Version of Record: https://www.sciencedirect.com/science/article/pii/S0305054820302884
Manuscript_04f8b03258fa925e0d4104f4a24dbd88

Branch & Memorize Exact Algorithms for Sequencing problems: Efficient embedding of Memorization into Search Trees

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

[a] *University of 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, enables to speed up algorithms at the price of increased space usage. In this work, we focus on search tree algorithms applied to sequencing problems. In these algorithms, on lower branching levels, isomorphic sub-problems may appear exponentially many times and the use of memorization is twofold: on the one hand it avoids repetitive solutions, as they correspond to identical sub-problems; on the other hand, it allows to check for dominance conditions among permutations of the same subset of elements. The idea of memorization appeared for a long time, however, to the best of the authors' knowledge, it was only seldom applied in the number of branching algorithms proposed in the literature. In this paper, we propose a unifying framework for implementing memorization in exact branching algorithms dedicated to sequencing problems. Our proposal leads to the paradigm of *Branch & Memorize* and its implementation to three classical single machine problems is validated by an extensive computational experimentation that shows that the mentioned paradigm consistently improves existing exact branching algorithms. These results emphasize the idea of more systematically embedding memorization in branching algorithms.

Keywords: scheduling, exact algorithms, memorization, branch and memorize, sequencing

1. Introduction

Memorization is a broad idea which has existed for a long time and can be casually described as "Memorize and learn from what has been done so far, to improve future decisions". The application of memorization to algorithms has the goal of speeding their processing, often at the price of an increased space usage. In this general sense, various algorithms in the literature that "intelligently" make use of computer memory can be classified as procedures with memorization embedded, 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

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

solutions again during the search: the so-called Tabu list is, then, the memory used for storage. SAT solvers deduce and then memorize conflict clauses during the tree search to perform non-chronological backtracking (*Conflict Driven Clause Learning*) (Zhang et al., 2001; Biere et al., 2009). Similar ideas also appear in *Artificial Intelligence* as *Intelligent Backtracking* or *Intelligent Back-jumping*, etc. Chvatal (1997) exploits the conflicting clauses memorization in addition with the memorization of already explored solutions to build its *resolution search* method. Resolution search has also been considered by Hanafi and Glover (2002) and Posta et al. (2011).
Dynamic programming algorithms typically involve memorization by storing all states necessary to compute by enumeration optimal solutions. This behavior enables to avoid solving multiple times identical sub-problems, often at the price of an exponential memory requirement. It is well-known that dynamic programming algorithms can be seen as particular search tree algorithms in which the search tree is explored level by level following the so-called *breadth first* search strategy. Search tree algorithms, like branch-and-bound algorithms, are based on the idea of enumerating all possibilities via a search tree that is created by a branching rule. 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 globally optimal solution. The critical question is how to prune the search tree such that the exploration of unpromising nodes is avoided. Dominance conditions and bounding procedures are commonly used to prune nodes, as in the case of branch-and-bound algorithms (e.g., see Morrison et al. (2016)). Similarly, memorization can also be viewed as another procedure that can help in pruning the search tree, including but not limited to identical sub-problems elimination as in dynamic programming. In this paper we focus on search tree algorithms applied to sequencing problems and on how to implement memorization so as to avoid solving either identical sub-problems or "dominated" sub-problems, with respect to the search of optimal solutions.

Memorization has already been considered with the intent of providing theoretical guarantees on the execution of exact exponential search tree algorithms (see, for instance, Fomin and Kratsch (2010)). The objective is to develop exact algorithms that provide a best possible worst-case running time guarantee. When such algorithms explore the search tree, identical sub-problems may appear exponentially many times, and the idea of memorization is then to avoid solving them many times by storing, like in dynamic programming algorithms, the solutions of the already solved sub-problems. For example, this principle was applied to the solution of the *Maximum Independent Set* problem by Robson (1986). By exploiting graph theoretic properties, Robson proposed an algorithm with a worst-case time complexity in $O(1.2109^n)$. This algorithm remained the exact exponential algorithm with the smallest worst-case time complexity until the $O(1.1996^n)$ algorithm of Xiao and Nagamochi (2017) was introduced. Recently, Xiao and Tan (2017) applied memorization to derive an algorithm for the *Maximum Induced Matching* problem running in $O(1.3752^n)$ time and exponential space, while Garraffa et al. (2018) proposed a variation of this principle to the single machine total tardiness problem by introducing a node merging mechanism to avoid solving multiple times identical sub-problems in a search tree. The latter approach led to an exact algorithm that required polynomial space and whose time complexity converged to $O^*(2^n)$ with n the number of jobs to be scheduled.
In the literature, we can found works proposing some memorization techniques inside experimentally efficient exact search tree algorithms. Our literature review mainly focuses on works dealing with sequencing problems. By embedding a simple memorization technique into their search-tree algorithm, Szwarc et al. (2001) solve the single machine total tardiness problem on instances with up to 500 jobs while the previous state-of-the art exact methods with no memo-

rization techniques were limited to 100-150 jobs. Another work on standard memorization techniques applied to sequencing problems was presented in T'kindt et al. (2004), where the impact of such techniques on the effectiveness of search strategies is analyzed. Following this work, Kao et al. (2009), Sewell and Jacobson (2012), Morrison et al. (2014), Li et al. (2018), Li et al. (2020a) and Li et al. (2020b) introduced a branch, bound and remember algoritm for solving scheduling problems and assembly line balancing problems. This algorithm embeds the same memorization technique proposed byT'kindt et al. (2004) and experimental results show a strong improvement of the algorithm due to that technique. Baptiste et al. (2004) and Jouglet et al. (2004) tackled the solution of a single machine scheduling problem by means of branch-and-bound algorithms in which a *no-good recording* technique is used to prune dominated nodes in the search tree. It consists in applying, to each partial solution associated to a node, a fast local search to try to build another node which could "dominate" the current one. If such another node is found, the current one is then pruned. Even if no information is stored in memory, this technique relates to a form of memorization which we will call later on "predictive node memorization". Another interesting study was proposed by Sourd and Kedad-Sidhoum (2008) who considered the solution of a single machine scheduling problem and extended the no-good recording technique by proposing to keep in memory the solutions generated by some local search. This approach is what we will call "predictive node memorization". They noticed that this technique helped in reducing the computational time of the branching algorithm except for the largest instances for which their were faced with penalizing memory limitations.

All the above quoted works show the effectiveness of various more or less elaborated forms of memorization, under different terminologies: branch, bound and remember, no-good recording, resolution search or even dynamic programming dominance. When considering the large amount of publications dealing with branching algorithms, few are finally using memorization to prune the search tree. To the best of our knowledge, no general framework for applying memorization exists in the literature, even if some form of memorization was studied. This paper pursues multiple goals: (1) to extend the findings of T'kindt et al. (2004) by proposing a unified framework for sequencing problems integrating memorization into search tree algorithms (the *Branch & Memorize* paradigm), (2) to discuss properties, links and limitations of various forms of memorization, (3) to highlight the impacts of memorization on classic search strategies, (4) to provide extensive experiments on the application of this paradigm to three classical single machine sequencing problems, showing the effectiveness of branch and memorize algorithms.
In Table 1 are presented various state-of-the-art branching algorithms embedding a technique of memorization and their link with the relevant memorization scheme and related subsection proposed in this paper.

This paper promotes the idea of systematically incorporating memorization into branching algorithms not just for single machine sequencing problems but for any sequencing problem whose solution could be expressed by a permutation of a set of elements.

In the remainder, we first describe the *Branch & Memorize* paradigm (section 2), followed by some discussion on its implementation (section 3). Then, we apply it to three single machine scheduling problems (sections 4 and 5). Finally, we conclude our work in section 6.

2. The *Branch & Memorize* paradigm

Consider a $\mathcal{NP}$-hard minimization sequencing problem in which the goal is to find an optimal permutation of a vector V of n items ($V = \{1, ..., n\}$). We assume that a solution can be represented

Table 1: State-of-the-art methods and their link to memorization

	SM (sect. 2.2.1)	PaNM (sect. 2.2.2)	PrNM (sect. 2.2.3)
Resolution search			
Chvatal (1997)		x	
Hanafi and Glover (2002)		x	
Posta et al. (2011)		x	
Branch, bound and remember			
Sewell and Jacobson (2012)		x	
Morrison et al. (2014)		x	
Li et al. (2018)		x	
Li et al. (2020a)		x	
Li et al. (2020b)		x	
No-good recording			
Baptiste et al. (2004)			x
Jouglet et al. (2004)			x
Sourd and Kedad-Sidhoum (2008)			x
Miscellaneous			
Szwarc et al. (2001)	x		
T'kindt et al. (2004)		x	

SM: Solution memorization
PaNM: Passive node memorization
PrNM: Predictive node memorization.

by a permutation, i.e. a sequence, of the vector items and a related branching algorithm that builds nodes of the search tree corresponding to sub-problems where the position of several items in the permutation has been fixed. For instance, in single machine sequencing, the items permutation corresponds to the jobs sequence. We adopt an intuitive way of representing the content of a node, by using lower case letters for fixed items and capital letters for items subsets to be fixed. For example, a node $\sigma S = 123\{4, ..., n\}$ represents a sub-problem in which items $\{1, 2, 3\}$ have already been assigned according to the order $(1, 2, 3)$, to the first three positions of the permutation, while the items to be sequenced afterward are $\{4, ..., n\}$. Formally, any node of a search tree can be defined by $\sigma_1 S_1 \sigma_2 S_2 ... \sigma_k S_k$, with the $\sigma_j{}'s$ being partial sequences of items and the $S_j{}'s$ being sub-problems that remain to be solved. Notice that, formally speaking, σ_1 and S_k can be empty. At any iteration, a search tree algorithm is defined by a list of nodes previously created but not yet developed, called *active nodes*. Some nodes called *explored nodes*, have already been branched on leading to the creation of children nodes. The *branching rule* is the rule which defines how to create children nodes while the *search strategy* refers to the rule indicating how to select the next *active node* to branch on. Following classic strategies in branching algorithms, a bounding mechanism as well as dominance conditions (also called pruning rules) can be used to eliminate nodes that are not leading to an optimal solution. By extension, we say that node A dominates node B, if the best complete solution in the subtree rooted by A is not worse (in the sense of the objective function) than the one in the subtree rooted by B. We refer to Morrison et al. (2016) for a recent overview of some exact branching algorithms.
In the remainder, we also use the notion of *decomposable problems* as introduced in T'kindt et al. (2004).

Definition 1. *Let* $\{1, ..., i\}\{i + 1, ..., n\}$ *be a problem to be solved. It is decomposable if and only if an optimal solution of the sub-problem* $\{1, ..., i\}$ *does not depend on an optimal permutation of* $\{i + 1, ..., n\}$*, and vice versa.*

The *Branch & Memorize* paradigm, which is detailed in the context of sequencing problems, gathers all techniques which take advantage of the tree exploration to memorize information on the visited nodes in order to prune further this tree without missing optimal solutions of the problem. We propose here various implementations of this paradigm for sequencing problems. While presenting these versions of memorization in search tree algorithms, we also discuss some of their properties or limitations. Finally, we summarize these information by providing some guidelines on which version to choose, depending on the problem under consideration and its known properties. We first recall some basic notions related to search tree algorithms before introducing different implementations of the *Branch & Memorize* paradigm.

2.1. Branching schemes and search strategies

In this section we briefly review two important mechanisms of search tree based algorithms: the *branching rule* and the *search strategy*. They are discussed in the context of sequencing problems.

The *branching rule*, taking an active node, usually consists in assigning an item to a specific position in the permutation. So, a *branching scheme* defines, at a node, how to choose this item and the positions it can occupy. We consider three classic branching schemes: *forward branching*, *backward branching* and *decomposition branching*. *Forward branching* (resp. *backward*

branching) assigns the item being branched on to the first (resp. last) free position. This item is usually selected according to a rule that depends on the problem. Notice that, with *forward branching* (resp. *backward branching*), the sub-problem associated to a node is defined by σS (resp. $S\sigma$). *Decomposition branching* is less commonly seen on sequencing problems since it is strongly dependent on the identification of structural properties of the problem. When applied at a given node, the item that is being branched on is called a *decomposition item*. When a *decomposition item* is assigned to a position, two sub-problems are generated, implied by the free positions before and after the *decomposition item*. This scenario can be seen in some Divide-and-Conquer like algorithms, for instance the algorithm of Gurevich and Shelah (1987) solving the Hamiltonian Path problem. Certainly one may determine the items that should be sequenced before and after this position by enumerating all 2-partitions of items. However, there are problems for which the two sub-problems can be uniquely determined in polynomial time by making use of some specific problem properties. As we will see, this situation occurs in machine sequencing on the $1||\sum T_j$ problem which will be discussed later on.

The *search strategy* is a mechanism which is used to select, at each iteration of the algorithm, the next active node to branch on. The classic search strategies are *depth-first*, *best-first* and *breadth-first*. The *depth-first* strategy consists in selecting the active node with the highest number of items assigned to a position: break ties by selecting the one with the smallest value of the lower bound. The *breadth-first* strategy consists in selecting the active node with the smallest number of items assigned to a position (no matter how ties are broken). At last, the *best-first* strategy consists in selecting the active node with the smallest value of the lower bound. Notice that both *best-first* and *breadth-first* imply a super-polynomial space complexity, which may affect the performance of the search tree based algorithm. Consequently, in practice, the *depth-first* is the most commonly adopted search strategy.

2.2. Memorization schemes

The theoretical memorization scheme that is presented by Robson (1986) stores an 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. Other memorization schemes can be invented according to the information to memorize. Below, we discuss three different memorization schemes that are helpful for efficiently solving some sequencing problems. They constitute different implementations of the *Branch & Memorize* paradigm.

For the sake of simplicity, we explain the *memorization* schemes in the case of *forward branching*. We leave to the reader their application to other branching rules.

2.2.1. Solution memorization

Basically, *solution memorization* consists in storing in memory, for any node A, the best solution contained in the sub-tree rooted at A. Consider the situation illustrated in Figure 1, where active nodes are colored in black. Node B is the current node to branch on, while σ, σ' and σ'' are different permutations of the same items. 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 and an optimal sequence of S has been memorized, then it may be used directly to solve nodes B and C and it may be no longer necessary to branch on these nodes. This is valid, for instance, if the problem is decomposable.

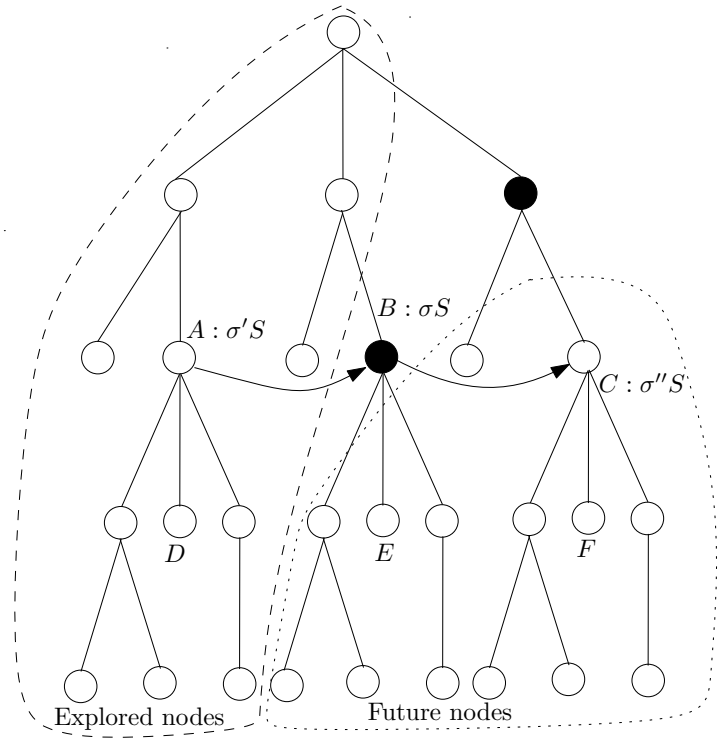

Figure 1: Solution Memorization

The idea being simple, some practical aspects have to be considered. In some situations, it may be not obvious to obtain an optimal solution from the sub-tree rooted at node A. For instance, the leaf node corresponding to an optimal solution of the node A, *i.e.* an optimal sequencing of items in S, might be cut due to some specific dominance conditions used in the search tree algorithm. We call them *context-dependent dominance conditions* since they are dependent on the fixed partial sequence. In Figure 1, assume that node D should have led to the best solution reachable from node A but was pruned by a dominance condition. Applying *solution memorization* may then lead to the memorization of another solution β to S that is not the best one with respect to S. Later, when re-using this solution to avoid solving B or C, an optimal solution of the problem may be missed if this one is in the sub-tree, for instance, rooted at node B, *i.e.* if it is the concatenation of σ and a permutation of S.

We can certainly disable context-dependent dominance conditions while applying *solution memorization*. However, this may slow down the algorithm, even if *solution memorization* is effective. An alternative, in this case, is to extend the memorization from "optimal solutions" to "lower bounds". Whenever an optimal solution of a node A cannot be obtained, we can still memorize the best lower bound obtained during the exploration of the sub-tree of A. This information might be useful when nodes B and C are encountered and it may help to cut these nodes by the bounding procedure. Moreover, the lower bound computation at node B and C, which may be time-consuming, is saved. This situation occured, for instance, when considering problem $1|\tilde{d}_j|\sum_j w_j C_j$ (section 4.2).

Note that the memorization of lower bounds is compatible with the memorization of optimal solutions. 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 known solution" of the problem that is associated with a node. With respect to the literature, the work by Szwarc et al. (2001) can be seen as a preliminary implementation of *solution memorization* as indicated in Table 1.

2.2.2. Passive node memorization

Passive node memorization consists in storing information on the active node σS selected for being branched on. Typically, partial sequence σ is memorized with possibly additional information. 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 "prune" nodes. Consider a search tree that is being explored following the breadth-first search strategy (Figure 2). Again, active nodes are colored in black and B is the current node. Assume that an explored node A exists, with σ' being a different permutation of the same items used in σ. If the partial sequence σ' has been memorized, then one of two situations occurs. If σ' dominates σ then node B can be pruned since it cannot lead to a better solution than A. If σ' does not dominate σ then σ can be memorized to possibly prune a future node such as node C. Notice that, a current node to be branched on, is compared with both explored nodes and active nodes (those, waiting to be branched on).

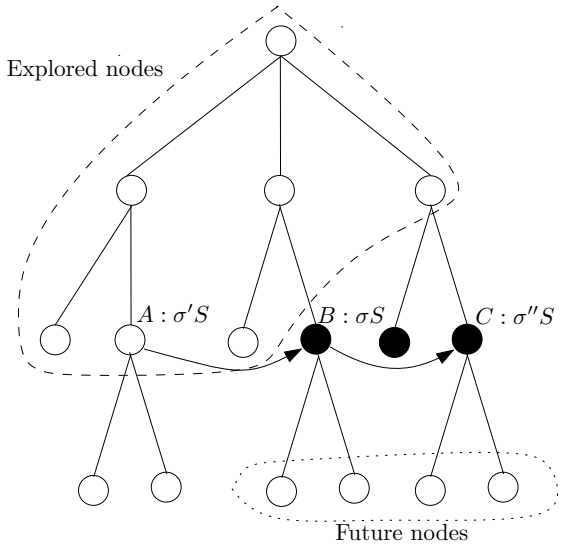

Figure 2: Passive node memorization

The notion of dominance between sequences is problem dependent and strongly influences the effectiveness of the memorization mechanism in pruning nodes. Therefore, for a given sequencing problem, to which passive memorization is applied, we assume the existence of a function $check(\sigma, \sigma')$ which returns `1` if σ' dominates σ. Obviously, this function corresponds to a mathematical condition that has been proved: it can be a sufficient condition or even a necessary and sufficient condition that states the dominance of a partial sequence σ' over another

partial sequence σ. Examples are given in section 4. With respect to the literature, the works by T'kindt et al. (2004), Sewell and Jacobson (2012) and Morrison et al. (2014) describe preliminary implementations of *passive node memorization* as indicated in Table 1. Same consideration holds for the works by Li et al. (2018), Li et al. (2020a) and Li et al. (2020b). Notice that, with respect to dynamic programming, when $check(\sigma, \sigma') = 1$, the state corresponding to σ is not discarded, unlike passive node memorization. The larger the number of such states, the larger the computational time saving with respect to the overlapping sub-problems elimination of dynamic programming.

2.2.3. Predictive node memorization

Predictive node memorization relies on the same concept present in *passive node memorization*, but with further 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 pruned by a sequence σ' previously memorized, *e.g.* 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 as in the so-called *no-good recording* technique of Baptiste et al. (2004). 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 was used by Jouglet et al. (2004) under the name *Dominance Rules Relying on Scheduled Jobs*. However, notice that they do not memorize the dominant solution found. Sourd and Kedad-Sidhoum (2008) have gone beyond the work of Jouglet et al. by storing the generated solutions. All these works describe preliminary implementations of *predictive node memorization* as indicated in Table 1. Notice that, we may also make use of another exact algorithm to optimize a part of σ to get π, as long as this algorithm is fast. This idea is strongly related to the theoretical mechanism called *merging* (Garraffa et al., 2018) which is designed to provide good worst-case time complexities for exact exponential algorithms. If such a sequence π can be constructed, then the current node σS is cut and sequence π is memorized. Note that node πS has not yet been encountered in the search tree when dealing with node σS (for example, consider $\pi = \sigma''$ inf Figure 3). Thus, it is important when applying *predictive node memorization* to remember that πS still needs to be branched on. Moreover, the extra cost of generating π should be limited to avoid excessive CPU time consumption.

3. Decision guidelines

In this section, we provide some guidelines on how to choose the appropriate memorization scheme according to the branching rule and the search strategy. The main results are summarized in the decision tree of Figure 4. Consider the following Definition 2 and Property 1 that are valid for forward, backward and decomposition branching schemes.

Definition 2. *(Concordance Property) Let LB(A) be the lower bound value computed at node A. Let be two nodes A and B defined by* $A = \sigma_1 S_1 \sigma_2 S_2 ... \sigma_k S_k$ *and* $B = \pi_1 S'_1 \pi_2 S'_2 ... \pi_{k'} S'_{k'}$*, with* σ_1 *and* π_1 *involving the same set of items in case of forward or decomposition branching schemes. In case of backarwd branching scheme,* σ_k *and* $\pi_{k'}$ *must involve the same set of items.*
A search tree based algorithm satisfies the concordance property if and only if, for any node A and B as defined above, $LB(A) < LB(B) \Leftrightarrow check(\pi_1, \sigma_1) = 1$ *(or* $check(\pi_k, \sigma_{k'}) = 1$ *in case of backward branching), i.e. node A dominates node B if and only if* $LB(A) < LB(B)$*.*

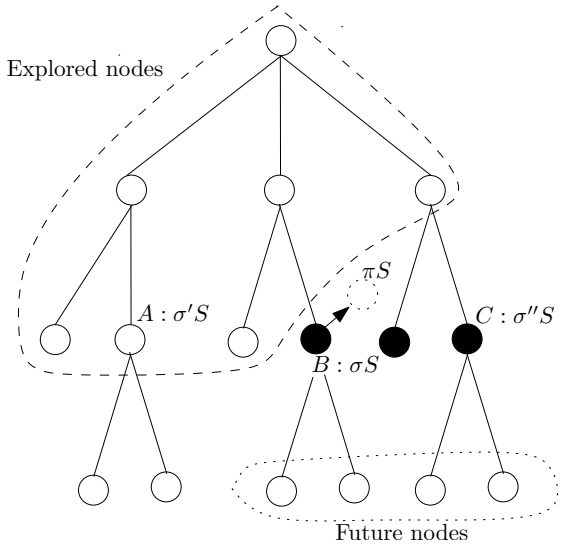

Figure 3: Predictive node memorization

Figure 4: Decision tree for choosing the memorization scheme

Property 1. *For the memorization schemes, we can state the following general results that are independent from the branching scheme:*

1. *Solution memorization can only be applied together with depth-first search strategy since an optimal solution in the sub-tree rooted at any node must be found first to be memorized.*
2. *For passive and predictive node memorizations, when the concordance property is verified and best-first search strategy is applied, the current node only needs to be compared with the explored nodes, instead of all nodes.*
3. *For passive node memorization, when breadth-first search strategy is applied, the current node only needs to be compared with the active nodes, instead of all nodes. In that case, predictive node memorization does not prune more nodes than passive node memorization.*

Proof. Result 1 is straightforward and follows from the definition of *solution memorization*.
Result 2 is a consequence of the *best-first* search strategy that always considers branching on the node with the lowest lower bound value : the concordance property implies that no active node can dominate it.
Result 3 follows from the fact that 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*, according to a *check* function, are discarded. If, in turn, *A* is dominated by another node, then it is pruned. Moreover, *predictive node memorization* cannot outperform *passive node memorization* since *passive node memorization* keeps the best node at each level. □

In the following, we discuss the choice of a memorization scheme depending on the branching rule and the search strategy. We use *node memorization* to refer to both *passive node memorization* and *predictive node memorization*.

3.1. Forward/backward branching

Without loss of generality, only *forward branching* is considered. According to the choice of the search strategy, and based on Property 1, Guideline 1 provides some advices on the choice of a memorization scheme.

Guideline 1. *Depending on the search strategy, we have:*

1. *With depth-first search strategy, if the problem is decomposable and solution memorization can memorize optimal solutions (instead of lower bounds), then solution memorization should be chosen as it is more effective both than passive node memorization and predictive node memorization.*
2. *With best-first search strategy, node memorization should be applied only to explored nodes if the concordance property (Definition 2) is satisfied.*
3. *With breadth-first search strategy, passive node memorization should be chosen and applied to active nodes.*

Remark 1. *In case 1, any node pruning that can be achieved by passive node memorization and predictive node memorization can also be obtained by solution memorization, but not conversely. We say that solution memorization is more effective than the two other memorization schemes. As an example, consider nodes $A = \sigma S$ and $B = \pi S$, where σ and π are two permutations of the same items set and A is visited before B. None of these nodes can be pruned by node memorization, if B dominates A. However, solution memorization will avoid exploring node B. If the problem is not decomposable, or context-dependent dominance conditions exist in the algorithm, then solution memorization memorizes lower bounds and it is not obvious to determine which memorization scheme is the best. 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 items set S at both nodes A and B. However, with passive node memorization, node B may be pruned whenever π is dominated by σ. Cases 2 and 3 are direct consequences of Property 1.*

3.2. Decomposition branching

With *decomposition branching*, at each level of the search tree a decomposition item can be put in any free position by the branching rule. Guideline 2 provides some advices on the choice of a memorization scheme.

Guideline 2. *With depth-first search strategy, none of the three memorization schemes is necessarily most effective than the others.*

We can imagine situations in which *solution memorization* or *passive node memorization* or *predictive node memorization* is the most effective. 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 sequencing the items subset S_1 after σ or π. Suppose σ and π contain different items but induce the same initial conditions for the following items (as an example in single machine sequencing, σ and π have the same completion time). This means that the sub-problems defined by S_1 are identical in A and B. Then, an 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 items and, hence, are not comparable. *Predictive node memorization* may or may not prune 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 items set. If $check(\pi, \sigma) = 1$, then node B can be pruned 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 the best choice 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 items 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 contain different items, may occur often for large size instances.

To the knowledge of the authors, *decomposition branching* being already not common for sequencing problems, its combination with best-first or breadth-first search strategy is even more rare. Therefore, we omit the discussion on these two cases. Interested readers may refer to Shang et al. (2018).

3.3. Implementation consideration

In order to effectively implement the *memorization* mechanism, it is required to take care about the **memory cleaning strategy** when the memory is full. Due to computer memory limitations, not all nodes or solutions can be stored during the processing of the search tree algorithm implementing memorization. Then, deciding which entry (node/solution) to keep in memory influences the effectiveness of the memorization in pruning nodes. We have experimentally tested several memory cleaning strategies like *First In First Out* (FIFO), *Biggest Entry First Out* (BEFO), etc. Finally, the most efficient one that we found is *Least Used First Out* (LUFO). As the name indicates, LUFO counts for each stored entry in the memory, the number of times that it has been queried. Each time the memory is full, all entries with minimum counter are discarded. This strategy is effective in practice since most of the memorized entries are never used, while others are typically used many times.

4. Application to the $1|r_j|\sum C_j$ and $1|\tilde{d}_j|\sum w_j C_j$ problems

In this section we focus on the application of the *Branch & Memorize paradigm* to branch-and-bound algorithms solving two single machine scheduling problems involving the minimization of the total (weighted for the second problem) completion time. All these algorithms implement a forward or backward branching rule and the three classic search strategies previously discussed are considered.

In this section, for each of these two problems, we apply memorization according to the provided guidelines and discuss the obtained results. For each problem, we compare several branching algorithms, which are named according to their features: Depth-, Best- and Breadth- refer to branch-and-bound algorithms with the corresponding search strategies and without memorization. Depth_X, Best_X and Breadth_X refer to the implementations with the corresponding search strategies and memorization of type X, where $X = S$ means *solution memorization*, $X = Pa$ means *passive node memorization* and $X = Pr$ means *predictive node memorization*. For *predictive node memorization*, we use the "dominance condition relying on scheduled jobs" (see Jouglet et al. (2004)) as the heuristic to search for dominant solutions. We name it $k - perm$ search, since at a given node σS, it enumerates all the permutations of the first or last k jobs in σ

to search for a dominant sequence. Besides, *k-perm* search is not performed when the breadth-first search strategy is used, since the memorization applied on active nodes already covers the effect of *k-perm*. Preliminary tests suggest that $k = 5$ should be chosen in our implementations to obtain the most efficient *predictive node memorization scheme*. Notice that *predictive node memorization* based algorithms could be further improved by adopting another local search strategy. But this is not dealt with in this paper.

All tests have been performed on an HP Z400 work station with 3.07GHz CPU and 8GB RAM. In the 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 by the branch-and-bound algorithms. The test results on instances of a given size are marked as *OOT* (out of time) if any instance of that size 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). Notice that 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.

4.1. Application to the $1|r_j|\sum C_j$ *problem*

The $1|r_j|\sum C_j$ problem requires n jobs to be scheduled on a single machine. Each job j is defined by a processing time p_j and a release date r_j before which the job cannot be processed. The machine can only process one job at a time and a schedule is a sequence of jobs in their order of processing. In a given schedule, each job j completes at time C_j and the aim is to find the sequence that minimizes $\sum C_j$. This problem is NP-hard in the strong sense and it has been widely studied in the literature with both exact and heuristic algorithms. The state-of-the-art exact algorithm was proposed by Tanaka and Fujikuma (2012). This algorithm, called *Sipsi*, uses a graph representation during the solution and the size of the graph depends on the value of the maximum processing time, denoted by p_{max}. Hence, when p_{max} is large, the algorithm is restricted by its memory usage. This algorithm proceeds by applying dynamic programming over that graph representation.

The memorization techniques introduced in this paper are applied to the baseline branch-and-bound introduced by Chu (1992) which was the most effective exact algorithm for a long time. This algorithm uses the *forward branching* rule, the *best-first search strategy* as well as a *k-perm* search. The latter is disconnected in the experiments done since it relates to *predictive node memorization*. The conducted experiments compare the impact of the different memorization techniques and show comparisons with *Sipsi* algorithm.

In *node memorization* techniques, the *check*() function used to compare nodes is that of T'kindt et al. (2004):

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

where S is the set of jobs that remain to be scheduled after sequences σ and π, and $E_{min}(\sigma) = \max(C(\sigma), \min_{r\in S} r_i)$, with $C(\sigma)$ denoting the completion time of σ. The item stored into memory is a tuple $\langle \sigma, C(\sigma), opt(\sigma|0)\rangle$ and $E_{min}(\sigma)$ can be computed when needed.

4.1.1. Experimental results

The problem is not *decomposable* due to the presence of release dates. Therefore, with the choice of *forward branching*, *node memorization* should be preferrably 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 1, 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.

The input was generated following the approach described by Chu (1992), *i.e.*, the processing times are generated uniformly from $[1, 100]$ and the release dates are generated uniformly from $[0, 50.5 \cdot n \cdot r]$, with r belonging to $\{0.2, 0.4, 0.6, 0.8, 1.0, 1.25, 1.50, 1.75, 2.0, 3.0\}$. Thirty instances were generated for each value of r, thereby leading to 300 instances for each size n from 70 to 350. The results are presented in Table 2. 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. Notice that when applying *predictive node memorization* we obtain better results than the baseline algorithm of Chu (1992) which integrates no memorization but only a *k-perm* search.

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*. As expected, *node memorization* provide better results than *solution 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 with FIFO as cleaning strategy. However, due to the hardness of another instance with 140 jobs, the algorithm Depth_Pa ran out of time.

The *Sipsi* algorithm is also tested on the same dataset and results are provided in Table 3. It is very efficient and is able to solve instances with up to 300 jobs. To evaluate the influence of the processing times values, we have considered instances where the processing times are generated uniformly from $[1, 1000]$. *Sipsi* and Depth_Pr algorithms are compared and the obtained results are reported in Table 4. It follows that both algorithms are only capable of solving instances with up to 130 jobs in size, Depth_Pr being faster. This shows that unlike *Sipsi*, Depth_Pr is not sensitive to the range of processing time of jobs.

The proposed approach can be extended in a similar way to the $F2||\sum C_j$ problem, where it can be applied, for instance, to the algorithm of Della Croce et al. (2002). The power of memorization still holds. However, the improvement in the size of solved instances is less significant and the resulting algorithm is less efficient than the current state of the art algorithm of Detienne et al. (2016). For this reason, we do not present results here, but we refer to Shang et al. (2018) for details.

Table 2: Results of the branch-and-bound algorithms on the $1|r_j|\sum C_j$ problem

	n	70	80	90	100	110	120	130	140
Without Memorization									
Depth-	Navg	141247.8	1778751.2	OOT					
	Nmax	17491232	276190737						
	Tavg	1.8	22.4						
	Tmax	217	3238						
Best-		OOT							
Breadth-		OOT							
Solution Memorization									
Depth_S	Navg	113438.69	1330100.19	OOT					
	Nmax	14321924	216164660						
	Tavg	2.19	24.81						
	Tmax	323.00	3748.50						
Node Memorization									
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_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_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 3: Results of the algorithm *Sipsi* on the $1|r_j|\sum C_j$ problem

	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 4: Results of algorithms on new instances with large processing times ($P_{max} = 1000$)

	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	

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

The $1|\tilde{d}_j|\sum w_jC_j$ problem requires n jobs to be scheduled on a single machine. Each job j is defined by a processing time p_j, a weight w_j and a deadline $\tilde{d}_j$ that must be met. The machine

can process one job at a time and, again, a schedule is defined by a sequence of jobs. The objective is to minimize the total weighted completion time $\sum w_j C_j$. This problem is NP-hard in the strong sense and has been solved by branch-and-bound algorithms (Posner, 1985; Potts and Van Wassenhove, 1983), with the performance of the algorithm of Posner being slightly superior. We adopt a combination of both branch-and-bound algorithms, by incorporating the lower bound and the dominance condition of Posner (1985) into the algorithm of Potts and Van Wassenhove (1983). The algorithm of Tanaka et al. (2009), called *Sips*, is known as the most efficient one for solving this problem and it follows the same approach of the *Sipsi* algorithm.

We adopt *backward branching* as branching scheme as done in Posner (1985); Potts and Van Wassenhove (1983). The *check*() function is defined as follows, where S is the set of jobs to be scheduled before σ and π.

$$check(\pi, \sigma) = \begin{cases} 1, \ if \ opt(\sigma | \sum_{i \in S} p_i) \leq opt(\pi | \sum_{i \in S} p_i) \\ 0, \ otherwise \end{cases} \tag{2}$$

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

4.2.1. Experimental results

This problem is *decomposable*. 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-and-bound algorithms: *node memorization* with the three search strategies and *solution memorization* based on *depth-first* search.

The concordance property is satisfied and so 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 as described in the previous section.

Regarding *solution memorization*, context-dependent dominance conditions are enabled in the algorithm (Theorem 2 in Potts and Van Wassenhove (1983)). Their removal has been experimentally proved to lead to an inefficient algorithm. Therefore, lower bounds are memorized during *solution memorization*, as described in section 2.2.1.

Following the test plan described by Potts and Van Wassenhove (1983), for each job j, the processing time p_j is an integer that is generated uniformly from $[1, 100]$ and its weight w_j is generated uniformly from $[1, 10]$. The total processing time $P = \sum_{j=1}^{n} p_j$ is then computed and for each job j, an integer deadline d_j 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 results are presented in Table 5. For *depth-first* search, without *memorization*, the most effective branch-and-bound algorithm is "out of time" on instances with 50 jobs, while *solution memorization* and *passive node memorization* enable to solve instances with up to 90 and 100 jobs, respectively. With the activation of *k-perm* search, Depth_Pr can solve 20 more jobs than Depth_Pa. 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. Consequently, embedding memorization inside the baseline branch-and-bound algorithm enables to solve instances more than three times larger.

Table 5: Results of the branch-and-bound algorithms on the $1|\tilde{d}_j|\sum w_jC_j$ problem

		40	50	60	70	80	90	100	110	120	130	140	150
Without Memorization													
Depth-	Navg	104915.3	OOT										
	Nmax	14536979											
	Tavg	0.9											
	Tmax	74.0											
Best-		OOT											
Breadth-		OOM											
Solution Memorization													
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						
Node Memorization													
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_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_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		

The results of *Sips* are presented in Table 6. It appears that it can efficiently solve instances with up to 140 jobs, which slightly improves upon the branch-and-bound algorithms with memorization. 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 uniformly from $[1, 1000]$ and job weights from $[1, 100]$. This leads to the results in Table 7. Both algorithms Breadth_Pa and *Sips* solve smaller instances than before, with Breadth_Pa being limited to 110 jobs and *Sips* being limited to 100 jobs.

Table 6: Results of the algorithm *Sips* on the $1|\tilde{d}_j|\sum w_jC_j$ problem

		40	50	60	70	80	90	100	110	120	130	140	150
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 7: Results of algorithms on new instances with larger processing times and weights ($p_{max} = 1000, w_{max} = 100$)

	n	90	100	110	120
Depth_Pr	Tavg	12.00	OOT		
	Tmax	603.35			
Best_Pr	Tavg	14.59	OOT		
	Tmax	1513.02			
Breadth_Pa	Tavg	8.869	115.62	121.93	OOM
	Tmax	464.71	16562.41	5244.38	
Sips	Tavg	19.14	33.00	OOM	
	Tmax	225.11	292.58		

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

5.1. Preliminaries

In this section we focus on a scheduling problem and a branching algorithm that involves a decomposition branching scheme. Even if this kind of branching scheme is not frequent in scheduling, there still exists works considering decomposition branching. Examples are for the $1|r_j, q_j|L_{max}$ problem (Carlier, 1982) and the $1|d_j = d < \sum_j p_j|\sum_j \alpha_j E_j + \beta_j T_j$ problem (Hoogeveen and van de Velde, 1991).
Let us focus on the $1||\sum T_j$ problem that requires to schedule a set of n jobs $N = \{1, 2, \ldots, n\}$ on a single machine. Each job j is defined by a processing time p_j and a due date d_j. The machine can only process one job at a time and a schedule is a sequence of the jobs. The aim is to find a sequence that minimizes $\sum T_j$ with $T_j = \max\{C_j - d_j, 0\}$ the tardiness of job j in any given schedule. This problem is known to be NP-hard in the ordinary sense (Du and Leung, 1990) and it has been extensively studied in the literature.

The current state-of-the-art exact method is a search-tree algorithm, denoted by SGDC2001, that 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 by Koulamas (2010). The main properties of the problem can be found in Szwarc et al. (2001), and some of them are given below. Without loss of generality, let $(1, 2, \ldots, n)$ be the sequence of jobs sorted by the LPT rule (Longest Processing Time first) and let $([1], [2], \ldots, [n])$ be the sequence given by the EDD rule (Earliest Due Date first). We first introduce two important decomposition properties.

Decomposition 1. *Lawler (1977) (Lawler's decomposition) Assume job* 1 *in the LPT sequence corresponds 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) Assume job k in the LPT sequence corresponds 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 decomposition branching scheme called *Double Decomposition* (Szwarc et al., 2001). From that branching scheme it follows that nodes of the search tree are of the form $\sigma_1 S_1 \sigma_2 S_2 ... \sigma_i S_i ... \sigma_k S_k$. The

Double Decomposition is always applied on S_1 and it 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. SGDC2001 algorithm employs the *depth-first* search strategy.

When branching from a node, another particular decomposition may occur as described in Property 2. 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 2 states when does such decomposition occur. In that case Decomposition 1 and Decomposition 2 are not applied, but rather two children nodes are created, each corresponding to one block of jobs (A or B), following Property 2 (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 2. *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_j and L_j of each job j 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 SGDC2001, even though it was called *Intelligent Backtracking* by the authors. Remarkably, lower bounds are not used in this algorithm due to the "Algorithmic Paradox" (Paradox 1) found by Szwarc et al. (2001). This one shows that the effectiveness of memorization largely surpasses the contribution of the lower bounding procedure 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 prune 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 pruned cannot be memorized.

The SGDC2001 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 2 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 algorithm a simple branching algorithm. Notice that *solution memorization* can be implemented in SGDC2001 as suggested in section 3.3. In SGDC2001, when the database of stored solutions was full, no cleaning strategy was used and no more partial solutions could be stored.

5.2. Experimental results

We take the reference algorithm SGDC2001 as the baseline search-tree algorithm in which the memorization techniques introduced in this paper are embedded. 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 relevance of considering *node memorization* instead of *solution memorization*. As already mentioned in section 3.2, it is not obvious to implement *node memorization*, for a decomposition 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 set 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 lead to better results than those obtained with *solution memorization*. Consequently, only *solution memorization* is considered and, as sketched in sections 3.2, 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 are retained. 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 SGDC2001, with two main changes. Since the memory usage was declared as a bottleneck of SGDC2001, we first retested SGDC2001 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 generated uniformly from $[1, 100]$ and due dates d_i are generated uniformly from $[p_i u, p_i v]$ 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 (R, T). The time limit for the solution of each instance is set to 7.5 hours. It is larger than the one used for the previous two problems in order to provide full information on the impact of memorization within the tested algorithms. An algorithm is considered as OOT (Out of Time) if it reaches this time limit. Additionally, when memorization is applied without a database cleaning strategy, the memory may be saturated leading the algorithm to be declared OOM (Out of Memory).

The results are reported in Table 8. Our implementation of SGDC2001 solves instances with up to 1000 jobs in size, knowing that the original program, as tested in 2001 was limited to instances with up to 500 jobs due to the memory size limit. Their tests were done on a Sun Ultra-Enterprise Station with a reduced CPU frequency (<450MHz). It is anyway interesting to see that with just the computer hardware evolution, memorization enables to solve instances with 500 jobs more. SGDC2001 is out of time for instances with 1100 jobs, and the memory size no longer seems to be the bottleneck. The first improvement we propose continues on the vein of Paradox 1.

Paradox 2. *Removing the Split procedure (Property 2) from SGDC2001 drastically accelerates the solution.*

The effect of Paradox 2 is pretty strong. The resulting algorithm *NoSplit* solves instances with an average solution time about twice faster than SGDC2001, and it also manages to solve instances with 1100 jobs. 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 SGDC2001 is run at 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 1200 jobs. Removing *Split* was not considered by Szwarc et al. (2001) because *Split* is a very strong component of the algorithm and the memory available at that time also discouraged this tentative.

The second improvement we provide to SGDC2001 relates to the database cleaning strategy which has been changed to LUFO strategy. The corresponding algorithm is referred to as NoSplit_LUFO in Table 8. All 200 instances with 1200 jobs are solved, with an average solution time of about half an hour, while SGDC2001 is limited to instances with 1000 jobs.

Table 8: Results of branch-and-bound algorithms for the $1||\sum T_j$ problem

	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											
SGDC2001	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	

The presented experiments show that correctly tuning the memorization mechanism may lead to considerable improvement to its efficiency. However, the striking point of these experiments relates to the comparison between the version of SGDC2001 without memorization (algorithm Depth-) and NoSplit_LUFO. Table 8 highlights the major contribution of a *Branch & Memorize* approach: Depth- being limited to instances with up to 200 jobs while NoSplit_LUFO is capable of solving all instances with 1200 jobs. To the best of authors' knowledge, NoSplit_LUFO is the currently most efficient exact algorithm for the $1||\sum T_j$ problem.

6. Conclusions

In this paper, we focused on the application of memorization techniques within search tree algorithms for the efficient solution of sequencing problems. Several memorization schemes are defined and some advices are provided for choosing the best memorization approach according to the branching scheme and the search strategy applied. The proposed *Branch & Memorize* approach has been tested on three single machine sequencing problems. Even though its performance depends on the problem, in all cases, it outperforms the related exact algorithms without

memorization. We provide below a summary of the conclusions obtained, under a CPU limit of 5h for problems $1|r_j|\sum C_j$and $1|\tilde{d_j}|\sum w_jC_j$ and of 7.5h for problem $1||\sum T_j$:

- Problem $1|r_j|\sum C_j$: best configuration is *depth-first* with *predictive node memorization*. Largest instances solved: 130 jobs against 80 jobs without memorization.
- Problem $1|\tilde{d_j}|\sum w_jC_j$: best configuration is *best-first*with *predictive node memorization*. Largest instances solved: 130 jobs against 40 jobs without memorization.
- Problem $1||\sum T_j$: best configuration is *depth-first*with *solution memorization*. Largest instances solved: 1200 jobs against 200 jobs without memorization.

Fundamentally, what we call the *Branch & Memorize paradigm* relies on a simple but potentially very efficient idea: memorizing what has already been done to avoid solving identical or dominated sub-problems in the rest of the solution process. The contribution of this paradigm strongly relies on the branching scheme which can induce more or less redundancy in the exploration of the solution space. It is noteworthy that the three sequencing 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. To our opinion, memorization techniques 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 since, under the hypothesis of infinite memory, memorization can be mathematically used to reduce the worst-case time complexity with respect to search tree 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 other combinatorial optimization problems. It may also be promising to see how the machine learning field could help in efficiently managing the database of stored partial solutions. 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 for providing us the code of his algorithms. This work has been partially supported by "Ministero dell'Istruzione, dell'Università e della Ricerca" Award "TESUN-83486178370409 finanziamento dipartimenti di eccellenza CAP. 1694 TIT. 232 ART. 6".

References

Baptiste, P., Jouglet, A., Carlier, J., 2004. A branch-and-bound procedure to minimize total tardiness on one machine with arbitrary release dates. European Journal of Operational Research 158, 595–608.

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.

Carlier, J., 1982. The one-machine sequencing problem. European Journal of Operational Research 11 (1), 42–47.

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.

Chvatal, V., 1997. Resolution search. Discrete Applied Mathematics 73, 81–99.

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., Kratsch, D., 2010. Exact exponential algorithms. Springer Science & Business Media.
Garraffa, M., Shang, L., Della Croce, F., T'Kindt, V., 2018. An exact exponential branch-and-merge algorithm for the single machine total tardiness problem. Theoretical Computer Science 745, 133–149.
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.
Gurevich, Y., Shelah, S., 1987. Expected computation time for hamiltonian path problem. SIAM Journal on Computing 16 (3), 486–502.
Hanafi, S., Glover, F., 2002. Resolution search and dynamic branch-and-bound. Journal of Combinatorial Optimization 6, 401–423.
Hoogeveen, H., van de Velde, S., 1991. Scheduling around a small common due date. European Journal of Operational Research 55, 237–242.
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.
Kao, G. K., Sewell, E. C., Jacobson, S. H., 2009. A branch, bound, and remember algorithm for the $1|r_i|\sum t_i$ scheduling problem. Journal of Scheduling 12, 163–175.
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.
Li, Z., Cil, Z., Mete, S., Kucukkoc, I., 2020a. A fast branch, bound and remember algorithm for disassembly line balancing problem. International Journal of Production Research 58 (11), 3220–3234.
Li, Z., Kucukkoc, I., Tang, Q., 2020b. A comparative study of exact methods for the simple assembly line balancing problem. Soft Computing 24, 11459–11475.
Li, Z., Kucukkoc, I., Zhang, Z., 2018. Branch, bound and remember algorithm for u-shaped assembly line balancing problem. Computers & Industrial Engineering 124, 24–35.
Morrison, D., Jacobson, S., Sauppe, J., Sewell, E., 2016. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. Discrete Optimization 19, 79–102.
Morrison, D. R., Sewell, E. C., Jacobson, S. H., 2014. An application of the branch, bound, and remember algorithm to a new simple assembly line balancing dataset. European Journal of Operational Research 236, 403–409.
Posner, M. E., 1985. Minimizing weighted completion times with deadlines. Operations Research 33 (3), 562–574.
Posta, M., Ferland, J., Michelon, P., 2011. Generalized resolution search. Discrete Optimization 8, 215–228.
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., T'Kindt, V., Della Croce, F., Jun. 2018. The Memorization Paradigm: Branch & Memorize Algorithms for the Efficient Solution of Sequencing Problems. Research report, University of Tours.
URL `https://hal.archives-ouvertes.fr/hal-01599835`
Sourd, F., Kedad-Sidhoum, S., 2008. A faster branch-and-bound algorithm for the earliness-tardiness scheduling problem. Journal of Scheduling 11, 49–58.
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.