An Exact Node-Merging Algorithm for the Single Machine Total Tardiness Problem : Experimental Considerations

Lei Shang[1], Michele Garraffa[2], Federico Della Croce[3] and Vincent T'kindt[1]

[1] Université François-Rabelais de Tours, Laboratoire d'Informatique (EA 6300)
ERL CNRS OC 6305, 64 avenue Jean Portalis, 37200 Tours, France
shang, tkindt@univ-tours.fr

[2] LIPN CNRS (UMR7030), Université Paris 13, Sorbonne Paris Cité
99 Avenue J-B Clément, 93430 Villetaneuse, France
garraffa@lipn.univ-paris13.fr

[3] Politecnico di Torino, DAUIN, Corso Duca degli Abruzzi 24, 10129 Torino, Italy
federico.dellacroce@polito.it

Keywords: exponential algorithms, branch and merge, single machine total tardiness.

1 Introduction

The design of exact exponential algorithms with worst-case running time guarantee for NP-hard problems has always been a challenging issue. For NP-hard scheduling problems, the results are particularly limited (see [5]). Here, we tackle the single machine total tardiness $1||\sum T_j$ problem where a jobset $N = \{1, 2, \ldots, n\}$ must be scheduled on a single machine. For each job j, a processing time p_j and a due date d_j are given. The problem asks for arranging the jobset in a sequence $S = (a_1, ..., a_n)$ so as to minimize $T(N, S) = \sum_{j=a_1}^{a_n} \max\{\sum_{i=a_1}^{j} p_i - d_j, 0\}$. The problem is NP-hard in the ordinary sense and has been extensively studied in the literature. The current state-of-the-art exact method in practice is a *Branch & Bound* algorithm which solves to optimality problems with up to 500 jobs [6]. However, considering the worst-case time complexity, the best we can have is still the conventional *Dynamic Programming* algorithm which runs in $\mathcal{O}^*(2^n)$ in time and space. Latest theoretical developments for the problem can be found in the survey of Koulamas [3].

Recently we have discovered some interesting structural properties in the branching tree of the problem and an exact algorithm called *Branch & Merge* (BM) has been proposed and proved to have a worst-case time complexity converging to $\mathcal{O}^*(2^n)$ with a polynomial space complexity [1]. In this paper we extend this theoretical result with some experimental feedback. Some insights on the characteristics of hard instances are also provided.

2 Branch & Merge

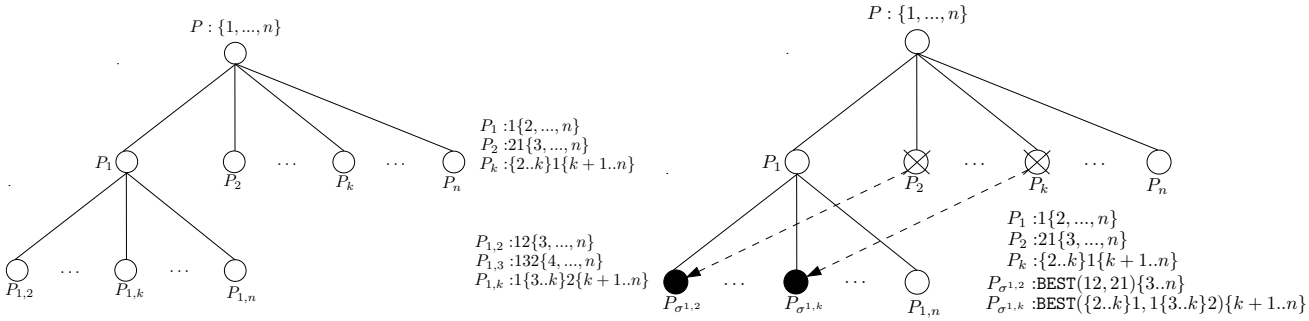
We first recall the core part of BM [1] which relies on Property 1. Let $(1, 2, \ldots, n)$ be a LPT (Longest Processing Time first) sequence and $([1], [2], \ldots, [n])$ be an EDD (Earliest Due Date first) sequence of all jobs.

Property 1. *[4] Let job* 1 *in LPT sequence correspond to job* $[k]$ *in EDD sequence. 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 1 naturally implies a *Branch & Reduce* (BR) algorithm [2]: branching the longest job on every possible positions so as to decompose the initial problem into smaller subproblems. In the resulting search tree, the authors found that many identical subproblems exist on specific

tree positions and these subproblems can be "merged" systematically at each node to avoid solving the same problems multiple times. The *Merge* decision is based on the dominance relation between two search tree nodes sharing the same jobset of fixed jobs.

As an example, Figure 1 illustrates *Merge* on a worst-case instance where LPT=EDD, i.e. the longest job (job 1) can be branched on any position. Node $P_2 = (21\{3..n\})$, consisting of a subproblem with jobs $\{3..n\}$ to schedule after jobs 2 and 1, can be merged with $P_{1,2} = (12\{3..n\})$. The resulting node $P_\sigma^{1,2}$ will be P_2 if the sequence (21) has a lower total tardiness than (12). Note that we know exactly the position of nodes to merge, which are pretty close in the search tree and therefore allow the *Merge* operations to be performed in polynomial time and space. Note that the example here only concerns some left branches in the tree while *Merge* operations can also be performed on right branches with a more complex structure.



(a) Before *Merge*

(b) After *Merge*

FIG. 1: Example of Branch & Merge on some left branches

3 Experimental results

The whole mechanism of BM has been implemented and tested on instances generated in the same way as in [6]. Before comparing it to the state-of-the-art algorithm in [6], we first describe the latter one more clearly. That algorithm, named BR-SPM, is based on the branching structure of BR, with the following three extra features integrated.

1. *Split*, which decomposes a problem according to precedence relations;
2. *PosElim*, which eliminates non-promising branching positions before each branching;
3. *Memorization*, which avoids solving a problem more than once by saving its solution to a database (the "memory") and retrieve it whenever the same problem appears again.

We implemented BR-SPM in the following way: on a given problem P, BR-SPM first tries to find the solution in the *memory*; if failed, it applies *Split* to decompose P into subproblems then solve them recursively; if *Split* does not decompose, *PosElim* is called which returns the list of positions on which the longest job can be branched on. The branching then occurs and the resulting subproblems are then solved recursively. Each time we solve a subproblem by branching, the solution sequence is saved to the memory for further query. Our implementation of BR-SPM successfully solves instances with up to 800 jobs in size, knowing that the original program was limited to instances with up to 500 jobs due to memory size limit.

We now provide our experimental results. In order to verify the concept of *Merge* without extra features, we first compare BR with BM on the hardest subset of instances of each size. Table 1, depicting (minimum, average and maximum) CPU time, average number of *Merge* hits and total number of explored nodes, shows that the *Merge* mechanism strongly accelerates the solution. However BM is still limited to 50 jobs in size only.

To improve the performances, we enable *Split* and *PosElim*. The resulting algorithms are called BR-SP and BM-SP. Now both algorithms can handle instances with up to 300 jobs (see

	TMin	TAvg	TMax	#Merge	#Nodes
BR	52.0	1039.0	3127.0	0	1094033204
BM	3.0	67.6	319.0	11277311	47143367

TAB. 1: Results for instances of size 40

Table 2). Surprisingly, however, even with a considerable number of merged nodes, BM-SP turns out to be slower than BR-SP.

	TMin	TAvg	TMax	#Merge	#Nodes
BR-SP	504.0	3000.8	7580.0	0	634569859
BM-SP	521.0	3097.9	7730.0	608986	508710322

TAB. 2: Results for instances of size 300

Auxiliary tests show that *Split* and *PosElim* negatively affect the *Merge* mechanism. Solving a small problem by *Split*, which sometimes finds directly the solution sequence according to precedence relations, may be faster than *Merge* two nodes. *PosElim* is also powerful as the average number of branching positions at each node after its application is approximately 2, i.e. most positions are already eliminated before *Merge*. This implies that the search tree explored by BR-SP may be even smaller than a binary tree, as there are also many nodes that are not counted: they only have a single child node. These observations show that it is not straightforward to combine directly the current *Merge* scheme with existing solving techniques. The theoretical effectiveness and running time guarantee of *Merge* stays valid, however we need to find a new way to apply it in practice.

4 Merge vs Memorization

The fundamental idea of *Memorization* is similar to *Merge*: avoid solving a problem more than once. However, it is applied in a different way. The idea of *Memorization* can be summarized as "do not solve a problem that was already solved: just retrieve the solution from database", while *Merge* requires "not to solve a problem if it is dominated by another problem on the subset of jobs that have already been fixed". *Merge* is very structural: the search tree nodes attacked by *Merge* are located on specific positions in the tree; *Memorization*, instead, has no constraints on the search tree nodes positions. The recursive structure of *Merge* allows a detailed complexity analysis of BM yelding a worst-case running time guarantee converging to $\mathcal{O}^*(2^n)$ and requiring polynomial space; besides, the time complexity of BR-SPM is $\mathcal{O}^*(2.4143^n)$ [1] and its space requirement is exponential. If exponential memory is allowed, *Memorization* tends to cover *Merge* in the sense that any two nodes that can be merged can also be avoided to be solved twice by applying *Memorization*. and this implies that the time complexity of BR-SPM may also tend to be $\mathcal{O}^*(2^n)$ or possibly lower. However, the idea of *Merge* might still play a role in the scenario where memory usage is limited. To verify this, some variations of *Merge* are studied.

The main issue is what decision information can we get from the subset of fixed jobs in the search tree of BR-SPM, where, for each search tree node, the emphasis is on the job inducing the branch, which is the longest processing time job. At each branch, two subproblems (the left one and the right one) are generated. Property 2 allows to prune one of them when some conditions are satisfied.

Property 2. *We reuse the notation in Property 1. Let job* 1 *be put in position* h, $\max(k, 2) \leq h \leq n-1$, *hence two subproblems are generated corresponding to job sets* $B_1(h)$ *and* $A_1(h)$, *respectively. Let* s *be the optimal solution of the subproblem induced by jobset* $B_1(h) \cup \{1\}$.

If job 1 *is not the last job in* s*, the current node can be cut, i.e. the subproblem on* $A_1(h)$ *does not need to be solved.*

Proof. Omitted. □

Property 2 reflects the idea of *Merge*: consider the example in section 2, the current node is characterized by $\{2\}1\{3..n\}$. If the optimal sequence of $\{1, 2\}$ is (12), then the current node can be cut. The gain becomes interesting when $\frac{|A_1(h)|}{|B_1(h)|}$ (or $\frac{|B_1(h)|}{|A_1(h)|}$ if $|A_1(h)| < |B_1(h)|$) is large.

We first test this idea on BR-SP, which can be considered as a special case of BR-SPM with the memory size limited to 0. Table 3 shows that the new algorithm BR-SP-Cut does somehow accelerate the solution, with lots of nodes cut. However when *Memorization* is enabled, BR-SPM-Cut becomes less interesting with respect to BR-SPM, since the nodes that are cut could also be solved immediately by fetching the solution from the memory. Therefore BR-SP(M)-Cut are only interesting in memory limited scenarios. We also considered other variations of *Merge/Memorization* that will be discussed at the conference.

	TMin	TAvg	TMax	#Cut	#Nodes
BR-SP	504.0	3000.8	7580.0	0	634569859
BR-SP-Cut	494.0	2968.6	7470.0	2163175	631529558

TAB. 3: Results for instances of size 300

In conclusion, we analysed the branching tree structure of the problem and compared two node fusion techniques, *Merge* and *Memorization*, that avoid solving identical subproblems multiple times. We provide experimental results to show the interest of *Merge* since it can be performed in polynomial time and space. However, *Memorization* tends to be more effective than *Merge* when exponential space is allowed. By carefully combining the strength of memorization to the increased memory sizes of current computers, the size of instances that can be solved to optimality is now pushed up from 500 to 800 jobs, thing which is also an interesting issue to be reported. Some variations of *Merge/Memorization* are currently being considered and we expect to have better results at the time of conference. Besides, as a search tree pruning scheme, *Merge* may also be generalized to other problems and will most probably fit especially to scenarios where polynomial space is required.

References

[1] L. Shang, M. Garraffa, F. Della Croce and V. T'Kindt (2016). "An Exact Exponential Branch-and-Merge Algorithm for the Single Machine Total Tardiness Problem", In Proc. PMS'16, 182-185,Valencia (Espagne).

[2] F. Della Croce, M. Garraffa, L. Shang and V. T'kindt (2015), "A branch-and-reduce exact algorithm for the single machine total tardiness problem", In Proc. MISTA2015, 879-881, Prague, Czech Republic.

[3] C. Koulamas (2010),"The single-machine total tardiness scheduling problem: review and extensions", *European Journal of Operational Research*, 202, 1-7.

[4] E. L. Lawler (1977), "A pseudopolynomial algorithm for sequencing jobs to minimize total tardiness", *Annals of Discrete Mathematics* 1, 331–342.

[5] C. Lenté, M. Liedloff, A. Soukhal and V. T'Kindt (2014), "Exponential Algorithms for Scheduling Problems", HAL, https://hal.archives-ouvertes.fr/hal-00944382.

[6] W. Szwarc, A. Grosso and F. Della Croce (2001), "Algorithmic paradoxes of the single machine total tardiness problem", *Journal of Scheduling* 4, 93–104.