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,c]

[a] *Université François-Rabelais de Tours, LIFAT (EA 6300), ERL CNRS ROOT 7002, Tours, France*
[b] *DIGEP - Politecnico di Torino, Torino, Italy*
[c] *CNR, IEIIT, Torino, Italy*

Abstract

Memorization, as an algorithm design technique, allows algorithms to be sped up at the price of increased 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. This idea has existed for a long time; however, to the best of the authors' knowledge, it has not been systematically considered when designing branching algorithms and has been rarely considered for sequencing problems.

In this paper, we explore the power of Memorization in solving hard sequencing problems. We first describe a general framework of Memorization and provide some guidelines for its implementation. Then we apply the framework to four sequencing problems: 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 that Memorization should be systematically considered as a solving block inside search tree based algorithms such as Branch and Bound.

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

1. Introduction

Memorization is an algorithm design technique that allows algorithms to be sped up at the price of increased 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 on the *Maximum Independent Set* problem by Robson (1986) in 1986. By exploiting graph theoretic properties and by applying *Memorization* to avoid solving identical

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

sub-problems, Robson proposed an algorithm with a worst-case time complexity in $O(1.2109^n)$. This proposed algorithm remained the exact exponential algorithm with the smallest worst-case time complexity until 2013, when the $O(1.1996^n)$ algorithm of Xiao and Nagamochi (2017) was introduced. *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 *Exact Exponential Algorithms* (EEAs), where the objective is to develop exact algorithms that can provide a best possible worst-case running time guarantee. Very recently, Xiao and Tan (2017) applied *Memorization* to derive an algorithm running in $O^*(1.3752^n)$ time and exponential space, solving the *Maximum Induced Matching* problem.

Although a typical *Memorization* algorithm memorizes solutions of sub-problems that appear repeatedly, we prefer to interpret the concept 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 future decisions". In the literature, various algorithms can be classified as procedures that embed memorization techniques, although their implementations could be quite different depending on the problem structure and the information to be stored. For instance, *Tabu Search* (Glover, 1989, 1990) is a metaheuristic that memorizes recently visited solutions to avoid returning to these solutions again during the search. SAT solvers deduce and then memorize conflict clauses during the tree search to perform non-chronological backtracking (*Conflict Driven Clause Learning*) (Biere et al., 2009; Zhang et al., 2001). Similar ideas also appear in *Artificial Intelligence* as *Intelligent Backtracking* or *Intelligent Back-jumping*. Variations of memorization exist under diverse names such as "Branch, bound and remember (BB&R)" (Sewell and Jacobson, 2012), no-good recording (Jouglet et al., 2004), etc.

From a theoretical point of view, the drawback of *Memorization* relies on increases in memory consumption which can be exponential. This drawback limits the quantity of memorized information, such as in *Tabu Search* or SAT solvers. In this paper, we instantiate the *Memorization Paradigm* in a way similar to what has been done in the field of EEA: we set up a Memorization framework for search-tree-based exact algorithms, but with a control on the memory usage. Based on our intuition, we hypothesize that Memorization with limited memory can dramatically accelerate such algorithms in practice. By embedding a simple Memorization technique into their *Branch & Bound* algorithm, Szwarc et al. Szwarc et al. (2001) solve the single-machine total tardiness problem on instances with up to 500 jobs. Other work on standard *Memorization* techniques that are applied to sequencing problems has been presented in T'kindt et al. (2004), where the benefit of such techniques is well demonstrated.

Search tree algorithms are based on the idea of enumerating all possibilities via a search tree that is 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 re- cursively and returns the globally optimal solution. As the basic structure is simple, the critical question is how to prune the search tree such that the exploration of unpromising nodes is avoided. Dominance conditions are commonly used to cut nodes: if it is proved that a more promising node exists or can be easily found, then the current node is abandoned. This is also the case for *Branch & Bound*, in which the bounding procedure provides an optimistic estimate of the solution quality of each node. If the estimate is not better than the current best solution, in other words, the current node is dominated by the incumbent solution, then the node is cut without being further developed. Similar to the bounding procedure in *Branch & Bound*, *Memorization* can be viewed as another procedure that can help in pruning the search tree. In branching algorithms, especially on lower branching levels, isomorphic subproblems may appear exponentially many times and *Memorization* can be used to avoid solving

identical problems multiple times.

To the best of the authors' knowledge, Memorization has not yet been systematically considered when designing search-tree-based algorithms, such as the bounding procedure in *Branch & Bound*; certainly, it is rarely considered in sequencing problems. The aim of this paper is to promote the systematic incorporation of Memorization into search-tree-based algorithms to better prune search trees. In the following sections, we first describe the general framework of *Memorization* (section 2), followed by some guidelines for implementation (section 3). Then, we apply the framework to four scheduling problems: $1|r_i|\sum C_i$ (section 4.1), $1|\tilde{d}_i|\sum w_i C_i$ (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. Applying *Memorization* to 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, search strategies and characteristics of the problem. In this section, we consider possible scenarios that may arise in sequencing problems. Then, we present the possible schemes of Memorization and explain how to choose the correct scheme for a given scenario.

Consider a generic sequencing problem in which n jobs $J = \{1, ..., n\}$ are to be scheduled. Each job j is defined by a set of features such as a processing time p_j, and a due date d_j, which depends on the problem under consideration. We adopt an intuitive way of representing the content of a node or sub-problem: for example $123\{4, ..., n\}$ represents a sub-problem in which jobs $\{1, 2, 3\}$ have already been fixed by branching, to the first three positions of the sequence, while the jobs to be scheduled afterward are $\{4, ..., n\}$.

By *active nodes* we denote the nodes that have been 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*, as defined in 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, regardless of the internal order of the jobs inside (it is defined as the sum of the 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 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 of 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: *forward branching*, *backward branching* and *decomposition branching*.

As indicated by their names, *forward branching* (resp. *backward branching*) assign the job being branched to the first (resp. last) free position. As a contrast, *decomposition branching* is less commonly seen on sequencing problems. When applied at a given node, the job that is being branched 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.However, here we restrict our study to the situation in which

the two sub-problems can be uniquely determined in polynomial time by making use of some specific problem properties. This situation occurs, for instance, in the $1||\sum T_i$ problem which will be discussed later on.

2.2. Search strategies

During the execution of a 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 chooses the node to branch among active nodes 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 exponential space usage since the search tree is explored level by level. *Best-first* chooses the node to explore according to its lower bound. Therefore, the space usage in the worst case is usually also super-polynomial as in *breadth-first* search.

Conventionally, when constructing a search tree based algorithm, the *depth-first* is most commonly adopted. However, according to T'kindt et al. (2004), this choice is highly questionable.

2.3. Memorization schemes

The memorization scheme that is 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, various memorization approaches can be used. The differences among them lie in the choice of which information to store and the way in which the stored information is used. Below, we discuss three different memorization schemes that are helpful for efficiently solving some sequencing problems.

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

2.3.1. 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, implied by S. In that case, if A has already been solved (for instance, by 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, 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 that corresponds to the optimal solution of node A may be missed if one of its ascendant nodes is cut due to a dominance condition. In Figure 1, assume that node D should have led to the optimal solution of problem S but was cut by a dominance condition. Applying *solution memorization* may then lead to the memorization of another solution β to S that is not optimal with respect to S. Troubles may appear if the optimal solution to the original problem (associated with the root node) is, for instance, given by node E. Solution memorization may lead to not exploring node B and directly

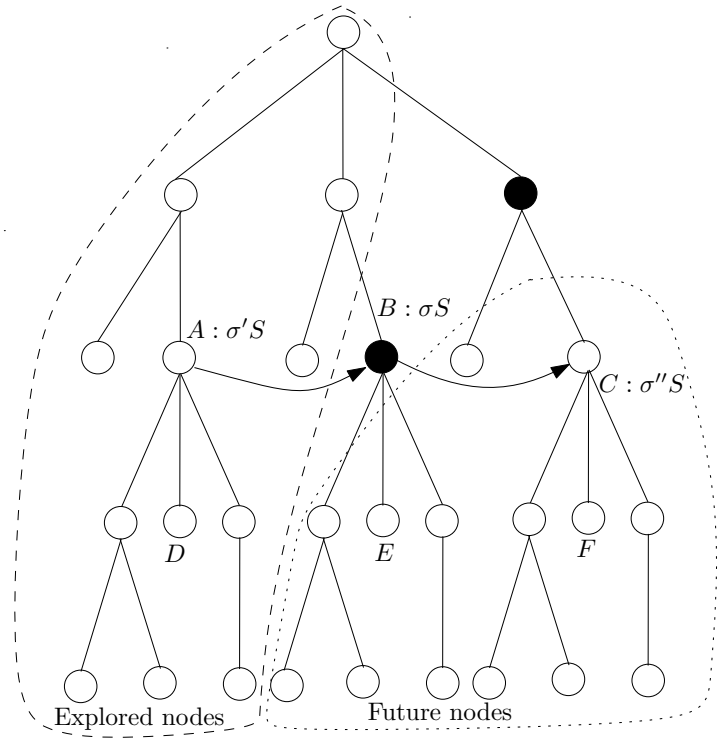


Figure 1: Solution Memorization

replacing S by the "best" solution that was found from node A. Consequently, the globally optimal solution is missed. This situation occurs whenever the dominance condition that pruned node D would not have pruned node E: in the remainder of the paper, conditions of this kind are referred to as *context-dependent dominance conditions* since they depend on the context of each node (typically, the initial partial sequence σ, σ' and σ''). In contrast, a *context-independent dominance condition* would have pruned nodes 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 play a very important role in the algorithm, this may slow down the algorithm, even if *solution memorization* is successful. Another approach for managing context-dependent dominance conditions is to extend the memorization from "solutions" to "lower bounds" when the branching algorithm involves a bounding mechanism. In this version of *Memorization*, we assume that all dominance conditions are retained in the algorithm. When node A is created, a lower bound is computed, which represents the best solution value that we may expect from the sub-tree of A. This lower bound is based on the cost function value of the sequence σ' which has already been 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 becomes increasingly precise. When all leaf nodes of the sub-tree of A have been explored, this value finally becomes tighter (higher) than the initial value that was computed at node A. Since the objective function value of σ' is known, we can then deduce the lower bound value that corresponds to S when it is scheduled after σ', and memorize it. Now

when node B is opened, instead of computing its lower bound, we can obtain it by retrieving the lower bound of S directly from memory and then adding the objective function value of σ. Using this approach, the lower bound that we obtain is tighter, and node B is more likely to be cut. Moreover, the lower bound computation at node B, which may be time-consuming, is saved. Note that the lower bound values, for nodes cut by context-dependent dominance conditions, still need to be computed and considered (which introduces an extra cost). Lower-bound memorization can be a good alternative to *solution memorization* with context-dependent dominance conditions turned off if 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 no nodes in a sub-tree 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 refer to the described memorization technique, including the memorization of optimal solutions and the memorization of lower bounds, as *solution memorization*, since both of them are related to the memorization of the "best solution" of the problem that is associated with a node.

2.3.2. Passive node memorization

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

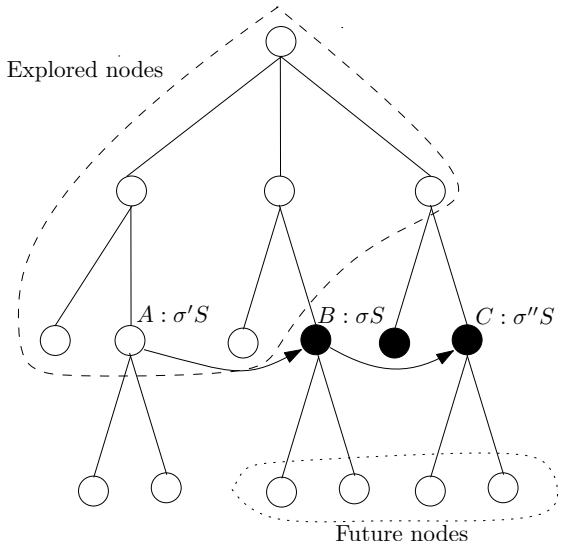


Figure 2: Passive node memorization)

Consider the branching situation depicted by Figure 2. Again, active nodes are colored in black and B is the current node. Assume that a node A exists along with the explored node, with σ' being a different permutation of the same jobs used in σ. If the partial sequence σ' has been memorized, then one of two situations occurs. If σ' dominates σ then B can be cut since it cannot lead to a better solution than A. If no such σ' that dominates σ is available, then σ can be memorized to possibly prune a future node such as C. Note that *solution memorization* and

passive node memorization may 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 nodes B and C, both *solution memorization* and *passive node memorization* indicate 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 function *check* must be evaluated on two different sequences of the same jobs that have the same starting time, and the implementation of this function is problem-dependent. Since the memorized sequence results from branching decisions, we call this *passive node memorization*.

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 the 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. A 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 concept as *passive node memorization*, but with additional operations. As illustrated in Figure 3, at a given node $B = \sigma S$, we first check, as 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, the improving sequence necessarily belongs to a part of the search tree that has not yet been 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 by searching for a neighboring 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 π, as long as this algorithm is fast. Notice that this idea is strongly related to the theoretical mechanism called *merging* (Shang et al., 2017)and is designed to provide good worst-case time complexities. If such a sequence π can be constructed, 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 node σS (for example, consider $\pi = \sigma''$). Thus, it is important when applying *predictive node memorization* to remember that πS still needs to be branched on. Additionally, the extra cost of generating π must be limited 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, the following property holds.

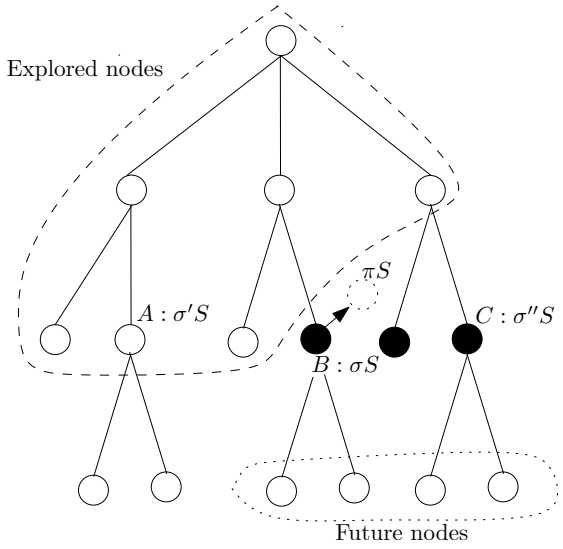


Figure 3: Predictive node memorization

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

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 $B = \pi S$, where σ and π are 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, in *solution memorization*, node B can also be pruned since the optimal solution of job 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 that dominates πS can be generated. Moreover, even if such a node αS is generated and πS is pruned, the same issue occurs in 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 it is impossible to determine which memorization scheme is dominant. However, in practice, *passive node memorization* may be preferred to *solution memorization*. Notably, if the problem is not decomposable, then it may be necessary to solve the sub-problem that consists of jobs S at both node A and and B. However, with *passive node memorization*, node B may be pruned whenever π is dominated by σ.

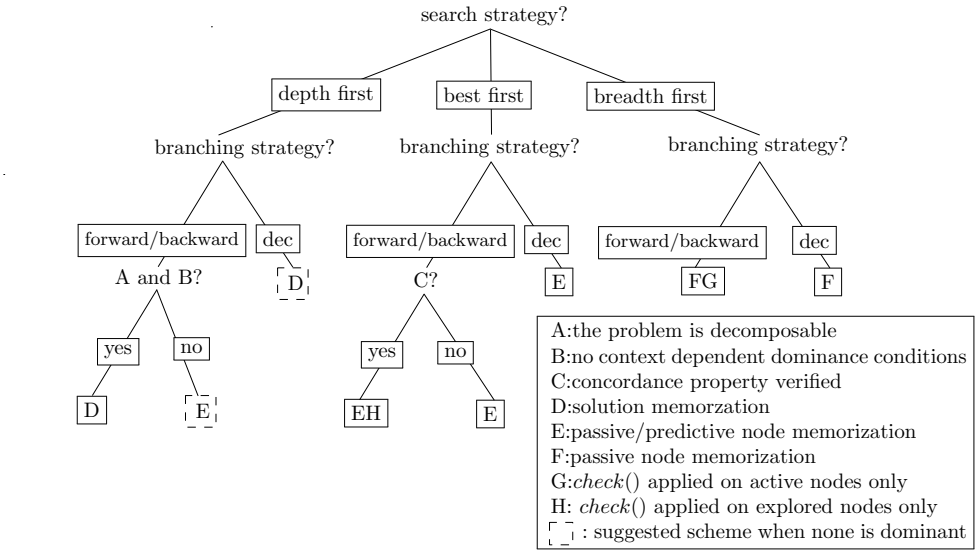


Figure 4: Decision tree for choosing the memorization scheme

2.4.2. Forward branching and best-first search strategy

The following property holds.

Property 2. *With forward branching and the best-first strategy, solution memorization is useless. Passive node memorization or predictive node memorization can be applied only to explored nodes if the Concordance Property (Definition 2) is satisfied.*

Proof. To apply *solution memorization* at a given node, the sub-problem concerning S must be solved first so that its optimal solution can be memorized. This strategy is not compatible with the *best-first* search strategy. In fact, no sequence can be stored before that the *best-first* search reaches a leaf node, while once a leaf node is reached, the optimal solution is obtained.

When *passive node memorization* and *predictive node memorization* are applied, the search, at a given node, of a dominant sub-sequence needs to be performed only in the set of explored nodes when the concordance property holds. As the *best-first* search strategy always considers 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 must consider both explored and active nodes for node pruning. Moreover, 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, both memorization schemes should be considered and compared to determine which is best.

2.4.3. Forward branching and breadth-first search strategy

With forward branching and breadth-first search, the following property holds.

Property 3. *With forward branching and the breadth-first strategy, solution memorization is useless. Passive node memorization should be chosen and 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 that are dominated by A are discarded. If, in turn, A is dominated by another node, then it is pruned. There is no need to consider explored nodes since explored nodes on higher levels have fewer fixed jobs; therefore, they are not comparable with the current node. Moreover, *predictive node memorization* cannot outperform *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 in 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 in which *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-problems defined by S_1 are identical in A and B. Then, the optimal sequence for S_1 that is found when solving A can be reused on B by *solution memorization*. In contrast, *passive node memorization* cannot handle this case since σ and π contain different jobs and, hence, are incomparable. *Predictive node memorization* may or may not cut B, depending on whether a dominant prefix can be generated or not.

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*. In contrast, 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 can be dominant in some cases, the memory limitation does not allow all of them to be applied and our experience suggests that it is preferable to apply *solution memorization*. This is due to the special structure of nodes $\sigma_1 S_1 ... \sigma_k S_k$, which makes the prefixed jobs more 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 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 the best-first strategy, solution memorization cannot be applied. Passive node memorization and predictive node memorization must be applied only to explored nodes whenever the concordance property holds and the check function for 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 both explored and active nodes.*

Proof. The proof is similar to that of Property 2. □

2.4.6. Decomposition branching and breadth-first search strategy

Property 5. *With decomposition branching and the breadth-first strategy, solution memorization cannot be applied. Whether node memorization should be applied to active nodes depends only 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 such a way 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 consideration

In this section we discuss efficient implementations of the memorization schemes and provide, when necessary, choices specific to the sequencing problems that are approached in the remainder of the paper. The key point is to have fast access to memorized partial solutions. Henceforth, we implement a database as a hash table that contains all the memorized solutions. By well choosing the hash function, the hash table 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 set of related jobs. The returned result should be $\langle \pi, opt(\pi|t_0) \rangle$ which is the optimal sequence that is associated with S when starting at time t_0, and its corresponding objective function value. Therefore, $\langle \pi, opt(\pi|t_0) \rangle$ defines the elements that 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 obtain a unique hash key for each given pair $\langle t_0, S \rangle$, even if it 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 requires $O(|S|)$ operations for each returned sequence. We may also incorporate the sum of the id's of jobs in S into h to obtain a more exact key, but this correspondingly increases the time needed to construct the key; however, it does not prevent checking whether a returned sequence π is a permutation of jobs in S or not.

For *passive* and *predictive node memorization*, implementation decisions are more dependent on the problem and 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 as follows:

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

where C_{max} refers to the makespan of a partial sequence, and $E_{min}(\pi)$ refers 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 the minimization of any regular objective function, 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 $\langle \sigma, C_{max}(\sigma), E_{min}(\sigma), opt(\sigma|t_0), ExpAct \rangle$, where $ExpAct$ is a flag that indicates 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 that is used for querying. Additionally, 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 that used in 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 $\langle 0, S_{\sigma_1} \rangle$ where S_{σ_1} refers 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, the elements dominated by the added one are removed. Moreover, due to memory limitations on the computer used for testing, in some instances, we need to clean the database when it is full. More precisely, in our experiments, the RAM is of size 8Gb; hence, the database size is also limited to 8Gb.

A cleaning strategy is needed to remove unpromising elements, i.e., those that are not expected 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 strategies during our experimentation.

FIFO: First In First Out

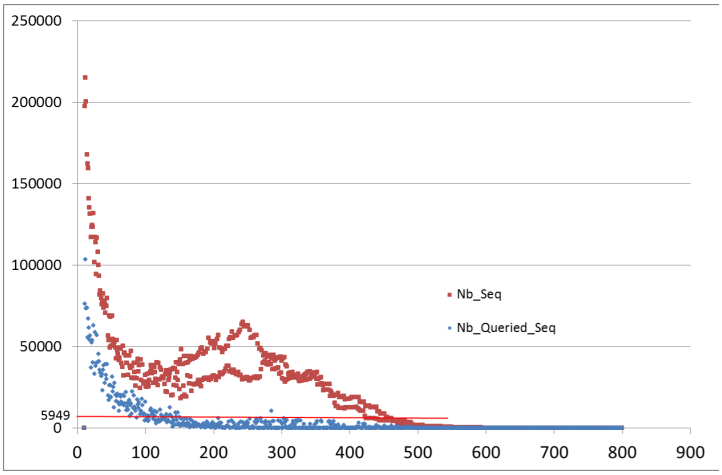
First In First Out is one of the most common database cleaning strategies: when the memory is full, we remove the elements that were added first. An extra structure is needed to record the order of the elements according to the times when they were added. When the database is full and a long sequence is waiting to be inserted, it may be necessary to remove more than one element to free up enough space.

BEFO: Biggest Entry First Out

In the Biggest-Entry-First-Out strategy, the biggest elements (the longest sequences) are removed from the database to free up enough continuous memory for storing new elements. For *solution memorization*, this means removing nodes at higher levels in the search tree. The impact of this cleaning strategy on *solution memorization*, intuitionally, is illustrated in 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. Sequences with many jobs (for example, 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 the sub-problems generated by one or several branchings can be found in memory.

However, for *passive and predictive node memorization*, the strategy involves removing nodes at lower levels of the search tree. These nodes correspond to sub-problems with many jobs that have already been fixed (and memorized) and few jobs to schedule. It may be possible that the extra cost of memorizing a long fixed partial sequence is higher than that of solving the corresponding small sub-problem directly without memorization. Since this is not obvious from a theoretical point of view, some preliminary experiments were performed to investigate whether it is better to also 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 free up a large amount of space to decrease the time cost of the cleaning operation.



Nb_Seq: number of sequences of a given size that are stored in memory.
Nb_Queried_Seq: number of sequences of a given size that are used to avoid solving identical problems more than once.

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

LUFO: Least Used First Out

Figure 5 also suggests another cleaning strategy, since many 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 maintain 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 former. This is because 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, which are not reported here, indicate that the FIFO strategy is not efficient for the considered scheduling problems. The BEFO strategy works better than FIFO, but its efficiency is not high enough to affect the computational results. The 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

To evaluate 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 to prune the search tree. According to the memorization framework, as defined in this paper, what they performed is *passive node memorization* with a database cleaning strategy that replaces the shortest stored sequence by the new one when the database is full.

Another aim of that paper was to choose the most suitable search strategy for solving these problems efficiently. In this section, for each of these three problems, we apply the previously defined *Memorization* framework 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 without memorization. Depth_X, Best_X and Breadth_X refer to *Branch & Bound* algorithms with the corresponding search strategies and *Memorization* of type X, where $X = S$ represents *solution memorization*, $X = Pa$ *passive node memorization* and $X = Pr$ *predictive node memorization*. For *predictive node memorization*, we use the *k-perm* heuristic to search for new sequences, as described in section 2.3.3.

The *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 $\sigma = \sigma^0\sigma^k$, where k is an input parameter and σ^k is the sub-sequence of the last k jobs in σ. The *k-perm* heuristic consists of enumerating all permutations of jobs in σ^k to obtain sequence σ^ℓ. Then, the first sequence $\sigma^0\sigma^\ell$ that is found that dominates $\sigma^0\sigma^k$, 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 that $k = 5$ should be chosen in our implementations to obtain the most efficient *predictive node memorization scheme*. Notice that *k-perm* search is not performed when the *breadth-first* strategy is used, since the memorization applied on active nodes already covers the effect of *k-perm*.

In result tables, Tavg and Tmax denote the average and maximum solution time in seconds. Navg and Nmax are respectively the average and maximum number of nodes created during the tree search. 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, which is 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 multiple cleanings. Also note that LUFO is chosen as the cleaning strategy according to preliminary experimentation.

All tests have been performed on an 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 requires n jobs to be scheduled on one machine in a way that minimizes the sum of the 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.

When the processing time of jobs is generated with $p_{max} = 100$, the best computational results to date is reported by Tanaka and Fujikuma (2012). This algorithm called Sipsi uses a graph representation during the solution and the size of the graph depends on p_{max}, hence, when p_{max} is large, the algorithm is restricted by its memory usage. Also, since the algorithm is memory consuming and is not a search tree algorithm, it is not obvious to integrate the *Memorization* mechanism with it. Therefore, we choose the algorithm described by T'kindt et al. (2004) to show the effectiveness of *Memorization* and then we also provide a comparison of the results obtained by *Memorization* with the result given by Sipsi.

The work of T'kindt et al. (2004) uses the *Branch & Bound* algorithm of Chu (1992) as a basis, so *forward branching* is adopted as the branching strategy. The lower bounds and dominance

conditions from Chu (1992) are maintained. A so called *DP Property*, which is added as a new feature by T'kindt et al. (2004), is equivalent to *passive node memorization* in our terminology. The *check*() function is based on a dominance condition that was given in Chu (1992), and was defined by T'kindt et al. (2004) as follows:

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

where Ω is the set of jobs that remain to be scheduled after sequences σ and π, and $E_{min}(\sigma) = \max(C(\sigma), \min_{r\in\Omega} r_i)$, with $C(\sigma)$ denoting 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 Equation 1, and if the *check* in Equation 1 returns 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 that is defined in Equation 2, it is not clear whether the concordance property is satisfied. Hence, when *passive node memorization* is applied with the *best-first* strategy, all nodes need to be considered in the comparisons, while when it is applied with the *breadth-first* strategy, only active nodes need to be considered. Therefore, the choices made by T'kindt et al. (2004) with respect to memorization are maintained. 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 add results for *predictive node memorization* and *solution memorization*. The input was generated following the approach described by Chu (1992), i.e., the processing times were generated uniformly from $[1, 100]$ and the release dates were generated between 0 and $50.5 \cdot n \cdot r$, where r belonging to $\{0.2, 0.4, 0.6, 0.8, 1.0, 1.25, 1.50, 1.75, 2.0, 3.0\}$. Thirty instances were generated for each value of r, thereby leading to 300 instances for each size n from 70 to 350.

Results related to *node memorization* are presented in Table 1. For all three search strategies, *passive node memorization* enables much larger instances to be solved, in comparison to the versions without *Memorization*. This is sufficient to prove the power of *Memorization* in solving this problem.

Depth_Pa and Depth_Pr solve instances with up to 130 jobs. The impact of *k-perm* search on this problem is very limited: *predictive node memorization* leads to almost the same result as *passive node memorization*. It is also worth mentioning that the database cleaning strategy LUFO enables faster solution of large instances. For example, we found that an instance with 140 jobs is solved in 1.6 hours by Depth_Pa with LUFO, while 14 hours are required to solve it when the cleaning strategy of T'kindt et al. (2004) is utilized instead. However, due to the hardness of another instance with 140 jobs, the algorithm Depth_Pa runs out of time.

The Sipsi algorithm is also tested on the same dataset and result is provided in Table 2. It is very efficient and is able to solve instances with up to 300 jobs. However, when using a newly generated dataset with $p_{max} = 1000$ instead of 100, Sipsi can only solve instances with up to 130

	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 the new algorithms on the $1|r_i|\sum C_i$ problem

jobs, as Depth_Pr can (see Table 3). This shows that unlike Sipsi, Depth_Pr is not sensitive on the range of processing time of jobs.

	n	130	140	150	200	250	300	350
Sipsi	Tavg	25.98	35.42	56.72	227.20	642.77	1307.56	OOM
	Tmax	231.69	351.13	1172.89	3993.28	5731.45	10683.34	

Table 2: Results of the algorithm Sipsi on the $1|r_i|\sum C_i$ problem

	n	70	80	90	100	110	120	130	140
Depth_Pr	Tavg	0.07	0.23	0.48	2.21	6.62	23.11	49.26	OOT
	Tmax	2.87	10.34	18.94	113.69	843.75	3438.79	4408.37	
Sipsi	Tavg	27.93	48.76	75.26	119.95	159.15	251.41	328.50	OOM
	Tmax	133.23	283.94	418.68	1616.69	1163.64	3271.22	2463.89	

Table 3: Results of algorithms on new instances with greater processing time ($P_{max} = 1000$)

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

The $1|\tilde{d}_i|\sum w_iC_i$ problem requires n jobs to be scheduled on a single machine. Each job i has a processing time p_i, a weight w_i and a deadline $\tilde{d}_i$ that must be met. The objective is to

minimize the total weighted completion time $\sum w_i C_i$. 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 the performance of the algorithm of Posner (1985) being slightly superior. The basic algorithm described by T'kindt et al. (2004) is a combination of the algorithms of Posner (1985); Potts and Van Wassenhove (1983), which is obtained by incorporating the lower bound and the dominance condition of Posner (1985) into the *Branch & Bound* algorithm of Potts and Van Wassenhove (1983). The algorithm of Tanaka et al. (2009) called Sips is known as efficient on solving this problem, but again this algorithm is not a search tree algorithm and is already memory consuming, hence we base our work on the algorithm of T'kindt et al. (2004) and provide the test result of Sips at the end.

We adopt *backward branching* as branching scheme as done in Posner (1985); Potts and Van Wassenhove (1983). Similar to what is done on the $1|r_i|\sum C_i$ problem, the *DP Property* is considered by T'kindt et al. (2004), which is equivalent to *passive node memorization*. The *check*() function is defined as follows, where Ω is the set of jobs to be scheduled before σ and π.

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

The items stored in the database are $\langle \sigma, opt(\sigma|\sum_{i\in\Omega} p_i), ExpAct \rangle$.

4.2.1. Application of the memorization framework and improved results

This problem is *decomposable* according to Definition 1. According to the decision tree in Figure 4, 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. In this section we compare four *Branch & Bound* algorithms: the *node memorization* with the three search strategies and the *solution memorization* based on *depth-first* search.

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

Regarding *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 *solution memorization*, as described in section 2.3.1.

Proposition 1. *With the check() function that is defined in Equation 3, our algorithms satisfy the Concordance Property (Definition 2).*

Proof. Consider two nodes $S\sigma$ and $S\pi$. First notice that the same sub-problem is to be solved in both nodes, which consists of scheduling jobs from S, starting from time 0. The lower bound that is used in the algorithm (see Posner (1985); Potts and Van Wassenhove (1983)) returned on the sub-problems on S is the same for both 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 $LB(S\sigma) \leq LB(S\pi)$.

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, according to the same reasoning; hence, $check(\pi, \sigma) = 1$. □

Following the test plan described by Potts and Van Wassenhove (1983), for each job i, the processing time p_i is an integer that is 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 increasing from 0.6 to 1.0 in steps of 0.1 and R increasing from 0.2 to 1.6 in steps of 0.2. To avoid generating infeasible instances, an (L, R) pair is only used when $L + R/2 > 1$; hence, only 30 (L, R) pairs are used, for each of which 10 feasible instances are generated, thereby yielding a total of 300 instances for each value of n from 40 to 140.

The result is presented in Table 4. For *depth-first* search, without *memorization* the program is "out of time" on instances with 50 jobs, while *solution memorization* and *passive node memorization* enable it to solve instances with up to respectively 90 and 100 jobs, with faster performance achieved using *passive node memorization*. With the activation of *k-perm* search, Depth_Pr can solve 20 more jobs than Depth_Pa. This strongly proves the power of all three memorization schemes.

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

On *breadth-first*, without *Memorization* Breadth- cannot even solve all instances with 40 jobs, while with *passive node memorization* instances of 130 jobs are all solved in an average solution time of 65.5 seconds. Again, as for the $1|r_i|\sum C_i$ problem, LUFO accelerates the solution process, but does not enable the solution of larger instances.

The result of Sips is presented at the end of the table, it can efficiently solve instances with up to 140 jobs. In order to test the sensitivity of these algorithms on the range of input data values, we generated a new dataset with processing time generated from $[1, 1000]$ and job weights from $[1, 100]$. This leads to the results in Table 5. Both algorithm Breadth_Pa and Sips solve less instances than before, with Breadth_Pa solves up to 110 jobs and Sips solves up to 100 jobs.

		40	50	60	70	80	90	100	110	120	130	140	150
Depth-	Navg	104915.3	OOT										
	Nmax	14536979											
	Tavg	0.9											
	Tmax	74.0											
Depth_S	Navg	763.4	2509.5	7919.1	27503.2	135724.0	189719.1	OOT					
	Nmax	17699	60462	228940	1660593	9841123	14388210						
	Tavg	0.4	0.5	0.9	2.5	22.1	38.7						
	Tmax	1.0	2.0	14.1	275.0	2876.1	7603.2						
Depth_Pa	Navg	577.4	1973.6	5850.6	21644.8	107804.7	146216.4	430330.1	OOT				
	Nmax	11963	83075	137580	1004546	12052793	4321070	13234264					
	Tavg	0.4	0.4	0.5	0.9	7.4	5.9	21.2					
	Tmax	1.0	1.0	2.3	39.0	1488.2	312.0	1055.7					
Depth_Pr	Navg	342.4	902.9	2512.6	7233.0	20196.3	35458.0	99387.1	274871.1	551713.3	OOT		
	Nmax	3865	17447	50003	187425	665376	768802	1781123	14713483	11236833			
	Tavg	0.4	0.4	0.4	0.6	1.0	1.3	4.1	14.3	34.3			
	Tmax	0.4	1.0	1.3	5.0	30.0	21.0	64.3	901.0	1255.0			
Best-		OOT											
Best_Pa	Navg	350.9	885.9	2125.7	6866.0	20700.7	28155.0	71459.7	OOT				
	Nmax	3912	20889	43000	440623	1348082	1252600	1668977					
	Tavg	0.4	0.4	0.4	0.6	2.0	2.0	6.4					
	Tmax	0.405	1.0	1.1	30.0	241.0	130.0	391.2					
Best_Pr	Navg	313.7	730.6	1680.8	4494.6	11060.6	16305.9	39053.7	132949.2	220989.7	390481.2	OOT	
	Nmax	3865	11762	28253	120259	319068	299540	607871	10659343	7578570	7630213		
	Tavg	0.4	0.4	0.4	0.5	0.8	1.0	2.5	35.0	20.5	83.6		
	Tmax	0.4	1.0	1.0	4.0	23.0	19.9	58.2	5008.0	1137.0	6806.5		
Breadth-		OOM											
Breadth_Pa	Navg	364.2	922.9	2074.1	6375.8	16474.2	24731.0	59474.3	105989.8	225013.9	464121.4	OOT	
	Nmax	4701	16952	36960	437697	881817	868876	1561063	5975094	7577492	23966269		
	Tavg	0.0	0.0	0.0	0.2	0.4	0.9	2.2	9.1	16.8	65.5		
	Tmax	0.015	0.1	0.7	9.0	31.1	31.0	67.2	1353.0	1135.0	8232.3		
Sips	Tavg	0.070	0.160	0.320	0.550	1.057	1.829	2.450	5.672	5.675	9.107	12.013	OOM
	Tmax	0.468	1.030	2.278	3.916	19.609	61.683	20.062	410.361	68.094	246.138	175.969	

Table 4: Results of the new algorithms on the $1|\tilde{d}_i|\sum w_iC_i$ problem

	n	90	100	110	120
Breadth_Pa	Tavg	8.869	115.62	121.93	OOM
	Tmax	464.711	16562.41	5244.38	
Sips	Tavg	19.14	33.00	OOM	
	Tmax	225.11	292.58		

Table 5: Results of algorithms on new instances with greater processing time and weights ($p_{max} = 1000, w_{max} = 100$)

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

In the $F2||\sum C_i$ problem, n jobs must be scheduled on two machines, namely, M_1 and M_2. First, each job i needs to be processed on M_1 for $p_{1,i}$ time units and then processed on M_2 for $p_{2,i}$ time units. The objective is to minimize the sum of the completion times of jobs. We restrict to the set of permutation schedules in which there always exists 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. The *Branch & Bound* algorithm that was proposed by T'kindt et al. (2004) based on the *Branch & Bound* algorithm of Della Croce et al. (2002) was the referential one until 2016, when Detienne et al. (2016) proposed a new and very efficient *Branch & Bound* algorithm that is capable of solving instances with up to 100 jobs. But again, the algorithm of Detienne et al. (2016) is based on a graph representation and is memory consuming depending on the range of processing time values. We therefore again base the *Memorization* on the algorithms described by T'kindt et al. (2004).

The branching scheme that is adopted in this algorithm is *forward branching*, and all the three search strategies are considered. 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, and $C_2(\cdot)$ is the completion time of a given sequence on the second machine. The memorized items 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, where σ is a fixed sequence, the optimal solution of sub-problem S depends on the order of the jobs in σ. According to the decision tree in Figure 4, 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 this section we compare four *Branch & Bound* algorithms: *node memorization* applied to the three search strategies and *solution memorization* based on *depth-first* search with.

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 satisfied. 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 with the *best-first* strategy. For the *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 *solution memorization*, as described in section 2.3.1. The items stored into 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. In addition, $opt(\pi|(t_1, t_2))$ is the sum of the 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 are the same as per T'kindt et al. (2004), as described in the previous section.

Thirty instances are generated for each size n from 10 to 130, with the processing times generated randomly from a uniform distribution in $[1, 100]$. The results are given in Table 6. 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*. 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.

Additionally, 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.

The power of *Memorization* is still illustrated on this problem, even though it seems not so strong with respect to the previous two problems. The algorithm of Detienne et al. (2016) is also

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

tested and the result in Table 7 shows that it is far more efficient than our algorithms. We then also tested these algorithms on new instances with processing times generated from $[1, 1000]$ (see Table 8). This has a negative impact on the algorithm of Detienne et al. (2016) which now only solves instances with up to 90 jobs instead of 130 jobs. However this result is still much better than Best_Pr which solves instances with up to only 40 jobs.

	n	40	50	100	110	120	130
Detienne	Tavg	9.05	19.06	353.35	601.0	984.93	OOM
	Tmax	17.32	28.2	563.38	1194.15	2274.12	

Table 7: Results of the algorithm Detienne on the $F2||\sum C_i$ problem

	n	40	50	60	70	80	90	100
Best_Pr	Tavg	192.5	OOT					
	Tmax	2459.0						
Detienne	Tavg	66.16	144.95	380.45	897.16	1518.82	2729.07	OOM
	Tmax	258.88	454.45	1133.72	3015.13	3659.89	7293.87	

Table 8: Results of algorithms on new instances with greater processing time ($p_{max} = 1000$)

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 the main properties and existing results of the problem, then determine parameters for *Memorization* and finally report the computational results.

5.1. Preliminaries

The problem involves scheduling 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 instances with up to 500 jobs in size Szwarc et al. (2001). The latest theoretical developments for the problem can be found in the survey of Koulamas Koulamas (2010). The main properties of the problem can be found in Szwarc et al. (2001), and some of them are given below.

Let $(1, 2, \ldots, n)$ be an 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 the LPT sequence correspond to job* $[k]$ *in the 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 the LPT sequence correspond to job* [1] *in the 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 in positions before or after S_i, implying a structure such as $\sigma_1 S_1 \sigma_2 S_2 ... \sigma_i S_i ... \sigma_k S_k$ 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 job e, which is $\emptyset$; the jobs between e and ℓ; and finally the jobs after ℓ. In the same way, the other 3 branchings 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 in B are scheduled before those in 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 rather two child nodes are created, each 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. follows) 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* algorithm 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 because many 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-problem once, memorizing the solution and reusing it whenever the sub-problem appears again. In addition, 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 into 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 position. 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 such as *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 obvious that we will 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* that could plausible lead to better results than those obtained with *solution memorization*. Consequently, only *solution memorization* 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 made by Szwarc et al. (2001) with respect to *Memorization* were good choices. In the remainder, we investigate the limitations of the *Memorization* technique as implemented by Szwarc et al. (2001) and propose improvements that 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 first retest BB2001 on our machine: an 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 a 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 9 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 7.5 hours, and the program is considered as OOT (Out of Time) if it reaches the time limit. Additionally, 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 9, 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-Enterprise Station with a reduced CPU frequency (<450MHz) and an 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 more jobs.

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

Paradox 2. *Removing the Split procedure (Property 6) from BB2001 drastically accelerates the solution.*

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 192s to 9s (see Table 9). 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 in a way 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. Removing *Split* 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 discouraged this tentative.

At this point, we have a better understanding of the power of *solution memorization* on this problem and we become curious about the effectiveness of memorized solutions. In other words, what are the stored solutions that are used effectively? 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 the BB2001 algorithm.

In Table 9, 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 192s, while BB2001 is limited to instances with 900 jobs.

	n	200	300	400	500	600	700	800	900	1000	1100	1200	1300
Depth-	Navg	306205.8	OOT										
	Nmax	11020671											
	Tavg	1.4											
	Tmax	77											
BB2001	Navg	12244.7	50662.9	130325.4	312115.8	521479.8	917491.0	1472547.3	2213671.2	3149954.5	OOT		
	Nmax	135242	799870	1313084	3371277	5462573	8522132	13866537	20453973	27246555			
	Tavg	0.1	2.0	8.3	33.7	81.1	209.5	463.4	855.3	1586.2			
	Tmax	3	50	117	491	1140	2579	5858	10003	18097			
NoSplit	Navg	11307.2	47835.0	122962.4	295104.1	493047.5	870015.8	1406250.0	2110070.2	3009635.4	4102758.9	OOM	
	Nmax	132497	776561	1335920	3239773	5348951	8302464	13410893	20227299	26691043	38293210		
	Tavg	0.1	1.0	4.5	18.0	42.7	107.5	230.3	417.7	778.2	1258.6		
	Tmax	2	28	64	265	613	1374	2886	4911	9151	16855		
NoSplit_LUFO	Navg	11307.2	47835.0	122962.4	295104.1	493047.5	870015.8	1406250.0	2110070.2	3009635.4	4102758.9	5314954.0	OOT
	Nmax	132497	776561	1335920	3239773	5348951	8302464	13410893	20227299	26691043	38293210	54926916	
	Tavg	0.1	1.0	4.5	18.0	42.7	107.5	230.3	417.7	778.2	1258.6	1991.7	
	Tmax	2	28	64	265	613	1374	2886	4911	9151	16855	26115	

Table 9: 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 to its efficiency. However, the striking point of these experiments relates to the comparison between the version of BB2001 without the *Memorization* mechanism (algorithm Depth-) and NoSplit_LUFO. Table 9 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. Advice is provided for choosing the best memorization approach according to the branching scheme and the search strategy of the algorithm.

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 10 provides a summary of the conclusions obtained.

Problem	Largest instances solved		Features of the best algorithm with memorization
	Without memorization	With memorization	
$1\|r_i\|\sum C_i$	80 jobs	130 jobs	*depth-first+ predictive node memorization*
$1\|\tilde{d}_i\|\sum w_iC_i$	40 jobs	130 jobs	**+** *predictive node memorization*
$F2\|\|\sum C_i$	35 jobs	40 jobs	*best-first+ predictive node memorization*
$1\|\|\sum T_i$	300 jobs	1200 jobs	*depth-first+ solution memorization*

Table 10: Conclusions on the tested problems

Fundamentally, what we call the *Memorization Paradigm* relies on a simple but potentially very efficient idea: memorizing what has already been done to avoid solving identical subproblems 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. It is noteworthy that the four scheduling problems dealt with in this paper mainly serve as applications illustrating how *Memorization* can be done in an efficient way. However, it is also clear that it can be applied to other hard combinatorial optimization problems, making this contribution interesting beyond scheduling theory. In 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. In addition, 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 also be very promising to see how the 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 or which solutions to remove from the database, through a learning process.

Acknowledgement

We thank Shunji Tanaka and Boris Detienne who kindly provided us the code of their algorithms.

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, Ch. 13.

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.

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.

Sewell, E. C., Jacobson, S. H., 2012. A branch, bound, and remember algorithm for the simple assembly line balancing problem. INFORMS Journal on Computing 24 (3), 433–442.

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.

Tanaka, S., Fujikuma, S., 2012. A dynamic-programming-based exact algorithm for general single-machine scheduling with machine idle time. Journal of Scheduling 15 (3), 347–361.

Tanaka, S., Fujikuma, S., Araki, M., 2009. An exact algorithm for single-machine scheduling without machine idle time. Journal of Scheduling 12 (6), 575–593.

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., 2017. Exact algorithms for maximum independent set. Information and Computation 255(1), 126–146.

Xiao, M., Tan, H., 2017. Exact algorithms for maximum induced matching. Information and Computation 256, 196–211.

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.