ the time required in the worst-case to exactly solve the considered combinatorial optimization problem of size n, i.e. the number of jobs in our context. As an example, the complexity of dynamic programming across the subsets for the total tardiness problem can be expressed as $\mathcal{O}^*(2^n)$.

To the authors' knowledge, there is no available exact algorithm for this problem running in $\mathcal{O}^*(c^n)$ (c being a constant) and polynomial space. Admittedly, one could possibly apply a divide-and-conquer approach as in [7] and [1]. This would lead to an $\mathcal{O}^*(4^n)$ complexity in time requiring polynomial space. Aim of this work is to present an improved exact algorithm exploiting known decomposition properties of the problem. Different versions of the proposed approach are described in Section 2. A final version making use of a new technique called branch-and-merge that avoids the solution of several equivalent subproblems in the branching tree is presented in Section 3. This version is shown to have a complexity that tends to $\mathcal{O}^*(2^n)$ in time and requires polynomial space. Finally, Section 4 concludes the paper with final remarks.

2 A Branch-and-Reduce approach

We recall here some basic properties of the total tardiness problem and introduces the notation used along the paper. Given the jobset $N = \{1, 2, \ldots, n\}$, let $(1, 2, \ldots, n)$ be a LPT (Longest Processing Time first) sequence, where $i < j$ whenever $p_i > p_j$ (or $p_i = p_j$ and $d_i \leq d_j$). Let also $([1], [2], \ldots, [n])$ be an EDD (Earliest Due Date first) sequence, where $i < j$ whenever $d_{[i]} < d_{[j]}$ (or $d_{[i]} = d_{[j]}$ and $p_{[i]} \leq p_{[j]}$). As the cost function is a regular performance measure, we know that in an optimal solution, the jobs are processed with no interruption starting from time zero. Let B_j and A_j be the sets of jobs that precede and follow job j in an optimal sequence. Correspondingly, $C_j = \sum_{k \in B_j} p_k + p_j$. Similarly, if job j is assigned to position k, we denote by $C_j(k)$ the corresponding completion time and by $B_j(k)$ and $A_j(k)$ the sets of predecessors and successors of j, respectively.

The main known theoretical properties are the following.

Property 1. *[5] Consider two jobs i and j with $p_i < p_j$. Then, i precedes j in an optimal schedule if $d_i \leq \max\{d_j, C_j\}$, else j precedes i in an optimal schedule if $d_i + p_i > C_j$.*

Property 2. *[9] Let job 1 in LPT order correspond to job $[k]$ in EDD order. Then, job 1 can be set only in positions $h \geq k$ and the jobs preceding and following job 1 are uniquely determined as $B_1(h) = \{[1], [2], \ldots, [k-1], [k+1], \ldots, [h]\}$ and $A_1(h) = \{[h+1], \ldots, [n]\}$.*

Property 3. *[9,12,14] Consider $C_1(h)$ for $h \geq k$. Job 1 cannot be set in position $h \geq k$ if:*

(a) $C_1(h) \geq d_{[h+1]}$, $h < n$;
(b) $C_1(h) < d_{[r]} + p_{[r]}$, for some $r = k+1, \ldots, h$.

Property 4. *([16]) For any pair of adjacent positions $(i, i+1)$ that can be assigned to job 1, at least one of them is eliminated by Property 3.*

In terms of complexity analysis, we recall (see, for instance, [6]) that, if it is possible to bound above $T(n)$ by a recurrence expression of the type $T(n) \leq \sum_{i=1}^{h} T(n-r_i) + \mathcal{O}(p(n))$, then we have $\sum_{i=1}^{h} T(n-r_i) + \mathcal{O}(p(n)) = \mathcal{O}^*(\alpha(r_1, \ldots, r_h)^n)$ where $\alpha(r_1, \ldots, r_h)$ is the largest root of the function $f(x) = 1 - \sum_{i=1}^{h} x^{-r_i}$.

A basic branch-and-reduce algorithm TTBR1 (Total Tardiness Branch-and-Reduce version 1) can be designed by exploiting Property 2, which allows to decompose the problem into two smaller subproblems when the position of the longest job 1 is given. The basic idea is to iteratively branch by assigning job 1 to every possible position $(1, ..., n)$ and correspondingly decompose the problem. Each time job 1 is assigned to a certain position i, two different subproblems are generated, corresponding to schedule the jobs before l (inducing subproblem $B_l(i)$) or after l (inducing subproblem $A_l(i)$), respectively. The algorithm operates by applying to any given jobset S starting at time t function $TTBR1(S, t)$ that computes the corresponding optimal solution. With this notation, the original problem is indicated by $N = \{1, ..., n\}$ and the optimal solution is reached when function $TTBR1(N, 0)$ is computed.

The algorithm proceeds by solving the subproblems along the branching tree according to a depth-first strategy and runs until all the leaves of the search tree have been reached. Finally, it provides the best solution found as an output. Algorithm 1 summarizes the structure of this approach, while Proposition 1 states its worst-case complexity.

Algorithm 1 Total Tardiness Branch-and-Reduce version 1 (TTBR1)

Input: $N = \{1, ..., n\}$ is the problem to be solved
function TTBR1(S, t)
  seqOpt $\leftarrow$ a random sequence of jobs
  $l \leftarrow$ the longest job in N
  for $i = 1$ to n **do**
    Branch by assigning job l to position i
    $seqLeft \leftarrow$ TTBR1($B_l(i), t$)
    seqRight $\leftarrow$ TTBR1($A_l(i), t + \sum_{k \in B_l(i)} p_k + p_l$)
    seqCurrent $\leftarrow$ concatenation of $seqLeft$, l and *seqRight*
    seqOpt $\leftarrow$ best solution between *seqOpt* and *seqCurrent*
  end for
  return *seqOpt*
end function

Proposition 1. *Algorithm TTBR1 runs in $\mathcal{O}^*(3^n)$ time and polynomial space in the worst case.*

Proof. Whenever the longest job 1 is assigned to the first and the last position of the sequence, two subproblems of size $n - 1$ are generated. For each $2 \leq i \leq n - 1$, two subproblems with size $i - 1$ and $n - i$ are generated. Hence, the total number of generated subproblems is $2n - 2$ and the time cost related to computing the best solution of size n starting from these subproblems is $\mathcal{O}(p(n))$. This induces the following recurrence for the running time $T(n)$ required by TTBR1:

$$T(n) = 2T(n-1) + 2T(n-2) + ... + 2T(2) + 2T(1) + \mathcal{O}(p(n)) \quad (1)$$

By replacing n with $n - 1$, the following expression is derived:

$$T(n-1) = 2T(n-2) + ... + 2T(2) + 2T(1) + \mathcal{O}(p(n-1)) \quad (2)$$

Expression 2 can be used to simplify the right hand side of expression 1 leading to:

$$T(n) = 3T(n-1) + \mathcal{O}(p(n)) \tag{3}$$

that induces as complexity $\mathcal{O}^*(3^n)$. The space requirement is polynomial since the branching tree is explored according to a depth-first strategy. □

An improved version of the algorithm is defined by taking into account Property 3 and Property 4, which state that for each pair of adjacent positions $(i, i+1)$, at least one of them can be discarded. The worst case occurs when the largest possible subproblems are kept. This corresponds to solving problems with size $n-1, n-3, n-5, \ldots$, that arise by branching on positions i and $n-i+1$ with i odd. The resulting algorithm is referred to as TTBR2 (Total Tardiness Branch and Reduce version 2). Its structure is equal to the one of TTBR1 depicted in Algorithm 1, but lines 5-9 are executed only when l can be set on position i according to Property 3. The complexity of the algorithm is discussed in Proposition 2.

Proposition 2. *Algorithm TTBR2 runs in $\mathcal{O}^*((1+\sqrt{2})^n) = \mathcal{O}^*(2.4143^n)$ time and polynomial space in the worst case.*

Proof. The proof is close to that of Proposition 1. We refer to problems where n is odd, but the analysis for n even is substantially the same. The algorithm induces a recursion of the type:

$$T(n) = 2T(n-1) + 2T(n-3) + ... + 2T(4) + 2T(2) + \mathcal{O}(p(n)) \tag{4}$$

as the worst case occurs when we keep the branches that induce the largest possible subproblems. Analogously to Proposition 1, we replace n with $n-2$ in the previous recurrence and we obtain:

$$T(n-2) = 2T(n-3) + 2T(n-5) + ... + 2T(4) + 2T(2) + \mathcal{O}(p(n-2)) \tag{5}$$

Again, we plug the latter expression into the former one and obtain the recurrence:

$$T(n) = 2T(n-1) + T(n-2) + \mathcal{O}(p(n)) \tag{6}$$

that induces as complexity $\mathcal{O}^*((1+\sqrt{2})^n) = \mathcal{O}^*(2.4143^n)$. The space complexity is still polynomial. □

3 A Branch-and-Merge Algorithm

In this section, we describe how to get an algorithm running with complexity arbitrarily close to $\mathcal{O}^*(2^n)$ in time and polynomial space by integrating a node-merging procedure into TTBR1. We recall that in TTBR1 the branching scheme is defined by assigning the longest unscheduled job to each available position and accordingly divide the problem into two subproblems. To facilitate the description of the algorithm, we focus on the scenario where the LPT sequence $(1, ..., n)$ coincides with the EDD sequence $([1], ..., [n])$, for convenience we write $LPT = EDD$. The extension of the algorithm to the case $LPT \neq EDD$ will be presented at the end of the section.

Figures 1 shows how an input problem $\{1, ..., n\}$ is decomposed by the branching scheme of TTBR1. Each node is labelled by the corresponding subproblem P_j (P denotes the input problem). Notice that from now on $P_{j_1,j_2,...,j_k}, 1 \leq k \leq n$, denotes the problem (node in the search tree) induced by the branching scheme of TTBR1 when the largest processing time job 1 is in position j_1, the second largest processing time job 2 is in position j_2 and so on till the k-th largest processing time job k being placed in position j_k.

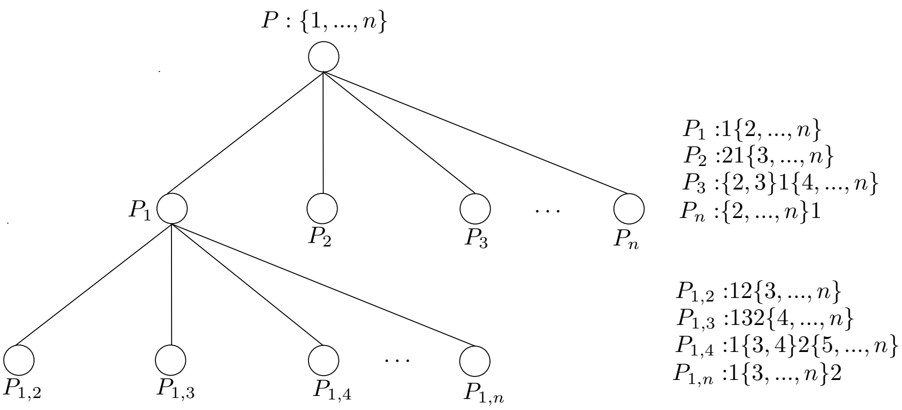


Fig. 1: The branching scheme of TTBR1 at the root node

To roughly illustrate the guiding idea of the merging technique introduced in this section, consider Figures 1. Noteworthy, nodes P_2 and $P_{1,2}$ are identical except for the initial subsequence (21 vs 12). This fact implies, in this particular case, that the problem of scheduling jobset $\{3, ..., n\}$ at time $p_1 + p_2$ is solved twice. This kind of redundancy can however be eliminated by merging node P_2 with node $P_{1,2}$ and creating a single node in which the best sequence among 21 and 12 is scheduled at the beginning and the jobset $\{3, ..., n\}$, starting at time $p_1 + p_2$, remains to be branched on. Furthermore, the best subsequence (starting at time $t = 0$) between 21 and 12 can be computed in constant time. Hence, the node created after the merging operation involves a constant time preprocessing step plus the search for the optimal solution of jobset $\{3, ..., n\}$ to be processed starting at time $p_1 + p_2$. We remark that, in the branching scheme of TTBR1, for any constant $k \geq 3$, the branches corresponding to P_i and P_{n-i+1}, with $i = 2, ..., k$, are decomposed into two problems where one subproblem has size $n - i$ and the other problem has size $i - 1 \leq k$. Correspondingly, the merging technique presented on problems P_2 and $P_{1,2}$ can be generalized to all branches inducing problems of sizes less than k. Notice that, by means of algorithm TTBR2, any problem of size less than k requires at most $\mathcal{O}^*(2.4143^k)$ time (that is constant time when k is fixed). In the remainder of the paper, for any constant $k \leq \frac{n}{2}$, we denote by left-side branches the search tree branches corresponding to problems $P_1, ..., P_k$ and by right-side branches the ones corresponding to problems $P_{n-k+1}, ..., P_n$.

In the following subsections, we show how the node-merging procedure can be systematically performed to improve the time complexity of TTBR1. Basically, two different recurrent structures hold respectively for left-side and right-side branches and allow to generate fewer subproblems at each recursion level. The node-merging mechanism is described by means of two distinct procedures, called `LEFT_MERGE` (applied to left-side branches) and `RIGHT_MERGE` (applied to right-side branches), which are discussed in Sections 3.1 and 3.2, respectively. The final branch-and-merge algorithm is described in Section 3.3 and embeds both procedures in the structure of TTBR1.

3.1 Merging left-side branches

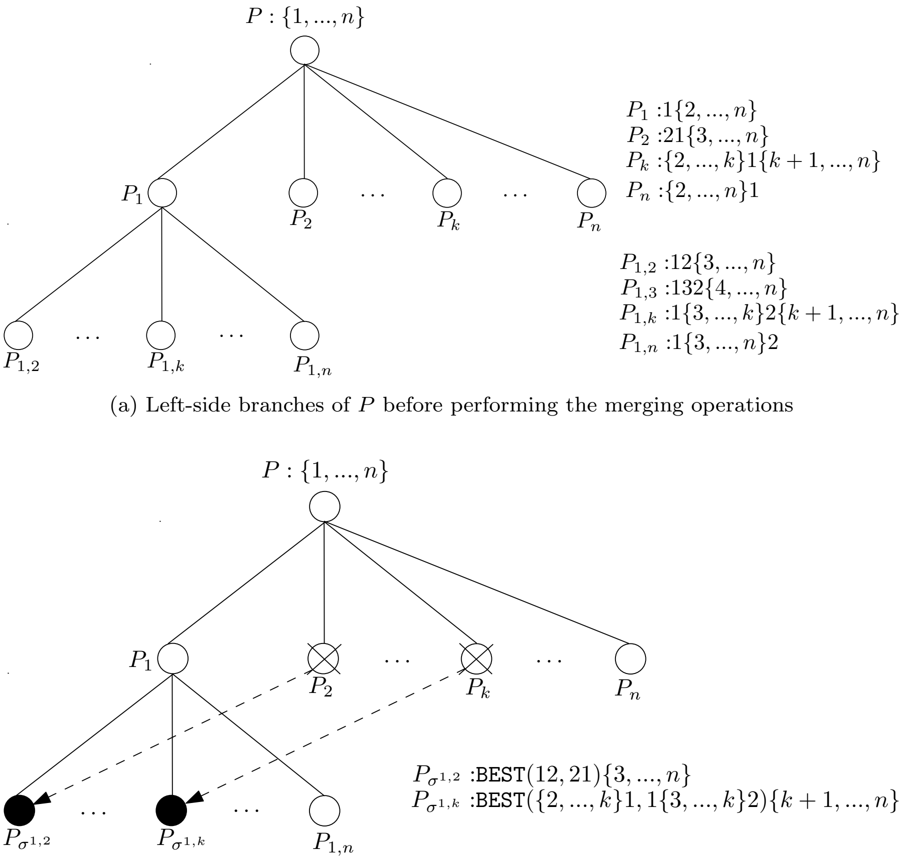
The first part of the section aims at illustrating the merging operations on the root node. The following proposition highlights two properties of the couples of problems P_j and $P_{1,j}$ with $2 \leq j \leq k$.

Lemma 1 *For a couple of problems P_j and $P_{1,j}$ with $2 \leq j \leq k$, the following conditions hold:*

1. *The solution of problems P_j and $P_{1,j}$ involves the solution of a common subproblem which consists in scheduling jobset $\{j+1, ..., n\}$ starting at time $t = \sum_{i=1,...,j} p_i$.*
2. *Both in P_j and $P_{1,j}$, at most k jobs have to be scheduled before jobset $\{j + 1, ..., n\}$.*

Proof. As problems P_j and $P_{1,j}$ are respectively defined by $\{2, ..., j\}1\{j + 1, ..., n\}$ and $1\{3, ..., j\}2\{j + 1, ..., n\}$, the first part of the property is straightforward.
The second part can be simply established by counting the number of jobs to be scheduled before jobset $\{j + 1, ..., n\}$ when j is maximal, *i.e.* when $j = k$. In this case, jobset $\{k + 1, ..., n\}$ has $(n - k)$ jobs which implies that k jobs remain to be scheduled before that jobset. □

Each couple of problems indicated in Proposition 1 can be merged as soon as they share the same subproblem to be solved. More precisely, $(k - 1)$ problems P_j (with $2 \leq j \leq k$) can be merged with the corresponding problems $P_{1,j}$.



(a) Left-side branches of P before performing the merging operations

(b) Left-side branches of P after performing the merging operations

Fig. 2: Left-side branches merging at the root node

Figure 2 illustrates the merging operations performed at the root node on its left-side branches, by showing the branch tree before and after (Figure 2a and Figure 2b) such merging operations. For any given $2 \leq j \leq k$, problems P_j and $P_{1,j}$ share the same subproblem $\{j+1, ..., n\}$ starting at time $t = \sum_{i=1}^{j} p_i$. Hence, by merging the left part of both problems which is constituted by jobset $\{1, ..., j\}$ having size $j \leq k$, we can delete node P_j and replace node $P_{1,j}$ in the search tree by the node $P_{\sigma^{1,j}}$ which is defined as follows (Figure 2b):

- Jobset $\{j+1, ..., n\}$ is the set of jobs on which it remains to branch.
- Let $\sigma^{1,j}$ be the sequence of branching positions on which the j longest jobs $1, ..., j$ are branched, that leads to the best jobs permutation between

$\{2, ..., j\}1$ and $1\{3, ..., j\}2$. This involves the solution of two problems of size at most $k-1$ (in $\mathcal{O}^*(2.4143^k)$ time by TTBR2) and the comparison of the total tardiness value of the two sequences obtained.

In the following, we describe how to apply analogous merging operations on any node of the tree. With respect to the root node, the only additional consideration is that the children nodes of a generic node may have already been affected by a previous merging.

In order to define the branching scheme used with the `LEFT_MERGE` procedure, a data structure $\mathcal{L}_\sigma$ is associated to a problem P_σ. It represents a list of $k-1$ subproblems that result from a previous merging and are now the first $k-1$ children nodes of P_σ. When P_σ is created by branching, $\mathcal{L}_\sigma = \emptyset$. When a merging operation sets the first $k-1$ children nodes of P_σ to $P_{\sigma^1}, ..., P_{\sigma^{k-1}}$, we set $\mathcal{L}_\sigma = \{P_{\sigma^1}, ..., P_{\sigma^{k-1}}\}$. As a conclusion, the following branching scheme for a generic node of the tree holds.

Definition 1 The branching scheme for a generic node P_σ is defined as follows:

- If $\mathcal{L}_\sigma = \emptyset$, use the branching scheme of TTBR1;
- If $\mathcal{L}_\sigma \neq \emptyset$, extract problems from $\mathcal{L}_\sigma$ as the first $k-1$ branches, then branch on the longest job in the available positions from the k-th to the last according to Property 2.

This branching scheme, whenever necessary, will be referred to as **improved branching**.

Before describing how merging operations can be applied on a generic node P_σ, we highlight its structural properties by means of Proposition 3.

Proposition 3. *Let P_σ be a problem to branch on, and σ be the permutation of positions assigned to jobs $1, \ldots, |\sigma|$, with σ empty if no positions are assigned. The following properties hold:*

1. *$j^* = |\sigma| + 1$ is the job to branch on,*
2. *j^* can occupy in the branching process, positions $\{\ell_b, \ell_b + 1, \ldots, \ell_e\}$, where*

$$\ell_b = \begin{cases} |\sigma| + 1 & \textit{if } \sigma \textit{ is a permutation of } 1, \ldots, |\sigma| \textit{ or } \sigma \textit{ is empty} \\ \rho_1 + 1 & \textit{otherwise} \end{cases}$$

with $\rho_1 = \max\{i : i > 0,\ \textit{positions } 1, \ldots, i \textit{ are in } \sigma\}$ and

$$\ell_e = \begin{cases} n & \textit{if } \sigma \textit{ is a permutation of } 1, \ldots, |\sigma| \textit{ or } \sigma \textit{ is empty} \\ \rho_2 - 1 & \textit{otherwise} \end{cases}$$

with $\rho_2 = \min\{i : \ i > \rho_1,\ i \in \sigma\}$

Proof. According to the definition of the notation P_σ, σ is a sequence of positions that are assigned to the longest $|\sigma|$ jobs. Since we always branch on the longest unscheduled job, the first part of the proposition is straightforward. The second part aims at specifying the range of positions that job j^* can occupy. Two cases are considered depending on the content of σ:

– If σ is a permutation of $1, \ldots, |\sigma|$, it means that the longest $|\sigma|$ jobs are set on the first $|\sigma|$ positions, which implies that the job j^* should be branched on positions $|\sigma| + 1$ to n
– If σ is not a permutation of $\{1, \ldots, |\sigma|\}$, it means that the longest $|\sigma|$ jobs are not set on consecutive positions. As a result, the current unassigned positions may be split into several ranges. As a consequence of the decomposition property, the longest job j^* should necessarily be branched on the first range of free positions, that goes from ρ_1 to ρ_2. Let us consider as an example $P_{1,9,2,8}$, whose structure is $13\{5, \ldots, 9\}42\{10, \ldots, n\}$ and the job to branch on is 5. In this case, we have: $\sigma = (1, 9, 2, 8)$, $\ell_b = 3$, $\ell_e = 7$. It is easy to verify that 5 can only be branched on positions $\{3, \ldots, 7\}$ as a direct result of Property 2.

□

Corollary 1 emphasises the fact that even though a node may contain several ranges of free positions, only the first range is the current focus since we only branch on the longest job in eligible positions.

Corollary 1. *Problem P_σ has the following structure:*

$$\pi\{j^*, \ldots, j^* + \ell_e - \ell_b\}\Omega$$

with π the subsequence of jobs on the first $\ell_b - 1$ positions in σ and Ω the remaining subset of jobs to be scheduled after position ℓ_e (some of them can have been already scheduled). The merging procedure is applied on jobset $\{j^*, \ldots, j^* + \ell_e - \ell_b\}$ starting at time $t_\pi = \sum_{i \in \Pi} p_i$ where Π is the jobset of π.

The validity of merging on a general node still holds as indicated in Proposition 4, which extends the result stated in Proposition 1.

Proposition 4. *Let P_σ be a generic problem and let $\pi, j^*, \ell_b, \ell_e, \Omega$ be computed relatively to P_σ according to Corollary 1. If $\mathcal{L}_\sigma = \emptyset$ the j-th child node P_{σ^j} is P_{σ,ℓ_b+j-1} for $1 \le j \le k$. Otherwise, the j-th child node P_{σ^j} is extracted from $\mathcal{L}_\sigma$ for $1 \le j \le k-1$, while it is created as P_{σ,ℓ_b+k-1} for $j = k$. For any couple of problems P_{σ^j} and P_{σ^1,ℓ_b+j-1} with $2 \le j \le k$, the following conditions hold:*

1. *Problems P_{σ^j} and P_{σ^1,ℓ_b+j-1} with $2 \le j \le k$ have the following structure:*
   – P_{σ^j}:
$$\begin{cases} \pi^j\{j^*+j, \ldots, j^*+\ell_e-\ell_b\}\Omega & 1 \le j \le k-1 \text{ and } \mathcal{L}_\sigma \neq \emptyset \\ \pi\{j^*+1, \ldots, j^*+j-1\}j^*\{j^*+j, \ldots, j^*+\ell_e-\ell_b\}\Omega & (1 \le j \le k-1; \mathcal{L}_\sigma = \emptyset) \\ & \text{or } j = k \end{cases}$$
   – P_{σ^1,ℓ_b+j-1}:
   $\pi^1\{j^*+2, \ldots, j^*+j-1\}(j^*+1)\{j^*+j, \ldots, j^*+\ell_e-\ell_b\}\Omega$
2. *By solving all the problems of size less than k, that consist in scheduling the jobset $\{j^*+1, \ldots, j^*+j-1\}$ between π and j^* and in scheduling $\{j^*+2, \ldots, j^*+j-1\}$ between π^1 and j^*+1, both P_{σ^j} and P_{σ^1,ℓ_b+j-1} consist in scheduling $\{j^*+j, ..., j^*+\ell_e-\ell_b\}\Omega$ starting at time $t_{\pi^j} = \sum_{i \in \Pi^j} p_i$ where Π^j is the jobset of π^j.*

Proof. The first part of the statement follows directly from Definition 1 and simply defines the structure of the children nodes of P_σ. The problem P_{σ^j} is the result of a merging operation with the generic problem P_{σ,ℓ_b+j-1} and it could possibly coincide with P_{σ,ℓ_b+j-1}, for each j=1, ..., k−1. Furthermore, P_{σ^j} is exactly P_{σ,ℓ_b+j-1} for j=k. The generic structure of P_{σ,ℓ_b+j-1} is $\pi\{j^*+1,\ldots,j^*+j-1\}j^*\{j^*+j,\ldots,j^*+\ell_e-\ell_b\}\Omega$, and the merging operations preserve the jobset to schedule after j^*. Thus, we have $\Pi^j=\Pi\cup\{j^*,...,j^*+j-1\}$ for each j=1, ..., k−1, and this proves the first statement. Analougosly, the structure of P_{σ^1,ℓ_b+j-1} is $\pi^1\{j^*+2,\ldots,j^*+j-1\}(j^*+1)\{j^*+j,\ldots,j^*+\ell_e-\ell_b\}\Omega$. Once the subproblem before j^*+1 of size less than k is solved, P_{σ^1,ℓ_b+j-1} consists in scheduling the jobset $\{j^*+j,...,j^*+\ell_e-\ell_b\}$ at time $t_{\pi^j}=\sum_{i\in\Pi^j}p_i$. In fact, we have that $\Pi^j=\Pi^1\cup\{j^*+2,\ldots,j^*+j-1\}\cup\{j^*+1\}=\Pi\cup\{j^*,\ldots,j^*+j-1\}$. □

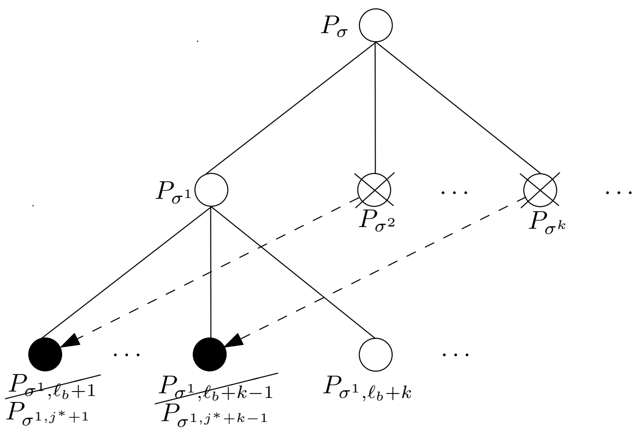


Fig. 3: Merging for a generic left-side branch

Analogously to the root node, each couple of problems indicated in Proposition 4 can be merged. Again, $(k-1)$ problems P_{σ^j} (with $2\le j\le k$) can be merged with the corresponding problems P_{σ^1,ℓ_b+j-1}. P_{σ^j} is deleted and P_{σ^1,ℓ_b+j-1} is replaced by $P_{\sigma^{1,j^*+j-1}}$ (Figure 3), defined as follows:

– Jobset $\{j^*+j,...,j^*+\ell_e-\ell_b\}\Omega$ is the set of jobs on which it remains to branch on.
– Let σ^{1,j^*+j-1} be the sequence of positions on which the j^*+j-1 longest jobs $1,...,j^*+j-1$ are branched, that leads to the best jobs permutation between π^j and $\pi^1\{j^*+2,\ldots,j^*+j-1\}(j^*+1)$ for $2\le j\le k-1$, and between $\pi\{j^*+1,\ldots,j^*+j-1\}j^*$ and $\pi^1\{j^*+2,\ldots,j^*+j-1\}(j^*+1)$ for $j=k$. This involves the solution of one or two problems of size at most $k-1$ (in $\mathcal{O}^*(2.4143^k)$ time by TTBR2) and the finding of the sequence that has the smallest total tardiness value knowing that both sequences start at time 0.

The LEFT_MERGE procedure is presented in Algorithm 2. Notice that, from a technical point of view, this algorithm takes as input one problem and produces as an output its first child node to branch on, which replaces all its k left-side children nodes.

Algorithm 2 LEFT_MERGE Procedure

Input: P_σ an input problem of size n, with ℓ_b, j^* accordingly computed
Output: Q: a list of problems to branch on after merging
1: **function** LEFT_MERGE(P_σ)
2: $Q \leftarrow \emptyset$
3: **for** j=1 to k **do**
4: Create $P_{\sigma j}$ (j-th child of P_σ) by the improved branching with the subproblem induced by jobset $\{j^*+1, \ldots, j^*+j-1\}$ solved if $\mathcal{L}_\sigma = \emptyset$ or $j=k$
5: **end for**
6: **for** j=1 to $k-1$ **do**
7: Create $P_{\sigma^1 j}$ (j-th child of P_{σ^1}) by the improved branching with the subproblem induced by jobset $\{j^*+2, \ldots, j^*+j-1\}$ solved if $\mathcal{L}_{\sigma^1} = \emptyset$ or $j=k$
8: $\mathcal{L}_{\sigma^1} \leftarrow \mathcal{L}_{\sigma^1} \cup$ BEST($P_{\sigma j+1}, P_{\sigma^1 j}$)
9: **end for**
10: $Q \leftarrow Q \cup P_{\sigma^1}$
11: **return** Q
12: **end function**

Lemma 2 *The LEFT_MERGE procedure returns one node to branch on in $\mathcal{O}(n)$ time and polynomial space. The corresponding problem is of size $n-1$.*

Proof. The creation of problems P_{σ^1, ℓ_b+j-1}, $\forall j = 2, \ldots, k$, can be done in $\mathcal{O}(n)$ time. The call of TTBR2 costs constant time. The BEST function called at line 8 consists in computing then comparing the total tardiness value of two known sequence of jobs starting at the same time instant: it runs in $\mathcal{O}(n)$ time. The overall time complexity of LEFT_MERGE procedure is then bounded by $\mathcal{O}(n)$ time as k is a constant. Finally, as only node P_{σ^1} is returned, its size is clearly $n-1$ when P_σ has size n. □

In the final part of this section, we discuss the extension of the algorithm in the case where $LPT \neq EDD$. In this case, Property 2 allows to discard subproblems associated to branching in some positions. Notice that if a problem P can be discarded according to this property, then we say that P does not exist and its associated node is empty.

Lemma 3 *Instances such that $LPT = EDD$ correspond to worst-case instances for which the LEFT_MERGE procedure returns one node of size $n-1$ to branch on, replacing all the k left-side children nodes of its parent node.*

Proof. Let us consider the improved branching scheme. The following exhaustive conditions hold:

1. $1 = [1]$ and $2 = [2]$;
2. $1 = [j]$ with $j \geq 2$;

3. $1 = [1]$ and $2 = [j]$ $j \geq 3$.

In case 1, the branching scheme matches the one of Figure 2, hence Lemma 3 holds according to 2. In case 2, the problem P_{σ^1} is empty if no problem has been merged to its position in the tree previously. The node associated to $P_{\sigma^1,\ell_b+\ell-1}$, $\forall \ell \leq k$, can then be considered as empty node, hence the merging can be done by simply moving the problem P_{σ^ℓ} into $P_{\sigma^1,\ell_b+\ell-1}$. As a consequence, the node returned by `LEFT_MERGE` only contains the merged nodes as children nodes, whose solution is much faster than solving a problem of size $n-1$. If P_{σ^1} is not empty due to a previous merging operation, the merging can be performed in the ordinary way. In case 3, the nodes associated to $P_{\sigma^1,\ell_b+1}, ..., P_{\sigma^1,\ell_b+j-2}$ may or may not be empty depending on the previous merging operation concerning P_{σ^1}, in either case the merging can be done. The same reasoning holds for nodes associated to P_{σ^ℓ} and $P_{\sigma^1,\ell_b+\ell-1}$ for $\ell \geq j$.

In general, the solution of problems P_{σ^ℓ}, $\forall \ell = 2, \dots, k$, can always be avoided. In the worst case, the node associated to P_{σ^1} contains a subproblem of size $n-1$, otherwise with the application of Property 2, it contains a problem whose certain children are set as empty.

□

3.2 Merging right-side branches

Due to the branching scheme, the merging of right-side branches involves a more complicated procedure than the merging of left-side branches. In the merging of left-side branches, it is possible to merge some nodes associated to problems P_ℓ with children nodes of P_1, while for the right-side branches, it is not possible to merge some nodes P_ℓ with children nodes of P_n. We can only merge children nodes of P_ℓ with children nodes of P_n. Let us more formally introduce the right merging procedure and, again, let $k < \frac{n}{2}$ be the same constant parameter as used in the left merging.

Figure 4 shows an example on the structure of merging for the k right-side branches with $k = 3$. The root problem P consists in scheduling jobset $\{1, \dots, n\}$. Unlike left-side merging, the right-side merging is done horizontally for each level. Nodes that are involved in merging are colored. For instance, the black square nodes at level 1 can be merged. Similarly, the black circle nodes at level 1 can be merged, the grey square nodes at level 2 can be merged and the grey circle nodes at level 2 can be merged. Notice that each right-side branch of P is expanded to a different depth which is actually an arbitrary decision: the expansion stops when the first child node has size $n-k-1$ as indicated in the figure. This eases the computation of the final complexity.

More generally, Figure 5 shows the right-side search tree and the content of the nodes involved in the merging in a generic way.

The rest of this section intends to describe the merging by following the same lines as for left merging. We first extend the notation P_σ in the sense

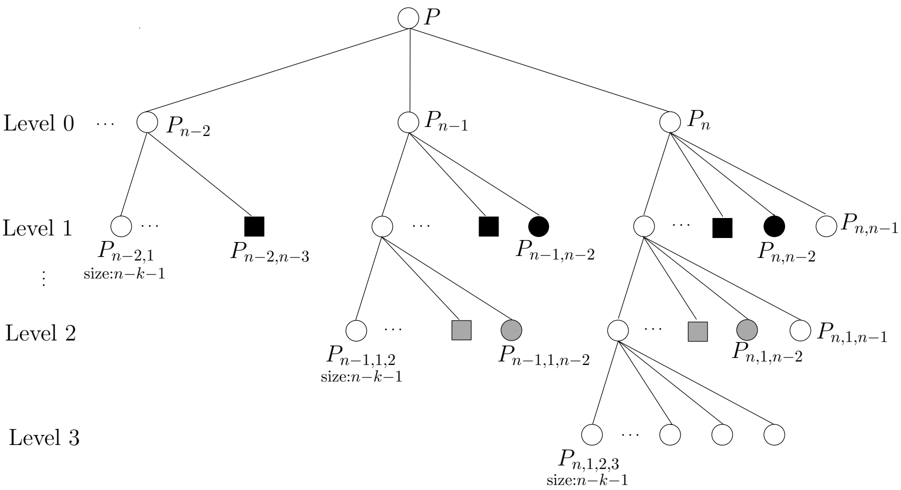
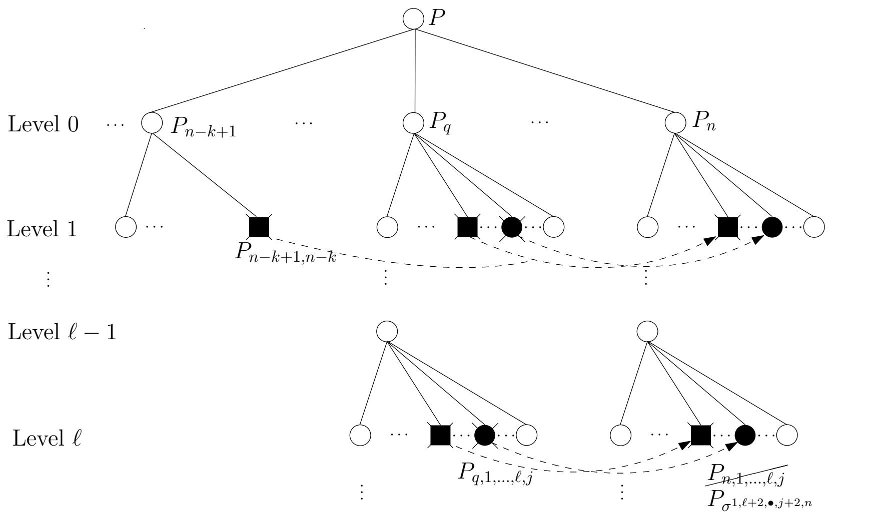


Fig. 4: An example of right-side branches merging for $k = 3$



$P_{q,1,\dots,\ell,j} : (2,\dots,\ell{+}1)\{\ell{+}3,...,j{+}1\}(\ell{+}2)\{j{+}2,...,q\}1\{q{+}1,...,n\}$
$P_{\sigma^{1,\ell+2,\bullet,j+2,n}} : (2,\dots,\ell{+}1)\{\ell{+}3,...,j{+}1\} \underset{\max\{j+1,n-k+\ell+1\}\leq q\leq n}{\texttt{BEST}} ((\ell{+}2)\{j{+}2,...,q\}1\{q{+}1,...,n\})$

Fig. 5: Generic right-side merging at the root node

that σ may now contain placeholders. The i-th element of σ is either the position assigned to job i if i is fixed, or $\bullet$ if job i is not yet fixed. The $\bullet$ sign is used as placeholder, with its cardinality below indicating the number of consecutive $\bullet$. As an example, the problem $\{2,\dots,n-1\}1n$ can now be denoted by $P_{n-1,\underset{n-2}{\bullet},n}$. The cardinality of $\bullet$ may be omitted whenever it is not important for the presentation or it can be easily deduced as in the above example. Note that this adapted notation eases the presentation of right merge while it has no impact on the validity of the results stated in the previous section.

Proposition 5. *Let P_σ be a problem to branch on. Let $j^*, \ell_b, \ell_e, \rho_1$ and ρ_2 be defined as in Proposition 3. Extending Corollary 1, problem P_σ has the following structure:*

$$\pi\{j^*, \ldots, j^* + \ell_e - \ell_b\}\gamma\Omega'$$

where π is defined as in Corollary 1 and γ is the sequence of jobs on positions $\rho_2, \ldots, \rho_3$ with $\rho_3 = \max\{i : i \geq \rho_2, positions\ \rho_2, \ldots, i\ are\ in\ \sigma\}$ and Ω' the remaining subset of jobs to be scheduled after position ρ_3 (some of them can have been already scheduled). The merging procedure is applied on jobset $\{j^*, \ldots, j^* + \ell_e - \ell_b\}$ preceded by a sequence of jobs π and followed by $\gamma\Omega'$.

Proof. The problem structure stated in Corollary 1 is refined on the part of Ω. Ω is split into two parts: γ and Ω'. The motivation is that γ will be involved in the right merging, just like the role of π in left merging. $\square$

Proposition 6 shed lights on how to merge the right side branches originated from the root node.

Proposition 6. *For each problem in the set*

$$\mathcal{S}_{\ell,j} = \left\{ P_\sigma : \begin{array}{l} |\sigma| = \ell+2, \\ \max\{j+1, n-k+\ell+1\} \leq \sigma_1 \leq n, \\ \sigma_i = i-1,\ \forall i \in \{2, \ldots, \ell+1\}, \\ \sigma_{\ell+2} = j \end{array} \right\}^1$$

with $0 \leq \ell \leq k-1$, $n-k \leq j \leq n-1$, and with σ_i referring to the position of job i in σ, we have the two following properties:

1. *The solution of problems in $\mathcal{S}_{\ell,j}$ involves the solution of a common subproblem which consists in scheduling jobset $\{\ell+3, ..., j+1\}$ starting at time $t_\ell = \sum_{i=2}^{\ell+1} p_i$.*
2. *For any problem in $\mathcal{S}_{\ell,j}$, at most $k+1$ jobs have to be scheduled after jobset $\{\ell+3, ..., j+1\}$.*

Proof. As each problem P_σ is defined by $(2, \ldots, \ell+1)\{\ell+3, ..., j+1\}(\ell+2)\{j+2, ..., \sigma_1\}1\{\sigma_1+1, ..., n\}$, the first part of the property is straightforward. Besides, the second part can be simply established by counting the number of jobs to be scheduled after jobset $\{\ell+3, ..., j+1\}$ when j is minimal, *i.e.* when $j=n-k$. In this case, $(\ell+2)\{j+2, ..., \sigma_1\}1\{\sigma_1+1, ..., n\}$ contains $k+1$ jobs. $\square$

The above proposition highlights the fact that some nodes can be merged as soon as they share the same initial subproblem to be solved. More precisely, at most $k-\ell-1$ nodes associated to problems $P_{q,1..\ell,j}$, $\max\{j+1, n-k+\ell+1\} \leq q \leq (n-1)$, can be merged with the node associated to problem $P_{n,1..\ell,j}$,

[1] Placeholders do not count in the cardinality of σ

$\forall j = (n-k), ..., (n-1)$. The node $P_{n,1..\ell,j}$ is replaced in the search tree by the node $P_{\sigma^{1,\ell+2,\bullet,j+2,n}}$ defined as follows (Figure 5):

- Jobset $\{\ell+3, ..., j+1\}$ is the set of jobs on which it remains to branch.
- Let $\sigma^{1,\ell+2,\bullet,j+2,n}$ be the sequence containing positions of jobs $\{1, \ldots, \ell+2, j+2, \ldots, n\}$ and placeholders for the other jobs, that leads to the best jobs permutation among $(\ell+2)\{j+2, ..., q\}1\{q+1, ..., n\}$, $\max\{j+1, n-k+\ell+1\} \leq q \leq n$. This involves the solution of at most k problems of size at most $k+1$ (in $\mathcal{O}^*(k \times 2.4143^{k+1})$ time by TTBR2) and the determination of the best of the computed sequences knowing that all of them start at time t, namely the sum of the jobs processing times in $(2, \ldots, \ell+1)\{\ell+3, ..., j+1\}$.

The merging process described above is applied at the root node, while an analogous merging can be applied at any node of the tree. With respect to the root node, the only additional consideration is that the right-side branches of a general node may have already been modified by previous mergings. As an example, let us consider Figure 6. It shows that, subsequently to the merging operations performed from P, the right-side branches of P_n may not be the subproblems induced by the branching scheme. However, it can be shown in a similar way as per left-merge, that the merging can still be applied.

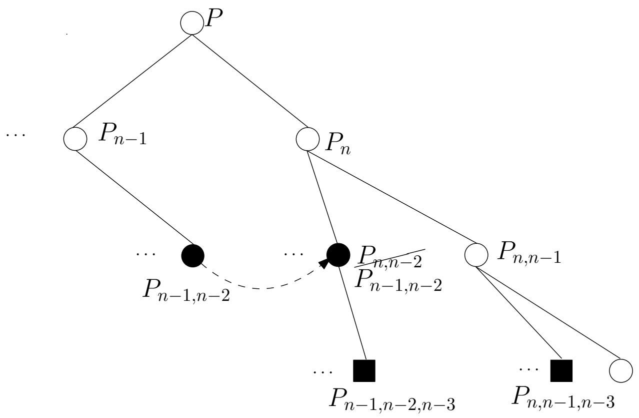


Fig. 6: The right branches of P_n have been modified when performing right-merging from P

In order to define the branching scheme used with the `RIGHT_MERGE` procedure, a data structure $\mathcal{R}_\sigma$ is associated to a problem P_σ. It represents a list of subproblems that result from a previous merging and are now the k right-side children nodes of P_σ. When a merging operation sets the k right-side children nodes of P_σ to $P_{\sigma^{n-k+1}}, ..., P_{\sigma^n}$, we set $\mathcal{R}_\sigma = \{P_{\sigma^{n-k+1}}, ..., P_{\sigma^n}\}$, otherwise we have $\mathcal{R}_\sigma = \emptyset$. As a conclusion, the following branching scheme for a generic node of the tree is defined. It is an extension of the branching scheme defined in Definition 1.

Definition 2 The branching scheme for a generic node P_σ is defined as follows:

- If $\mathcal{R}_\sigma = \emptyset$, use the branching scheme defined in Definition 1;
- If $\mathcal{L}_\sigma = \emptyset$ and $\mathcal{R}_\sigma \neq \emptyset$, branch on the longest job in the available positions from the 1st to the $(n-k)$-th, then extract problems from $\mathcal{R}_\sigma$ as the last k branches.
- If $\mathcal{L}_\sigma \neq \emptyset$ and $\mathcal{R}_\sigma \neq \emptyset$, extract problems from $\mathcal{L}_\sigma$ as the first $k-1$ branches, then branch on the longest job in the available positions from the k-th to the $n-k$-th, finally extract problems from $\mathcal{R}_\sigma$ as the last k branches.

This branching scheme, whenever necessary, will be referred to as improved branching. It generalizes, also replaces, the one introduced in Definition 1

Proposition 7 states the validity of merging a general node, which extends the result in Proposition 6.

Proposition 7. *Let P_σ be a generic problem and let $\pi, j^*, \ell_b, \ell_e, \gamma, \Omega'$ be computed relatively to P_σ according to Proposition 5. If $\mathcal{R}_\sigma{=}\emptyset$, the right merging on P_σ can be easily performed by considering P_σ as a new root problem. Suppose $\mathcal{R}_\sigma{\neq}\emptyset$, the q-th child node P_{σ^q} is extracted from $\mathcal{R}_\sigma$, $\forall n'{-}k{+}1{\leq}q{\leq}n'$, where $n'{=}\ell_e{-}\ell_b{+}1$ is the number of children nodes of P_σ. The structure of P_{σ^q} is $\pi\{j^*{+}1,...,j^*{+}q{-}1\}\gamma^q\Omega'$.*

For $0{\leq}\ell{\leq}k{-}1$ and $n'{-}k{\leq}j{\leq}n'{-}1$, the following conditions hold:

1. *Problems in $\mathcal{S}^\sigma_{\ell,j}$ have the following structure:* $\pi(j^*{+}1,\ldots,j^*{+}\ell)\{j^*{+}\ell{+}2,...,j^*{+}j\}(j^*{+}\ell{+}1)\{j^*{+}j{+}1,...,j^*{+}q{-}1\}\gamma^q\Omega'$ *with q varies from* $\max\{j{+}1, n{-}k{+}\ell{+}1\}$ *to n'.*
2. *The solution of all problems in $\mathcal{S}^\sigma_{\ell,j}$ involves the scheduling of a jobset* $\{j^*{+}j{+}1,...,j^*{+}q{-}1\}$, $\max\{j{+}1, n{-}k{+}\ell{+}1\}{\leq}q{\leq}n'$, *which is of size less than k. Besides, for all problems in $\mathcal{S}^\sigma_{\ell,j}$ it is required to solve a common subproblem made of jobset $\{j^*{+}\ell{+}2,...,j^*{+}j\}$ starting after $\pi(j^*{+}1,\ldots,j^*{+}\ell)$ and before $(j^*{+}\ell{+}1)\{j^*{+}j{+}1,...,j^*{+}q{-}1\}\gamma^q\Omega'$.*

Proof. The proof is similar to the one of Proposition 4. The first part of the statement follows directly from Definition 2 and simply defines the structure of the children nodes of P_σ. For the second part, it is necessary to prove that $\{j^*{+}j{+}1,...,j^*{+}q{-}1\}\gamma^q$ consists of the same jobs for any valid value of q. Actually, since right-merging only merges nodes that have common jobs fixed after the unscheduled jobs, the jobs present in $\{j^*{+}j{+}1,...,j^*{+}q{-}1\}\gamma^q$ and the jobs present in $\{j^*{+}j{+}1,...,j^*{+}q{-}1\}j^*\{j^*{+}q,...,j^*{+}n'{-}1\}\gamma$, $\max\{j{+}1, n{-}k{+}\ell{+}1\}{\leq}q{\leq}n'$, must be the same, which proves the statement. □

Analogously to the root node, given the values of ℓ and j, all the problems in $\mathcal{S}^\sigma_{\ell,j}$ can be merged. More precisely, we rewrite σ as $\alpha\underset{n'}{\bullet}\beta$ where α is the sequence of positions assigned to jobs $\{1,\ldots,j^*{-}1\}$, $\underset{n'}{\bullet}$ refers to the jobset to branch on and β contains the positions assigned to the rest of jobs. At most $k-\ell-1$ nodes associated to problems $P_{\alpha,\ell_b+q-1,\ell_b..\ell_b+\ell-1,\ell_b+j-1,\bullet,\beta}$, with $\max\{j+1, n'-k+\ell+1\} \leq q \leq n'-1$, can be merged with the node associated to problem $P_{\alpha,\ell_e,\ell_b..\ell_b+\ell-1,\ell_b+j-1,\bullet,\beta}$.

Node $P_{\alpha,\ell_e,\ell_b..\ell_b+\ell-1,\ell_b+j-1,\bullet,\beta}$ is replaced in the search tree by node $P_{\alpha,\sigma^{\ell,\ell_b,j},\bullet,\beta}$ defined as follows:

- Jobset $\{j^*+\ell+2, ..., j^*+j\}$ is the set of jobs on which it remains to branch.
- Let $\sigma^{\ell,\ell_b,j}$ be the sequence of positions among

$$\{(\ell_b+q-1, \ell_b..\ell_b+\ell-1, \ell_b+j-1) : \max\{j+1, n'-k+\ell+1\} \leq q \leq n'-1\}$$

  associated to the best job permutation on $(j^*+\ell+1)\{j^*+j+1, ..., j^*+q-1\}\gamma^q$, $\forall \max\{j+1, n'-k+\ell+1\} \leq q \leq n'$. This involves the solution of k problems of size at most $k+1$ (in $\mathcal{O}^*(k \times 2.4143^{k+1})$ time by TTBR2) and the determination of the best of the computed sequences knowing that all of them start at time t, namely the sum of the jobs processing times in $\pi(j^*+1, \ldots, j^*+\ell)\{j^*+\ell+2, ..., j^*+j\}$.

The RIGHT_MERGE procedure is presented in Algorithm 3. Notice that, similarly to the LEFT_MERGE procedure, this algorithm takes as input one problem P_σ and provides as an output a set of nodes to branch on, which replaces all its k right-side children nodes of P_σ. It is interesting to notice that the LEFT_MERGE procedure is also integrated.

A procedure MERGE_RIGHT_NODES (Algorithm 4) is invoked to perform the right merging for each level $\ell = 0, ..., k-1$ in a recursive way. The initial inputs of this procedure (line 13 in RIGHT_MERGE) are the problem P_σ and the list of its k right-side children nodes, denoted by *rnodes*. They are created according to the improved branching (lines 4-12 of Algorithm 3). Besides, the output is a list Q containing the problems to branch on after merging. In the first call to MERGE_RIGHT_NODES, the left merge is applied to the first element of *rnodes* (line 2), all the children nodes of nodes in *rnodes* not involved in right nor left merging, are added to Q (lines 3-7). This is also the case for the result of the right merging operations at the current level (lines 8-11). In Algorithm 4, the value of r indicates the current size of *rnodes*. It is reduced by one at each recursive call and the value $(k-r)$ identifies the current level with respect to P_σ. As a consequence, each right merging operation consists in finding the problem with the best total tardiness value on its fixed part, among the ones in set $\mathcal{S}^{\sigma}_{k-r,j}$. This is performed by the BEST function (line 10 of MERGE_RIGHT_NODES) which extends the one called in Algorithm 2 by taking at most k subproblems as input and returning the dominating one.

The MERGE_RIGHT_NODES procedure is then called recursively on the list containing the first child node of the 2nd to r-th node in *rnodes* (lines 13-17). Note that the procedure LEFT_MERGE is applied to every node in *rnodes* except the last one. In fact, for any specific level, the last node in *rnodes* belongs to the last branch of P_σ, which is $P_{\sigma,l_b+n-1,\bullet,\beta}$. Since $P_{\sigma,l_b+n-1,\bullet,\beta}$ is put into Q at line 14 of RIGHT_MERGE, it means that this node will be re-processed later and LEFT_MERGE will be called on it at that moment. Since the recursive call of MERGE_RIGHT_NODES (line 18) will merge some nodes to the right-side children nodes of $P_{\alpha,\ell_b,\underset{n_r-1}{\bullet},\beta^r}$, the latter one must be added to the list $\mathcal{L}$ of $P_{\alpha,\underset{n_r}{\bullet},\beta^r}$ (line 19). In addition, since we defined $\mathcal{L}$ as a list of size either 0 or $k-1$, lines 20-24 add the other $(k-2)$ nodes to $\mathcal{L}_{\alpha,\underset{n_r}{\bullet},\beta^r}$.

It is also important to notice the fact that a node may have its $\mathcal{L}$ or $\mathcal{R}$ structures non-empty, if and only if it is the first or last child node of its parent node. A direct result is that only one node among those involved in a merging may have its $\mathcal{L}$ or $\mathcal{R}$ non-empty. In this case, these structures need to be associated to the resulting node. The reader can always refer to Figure 4 for a more intuitive representation.

Algorithm 3 `RIGHT_MERGE` Procedure

Input: $P_\sigma = P_{\alpha,\underset{n}{\bullet},\beta}$ a problem of size n, with ℓ_b, j^* computed according to Proposition 3
Output: Q : a list of problems to branch on after merging
1: **function** `RIGHT_MERGE`(P_σ)
2: $\quad Q \leftarrow \emptyset$
3: $\quad nodes \leftarrow \emptyset$
4: $\quad$ **if** $\mathcal{R}_\sigma = \emptyset$ **then**
5: $\qquad$ **for** $q = n-k+1$ to n **do**
6: $\qquad\quad$ Create $P_{\alpha,\ell_b+q-1,\bullet,\beta}$ by branching
7: $\qquad\quad \delta \leftarrow$ the sequence of positions of jobs $\{j^*+q,\ldots,j^*+n-1\}$ fixed by TTBR2
8: $\qquad\quad nodes \leftarrow nodes + P_{\alpha,\ell_b+q-1,\bullet,\delta,\beta}$
9: $\qquad$ **end for**
10: $\quad$ **else**
11: $\qquad nodes \leftarrow \mathcal{R}_\sigma$
12: $\quad$ **end if**
13: $\quad Q \leftarrow Q \cup$ `MERGE_RIGHT_NODES`($nodes, P_\sigma$)
14: $\quad Q \leftarrow Q \cup nodes[k]$ ▷ The last node will be re-processed
15: $\quad$ **return** Q
16: **end function**

Lemma 4 *The* ***RIGHT_MERGE*** *procedure returns a list of $\mathcal{O}(n)$ nodes in polynomial time and space.*
The solution of the associated problems involves the solution of 1 *subproblem of size $(n-1)$, of $(k-1)$ subproblems of size $(n-k-1)$, and subproblems of size i and $(n_q-(k-r)-i-1)$, $\forall r=2,...,k; q=1,...,(r-1); i=k,...,(n-2k+r-2)$.*

Proof. The first part of the result follows directly from Algorithm 3. The only lines where nodes are added to Q in `RIGHT_MERGE` are lines 13-14. In line 14, only one problem is added to Q, thus it needs to be proved that the call on `MERGE_RIGHT_NODES` (line 13) returns $\mathcal{O}(n)$ nodes. This can be computed by analysing the lines 2-7 of Algorithm 4. Considering all recursive calls, the total number of nodes returned by `MERGE_RIGHT_NODES` is $(\sum_{i=1}^{k-1}(k-i)(n-2k-i))+k-1$ which yields $\mathcal{O}(n)$. The number of all the nodes considered in right merging is bounded by a linear function on n. Furthermore, all the operations associated to the nodes (merging, creation, etc) have a polynomial cost. As a consequence, Algorithm 3 runs in polynomial time and space.

Algorithm 4 MERGE_RIGHT_NODES Procedure

Input: $rnodes = [P_{\alpha,\underset{n_1}{\bullet},\beta^1}, \ldots, P_{\alpha,\underset{n_r}{\bullet},\beta^r}]$, ordered list of r last children nodes with ℓ_b defined on any node in $rnodes$. $|\alpha|+1$ is the job to branch on and $n_r = n_1 + r - 1$.
Output: Q, a list of problems to branch on after merging

1: **function** MERGE_RIGHT_NODES($rnodes, P_\sigma$)
2: $\quad Q \leftarrow$ LEFT_MERGE($P_{\alpha,\underset{n_1}{\bullet},\beta^1}$)
3: $\quad$ **for** $q = 1$ to $r-1$ **do**
4: $\quad\quad$ **for** $j = \ell_b + k$ to $\ell_b + n_1 - 1$ **do**
5: $\quad\quad\quad Q \leftarrow Q \cup P_{\alpha,j,\underset{n_q-1}{\bullet},\beta^q}$
6: $\quad\quad$ **end for**
7: $\quad$ **end for**
8: $\quad$ **for** $j = \ell_b + n_1$ to $\ell_b + n_r$ **do**
9: $\quad\quad$ Solve all the subproblems of size less than k in $\mathcal{S}^\sigma_{k-r,j}$
10: $\quad\quad \mathcal{R}_{\alpha,\underset{n_r}{\bullet},\beta^r} \leftarrow \mathcal{R}_{\alpha,\underset{n_r}{\bullet},\beta^r} +$ BEST($\mathcal{S}^\sigma_{k-r,j}$)
11: $\quad$ **end for**
12: $\quad$ **if** $r > 2$ **then**
13: $\quad\quad newnodes \leftarrow \emptyset$
14: $\quad\quad$ **for** $q = 2$ to $r-1$ **do**
15: $\quad\quad\quad newnodes \leftarrow newnodes +$ LEFT_MERGE($P_{\alpha,\underset{n_q}{\bullet},\beta^q}$)
16: $\quad\quad$ **end for**
17: $\quad\quad newnodes \leftarrow newnodes + P_{\alpha,\ell_b,\underset{n_r-1}{\bullet},\beta^r}$
18: $\quad\quad Q \leftarrow Q \cup$ MERGE_RIGHT_NODES($newnodes, P_\sigma$)
19: $\quad\quad \mathcal{L}_{\alpha,\underset{n_r}{\bullet},\beta^r} \leftarrow P_{\alpha,\ell_b,\underset{n_r-1}{\bullet},\beta^r}$
20: $\quad\quad$ **for** $q = 2$ to $k-1$ **do**
21: $\quad\quad\quad$ Create $P_{\alpha,\ell_b+q-1,\underset{n_r-1}{\bullet},\beta^r}$ by branching
22: $\quad\quad\quad \delta \leftarrow$ the sequence of positions of jobs $\{|\alpha|+2, \ldots, |\alpha|+q\}$ fixed by TTBR2
23: $\quad\quad\quad \mathcal{L}_{\alpha,\underset{n_r}{\bullet},\beta^r} \leftarrow \mathcal{L}_{\alpha,\underset{n_r}{\bullet},\beta^r} + P_{\alpha,\ell_b+q-1,\delta,\underset{n_r-1}{\bullet},\beta^r}$
24: $\quad\quad$ **end for**
25: $\quad$ **end if**
26: $\quad$ **return** Q
27: **end function**

Regarding the sizes of the subproblems returned by RIGHT_MERGE, the node added in line 14 of Algorithm 3 contains one subproblem of size $(n-1)$, corresponding to branching the longest job on the last available position. Then, the problems added by the call to MERGE_RIGHT_NODES are added to Q. In line 2 of Algorithm 4, the size of the problem returned by LEFT_MERGE is reduced by one unit when compared to the input problem which is of size $(n-k-(k-r))$. Note that $(k-r)$ is the current level with respect to the node tackled by Algorithm 4. As a consequence, the size of the resulting subproblem is $(n-k-(k-r)-1)$. Note that this line is executed $(k-1)$ times, $\forall r = k, \ldots, 2$, corresponding to the number of calls to MERGE_RIGHT_NODES. In line 5 of Algorithm 4, the list of nodes which are not involved in any merging operation are added to Q. This corresponds to couples of problems of size i and $(n_q-(k-r)-i-1)$, $\forall i = k, ..., (n-k-1)$ and this proves the last part of the lemma. □

Lemma 5 *Instances such that $LPT = EDD$ correspond to worst-case instances for which the `RIGHT_MERGE` procedure returns $\mathcal{O}(n)$ nodes to branch on, whose subproblems are listed in Lemma 4, replacing all the k right-side children nodes of its parent node.*

Proof. The proof follows similar reasoning as the one in Lemma 3. In general, if $LPT \neq EDD$ then the number of nodes in $\mathcal{S}^{\sigma}_{\ell,j}$ (defined in Proposition 7) could be less, since some nodes may not be created due to Property 2. However, all the nodes inside $\mathcal{S}^{\sigma}_{\ell,j}$ can still be merged to one except when $\mathcal{S}^{\sigma}_{\ell,j}$ is empty. In either case, we can achieve at least the same reduction as the case of $LPT = EDD$. □

3.3 Complete algorithm and analysis

We are now ready to define the main procedure `TTBM` (Total Tardiness Branch-and-Merge), stated in Algorithm 5 which is called on the initial input problem $P : \{1, ..., n\}$. The algorithm has a similar recursive structure as TTBR1. However, each time a node is opened, the sub-branches required for the merging operations are generated, the subproblems of size less than k are solved and the procedures `LEFT_MERGE` and `RIGHT_MERGE` are called. Then, the algorithm proceeds recursively by extracting the next node from Q with a depth-first strategy and terminates when Q is empty.

Proposition 8 determines the time complexity of the proposed algorithm. In this regard, the complexity of the algorithm depends on the value given to k. The higher it is, the more subproblems can be merged and the better is the worst-case time complexity of the approach.

Proposition 8. *Algorithm TTBM runs in $\mathcal{O}^*((2+\epsilon)^n)$ time and polynomial space, where $\epsilon \to 0$ for large enough values of k.*

Proof. The proof is based on the analysis of the number and the size of the subproblems put in Q when a single problem P^* is expanded. As a consequence of Lemma 3 and Lemma 5, TTBM induces the following recursion:

$$\begin{aligned}
T(n) =& 2T(n-1) + 2T(n-k-1) + ... + 2T(k) \\
&+ \sum_{r=2}^{k} \sum_{q=1}^{r-1} \sum_{i=k}^{n_1-(k-r)-2} (T(i) + T(n_q - (k-r) - i - 1)) \\
&+ (k-1)T(n_1 - 1) + \mathcal{O}(p(n))
\end{aligned}$$

First, a simple lower bound on the complexity of the algorithm can be derived by the fact that the procedures `RIGHT_MERGE` and `LEFT_MERGE` provide (among the others) two subproblems of size $n-1$, based on which the following inequality holds:

$$T(n) > 2T(n-1) \tag{7}$$

Algorithm 5 Total Tardiness Branch and Merge (TTBM)

```
Input: P : {1, ..., n}: input problem of size n
       n/2 > k ≥ 2: an integer constant
Output: seqOpt: an optimal sequence of jobs
function TTBM(P,k)
    Q ← P
    seqOpt ← a random sequence of jobs
    while Q ≠ ∅ do
        P* ← extract next problem from Q (depth-first order)
        if the size of P* < 2k then
            Solve P* by calling TTBR2
        end if
        if all jobs {1, ..., n} are fixed in P* then
            seqCurrent ← the solution defined by P*
            seqOpt ← best solution between seqOpt and seqCurrent
        else
            Q ← Q ∪ LEFT_MERGE(P*)                                  ▷ Left-side nodes
            for i = k + 1, ..., n − k do
                Create the i-th child node P_i by branching scheme of TTBR1
                Q ← Q ∪ P_i
            end for
            Q ← RIGHT_MERGE(P*)                                     ▷ Right-side nodes
        end if
    end while
    return seqOpt
end function
```

By solving the recurrence, we obtain that $T(n) = \omega(2^n)$. As a consequence, the following inequality holds:

$$T(n) > T(n-1) + \ldots + T(1) \tag{8}$$

In fact, if it does not hold, we have a contradiction on the fact $T(n) = \omega(2^n)$. Now, we consider the summation $\sum_{i=k}^{n_1-(k-r)-2}(T(n_q-(k-r)-i-1))$. Since $n_q = n_1 + q - 1$, we can simply expand the summation as follows:

$$\sum_{i=k}^{n_1-(k-r)-2}(T(n_q-(k-r)-i-1)) = T(q) + ... + T(n_1-(k-r)+q-k-2)$$

. We know that $k \geq q$, then $q - k \leq 0$ and the following inequality holds:

$$T(q) + ... + T(n_1-(k-r)+q-k-2) \leq \sum_{i=q}^{n_1-(k-r)-2} T(i)$$

.

As a consequence, we can bound above $T(n)$ as follows:

$$\begin{aligned}
T(n) =& 2T(n-1) + 2T(n-k-1) + ... + 2T(k) \\
&+ \sum_{r=2}^{k}\sum_{q=1}^{r-1}\sum_{i=k}^{n_1-(k-r)-2} (T(i) + T(n_q - (k-r) - i - 1)) \\
&\leq 2T(n-1) + 2T(n-k-1) + ... + 2T(k) \\
&\quad + \sum_{r=2}^{k}\sum_{q=1}^{r-1}\sum_{i=q}^{n_1-(k-r)-2} 2T(i) + (k-1)T(n_1-1) + \mathcal{O}(p(n)) \\
&\leq 2T(n-1) + 2T(n-k-1) + ... + 2T(k) \\
&\quad + \sum_{r=2}^{k}\sum_{q=1}^{r-1}\sum_{i=1}^{n_1-(k-r)-2} 2T(i) + (k-1)T(n_1-1) + \mathcal{O}(p(n))
\end{aligned}$$

By using Equation 8, we obtain the following:

$$\begin{aligned}
T(n) &\leq 2T(n-1) + 2T(n-k-1) + ... + 2T(k) \\
&\quad + \sum_{r=2}^{k}\sum_{q=1}^{r-1}\sum_{i=1}^{n_1-(k-r)-2} 2T(i) + (k-1)T(n_1-1) + \mathcal{O}(p(n)) \\
&\leq 2T(n-1) + 2T(n-k-1) + ... + 2T(k) \\
&\quad + \sum_{r=2}^{k}\sum_{q=1}^{r-1} 2T(n_1 - (k-r) - 1) + (k-1)T(n_1-1) + \mathcal{O}(p(n))
\end{aligned}$$

Finally, we apply some algebraic steps and we use the equality $n_1 = n - k$ to derive the following upper limitation of $T(n)$:

$$\begin{aligned}
T(n) &\leq 2T(n-1) + 2T(n-k-1) + ... + 2T(k) \\
&\quad + \sum_{r=2}^{k}(r-1)2T(n_1 - (k-r) - 1) + (k-1)T(n_1-1) + \mathcal{O}(p(n)) \\
&\leq 2T(n-1) + 2T(n-k-1) + ... + 2T(k) + 2(k-1)T(n_1-1) \\
&\quad + \sum_{r=2}^{k-1}(r-1)2T(n_1 - (k-r) - 1) + (k-1)T(n_1-1) + \mathcal{O}(p(n)) \\
&\leq 2T(n-1) + 2T(n-k-1) + ... + 2T(k) \\
&\quad + (k-1)4T(n_1-1) + (k-1)T(n_1-1) + \mathcal{O}(p(n)) \\
&\leq 2T(n-1) + 4T(n-k-1) + 5(k-1)T(n-k-1) + \mathcal{O}(p(n)) \\
&= 2T(n-1) + (5k-1)T(n-k-1) + \mathcal{O}(p(n))
\end{aligned}$$

k	$T(n)$
3	$\mathcal{O}^*(2.5814^n)$
4	$\mathcal{O}^*(2.4302^n)$
5	$\mathcal{O}^*(2.3065^n)$
6	$\mathcal{O}^*(2.2129^n)$
7	$\mathcal{O}^*(2.1441^n)$
8	$\mathcal{O}^*(2.0945^n)$
9	$\mathcal{O}^*(2.0600^n)$
10	$\mathcal{O}^*(2.0367^n)$
11	$\mathcal{O}^*(2.0217^n)$
12	$\mathcal{O}^*(2.0125^n)$
13	$\mathcal{O}^*(2.0070^n)$
14	$\mathcal{O}^*(2.0039^n)$
15	$\mathcal{O}^*(2.0022^n)$
16	$\mathcal{O}^*(2.0012^n)$
17	$\mathcal{O}^*(2.0007^n)$
18	$\mathcal{O}^*(2.0004^n)$
19	$\mathcal{O}^*(2.0002^n)$
20	$\mathcal{O}^*(2.0001^n)$

Table 1: The time complexity of TTBM for values of k from 3 to 20

Note that $\mathcal{O}(p(n))$ includes the cost for creating all nodes for each level and the cost of all the merging operations, performed in constant time.

The recursion $T(n) = 2T(n-1) + (5k-1)T(n-k-1) + \mathcal{O}(p(n))$ is an upper limitation of the running time of TTBM. Recall that its solution is $T(n) = \mathcal{O}^*(c^n)$ where c is the largest root of the function:

$$f_k(x) = 1 - \frac{2}{x} - \frac{5k-1}{x^{k+1}} \tag{9}$$

.

As k increases, the function $f_k(x)$ converges to $1 - \frac{2}{x}$, which induces a complexity of $\mathcal{O}^*(2^n)$. Table 1 shows the time complexity of TTBM obtained by solving Equation 9 for all the values of k from 3 to 20. The base of the exponential is computed by solving Equation 9 by means of a mathematical solver and rounding up the fourth digit of the solution. The table shows that the time complexity is $\mathcal{O}^*(2.0001^n)$ for $k \geq 20$. $\square$

4 Conclusions

This paper focused on the design of exact branching algorithms for the single machine total tardiness problem. By exploiting some inherent properties of the problem, we first proposed two branch-and-reduce algorithms, indicated with TTBR1 and TTBR2. The former runs in $\mathcal{O}^*(3^n)$, while the latter achieves a better time complexity in $\mathcal{O}^*(2.4143^n)$. The space requirement is polynomial in both cases. Furthermore, a technique called branch-and-merge, is presented and applied onto TTBR1 in order to improve its performance. The final achievement is a new algorithm (TTBM) with time complexity converging to $\mathcal{O}^*(2^n)$ and polynomial space. The same technique can be tediously adapted to improve the performance of TTBR2, but the resulting algorithm achieves the same asymptotic time complexity as TTBM, and thus it was omitted. To the best of authors' knowledge, TTBM is the polynomial space algorithm that has the best worst-case time complexity for solving this problem.

Beyond the new established complexity results, the main contribution of the paper is the branch-and-merge technique. The basic idea is very simple, and it consists of speeding up branching algorithms by avoiding to solve identical problems. The same goal is traditionally pursued by means of *Memorization* [2], where the solution of already solved subproblems are stored and then queried when an identical subproblem appears. This is at the cost of exponential space. In contrast, branch-and-merge discards identical subproblems but by appropriately merging, in polynomial time and space, nodes involving the solution of common subproblems. When applied systematically in the search tree, this technique enables to achieve a good worst-case time bound. On a computational side, it is interesting to notice that node merging can be relaxed to avoid solving in $\mathcal{O}^*(2.4143^k)$, with k fixed, subproblems at merged nodes. Thus, we reduce to the comparison of active nodes with already branched nodes with the requirement of keeping use of a polynomial space. This can also be seen as memorization but with a fixed size memory used to store already explored nodes. This leads to the lost of a reduced worst-case time bound but early works [17] have shown that this can lead to substantially good practical results, at least on some scheduling problems.

As a future development of this work, our aim is twofold. First, we aim at applying the branch-and-merge algorithm to other combinatorial optimization problems in order to establish its potential generalization to other problems. Second, we want to explore the pratical efficiency of this algorithm on the single machine total tardiness problem and compare it with relaxed implementation where a node comparison procedure is implemented with a fixed memory space used to store already branched nodes, in a similar way than in [17].

References

1. H.L. Bodlaender, F.V. Fomin, A.M.C.A. Koster, D. Kratsch and D.M. Thilikos (2012), "On Exact Algorithms for Treewidth", *ACM Transactions on Algorithms*, 9 (1), article 12, 23 pages.
2. F.V. Fomin and D. Kratsch, "Exact exponential algorithms", **Springer Science & Business Media**, 2010.
3. F. Della Croce, R. Tadei, P. Baracco and A. Grosso (1998), "A new decomposition approach for the single machine total tardiness scheduling problem", *Journal of the Operational Research Society* 49, 1101–1106.
4. J. Du and J. Y. T. Leung (1990), "Minimizing total tardiness on one machine is NP–hard", *Mathematics of Operations Research* 15, 483–495.
5. H. Emmons (1969), "One-machine sequencing to minimize certain functions of job tardiness", *Operations Research* 17, 701–715.
6. D. Eppstein (2001), "Improved algorithms for 3-coloring, 3-edge-coloring, and constraint satisfaction"'. In Proc. Symposium on Discrete Algorithms, SODA'01, 329-337.

7. Y. Gurevich and S. Shelah (1987), "Expected computation time for the hamiltonian path problem", *Siam Journal on Computing*, 16, 486-502.
8. C. Koulamas (2010),"The single-machine total tardiness scheduling problem: review and extensions", *European Journal of Operational Research*, 202, 1-7.
9. E. L. Lawler (1977), "A pseudopolynomial algorithm for sequencing jobs to minimize total tardiness", *Annals of Discrete Mathematics* 1, 331–342.
10. C. Lenté, M. Liedloff, A. Soukhal and V. T'Kindt (2013), "On an extension of the Sort & Search method with application to scheduling theory", *Theoretical Computer Science*, 511, 13-22.
11. C. Lenté, M. Liedloff, A. Soukhal and V. T'Kindt (2014), "Exponential Algorithms for Scheduling Problems"', HAL, https://hal.archives-ouvertes.fr/hal-00944382.
12. C. N. Potts and L. N. Van Wassenhove (1982), "A decomposition algorithm for the single machine total tardiness problem", *Operations Research Letters* 5, 177–181.
13. J.M. Robson (1986), "Algorithms for maximum independent sets", *Journal of Algorithms*, 7, 425-440.
14. W. Szwarc (1993), "Single machine total tardiness problem revisited", Y. Ijiri (ed.), Creative and Innovative Approaches to the Science of Management, **Quorum Books, Westport, Connecticut (USA)**, 407–419.
15. W. Szwarc, A. Grosso and F. Della Croce (2001), "Algorithmic paradoxes of the single machine total tardiness problem", *Journal of Scheduling* 4, 93–104.
16. W. Szwarc and S. Mukhopadhyay (1996), "Decomposition of the single machine total tardiness problem", *Operations Research Letters* 19, 243–250.
17. V. T'kindt, F. Della Croce and C. Esswein (2004). "Revisiting branch and bound search strategies for machine scheduling problems", *Journal of Scheduling* 7(6), 429–440.
18. A. Tucker (2012), "Applied combinatorics" 6th Edition, **New York: Wiley**.
19. G.J. Woeginger (2003), "Exact algorithms for NP-hard problems: a survey". In M. Juenger, G. Reinelt, and G. Rinaldi, (eds.) Combinatorial Optimization - Eureka! You shrink!, volume 2570 of Lecture Notes in Computer Science, 185-207, **Springer-Verlag**.