The Memorization Paradigm: Branch & Memorize Algorithms for the Efficient Solution of Sequencing Problems

Lei Shang, Vincent t'Kindt, Federico Della Croce

The Memorization Paradigm: Branch & Memorize Algorithms for the Efficient Solution of Sequencing Problems

Lei Shang[a,*], Vincent T'Kindt[a], Federico Della Croce[b]

[a] *Université François-Rabelais de Tours,*
LI (EA 6300), ERL CNRS OC 6305
64 avenue Jean Portalis
37200 Tours, France
[b] *Politecnico di Torino, DIGEP*
Corso Duca degli Abruzzi 24
10129 Torino, Italy

Abstract

Memorization as an algorithm design technique allows to speed up algorithms at the price of more space usage. Typically, in search tree algorithms, on lower branching levels, isomorphic sub-problems may appear exponentially many times and the idea of Memorization is to avoid repetitive solutions as they correspond to identical sub-problems. The idea exists since a long time but apparently has not been systematically considered when designing branching algorithms. It is at least rare for sequencing problems, to the authors' knowledge.

In this paper, we explore the power of Memorization on solving hard sequencing problems. We first describe a general framework of Memorization with some guidelines provided on the implementation. Then we apply the framework to four sequencing problems including the two-machine flowshop problem minimizing the sum of completion time and three single machine problems whose objective functions to minimize are the total tardiness, the sum of completion time with release date and the sum of weighted completion time with deadline. The global results suggest systematically considering Memorization as a solving block inside search tree based algorithms like Branch and Bound.

Keywords: scheduling, exact algorithms, memorization, sequencing, branch and memorize, total tardiness, sum of completion times, flowshop, single machine

1. Introduction

Memorization as an algorithm design technique allows to speed up algorithms at the price of more space usage. Typically, in search tree algorithms, on lower branching levels, isomorphic sub-problems may appear exponentially many times and the idea of Memorization is to avoid repetitive solutions as they correspond to identical sub-problems. The method was first applied

*Corresponding author

Email addresses: shang@univ-tours.fr (Lei Shang), tkindt@univ-tours.fr (Vincent T'Kindt), federico.dellacroce@polito.it (Federico Della Croce)

on the *Maximum Independent Set* problem by Robson (1986) in 1986. By exploiting graph theoretic properties and by applying *Memorization* to avoid solving identical sub-problems, Robson proposed an algorithm with a worst-case time complexity in $O(1.2109^n)$. It has remained the exact exponential algorithm with the smallest worst-case time complexity until 2013, when it was improved by the $O(1.1996^n)$ algorithm of Xiao and Nagamochi (2013). *Memorization* is sometimes used to speed up search tree algorithms (Chandran and Grandoni, 2005; Fomin et al., 2005; Fomin and Kratsch, 2010) in the context of EEA (*Exact Exponential Algorithms*), where the objective is to conceive exact algorithms that can provide a best possible worst-case running time guarantee.

Despite the fact that a typical *Memorization* algorithm memorizes solutions of sub-problems that appear repeatedly, we prefer to interpret the idea in a more general way.

What we call the *Memorization Paradigm* can be formulated as "Memorize and learn from what you have done so far, to improve your next decisions". In the literature, various algorithms can be classified as procedures embedding memorization techniques, though the implementation could be quite different depending on the problem structure and the information to store. For instance, *Tabu Search* (Glover, 1989, 1990) is a metaheuristic, which memorizes recently visited solutions in order to avoid returning back to these solutions again during the search. SAT solvers deduce and then memorize conflict clauses during the tree search in order to perform non-chronological backtracking (*Conflict Driven Clause Learning*) (Biere et al., 2009; Zhang et al., 2001). Similar ideas also appear in *Artificial Intelligence* area as *Intelligent Backtracking* or *Intelligent Back-jumping*.

From a theoretical point of view, the drawback relies on the memory consumption of *Memorization* which can be exponential.

This drawback turns out to limit the quantity of memorized information like in *Tabu Search* or SAT solvers. In this paper we instantiate the *Memorization Paradigm* in a way similar to what is done in the field of EEA, i.e. we set up a *Memorization* framework for search tree based exact algorithms but with a control on the memory usage. We have the intuition that a *Memorization* with limited memory could already dramatically accelerate the solution in practice. By embedding a simple *Memorization* technique into their *Branch & Bound* algorithm, Szwarc et al. Szwarc et al. (2001) solved the single machine total tardiness problem on instances with up to 500 jobs in size. Other works presenting standard memorization techniques applied to sequencing problems have been done by T'kindt et al. (2004) where the benefit of such technique is well shown.

Search tree algorithms are based on the idea of enumerating all possibilities via a search tree created by a branching mechanism.

For each decision variable, the algorithm *branches* on all possible values, each time creating a new sub-problem (a node in the search tree) of a reduced size. The algorithm continues recursively and returns the global optimal solution. The basic structure being simple, the critical question is how to prune the search tree so as to avoid exploring unpromising nodes. Dominance conditions are commonly used to cut nodes: at a node, if it is proved that a more promising node exists or can be easily found, then the current one can be abandoned. This is also the case for *Branch & Bound*, in which at each node, the bounding procedure provides an optimistic estimation of the solution quality of that node. If the estimation value is not better than the currently best solution found, in other words, the current node is dominated by the incumbent solution, and then the node is cut without being further developed. Just like the bounding procedure in *Branch & Bound*, *Memorization* can be seen as another procedure which can help in pruning the search tree. In branching algorithms, especially on lower branching levels, isomorphic sub-problems

may appear exponentially many times and *Memorization* can be used to avoid solving identical problems multiple times.

Memorization, apparently, has not yet been systematically considered when designing search tree based algorithms, as the bounding procedure in *Branch & Bound*. It is at least rare in sequencing problems, to the authors' knowledge. The aim of this paper is to promote a systematic merge of *Memorization* into search tree based algorithms in order to better prune the search tree. In the following sections, we first describe a general framework of *Memorization* (section 2), followed by some guidelines on the implementation (section 3). Then, we apply the framework to four scheduling problems including $1|r_i|\sum C_i$ (section 4.1), $1|\tilde{d}_i|\sum w_iC_i$ (section 4.2), $F2||\sum C_i$ (section 4.3) and $1||\sum T_i$ (section 5). Finally, we conclude our work in section 6.

2. A general framework for *Memorization* in search trees

For a given minimization problem, the application of *Memorization* depends on several components of the search tree based algorithm such as the branching scheme, the search strategies and also the characteristics of the problem. In this section, we consider possible scenarios that may appear for sequencing problems. Then, we present the possible schemes of *Memorization* and how to choose the right scheme depending on the scenario.

Even though the general idea of *Memorization* can be generalized and applied to any combinatorial optimization problems, in this paper we focus on sequencing problems. Consider a generic sequencing problem where n jobs $J = \{1, ..., n\}$ are to be scheduled. Each job j is defined by a set of features like a processing time p_j, a due date d_j, etc, which depends on the problem under consideration. Some resources are available for the execution of jobs and an ordering of jobs must be found to minimize some cost function, usually depending on jobs' completion times. We adopt an intuitive way to represent the content of a node or a sub-problem: as an example $123\{4, ..., n\}$ represents a sub-problem in which jobs $\{1, 2, 3\}$ are already fixed by branching, to the first three positions of the sequence, while the jobs to be scheduled after are $\{4, ..., n\}$.

No matter the branching scheme, at any iteration of the algorithm, by *active nodes* we denote the nodes that are created but not yet developed, and by *explored nodes* the nodes that have already been branched on (children nodes have been created). We also adopt the notion of *decomposable problems* defined by T'kindt et al. (2004). Typically, for single machine scheduling problems, this often implies that the completion time of the prefixed job sequence of a node is constant no matter of the order of jobs inside (it is defined as the sum of processing times of the jobs in that sequence).

Definition 1. *Let $\{1, ..., i\}\{i + 1, ..., n\}$ be a problem to be solved. It is decomposable if and only if the optimal solution of the sub-problem $\{1, ..., i\}$ (resp. $\{i + 1, ..., n\}$) can be computed independently from $\{i + 1, ..., n\}$ (resp. $\{1, ..., i\}$), i.e. without knowing the optimal sequence of $\{i + 1, ..., n\}$ (resp. $\{1, ..., i\}$).*

2.1. Branching schemes

In common search tree based algorithms for scheduling (sequencing) problems, the branching operation consists in assigning a job to a specific position in the sequence. A *Branching Scheme* defines, at a node, how to choose this job and the positions to occupy.

We consider three classic branching schemes, namely *forward branching*, *backward branching* and *decomposition branching*.

When applying *forward branching* at a given node, each eligible free job is assigned to the first free position. For example, the nodes at the first level of the search tree correspond to the following sub-problems: $1\{2, ..., n\}$, $2\{1, 3, ..., n\}$, ..., $n\{1, ..., n-1\}$.

When applying *backward branching* at a given node, each eligible free job is assigned to the last free position. For example the nodes at the first level of the search tree correspond to the following sub-problems: $\{2, ..., n\}1$, $\{1, 3, ..., n\}2$, ..., $\{1, ..., n-1\}n$. This scheme is symmetric with *forward branching*, hence for the sake of simplicity we only discuss *forward branching* in this paper and add extra remarks on *backward branching* whenever necessary.

When applying *decomposition branching* at a given node, the job that is being considered to branch is called a *decomposition job*. When a *decomposition job* is assigned to a position, two sub-problems are generated implied by the free positions before and after the *decomposition job*. Certainly one may determine the jobs that should be scheduled before and after this position by enumerating all 2-partitions of jobs as the *Divide & Conquer* technique introduced by Fomin and Kratsch (2010), but here we restrict our study to the situation where the two sub-problems can be uniquely determined in polynomial time making use of some specific problem properties. As an example, the nodes at the first level of the search tree could contain $\{2, 3, 4\}1\{5, ..., n\}$, if job 1 is the *decomposition job* which is assigned to position 4 and generates two sub-problems corresponding to jobs $\{2, 3, 4\}$ and $\{5, ..., n\}$, respectively. This situation occurs, for instance, to

the $1||\sum T_i$ problem which will be tackled later on.

2.2. Search strategies

During the execution of search tree based algorithms, when two or more nodes are active, a strategy is needed to determine the next node to branch on. The classic search strategies are *depth first*, *best first* and *breadth first*.

Depth first is the most common strategy: the node to explore is an arbitrary active node at the lowest search tree level. The advantage of this strategy is that it only requires polynomial space.

Breadth first selects an active node with the highest search tree level. This leads to an exponential space usage since the search tree is explored level by level.

Best first

chooses the node to explore according to its lower bound. The space usage in the worst case is therefore also super-polynomial like in *breadth first*.

It seems conventional that when constructing a search tree based algorithm, the *depth first* is adopted. However, this choice is strongly questionable according to T'kindt et al. (2004).

2.3. Memorization schemes

The *memorization* presented by Robson (1986) stores the optimal solution of each sub-problem of a predetermined limited size and reuses that solution whenever such sub-problem appears again during the tree search. However different memorization approaches can be used. The differences rely on the choice of the information to store and the way in which the stored information is used. We discuss below three different memorization schemes that are helpful to efficiently solve some sequencing problems.

Taking account of the branching schemes introduced in section 2.1, any node of the search tree can be defined by $\sigma_1 S_1 \sigma_2 S_2 ... \sigma_k S_k$, the $\sigma_j{}'s$ being partial sequences of jobs and the $S_j{}'s$ being sub-problems which remain to be scheduled. For the sake of simplicity, we explain the *Memorization* schemes in the case of *forward branching*, i.e. $k = 1$, and a node corresponds to a problem σS.

2.3.1. Solution memorization

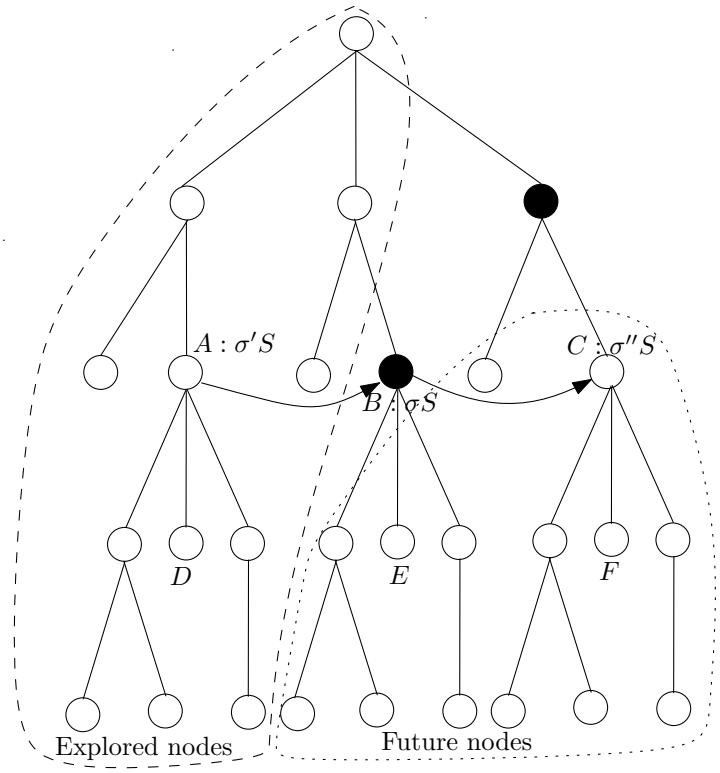


Figure 1: Solution Memorization

Consider the situation illustrated in Figure 1, where active nodes are colored in black. Node B is the current node, while σ, σ' and σ'' are different permutations of the same jobs. In other words, nodes A, B and C may contain the same sub-problem to solve, implied by S. In that case, if A has already been solved (consider for instance a depth-first search) and the optimal sequence of S has been memorized, then it can be used directly to solve nodes B and C and it is no longer necessary to branch on these nodes.

Note that, in order to successfully perform memorization, we must guarantee that the solution of S memorized at node A is optimal. Depending on the branching algorithm implementation, this may not be obvious: for instance in *Branch & Bound* algorithms, the leaf node corresponding to the optimal solution of node A may be missed if one of its ascendant node is cut due to a dominance condition. Looking at Figure 1, assume that node D should have led to the optimal solution of problem S but has been cut by a dominance condition. Applying *solution memorization* may then lead to memorize another solution β to S, which is not optimal with respect to S. Troubles may appear if the global optimal solution to the original problem (associated to the root node) is, for instance, given by node E. Solution memorization may imply not exploring node B and directly replacing S by the "best" solution found from node A. As a consequence, the global optimal solution is missed. This situation occurs whenever the dominance condition which has pruned node D would not have pruned node E: in the remainder, this kind of conditions are

refined to as *context dependent dominance conditions* since they depend on the context of each node (typically, the initial partial sequence σ, σ' and σ''). By opposition, a *context independent dominance condition* would have pruned node D, E and F.

A direct way to fix this is to disable dominance conditions whenever *solution memorization* is applied.

However, if these context dependent conditions are playing a very important role in the algorithm then this may slow down the algorithm even if *solution memorization* works. Another approach to manage context dependent dominance conditions is to extend the memorization from "solutions" to "lower bounds" when the branching algorithm involves a bounding mechanism. In that version of *Memorization*, we assume that all dominance conditions are kept in the algorithm. When node A is created, a lower bound is computed, which represents the best solution value we may expect from the sub-tree of A. This lower bound is based on the cost function value of the sequence σ' which is already fixed, and an evaluation on the unsolved part S. When branching down the sub-tree of A, jobs in S are fixed gradually, hence the evaluation on the remaining unscheduled jobs also becomes more and more precise. When all leaf nodes of the sub-tree of A are explored, this value finally becomes tighter (higher) than the initial value computed at node A. Since the objective function value of σ' is known, we can then deduce the lower bound value corresponding to S when scheduled after σ', and memorize it. Now when node B is opened, instead of computing its lower bound, we can get it by finding the lower bound of S directly from the memory and then add the objective function value of σ. In this way, the lower bound we get is tighter, and node B is more likely to be cut. Moreover, the lower bound computation at node B, which may be time costly, is saved. Notice that, for nodes cut by context dependent dominance conditions, their lower bound values still need to be computed and considered (hence introduces an extra cost). Lower bound memorization can be a good alternative to *solution memorization* with context dependent dominance conditions turned off as long as these conditions are efficient in pruning the search tree.

Note that the memorization of lower bounds is compatible with the memorization of optimal solutions: whenever in a sub-tree no nodes are cut by context dependent dominance conditions and the global upper bound is updated by some nodes from this sub-tree, the optimal solution of this sub-tree is memorized. Otherwise, the lower bound is memorized. We denote the described memorization technique including the memorization of optimal solutions and the memorization of lower bounds as *solution memorization* since both are related to the memorization of the "best solution" of the problem associated to a node.

2.3.2. Passive node memorization

At any node σS, another information that can be memorized is the partial sequence σ. Unlike *solution memorization* where the memorized sequences can be used to solve a node, *passive node memorization* is only used to cut nodes.

Consider the branching situation depicted by Figure 2. Again active nodes are black-colored and B is the current node. Assume a node A exists among explored node, with σ' being a different permutation of the same jobs used in σ. If the partial sequence σ' has been memorized then two situations may occur. If σ' dominates σ then B can be cut since it cannot lead to a solution better than A. If no such σ' dominating σ is available, then σ can be memorized in order to possibly prune a future node like C. Note that *solution memorization* and *passive node memorization* may possibly intersect. Consider the previous example and nodes A, B and C. If the optimal solution of sub-problem S has been obtained from the exploration of node A, then at node B and C both

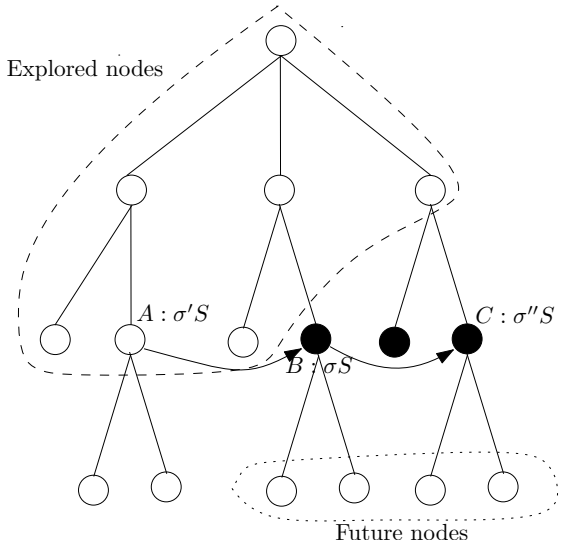


Figure 2: Passive node memorization)

solution memorization and *passive node memorization* imply not to branch on these nodes if σ' dominates σ and σ''.

The dominance test between sequences can be implemented as a function $check(\sigma, \sigma')$ which returns 1 if σ' dominates σ, as introduced by T'kindt et al. (2004). The check must be done on two different sequences of the same jobs, having the same starting time and its implementation is problem dependent. Since the memorized sequence results from branching decisions, we call it *passive node memorization*.

Any node, ready to be branched on, must be compared to explored and/or to active nodes depending on the search strategy. Additionally, it may be necessary to perform the check twice:

first once the node is created, then at the time of branching. Memorizing the partial sequence, when the node is created, ensures that the best sequence is kept before any exploration of nodes. Then, rechecking the dominance when branching on a node enables that node to be cut if a dominant partial sequence has been found meanwhile.

In the following, we introduce Definition 2 which relates the lower bounding mechanism of search tree based algorithms to the *check* function. When this test is verified, the current node only needs to be compared to explored nodes instead of all nodes when best first is chosen as the search strategy, as detailed in Section 2.4.

Definition 2. *(Concordance Test) Let LB(A) be the lower bound value computed at node A. The search tree based algorithm satisfies the concordance property if and only if, for any node* $A = \sigma S$ *and* $B = \pi S$, $LB(A) < LB(B) \Leftrightarrow check(\pi, \sigma) = 1$.

2.3.3. Predictive node memorization

Predictive node memorization relies on the same idea as *passive node memorization*, but with additional operations. As illustrated in Figure 3, at a given node $B = \sigma S$, we first check, like in *passive node memorization*, if the current node can be cut by σ' memorized at node A. If not, instead of directly memorizing σ, we search for an improving sequence π. Notice that, by the

way, the improving sequence necessarily belongs to a part of the search tree not yet explored when dealing with the node σS. There may be many ways to compute π. For instance, we may perform some local search on σ, searching for a neighbor sequence π that dominates σ. Alternatively, we may focus on a short sub-sequence of σ and solve it to optimality (in a brute-force way, for instance). The latter idea appears as *Dominance Rules Relying on Scheduled Jobs* (Jouglet et al., 2004). We may also make use of an exact algorithm to optimize a part of σ to get σ', as far as this algorithm is fast. Notice that this idea is strongly related to the theoretical mechanism called *merging* (Shang et al., 2017)and designed to provide good worst-case time complexities. If such a sequence π can be built, then the current node σS is cut and node πS is memorized. Note that node πS has not yet been encountered in the search tree when dealing with

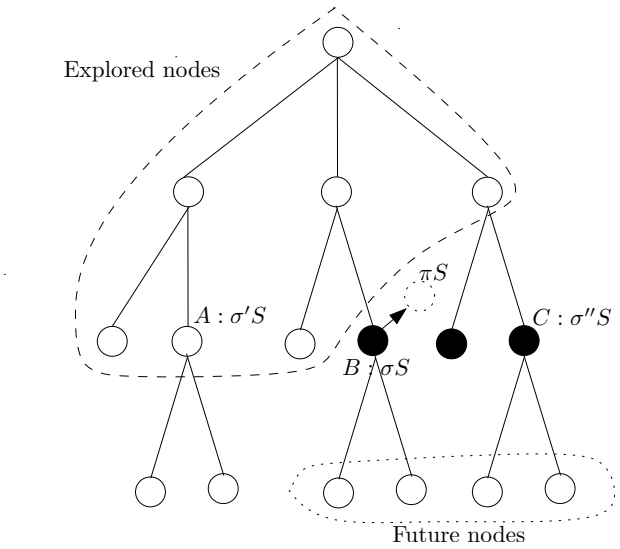


Figure 3: Predictive node memorization

node σS (consider, $\pi = \sigma''$). So, it is important when applying *predictive node memorization* to remember that πS still needs to be branched on.

Also, the extra cost of generating π must be limited in order to avoid excessive CPU time consumption.

2.4. Decision guidelines

In this section we provide some guidelines on how to choose the appropriate memorization scheme according to the branching scheme and the search strategy. The main results are summarized in the decision tree in Figure 4.

2.4.1. Forward branching and depth first search strategy

In *forward branching*, any node of the search tree can be defined as σS. When *depth first* is used as the search strategy we can state the following property.

Property 1. *With forward branching and depth first, if the problem is decomposable and solution memorization memorizes optimal solutions, then solution memorization dominates both passive node memorization and predictive node memorization.*

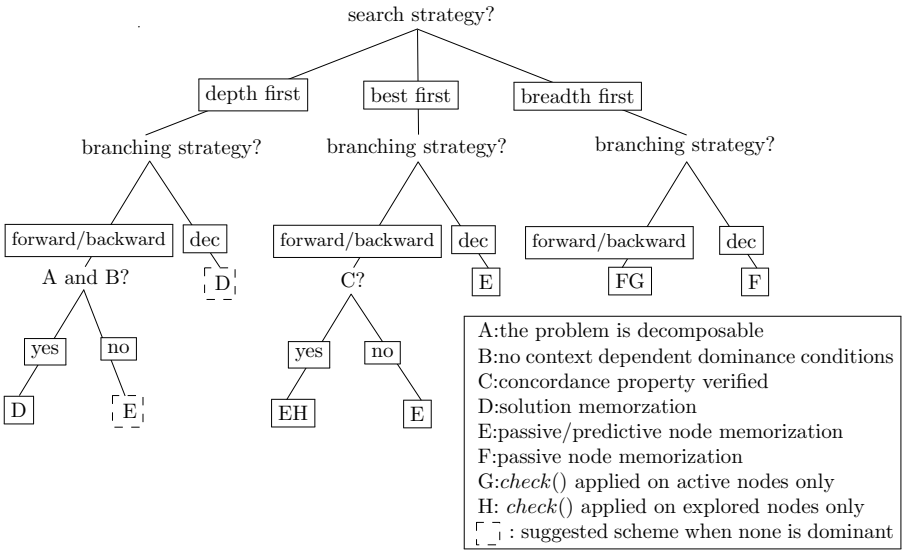


Figure 4: Decision tree for choosing the memorization scheme

Proof. Any node deletion that can be achieved by *passive node memorization* and *predictive node memorization* can also be achieved by *solution memorization*, but not conversely. Consider nodes $A = \sigma S$ and $B = \pi S$ with σ and π two permutations of the same jobs. As the problem is decomposable, solving sub-problem S at node A is equivalent to solving it at node B. Without loss of generality, we assume that A appears before B during the solution. In *passive node memorization* if $check(\pi, \sigma) = 1$, i.e. sequence σ dominates π, then B can be pruned. However, also in *solution memorization*, node B can be pruned since the optimal solution of jobs S has already been memorized from node A.

Now consider the case where $check(\sigma, \pi) = 1$, i.e. sequence π dominates σ. This implies that with *passive node memorization*, node B will not be pruned. However, as explained above, with *solution memorization*, node B is pruned. With *predictive node memorization*, the conclusion is the same since we have no guarantee that starting from node πS another node αS dominating πS can be generated. Besides, even if such αS is generated and πS is pruned, the same issue occurs to node αS when it is generated.

□

If the problem is not decomposable, or *context dependent dominance conditions* are used in the algorithm, then *solution memorization* memorizes lower bounds, and then which memorization scheme is dominant cannot be determined. However, in practice, *passive node memorization* may be preferred to *solution memorization*. Notably, as the problem is not decomposable, then it may be necessary to solve the sub-problem consisting of jobs S both at nodes A and B. However, with *passive node memorization*, node B may be pruned whenever π is dominated by σ.

2.4.2. Forward branching and best first search strategy

We can state the following property.

Property 2. *With forward branching and best first strategy, solution memorization does not apply. Passive node memorization or predictive node memorization can be applied only to explored nodes if the Concordance Test (Definition 2) is answered.*

Proof. To apply *solution memorization* at a given node, the sub-problem concerning S must be solved first in order to memorize its optimal solution. This is not compatible with *best first* search strategy. In fact, if *best first* reaches a leaf node, then also the optimal solution is reached and no sequence has been stored before.

When *passive node memorization* and *predictive node memorization* are applied, the search, at a given node, of a dominant sub-sequence needs only to be done in the set of explored nodes whenever the concordance property holds. As the *best first* search strategy always consider for branching the node with the lowest lower bound value, the concordance property implies that no active node can dominate it. □

When the concordance property does not hold, then node memorization techniques are required to consider both explored and active nodes for node pruning.

Besides, no dominance can be deduced a priori between *predictive node memorization* and *passive node memorization*. It depends on how the search for an improving sub-sequence is applied in *predictive node memorization*. Generally speaking, both memorization schemes should be considered and compared to find the best one.

2.4.3. Forward branching and breadth first search strategy

With forward branching and breadth first we can state the following property.

Property 3. *With forward branching and breadth first strategy, solution memorization does not apply. Passive node memorization should be chosen and should be applied to active nodes.*

Proof. Under this configuration, *solution memorization* is useless since leaf nodes are reached only at the end of the search tree. *Passive node memorization* can be applied to *active nodes* only. An active node A is selected for branching when all the nodes at the same level have been created, hence all other active nodes dominated by A are discarded. If in turn A it is dominated by another node, then it is pruned. There is no need to consider explored nodes since explored nodes on higher levels have less fixed jobs, therefore they are not comparable with the current node.

Also, *predictive node memorization* cannot do better than *passive node memorization* since *passive node memorization* already keeps the best node at each level.

□

2.4.4. Decomposition branching and depth first search strategy

With *decomposition branching*, at each level of the search tree a decomposition job can be put on any free position by the branching operation.

Under this configuration, no dominance can be deduced among the memorization schemes. In fact, we can imagine situations where either *solution memorization* or *passive node memorization* or *predictive node memorization* is dominant. Consider nodes $A = \sigma S_1 j_1 S_2$ and $B = \pi S_1 j_2 S_3$ with A being explored before B. In both nodes, the current sub-problem concerns scheduling jobs S_1 after σ or π. Suppose σ and π contain different jobs but have the same completion time, which means that the sub-problem defined by S_1 is identical in A and B. Then,

the optimal sequence for S_1 found when solving A can be reused on B by *solution memorization*, while *passive node memorization* cannot handle this case since σ and π contain different jobs hence, are incomparable. *Predictive node memorization* may or may not cut B depending whether a dominant prefix can be generated or not.

On the other hand, we may also imagine the case where $A = \sigma S_1 j_1 S_2$ and $B = \pi S_3 j_2 S_4$. Suppose σ and π are different permutations of the same jobs. If $check(\pi, \sigma) = 1$, then node B can be cut by *passive node memorization* or *predictive node memorization*, while this is not the case for *solution memorization* because sub-problems S_1 and S_3 do not consist of the same jobs.

In practice, even though every memorization scheme could be dominant in some cases, the memory limitation does not allow applying all of them and our experience suggests preferring *solution memorization*. This is due to the special structure of nodes $\sigma_1 S_1 ... \sigma_k S_k$, which makes the prefixed jobs much spread out (they are separated by S_i), and prevents the application of successful *passive node memorization* and *predictive node memorization*. Moreover, the case with nodes $\sigma_1 S \sigma_2$ and $\pi_1 S \pi_2$, where σ_1 and π_1 have the same completion time but contain different jobs, may occur pretty often for large instances if the jobs processing times do not present a large variance.

2.4.5. Decomposition branching and best first search strategy

Property 4. *With decomposition branching and best first strategy, solution memorization does not apply. Passive node memorization and predictive node memorization must only be applied to explored nodes whenever the concordance property holds and the check function comparing two nodes $\sigma_1 S_1 ... \sigma_k S_k$ and $\sigma_1' S_1' ... \sigma_{k'}' S_{k'}'$ only works on σ_1 and σ_1'. Otherwise, passive node memorization and predictive node memorization must be applied to explored and active nodes.*

Proof. Similar to that of Property 2.

□

2.4.6. Decomposition branching and breadth first search strategy

Property 5. *With decomposition branching and breadth first strategy, solution memorization does not apply. Whether node memorization should be applied to active nodes only depends on the definition of the check function.*

Proof. This configuration discourages *solution memorization* for the same reason as in Property 3. If the *check* function is defined in a way such that the explored nodes are not comparable to active nodes, then *passive node memorization* and *predictive node memorization* should be applied to *active nodes* only, otherwise they should be applied to all nodes. □

3. Implementation guidelines

In this section we discuss efficient implementations of the memorization schemes, providing, when necessary, choices specific to the sequencing problems tackled in the remainder. The key point is to have fast access to memorized partial solutions. Henceforth, we implement a database as a hashtable which contains all the memorized solutions.

By well choosing the hash function, a hashtable supports querying in $O(1)$ time to find the corresponding elements given a hash key.

For *solution memorization*, at a given node, the database is queried with $\langle t_0, S \rangle$, where t_0 is the starting time of the sub-problem and S is the related jobs. The returned result should

be $\langle \pi, opt(\pi|t_0) \rangle$ which is the optimal sequence associated to S when starting at time t_0, and its corresponding objective function value. So, $\langle \pi, opt(\pi|t_0) \rangle$ defines the elements which are memorized in the database. We define the hash key h as a combination of t_0 and $|S|$: seeing h as a set of bits, t_0 occupies the higher bits in h while $|S|$ occupies the lower bits. The aim is to have a unique hash key for each given pair $\langle t_0, S \rangle$, even if this is not necessarily bijective: i.e., two elements in the database with the same hash key may correspond to different pairs $\langle t_0, S \rangle$. As a consequence, when a list of elements is returned for a pair $\langle t_0, S \rangle$, it is also necessary to verify that the returned sequence is a sequence of jobs S. This takes $O(|S|)$ operations for each returned sequence. We may also include the sum of job id's of S into h in order to have a more exact key, but this correspondingly increases the time needed to construct the key, without preventing from checking whether a returned sequence π is a permutation of jobs S or not.

For *passive* and *predictive node memorization*, implementation decisions are more dependent on the problem and on the check function used to compare two partial sequences σ and π of the same jobs. For any such σ and π, a general definition of *check*() could be:

$$check(\pi, \sigma) = \begin{cases} 1, \ if \ C_{max}(\sigma) \leq \max(C_{max}(\pi); E_{min}(\pi)) \ and \ opt(\sigma|t_0) \leq opt(\pi|t_0) \\ 0, \ otherwise \end{cases} \quad (1)$$

with C_{max} referring to the makespan of a partial sequence, and $E_{min}(\pi)$ referring to the earliest starting time of the jobs scheduled after π. It is not difficult to see that if $check(\pi, \sigma) = 1$ then node σS dominates node πS. Indeed, for any regular objective function to minimize, with respect to the fixed jobs, $opt(\sigma|t_0) \leq opt(\pi|t_0)$ ensures that σ yields a smaller cost than π. Moreover, $C_{max}(\sigma) \leq \max(C_{max}(\pi); E_{min}(\pi))$ guarantees that the starting time of jobs S at node σS is not higher than in node πS. Therefore, σS dominates πS.

Consequently, an element of the database is a tuple $\langle \sigma, C_{max}(\sigma), E_{min}(\sigma), opt(\sigma|t_0), ExpAct \rangle$ with $ExpAct$ being a flag indicating whether this element corresponds to an explored or an active node. Notice that t_0 is not included since it appears in the hash key used for querying. Also, when the problem is decomposable, the *check* function reduces to:

$$check(\pi, \sigma) = \begin{cases} 1, \ if \ opt(\sigma|t_0) \leq opt(\pi|t_0) \\ 0, \ otherwise \end{cases}$$

where only $\langle \sigma, opt(\sigma|t_0), ExpAct \rangle$ need to be stored. For node memorization techniques the hash key, at a given node, is computed in a way similar to solution memorization. Consider, for example, *forward branching*: let $\sigma_1 S_1$ be the current node. As the dominance of another node is checked on σ_1, the database is queried with $\langle 0, S_{\sigma_1} \rangle$ with S_{σ_1} referring to the set of jobs in σ_1. Then, only S_{σ_1} needs to be binary encoded into the hash value.

With respect to the database management, notice that when an element is added, in node memorization techniques, then the elements dominated by the added one are removed. Besides, due to memory limitation on the computer used for testing, we may need to clean the database when it is full on some instances. More precisely, in our experiments, the RAM is of 8Gb and hence the database size is also limited to 8Gb.

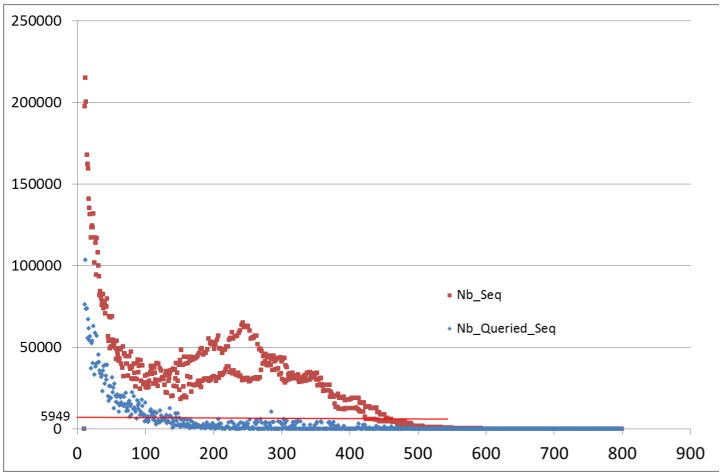
A cleaning strategy is needed to remove unpromising elements, i.e. those that are expected not to be used for pruning the search tree. As it is not clear which elements are unpromising, several strategies have been tested. We have implemented the following ones during our experimentation.

FIFO: First In First Out

This is one of the most common database cleaning strategy: when the memory is full, we first

remove the first added elements. An extra structure is needed to record the order of elements according to the time when they are added. When the database if full and a long sequence is waiting to be inserted, it may be necessary to remove more than one elements in order to free enough space.

BEFO: Biggest Entry First Out
This cleaning strategy suggests removing from the database the biggest elements (the longest sequences) in order to free enough continuous memory for storing new elements. For *solution memorization* it means removing nodes at higher levels in the search tree. An intuition of the impact of this cleaning strategy on *solution memorization* can be sketched from Figure 5 which presents the number of sequences memorized per size for an instance of the $1||\sum T_i$ scheduling problem with 800 jobs. It can be seen that sequences with "large number of jobs" (let's say more than 500 jobs) are not often used to prune nodes, and even if some large nodes could have been useful for node pruning, we may still expect that the solution of its sub-problems generated by one or several branching can be found from the memory.



Nb_Seq: number of sequences of a given size, stored in the memory.
Nb_Queried_Seq: number of sequences of a given size that are used to avoid solving twice identical problems.

Figure 5: Number of solutions and useful solutions in memory for an instance of $1||\sum T_i$ with 800 jobs

However, for *passive and predictive node memorization*, the strategy means removing nodes at lower levels of the search tree. These nodes refer to sub-problems with many jobs already fixed (and memorized) and few jobs to schedule. It may be possible that the extra cost of memorizing a long fixed partial sequence is not inferior to solving the corresponding small sub-problem directly without memorization. Since this is not obvious from a theoretic point of view, some preliminary experiments were performed in order to investigate whether it is better also to remove nodes at higher levels of the search tree in *node memorization*. Computational testing confirms that removing the longest elements is always preferred, at least on problems $1|r_i|\sum C_i$, $1|\tilde{d}_i|\sum w_i C_i$ and $F2||\sum C_i$.

At each cleaning, we also tend to clean up a large amount of space in order to decrease the time cost induced by the cleaning operation itself.

LUFO: Least Used First Out

Figure 5 also suggests another cleaning strategy since a lot of sequences are never used to prune nodes in the search tree. These sequences can be removed from the database to save space. To implement the LUFO cleaning strategy we keep a usage counter for each database element. The counter is incremented by 1 each time the element is queried and used to prune a node in the search tree, and it is decremented by 1 when a cleaning operation is performed. Elements whose counter is zero are removed by the cleaning operation. Note that in *node memorization*, when a database element is replaced by a new one, the latter should inherit the counter value of the old one. This is because that the counter value reflects the usefulness of a solution and the counter value of a newly added solution should not be smaller than the counter values of solutions that are dominated by the new one.

Preliminary results, not reported here, show that FIFO strategy is not efficient for the considered scheduling problems. BEFO strategy works better than FIFO, but its efficiency is not high enough to make a difference in the computational results. LUFO strategy is proved to be surprisingly efficient.

4. Application to the $1|r_i|\sum C_i$, $1|\tilde{d}_i|\sum w_iC_i$ and $F2||\sum C_i$ problems

In order to experiment the effectiveness of *Memorization* on scheduling problems, we first test it on three problems that were considered by T'kindt et al. (2004). In that work the authors used memory to apply the so-called *DP property* over nodes in order to prune the search tree. According to the memorization framework as defined in this paper, what they have done is *passive node memorization* with a database cleaning strategy which replaces the shortest stored sequence by the new one when the database if full.

The aim of that paper was also on choosing the most suitable search strategy when trying to solve these problems efficiently. In this section, for each of these three problems we apply the previously defined framework of *Memorization* with various considerations and discuss the obtained results. For each problem, we compare several *Branch & Bound* algorithms which are named according to their features: Depth-, Best- and Breadth- refer to *Branch & Bound* algorithms with the corresponding search strategies and no memorization included. Depth_X, Best_X and Breadth_X refer to a *Branch & Bound* algorithm with corresponding search strategies and *memorization* X used, with $X = S$ representing *solution memorization*, $X = Pa$ representing *passive node memorization* and $X = Pr$ the *predictive node memorization*. For *predictive node memorization*, we use *k-perm* heuristic to search for new sequences, as described in section 2.3.3.

k-perm heuristic also refers to a "dominance condition relying on scheduled jobs" as introduced by Jouglet et al. (2004). At a given node σS, assume that $\sigma = \sigma^0\sigma^k$ with σ^k the sub-sequence of the k last jobs in σ, k being an input parameter. The *k-perm* heuristic consists in enumerating all permutations of jobs in σ^k to obtain sequence σ^ℓ. Then, the first found sequence $\sigma^0\sigma^\ell$ dominating $\sigma^0\sigma^k$, if it exists, is used to prune node σS.

The dominating sequence can be memorized. The notion of dominance between sequences is the one used to define the *check* function in *node memorization*. Preliminary tests suggest us to choose $k = 5$ in our implementations in order to have the most efficient *predictive node memorization scheme*.

Notice that *k-perm* search is not performed when *breadth first* strategy is used, since the memorization applied on active nodes already covers the effect of *k-perm*.

The algorithms proposed by T'kindt et al. (2004) in 2004 are also tested on the same dataset and they are named as Depth_Pa_04, Best_Pa_04 and Breadth_Pa_04, respectively. Compared to our algorithms Depth_Pa, Best_Pa and Breadth_Pa, the main differences are that the RAM usage is limited to 450M for Depth_Pa_04, Best_Pa_04 and Breadth_Pa_04, in order to obtain similar results to that reported in 2004. Also, in our algorithms

LUFO is chosen as the database cleaning strategy.

The test results on instances of certain sizes are marked as *OOT* (out of time) if any instance is not solved after 5 hours. Analogously, with the application of *Memorization*, memory problems may occur and the limit on RAM usage may be reached, reported as *OOM* (out of memory). Note that according to our experiments, even when memory cleaning strategies are applied, *OOM* may still occur due to the fragmentation of the memory after a number of cleanings. Also note that LUFO is chosen as the cleaning strategy according to preliminary experimentations.

All tests have been done on a HP Z400 work station with 3.07GHz CPU and 8GB RAM.

4.1. Application to the $1|r_i|\sum C_i$ *problem*

The $1|r_i|\sum C_i$ problem asks to schedule n jobs on one machine to minimize the sum of completion times. Each job i has a processing time p_i and a release date r_i before which the job cannot be processed. The problem is NP-hard in the strong sense and it has been widely studied in the literature with both exact and heuristic algorithms considered. The referential computational results so far are done by T'kindt et al. (2004), in which with *forward branching* and *best first* and the application of a so-called *DP Property* the algorithm is able to solve instances with up to 130 jobs. A mixed integer programming approach is also reported by Kooli and Serairi (2014), which enables to solve instances with up to 140 jobs. However, in their experiments, only 5 instances are generated for each set of parameters. This makes their result less convincing due to the fact that the hardness of instances varies a lot even when generated with the same parameters, as observed during our study. Consequently, we consider in this section that the *Branch & Bound* algorithm provided by T'kindt et al. (2004) is at least as efficient as the approach of Kooli and Serairi (2014). In contrast, 30 instances are generated for each set of parameters considered by T'kindt et al. (2004), which leads to 300 instances for each size.

The work of T'kindt et al. (2004) uses the *Branch & Bound* algorithm of Chu (1992) as a basis, and so *forward branching* is adopted as the branching strategy. With respect to search strategies, *depth first*, *best first* and *breadth first* were all tested by T'kindt et al. (2004), aiming to explore the impact of different search strategies on the efficiency of the algorithm. The lower bounds and dominance conditions from Chu (1992) are kept. A so called *DP Property*, added as a new feature by T'kindt et al. (2004), is actually equivalent to *passive node memorization* in our terminology. The *check*() function is based on a dominance condition given by Chu (1992) and it was defined by T'kindt et al. (2004) as follows:

$$check(\pi,\sigma) = \begin{cases} 1, \ if\ opt(\sigma|0) \leq opt(\pi|0)\ and\ opt(\sigma|0) + |\Omega| * E_{min}(\sigma) \leq opt(\pi|0) + |\Omega| * E_{min}(\pi) \\ 0, \ otherwise \end{cases} \tag{2}$$

with Ω the jobs that remain to be scheduled after sequence σ and π. We also have $E_{min}(\sigma) = \max(C(\sigma), \min_{r\in\Omega} r_i)$, with $C(\sigma)$ the completion time of σ. The item stored into memory is a tuple $\langle \sigma, C(\sigma), opt(\sigma|0), ExpAct \rangle$ and $E_{min}(\sigma)$ can be computed when needed. Note that this definition of *check* is an adaption of the general Equation 1 and if the *check* in Equation 1 return 1 then this *check* also returns 1.

4.1.1. Application of the memorization framework and improved results

The problem is not *decomposable* due to the existence of release dates. Therefore, with the choice of *forward branching*, *node memorization* should be chosen, according to the decision tree in Figure 4.

The lower bound used in the algorithm is based on the SRPT (Shortest Remaining Processing Time) rule. Together with the *check*() function defined in Equation 2, it is not clear whether the concordance property is answered.

Hence, when *passive node memorization* is applied upon *best first*, all nodes need to be considered for the comparisons, while when it is applied with *breadth first*, only active nodes need to be considered. Therefore, the choices made by T'kindt et al. (2004) with respect to memorization are kept. The *check*() function also remains the same, as defined in Equation 2.

Here we refresh the computational results of T'kindt et al. (2004) on new randomly generated input and also add results for *predictive node memorization* and *solution memorization*. The input is generated following the way described by Chu (1992), i.e. the processing times are generated uniformly from [1, 100] and release dates are generated between 0 and $50.5 \cdot n \cdot r$ with r belonging to $\{0.2, 0.4, 0.6, 0.8, 1.0, 1.25, 1.50, 1.75, 2.0, 3.0\}$. 30 instances are generated for each value of r, hence leading to 300 instances for each size n from 70 to 140.

We first ran the algorithms from T'kindt et al. (2004) (Depth_Pa_04, Best_Pa_04 and Breadth_Pa_04) on these newly generated instances.

All these three algorithms are able to solve instances with up to 110 jobs. On the running time, Best_Pa_04 is the fastest one, with a maximum running time of 27 seconds for instances with 110 jobs. This value becomes 40 seconds for Depth_Pa_04 and 1082 seconds for Breadth_Pa_04. It was reported by T'kindt et al. (2004) that the algorithms with best first and breadth first can solve respectively instances with 130 jobs and 120 jobs which is different from what we obtain. This reveals that the newly generated instances are harder than that of T'kindt et al. (2004), knowing that the computational power of our test environment is much better than in 2004. Moreover, we also observed that for Depth_Pa_04, the hardest instance of 110 jobs is solved in 40 seconds but the hardest instance of 90 jobs is solved in 1691 seconds. This is why we think that the hardness of instances vary a lot when generated randomly.

Results related to *node memorization* are put in Table 1.

For all the three search strategies, *passive node memorization* enables to solve much larger instances with respect to the versions without memorization. This is sufficient to prove the power of memorization on this problem.

Depth_Pa, Best_Pa and Breadth_Pa are all more powerful than their counterparts of 2004, i.e. Depth_Pa_04, Best_Pa_04 and Breadth_Pa_04. This is especially visible on *depth first*, where Depth_Pa_04 solves instances with up to 110 jobs while Depth_Pa solves instances with up to 130 jobs. This makes Depth_Pa the global best algorithm and shows that when more physical memory is available and a larger database with an appropriate cleaning strategy is set, the memorization can be further boosted and the gain can be important.

The impact of *k-perm* search on this problem is very limited: *predictive node memorization* basically leads to the same result as *passive node memorization*.

In addition, we also tested *solution memorization* on this problem since no theoretical dominance between the memorization schemes can be established for this problem. Since context dependent dominance conditions are enabled in the algorithm, we first disabled them in order to obtain the optimal solution of each node. But this turned out to be very inefficient. Therefore, we also implemented the memorization of lower bounds, as described in section 2.3.1.

However, the resulting algorithm can only solve instances with up to 80 jobs, hence not competitive compared to *node memorization*, as predicted according to the decision tree in Figure 4.

It is also worth to be mentioned that the database cleaning strategy LUFO enables a faster solution of large instances. As an example, we found an instance with 140 jobs is solved in 1.6 hours by Depth_Pa with LUFO, while it needs 14 hours to be solved when the cleaning strategy of T'kindt et al. (2004) is kept instead. However, due to the hardness of another instance with 140 jobs, the algorithm Depth_Pa is finally out of time.

	n	70	80	90	100	110	120	130	140
Depth-	Navg	141247.8	1778751.2	OOT					
	Nmax	17491232	276190737						
	Tavg	1.8	22.4						
	Tmax	217	3238						
Depth_Pa	Navg	2583.4	5756.2	18639.9	26827.4	48502.9	174545.5	192409.4	OOT
	Nmax	147229	314707	2253897	644151	1281097	16575522	7742714	
	Tavg	0.0	0.0	0.3	0.7	1.3	7.1	9.1	
	Tmax	2	7	64	27	41	754	295	
Depth_Pr	Navg	1771.1	4455.1	12625.7	19621.7	30380.4	117865.6	128277.5	OOT
	Nmax	82765	267416	1455743	588429	1096520	11126694	5132228	
	Tavg	0.0	0.0	0.3	0.5	0.9	4.7	6.6	
	Tmax	1	7	46	28	39	488	252	
Best-		OOT							
Best_Pa	Navg	1230.5	3299.4	5235.1	9494.8	13658.5	38574.5	43986.9	OOT
	Nmax	36826	256534	292929	216293	228848	2675337	1449900	
	Tavg	0.0	0.2	0.2	0.4	0.6	15.3	11.8	
	Tmax	0	46	38	27	25	3595	1630	
Best_Pr	Navg	1229.6	3298.2	5229.0	9490.7	13545.7	38560.1	43989.8	OOT
	Nmax	36826	256529	292927	216037	228832	2674776	1449872	
	Tavg	0.0	0.2	0.2	0.4	0.7	15.4	11.9	
	Tmax	1	47	39	28	25	3579	1636	
Breadth-		OOT							
Breadth_Pa	Navg	1947.7	6745.0	9893.8	21308.5	27383.1	OOT		
	Nmax	90494	709607	733980	575430	1209481			
	Tavg	0.0	4.6	3.4	5.3	5.7			
	Tmax	9	1319	897	483	935			

Table 1: Results of new algorithms on the $1|r_i|\sum C_i$ problem

4.2. Application to the $1|\tilde{d}_i|\sum w_iC_i$ *problem*

The $1|\tilde{d}_i|\sum w_iC_i$ problem asks to schedule n jobs on a single machine. Each job i has a processing time p_i, a weight w_i and a deadline $\tilde{d}_i$ which has to be answered. The objective is to minimize the total weighted completion time $\sum w_iC_i$. The problem is NP-hard in the strong sense and has been solved by *Branch & Bound* algorithms Posner (1985); Potts and Van Wassenhove (1983), with Posner (1985) slightly superior. The basic algorithm described by T'kindt et al. (2004) is a combination of algorithms of Posner (1985); Potts and Van Wassenhove (1983) by incorporating the lower bound and the dominance condition of Posner (1985) into the *Branch & Bound* algorithm of Potts and Van Wassenhove (1983). With respect to search strategies, all the three strategies, i.e. *depth first*, *best first* and *breadth first* were considered by T'kindt et al. (2004) and *backward branching* is adopted as the branching scheme as for Posner (1985); Potts and Van Wassenhove (1983). Similarly to what is done on the $1|r_i|\sum C_i$ problem, the *DP Property*

is also considered by T'kindt et al. (2004), which is actually *passive node memorization*. The *check*() function is defined as follows, where Ω is the set of jobs to be scheduled before σ and π.

$$check(\pi, \sigma) = \begin{cases} 1, \ if \ opt(\sigma|\sum_{i\in\Omega} p_i) \leq opt(\pi|\sum_{i\in\Omega} p_i) \\ 0, \ otherwise \end{cases} \quad (3)$$

The items stored in the database are $\langle \sigma, opt(\sigma|\sum_{i\in\Omega} p_i), ExpAct \rangle$. In 2003, Pan (2003) proposed another *Branch & Bound* algorithm with reported experiments showing that it can solve to optimality all instances with up to 90 jobs in size. As the testing protocol is identical to the one used by T'kindt et al. (2004), we can conclude that the algorithm of Pan is outperformed by the best one proposed in the work of T'kindt et al. (2004) (which is reported as being able to solve instances with up to 130 jobs in size).

4.2.1. Application of the memorization framework and improved results

This problem is *decomposable* according to Definition 1.

From the decision tree in Figure 4 we can derive that with the depth first search strategy, *solution memorization* should be considered, even though its superiority over *node memorization* depends on the presence of *context dependent dominance conditions* in the algorithm. According to T'kindt et al. (2004), *node memorization* was implemented with that strategy. Consequently, in this section we compare four *Branch & Bound* algorithms: the three versions of T'kindt et al. (2004), i.e. *node memorization* applied to the three search strategies and a version based on depth first with *solution memorization*.

The concordance property is answered (see Proposition 1) and hence the *passive node memorization* only considers explored node when the search strategy is *best first*, and only active nodes need to be considered in *breadth first*. For *solution memorization*, the items stored into the memory are $\langle \pi, opt(\pi|0) \rangle$. For *node memorization*, the *check*() function and the items stored remain the same as for T'kindt et al. (2004), as described in the previous section.

About *solution memorization*, context dependent dominance conditions are enabled in the algorithm. Their removal has been experimentally proved to lead to an inefficient algorithm. Therefore, lower bounds are memorized during the *solution memorization*, as described in section 2.3.1.

Proposition 1. *With the check() function defined in Equation 3, our algorithms verify the Concordance Test (Definition 2).*

Proof. Consider two nodes $S\sigma$ and $S\pi$. First notice that the sub-problem to solve in both nodes are the same, which consists in scheduling jobs from S starting from time 0. The lower bound used in the algorithm (see Posner (1985); Potts and Van Wassenhove (1983)) returned on the sub-problems on S are the same for the two nodes. Therefore, if $check(\pi, \sigma) = 1$, which means $opt(\sigma|\sum_{i\in\Omega} p_i) \leq opt(\pi|\sum_{i\in\Omega} p_i)$, then $LB(S\sigma) \leq LB(S\pi)$.

With the same reasoning, if $LB(S\sigma) \leq LB(S\pi)$, it can be deduced that the $opt(\sigma|\sum_{i\in\Omega} p_i) \leq opt(\pi|\sum_{i\in\Omega} p_i)$ must holds, and hence $check(\pi, \sigma) = 1$. □

Following the test plan described by Potts and Van Wassenhove (1983), for each job i, its processing time p_i is an integer generated randomly from the uniform distribution $[1, 100]$ and its weight w_i is generated uniformly from $[1, 10]$. The total processing time $P = \sum_{i=1}^{n} p_i$ is then computed and for each job i an integer deadline d_i is generated from the uniform distribution $[P(L - R/2), P(L + R/2)]$, with L increase from 0.6 to 1.0 in steps of 0.1 and R increases from 0.2

to 1.6 in steps of 0.2. In order to avoid generating infeasible instances, a (L, R) pair is only used when $L + R/2 > 1$, hence only 20 (L, R) pairs are actually used, for each of which 10 feasible instances are generated, yielding a total of 200 instances for each value of n from 40 to 140. We first present the results of *passive node memorization* algorithms (Depth_Pa_04, Best_Pa_04 and Breadth_Pa_04) from T'kindt et al. (2004). Both Depth_Pa_04 and Best_Pa_04 are stated by T'kindt et al. (2004) to solve instances with up to 110 jobs. However, they are only capable of solving instances with 70 jobs on the newly generated instances, with a maximum solution time 11 seconds and 285 seconds, respectively. Breadth_Pa_04 was reported to be able to solve instances with up to 130 jobs in 2004 but this falls down to 100 jobs in our tests with a maximum solution time of 36 seconds. This difference is not negligible and it reveals the fact that the newly generated instances seem much harder than those generated by T'kindt et al. (2004).

The results of the new algorithms are presented in Table 2. On *depth first*, without *memorization* the program is "out of time" on instances with 50 jobs, while both *solution memorization* and *passive node memorization* enable to solve instances with up to 100 jobs, with *passive node memorization* running faster. With the activation of *k-perm* search, Depth_Pr enables to solve 30 more jobs than Depth_Pa. This strongly proves the power of all the three memorization schemes. It also worth to be noticed that Depth_Pa solves instances with 30 more jobs with respect to Depth_Pa_04, knowing that the only differences between these two algorithm are that the database size in Depth_Pa is larger and the database cleaning strategy is different.

For *best first*, the same phenomenon can be observed, that is, Best_Pr is more efficient than Best_Pa, which is better than Best- and Best_Pa_04. Best_Pr can also solve instances with up to 130, and faster than Depth_Pr.

Breadth_Pa is the most powerful algorithm among all. It is surprising to see that without *memorization* Breadth- cannot even solve all instances of 40 jobs, while with *passive node memorization* instances of 130 jobs are all solved in an average solution time of 27 seconds. Again, as for the $1|r_i|\sum C_i$ problem, LUFO allows to accelerate the solution but it did not enable to solve larger instances.

	n	40	50	60	70	80	90	100	110	120	130	140
Depth-	Navg	116827.7	OOT									
	Nmax	14536979										
	Tavg	1										
	Tmax	74										
Depth_S	Navg	772.0	2718.0	6706.0	28463.0	114970.0	139382.0	563209.0	OOT			
	Nmax	17699	60462	137207	1660593	6180097	2803714	12335703				
	Tavg	0.4	0.5.0	1.0	3.0	15.0	12.0	113.0				
	Tmax	1	2	6	275	1544	474	5346				
Depth_Pa	Navg	559.1	2091.3	5240.1	20068.3	75727.4	139429.8	376206.9	OOT			
	Nmax	11963	83075	94189	1004546	1960891	4321070	5549747				
	Tavg	0.4	0.4	0.5	0.9	2.9	5.5	17.0				
	Tmax	1	1	2	39	157	312	515				
Depth_Pr	Navg	326.4	901.7	2184.1	6825.3	20429.0	32531.0	90375.3	266689.8	574824.4	1397463.6	OOT
	Nmax	3431	17447	28677	187425	665376	768802	1781123	14713483	11236833	103699138	
	Tavg	0.4	0.4	0.4	0.6	1.0	1.0	3.8	14.4	37.9	108.2	
	Tmax	0	1	1	5	30	21	51	901	1255	8732	
Best-		OOT										
Best_Pa	Navg	334.6	879.3	1859.0	7159.9	16581.8	27259.5	60349.0	OOT			
	Nmax	3800	20889	25574	440623	547165	1252600	798372				
	Tavg	0.4	0.4	0.4	0.7	1.3	1.8	4.3				
	Tmax	0	1	1	30	73	130	91				
Best_Pr	Navg	292.0	708.9	1486.9	4312.7	10301.7	14642.9	31891.2	145203.1	239837.4	330474.1	OOT
	Nmax	2435	11762	18051	120259	319068	276507	332022	10659343	7578570	6712266	
	Tavg	0.4	0.4	0.4	0.5	0.8	1.0	1.9	45.6	20.0	26.7	
	Tmax	0	1	1	4	23	13	25	5008	1137	716	
Breadth-		OOM										
Breadth_Pa	Navg	348.6	940.9	1833.1	6533.4	14964.4	23725.0	53309.2	102633.5	239512.5	329902.3	OOT
	Nmax	4701	16952	24559	437697	453506	868876	789310	5975094	7577492	6702080	
	Tavg	0.0	0.0	0.0	0.2	0.4	0.8	2.0	10.1	20.0	26.7	
	Tmax	0	0	0	9	15	31	36	1353	1135	718	

Table 2: Results of the new algoritihms on the $1|\tilde{d_i}|\sum w_i C_i$ problem

4.3. Application to the $F2||\sum C_i$ problem

The $F2||\sum C_i$ problem asks to schedule n jobs are to be scheduled on two machines M_1 and M_2. Each job i needs first to be processed on M_1 for $p_{1,i}$ time units then be processed on M_2 for $p_{2,i}$ time. The objective is to minimize the sum of completion times of jobs. We restrict to the set of permutation schedules in which there always exist an optimal solution. A permutation schedule is a schedule in which the jobs sequences on the two machines are the same. The problem is NP-hard in the strong sense. Up to 2016, the best exact algorithm was the *Branch & Bound* algorithm proposed by T'kindt et al. (2004) and based on the *Branch & Bound* algorithm of Della Croce et al. (2002). Recently, Detienne et al. Detienne et al. (2016) proposed a new and very efficient *Branch & Bound* algorithm capable of solving instances with up to 100 jobs in size. This is definitely the state-of-the-art exact method for solving the $F2||\sum C_i$ problem. However, in order to evaluate the impact of using *Memorization* in a *Branch & Bound* algorithm we make use of the algorithms described by T'kindt et al. (2004) since their code was directly available to us.

The adopted branching scheme in this algorithm is *forward branching* and all the three search strategies were considered. The *DP Property* is also considered by T'kindt et al. (2004), which is actually *passive node memorization*. The *check*() function is based on a result reported by Della Croce et al. (2002) and is defined as follows:

$$check(\pi,\sigma) = \begin{cases} 1, \ if\ opt(\sigma|0) \leq opt(\pi|0)\ and\ |\Omega| * (C_2(\sigma) - C_2(\pi)) \leq opt(\pi) - opt(\sigma) \\ 0, \ otherwise \end{cases} \quad (4)$$

where Ω is the set of jobs to be scheduled after σ and π, $C_2(\cdot)$ is the completion time of a given sequence on the second machine. The items stored into the database are $\langle \sigma, C_2(\sigma), opt(\sigma|0), ExpAct \rangle$.

4.3.1. Application of the memorization framework and improved results

This problem is not *decomposable* since

given a partial solution of the form σS with σ a fixed sequence, the optimal solution of subproblem S depends on the order of jobs in σ. From the decision tree in Figure 4 we can derive that with the depth first search strategy, *solution memorization* should be considered, even though its superiority over *node memorization* depends on the presence of *context dependent dominance conditions*. According to T'kindt et al. (2004), *node memorization* was implemented with that strategy. Consequently, in this section we compare four *Branch & Bound* algorithms: the three versions of T'kindt et al. (2004), i.e. *node memorization* applied to the three search strategies and a version based on depth first with *solution memorization*.

With the *check*() function defined in Equation 4 and the lower bound (a Lagrangian Relaxation based lower bound) used in the algorithm, the concordance property is not answered. We performed experiments to look for the case where for two nodes σS and πS, $check(\pi, \sigma) = 1$ but $LB(\pi) < LB(\sigma)$ and we found it. Therefore, the concordance property is not verified and both active and explored nodes need to be considered for *best first* strategy. For *breadth first* strategy, only active nodes need to be considered.

For *solution memorization*, since context dependent dominance conditions are enabled in the algorithm, and they are important for a fast solution of the problem, lower bounds are memorized during the *solution memorization*, as described in section 2.3.1.

The items stored into the memory are $\langle \pi, t_1, t_2, C_2(\pi), opt(\pi|(t_1, t_2)) \rangle$ where t_1 is the actual starting time of π on the first machine and t_2 is the actual starting time of π on the second machine. Besides, $opt(\pi|(t_1, t_2))$ is the sum of completion times of jobs in π, when π starts at time t_1 on the first machine and time t_2 on the second machine. For *node memorization*, the *check*() function and the stored item remain the same as per T'kindt et al. (2004), as described in the previous section.

30 instances are generated for each size n from 10 to 40, with the processing times generated randomly from an uniform distribution in $[1, 100]$. Again, we ran the algorithms from T'kindt et al. (2004) on these newly generated instances.

Depth_Pa_04 is able to solve instances of 40 jobs which is 5 jobs more than reported by T'kindt et al. (2004), with a maximum solution time about 3.4 hours. Both Best_Pa_04 and Breadth_Pa_04 solve instances with up to 35 jobs, as reported by T'kindt et al. (2004), with maximum solution times of 43 seconds and 806 seconds, respectively.

Other results are given in Table 3. Depth- is able to solve instances with 35 jobs. Best- is able to solve instances with 30 jobs and Breadth- can only solve up to 25 jobs. With *passive node memorization* enabled, Depth_Pa solves instances with 5 more jobs than Depth-. Best_Pa and Breadth_Pa solve instances with 10 more jobs than the versions without *Memorization*. With respect to algorithms X_Pa_04, algorithms X_Pa use a larger database, more precisely, the maximum number of solutions that can be stored is set to 6000000 instead of 350000. This enables Best_Pa to solve 5 more jobs than Best_Pa_04. However, Depth_Pa and Breadth_Pa are not able to solve larger instances with respect to Depth_Pa_04 and Breadth_Pa_04, even though they solve instances faster. Notice that the $F2||\sum C_i$ problem is a really hard problem, certainly more difficult than the two other problems previously tackled in this paper.

Also, the LUFO strategy is adopted for database cleaning but it did not enable to solve larger instances without having an "Out of Time" problem.

Predictive node memorization is not more efficient than *passive node memorization*: in fact no nodes are cut by undertaking a *k-perm* search. The result is hence even slightly slower due

to the time consumed by the call to the *k-perm* heuristic. Depth_S solve instances with less nodes generated compared to Depth-. However, its efficiency is even less than Depth-, due to the processing of lower bound memorization.

From a global point of view, the power of memorization is also illustrated on this problem, since we always have benefits in using it. As a perspective for this problem it could be interesting to evaluate the contribution of *Memorization* when embedded into the state-of-the-art algorithm presented by Detienne et al. (2016).

	n	10	15	20	25	30	35	40	45
Depth-	Navg	23.7	255.6	4137.7	21460.4	317102.0	3615780.0	OOT	
	Nmax	84	2367	83863	311742	3097479	53187978		
	Tavg	0.0	0.0	0.1	0.8	26.0	423.0		
	Tmax	0	0	2	17	248	6128		
Depth_S	Navg	24.0	228.0	3561.0	19733.0	294355.0	3425633.0	OOT	
	Nmax	84	1735	68070	273146	2712580	49360565		
	Tavg	0.0	0.0	0.1	1.0	29.0	497.0		
	Tmax	0	0	2	15	248	6933		
Depth_Pa	Navg	22.8	187.2	1573.0	8205.0	61337.0	337194.0	1894037.2	OOT
	Nmax	80	1083	17114	48459	291750	1568506	15472612	
	Tavg	0.0	0.0	0.0	0.1	4.1	35.0	328.3	
	Tmax	0	0	0	2	21	163	3627	
Depth_Pr	Navg	22.8	187.2	1573.0	8205.0	61361.3	337194.0	1894037.0	OOT
	Nmax	80	1083	17114	48459	291016	1568506	15472612	
	Tavg	0.0	0.0	0.0	0.1	4.1	32.8	332.8	
	Tmax	0	0	0	2	23	173	3664	
Best-	Navg	23.7	249.3	3993.1	21717.7	291131.9	OOM		
	Nmax	84	2253	83863	311742	2451152			
	Tavg	0.0	0.0	0.1	0.7	19.1			
	Tmax	0	0	2	17	197			
Best_Pa	Navg	20.9	139.5	957.3	4780.7	28957.0	112229.8	495186.5	OOM
	Nmax	72	624	6646	21022	152797	426641	3617824	
	Tavg	0.0	0.0	0.0	0.0	1.2	7.8	80.6	
	Tmax	0	0	0	1	4	43	1253	
Best_Pr	Navg	20.9	139.5	957.3	4780.7	28957.0	112229.8	495186.5	OOM
	Nmax	72	624	6646	21022	152797	426641	3617824	
	Tavg	0.0	0.0	0.0	0.0	1.4	8.3	83.1	
	Tmax	0	0	0	1	5	45	1283	
Breadth-	Navg	23.9	266.1	5181.8	39303.6	OOT			
	Nmax	84	2360	83863	311742				
	Tavg	0.0	0.0	0.1	1.6				
	Tmax	0	0	2	17				
Breadth_Pa	Navg	21.0	148.8	1369.5	8889.1	115219.2	345109.6	OOT	
	Nmax	72	692	9927	63485	2242263	2357023		
	Tavg	0.0	0.0	0.0	0.2	26.1	54.2		
	Tmax	0	0	0	3	711	665		

Table 3: Results of new algorithms on the $F2||\sum C_i$ problem

5. Application to the $1||\sum T_i$ problem

In this section, we report the results of the application of *Memorization* on solving the single machine total tardiness problem, referred to as $1||\sum T_i$. We first introduce main properties and existing results of the problem, then determine parameters for *Memorization* and finally report the computational results.

5.1. Preliminaries

The problem asks to schedule a set of n jobs $N = \{1, 2, \ldots, n\}$ on a single machine. For each job j, a processing time p_j and a due date d_j are given and the objective is to arrange the jobs into a sequence $S = (a_1, ..., a_n)$ so as to minimize $T(N, S) = \sum_{j=1}^{n} \max\{\sum_{i=1}^{j} p_{a_i} - d_{a_j}, 0\}$. This problem is a classic scheduling problem known to be NP-hard in the ordinary sense Du and Leung (1990). It has been extensively studied in the literature.

The current state-of-the-art exact method in practice is a *Branch & Bound* algorithm (named as BB2001 in this paper) which solves to optimality problems with up to 500 jobs in size Szwarc et al. (2001). Latest theoretical developments for the problem can be found in the survey of Koulamas Koulamas (2010). Main properties of the problem can be found in Szwarc et al. (2001), and some of them are reminded below.

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.

We first introduce two important decomposition properties.

Decomposition 1. *Lawler (1977) (Lawler's decomposition) 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* $A_1(h) = \{[h+1], \ldots, [n]\}$.

Decomposition 2. *Szwarc et al. (1999) Let job* k *in LPT sequence correspond to job* [1] *in EDD sequence. Then, job* k *can be set only in positions* $h \leq (n - k + 1)$ *and the jobs preceding job* k *are uniquely determined as* $B_k(h)$*, where* $B_k(h) \subseteq \{k+1, k+2, \ldots, n\}$ *and* $\forall i \in B_k(h), j \in \{n, n-1, \ldots, k+1\} \setminus B_k(h)$, $d_i \leq d_j$

The two above decomposition rules can be applied simultaneously to derive a decomposing branching scheme called *Double Decomposition* (Szwarc et al., 2001). At any node, let S_i be a set of jobs to schedule. Note that some other jobs may have already been fixed on positions before or after S_i, implying a structure like $\sigma_1 S_1 \sigma_2 S_2 ... \sigma_i S_i ... \sigma_k S_k$ over all positions, but a node only focuses on the solution of one sub-problem, induced by one subset of jobs (S_i here). With depth first, which is the search strategy retained in the *Branch & Bound* BB2001, the *Double Decomposition* is always applied on S_1. This works as follows. First find the longest job ℓ and the earliest due date job e in S_1. Then apply Decomposition 1 (resp. Decomposition 2) to get the lists L_ℓ (resp. L_e) of positions, on which ℓ (resp. e) can be branched on. As an example, suppose $L_e = \{1, 2\}$ and $L_\ell = \{5, 6\}$. Then, a double branching can be done by fixing job e on position 1 and fixing job ℓ on position 5, decomposing the jobs S_i to three subsets (sub-problems): the jobs before jobs job e, which is $\emptyset$; the jobs between e and ℓ; and finally the jobs after ℓ. In the same way, the other 3 branching can be performed by fixing jobs e and ℓ in all compatible position pairs: $(1, 6)$, $(2, 5)$ and $(2, 6)$.

When branching from a node, another particular decomposition may occur as described in Property 6. Assume that a given subset of jobs S is decomposed into two disjoint subsets B

and A, with $B \cup A = S$ and all jobs of B are scheduled before that of A in an optimal schedule of S: (B, A) is then called an optimal block sequence and Property 6 states when does such decomposition occur. In that case Decomposition 1 and Decomposition 2 are not applied but two child nodes are rather created each one corresponding to one block of jobs (A or B), following Property 6 (also called the *split* property).

Let E_j and L_j be the earliest and latest completion times of job j, that is if B_j (resp. A_j) is the currently known jobs that precedes (resp. follow) job j, then $E_j = p(B_j)+p_j$, and $L_j = p(N \setminus A_j)$.

Property 6. *Szwarc et al. (1999) (Split)*

(B, A) is an optimal block sequence if $\max_{i \in B} L_i \leq \min_{j \in A} E_j$.

The value of E_i and L_i of each job i can be obtained by applying Emmons' conditions (Emmons, 1969) following the $O(n^2)$ procedure provided by Szwarc et al. (1999).

An initial version of *solution memorization* has been already implemented in BB2001, even though it was called *Intelligent Backtracking* by the authors. Remarkably, lower bounds are not used in this *Branch & Bound* algorithms due to

the "Algorithmic Paradox" (Paradox 1) found in Szwarc et al. (2001). This one shows that the power of *Memorization* largely surpasses the power of the lower bounding procedures in the algorithm.

Paradox 1. *"...deleting a lower bound drastically improves the performance of the algorithm..."*

Paradox 1 is simply due to the fact that a lot of identical sub-problems occur during the exploration of the search tree. The computation time required by lower bounding procedures to cut these identical problems is much higher than simply solving that sub-problems once, memorizing the solution and reusing it whenever the sub-problem appears again. Besides, pruning nodes by the lower bound may negatively affect memorization since the nodes that are cut cannot be memorized.

The BB2001 algorithm uses a depth first strategy and for each node to branch on, the following procedure is applied:

1. Search the solution of the current problem, defined by a set of jobs and a starting time of the schedule, in "memory", and return the solution if found; otherwise go to 2.
2. Use Property 6 to split the problem to new sub-problems which are solved recursively starting from step 1. If no split can be done, go to step 3.
3. Combine Decompositions 1 and 2 to branch on the longest job and the smallest-due-date job to every candidate positions. For each branching case, solve sub-problems recursively, then store in memory the best solution among all branching cases and return it.

Note that due to Paradox 1, all lower bounding procedures are removed, which makes the *Branch & Bound* algorithm a simple branching algorithm. Notice that *solution memorization* can be implemented in BB2001 as suggested in section 3. In BB2001, when the database of stored solutions is full, no cleaning strategy is used and no more partial solutions can be stored. The memory limit of this database in BB2001 is not mentioned by Szwarc et al. (2001).

5.2. Application of the memorization framework and improved results

We take the reference algorithm BB2001 as a basis, in which *decomposition branching* and *solution memorization* are already chosen. The *decomposition branching* has been proved to be

very powerful, and there is no evidence that other branching schemes like *forward branching* or *backward branching* can lead to a better algorithm (see Szwarc et al. (2001)). The problem is *decomposable* according to Definition 1. The main discussion relies on the relevancy of considering *node memorization* instead of *solution memorization*. As already mentioned in section 2.4.4, it is not obvious to implement *node memorization*, for a decomposing branching scheme, which could outperform the *solution memorization*. Here a node is structured as $\sigma_1 S_1 ... \sigma_k S_k$ with the $\sigma_i's$ being the partial sequences to memorize in *node memorization*. Assume we have two nodes $\sigma_1 S_1 ... \sigma_k S_k$ and $\pi_1 S'_1 ... \pi_\ell S'_\ell$, it is not apparent to find σ_i and π_j, $i \in \{1, .., k\}$, $j \in \{1, .., \ell\}$, such that σ_i and π_j are of same jobs and have the same starting time. Moreover, it seems complicated to design an efficient *check*() function deciding which of these two nodes is dominating the other. We found no way to implement *node memorization* which could hopefully lead to better results than those obtained with *solution memorization*. Consequently, *solution memorization* only is considered and, as sketched in sections 2.4.5 and 2.4.6, there is no interest in considering *best first* or *breadth first* search strategies.

Henceforth, the choices done by Szwarc et al. (2001) with respect to memorization were good choices. In the remainder we investigate limitations of the memorization technique as implemented by Szwarc et al. (2001) and propose improvements which significantly augment the efficiency of the algorithm.

Our algorithm is based on BB2001, with two main changes.

Since the memory usage was declared as a bottleneck of BB2001, we firstly retest BB2001 on our machine: a HP Z400 work station with 3.07GHz CPU and 8GB RAM. 200 instances are generated randomly for each problem size using the same generation scheme as per Potts and Van Wassenhove (1982). Processing times are integers generated from an uniform distribution in the range $[1, 100]$ and due dates d_i are integers from a uniform distribution in the range $[p_i u, p_i v]$ where $u = 1 - T - R/2$ and $v = 1 - T + R/2$. Each due date is set to zero whenever its generated value is negative. Twenty combinations (R, T) are considered where $R \in \{0.2, 0.4, 0.6, 0.8, 1\}$, and $T \in \{0.2, 0.4, 0.6, 0.8\}$. Ten instances are generated for each combination and the combination $(R = 0.2, T = 0.6)$ yields the hardest instances as reported in the literature (see Szwarc et al. (1999)) and confirmed by our experiments. Table 4 presents the results we obtain when comparing different algorithms. For each version we compute the average and maximum CPU time T_{avg} and T_{max} in seconds for each problem size. The average and maximum number of explored nodes N_{avg} and N_{max} are also computed. The time limit for the solution of each instance is set to 4 hours, and the program is considered as OOT (Out of Time) if it reaches the time limit. Also, when memorization is enabled without a database cleaning strategy, the physical memory may be saturated by the program, in which case the program is indicated as OOM (Out of Memory).

Our implementation of BB2001 solves instances with up to 900 jobs in size as reported in Table 4, with an average solution time of 764s and a maximum solution time of 9403s for 900-job instances, knowing that the original program, as tested in 2001 was limited to instances with up to 500 jobs due to memory size limit. Their tests were done on a Sun Ultra-Entreprise Station with a reduced CPU frequency (<450MHz) and a RAM size not stated. It is anyway interesting to see that with just the computer hardware evolution, *Memorization* is augmented to solve instances with 400 jobs more.

BB2001 is out of time (>4h) for instances with 1000 jobs, and the memory size seems no longer to be the bottleneck. The first improvement we propose presume on the vein of Paradox 1.

Paradox 2. *Removing Split procedure (Property 6) from BB2001 drastically accelerate the so-*

lution.

The effect of Paradox 2 is astonishing. The resulting algorithm *NoSplit* solves instances with 700 jobs with an average solution time 20 times faster: from 192 seconds to 9 seconds (see Table 4). In fact, *Split* is performed based on precedence relations between jobs, induced by the computation of the $E_j's$ and $L_j's$. The computation of these precedence relations is time consuming in practice. Moreover, as already claimed, many identical problems appear in the search tree and the *Split* procedure in BB2001 is run each time. When *Split* is removed, identical problems are solved needing more time when first met, but then never solved twice thanks to *solution memorization*. However, the disadvantage is also clear: more solutions are added to the database and hence the database is filled faster than when *Split* is enabled. This is why *NoSplit* encounters memory problems on instances with 800 jobs.

This was not considered by Szwarc et al. (2001) because *Split* is a very strong component of the algorithm and the computer memory at that time also discourages this tentative. S_i, SDD_2

At this point, we have a better understanding of the power of *solution memorization* on this problem and we become curious on the effectiveness of memorized solutions. In other words, what are the stored solutions that are effectively used? To answer this question, we test cleaning strategies as defined in section 3, to remove useless solutions when the database memory is "full". The most efficient strategy is proved to be LUFO by preliminary experiments not reported here. Embedding such a memory cleaning strategy is our second contribution to BB2001 algorithm.

In Table 4, the final implementation of the memorization mechanism within the *Branch & Bound* algorithm for the $1||\sum T_i$ problem is referred to as NoSplit_LUFO.

All 200 instances with 1200 jobs are solved, with an average solution time of 192 seconds, while BB2001 is limited to instances with 900 jobs.

	n	300	400	500	600	700	800	900	1000	1100	1200	1300
Depth-	Navg	46046201	OOT									
	Nmax	2249342615										
	Tavg	155										
	Tmax	6499										
BB2001	Navg	61501	136452	290205	560389	880268	1534960	2053522	OOT			
	Nmax	663268	1884993	3585456	5784871	9802077	18199764	19352429				
	Tavg	2	9	31	85	192	469	763				
	Tmax	33	193	580	1263	2963	6817	9403				
NoSplit	Navg	202970	457918	985235	1934818	3053648	OOM					
	Nmax	2156144	6027604	13028651	20285112	33977553						
	Tavg	0	0	2	4	9						
	Tmax	4	13	34	59	114						
NoSplit_LUFO	Navg	202970	457918	985235	1934818	3086620	5408511	7697810	12578211	19100285	28223766	OOT
	Nmax	2156144	6027604	13028651	20285112	36853477	60151076	88909109	139698961	332937242	420974965	
	Tavg	0	0	2	5	9	20	31	61	112	192	
	Tmax	4	13	34	61	136	275	429	832	2504	3763	

Table 4: Results for the $1||\sum T_i$ problem

The experiments presented so far have shown that correctly tuning the memorization mechanism, notably by considering a cleaning strategy and studying interference with other components of the algorithm may lead to serious changes of its efficiency. However, the striking point of these experiments relates on the comparison between the version of BB2001 without the memorization mechanism (algorithm Depth-) and NoSplit_LUFO. Table 4 highlights the major contribution of memorization: Depth- being limited to instances with up to 300 jobs while NoSplit_LUFO is capable of solving all instances with 1200 jobs. It is evident that memorization is a very powerful mechanism.

6. Conclusion

In this paper we focus on the application of *Memorization* within search tree algorithms for the efficient solution of sequencing problems. A framework of *Memorization* is provided with several memorization schemes defined. Advices are provided to choose the best memorization approach according to the branching scheme and the search strategy of the algorithm. Some details on the efficient implementation of *Memorization* are also discussed.

The application of the framework has been done on four scheduling problems. Even if the impact of *Memorization* depends on the problem, for all the tackled problems it was beneficial to use it. Table 5 provides a summary of the conclusions obtained.

Problem	Largest instances solved		Features of the best algorithm with memorization	Best in literature?
	Without memorization	With memorization		
$1\|r_i\|\sum C_i$	80 jobs	130 jobs	depth first+ *predictive node memorization*	yes
$1\|\tilde{d}_i\|\sum w_iC_i$	40 jobs	130 jobs	breadth first+ *passive node memorization*	yes
$F2\|\|\sum C_i$	30 jobs	40 jobs	best first+ *passive node memorization*	no
$1\|\|\sum T_i$	300 jobs	1200 jobs	depth first+ *solution memorization*	yes

Table 5: Conclusions on the tested problems

Fundamentally, what we call the *Memorization Paradigm* relies on a simple but potentially very efficient idea: avoid solving multiple times the same sub-problems by memorizing what has already been done or what can be done in the rest of the solution process. The contribution of this paradigm strongly relies on the branching scheme which may induce more or less redundancy in the exploration of the solution space. Noteworthy, the four scheduling problems dealt with in this paper mainly serve as applications illustrating how memorization can be done in an efficient way. But, it is also clear that it can be applied to other hard combinatorial optimization problems, by the way making this contribution interesting beyond scheduling theory. To our opinion, the memorization paradigm should be embedded into any branching algorithm, so creating a new class of branching algorithms called *Branch & Memorize* algorithms. They may have a theoretical interest by offering the possibility of reducing the worst-case time complexity with respect to *Branch & Bound* algorithms. And they also have an interest from an experimental viewpoint, as illustrated in this paper.

As a future research line, we plan to evaluate *Branch & Memorize* algorithms on more combinatorial optimization problems. It may be also very promising to see how machine learning field could help in efficiently managing the database of stored partial solutions. More concretely, a more intelligent database managing strategy may be conceived, which decides which solutions to store into or which solutions to remove from the database, through a learning process.

References

Biere, A., Heule, M., van Maaren, H., Walsh, T., 2009. Conflict-driven clause learning sat solvers. Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, 131–153.

Chandran, L. S., Grandoni, F., 2005. Refined memorization for vertex cover. Information Processing Letters 93 (3), 125–131.

Chu, C., 1992. A branch-and-bound algorithm to minimize total flow time with unequal release dates. Naval Research Logistics (NRL) 39 (6), 859–875.

Della Croce, F., Ghirardi, M., Tadei, R., 2002. An improved branch-and-bound algorithm for the two machine total completion time flow shop problem. European Journal of Operational Research 139 (2), 293–301.

Detienne, B., Sadykov, R., Tanaka, S., 2016. The two-machine flowshop total completion time problem: branch-and-bound algorithms based on network-flow formulation. European Journal of Operational Research 252 (3), 750–760.

Du, J., Leung, J. Y.-T., 1990. Minimizing total tardiness on one machine is np-hard. Mathematics of operations research 15 (3), 483–495.

Emmons, H., 1969. One-machine sequencing to minimize certain functions of job tardiness. Operations Research 17 (4), 701–715.

Fomin, F. V., Grandoni, F., Kratsch, D., 2005. Some new techniques in design and analysis of exact (exponential) algorithms. Bulletin of the EATCS 87 (47-77), 0–288.

Fomin, F. V., Kratsch, D., 2010. Exact exponential algorithms. Springer Science & Business Media.

Glover, F., 1989. Tabu search—part i. ORSA Journal on computing 1 (3), 190–206.

Glover, F., 1990. Tabu search—part ii. ORSA Journal on computing 2 (1), 4–32.

Jouglet, A., Baptiste, P., Carlier, J., 2004. Branch-and-bound algorithms for totalweighted tardiness. In: Handbook of scheduling: Algorithms, models, and performance analysis. Chapman and Hall/CRC.

Kooli, A., Serairi, M., 2014. A mixed integer programming approach for the single machine problem with unequal release dates. Computers & Operations Research 51, 323–330.

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

Lawler, E. L., 1977. A "pseudopolynomial" algorithm for sequencing jobs to minimize total tardiness. Annals of discrete Mathematics 1, 331–342.

Pan, Y., 2003. An improved branch and bound algorithm for single machine scheduling with deadlines to minimize total weighted completion time. Operations Research Letters 31 (6), 492–496.

Posner, M. E., 1985. Minimizing weighted completion times with deadlines. Operations Research 33 (3), 562–574.

Potts, C. N., Van Wassenhove, L., 1982. A decomposition algorithm for the single machine total tardiness problem. Operations Research Letters 1 (5), 177–181.

Potts, C. N., Van Wassenhove, L. N., 1983. An algorithm for single machine sequencing with deadlines to minimize total weighted completion time. European Journal of Operational Research 12 (4), 379–387.

Robson, J. M., 1986. Algorithms for maximum independent sets. Journal of Algorithms 7 (3), 425–440.

Shang, L., Garraffa, M., Della Croce, F., T'Kindt, V., Aug. 2017. Merging nodes in search trees: an exact exponential algorithm for the single machine total tardiness scheduling problem. In: 12th International Symposium on Parameterized and Exact Computation (IPEC 2017). Vol. 89 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Vienna, Austria, pp. 28:1–28:12.

Szwarc, W., Della Croce, F., Grosso, A., 1999. Solution of the single machine total tardiness problem. Journal of Scheduling 2 (2), 55–71.

Szwarc, W., Grosso, A., Croce, F. D., 2001. Algorithmic paradoxes of the single-machine total tardiness problem. Journal of Scheduling 4 (2), 93–104.

T'kindt, V., Della Croce, F., Esswein, C., 2004. Revisiting branch and bound search strategies for machine scheduling problems. Journal of Scheduling 7 (6), 429–440.

Xiao, M., Nagamochi, H., 2013. Exact algorithms for maximum independent set. In: Cai, L., Cheng, S.-W., Lam, T.-W. (Eds.), Algorithms and Computation. Vol. 8283 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 328–338.

Zhang, L., Madigan, C. F., Moskewicz, M. H., Malik, S., 2001. Efficient conflict driven learning in a boolean satisfiability solver. In: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design. IEEE Press, pp. 279–285.