THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : **MATHÉMATIQUES & INFORMATIQUE**

Arrêté ministériel : 25 mai 2016

Présentée par

Luc LIBRALESSO

Thèse dirigée par
Louis ESPERET, Chargé de Recherche CNRS, G-SCOP
et co-encadrée par
Vincent JOST, Chargé de Recherche CNRS, G-SCOP
Thibault Honegger, Chief Scientific Officer, NETRI

préparée au sein du
Laboratoire des Sciences pour la Conception, l'Optimisation et la Production de Grenoble (G-SCOP)

dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique (ED-MSTII)**

Recherches arborescentes anytime pour l'optimisation combinatoire

Anytime tree search algorithms for combinatorial optimization

Thèse soutenue publiquement le **24 Juillet 2020**
devant le jury composé de :

Remerciments

Cette thèse est le résultat d'un grand nombre de rencontres, de collaborations au long de ces 3 dernières années. J'aimerais remercier en particulier:

- Christian Artigues, Jin-Kao Hao, Christine Solnon et Vincent T'Kindt. Merci d'avoir pris le temps de lire mes travaux et pour vos nombreux conseils et remarques.
- Abdel-Malik Bouhassoun, Aurélien Secardin, Pablo Andres Focke pour avoir accepté de faire vos stages avec moi. Ce fut un plaisir de travailler avec vous. Merci.
- Florian Fontan d'avoir accepté de participer avec moi au challenge ROADEF et pour toutes les discussions très enrichissantes. J'ai vraiment appris beaucoup grâce a toi. Merci.
- Thibault Honegger et Florian Larramendy. Votre force de travail et enthousiasme est une immense source d'inspiration pour moi. Je vous souhaite le meilleur pour la suite, bien que j'ai aucun doute que vous allez y arriver.
- Louis Esperet, Thibault Honegger, et Vincent Jost pour avoir accepté de diriger ma thèse. Grace a vous, j'ai passé 3 excellentes années, Merci d'avoir cru en moi. Merci pour tout.
- Van-Dat Cung et Hadrien Cambazard pour tous vos conseils et discussions, merci.
- Nadia Brauner pour m'avoir fait découvrir la recherche opérationnelle, et pour tous tes conseils, merci.
- L'ensemble des personnes que j'ai pu rencontrer à G-SCOP et à Netri. Merci pour tous ces bons moments.
- L'équipe administrative de G-SCOP, en particulier Marie-Jo, Fadila et Myriam pour m'avoir permis de faire ma thèse dans des conditions optimales.
- Une petite dédicasse à Aldric, Karine et Marlène, ainsi que tous les membres de *Eat & Move*. Merci pour toute la bonne humeur et toutes ces super séances.
- Ma partenaire de vie et ma meilleure amie, Adèle.
- Enfin, mes parents pour avoir pris grand soin de mon éducation et pour m'avoir transmis le goût des sciences.

Abstract

Tree search algorithms are used in a large variety of applications (MIP, CP, SAT, metaheuristics with Ant Colony Optimization and GRASP) and also in AI/planning communities. All of these techniques present similar components and many of those components can be transferred from one community to another. Preliminary results indicate that anytime tree search (search techniques from AI/planning) can be part of the operations research toolbox as they are simple and competitive compared to commonly used metaheuristics in operations research.

In this work, we detail a state of the art and a classification of the different tree search techniques that one can find in metaheuristics, exact methods, and AI/planning. Then, we present a generic framework that allows the rapid prototyping of tree search algorithms. Finally, we use this framework to develop anytime tree search algorithms that are competitive with the commonly-used metaheuristics in operations research. We report new tree search applications for some combinatorial optimization problems and new best-known solutions.

Main messages:

- anytime tree search algorithms can be competitive with classical meta-heuristics on large scale discrete optimization instances by using anytime tree search strategies.
- The combination of algorithmic components from different communities (AI, CP, meta-heuristics, branch-and-bounds) combined with a study on the contribution of each component leads to new algorithms that are competitive, and sometimes, even simpler than the state-of-the-art methods from each of these communities.

Résumé

Les recherches arborescentes sont utilisées dans un grand nombre d'applications (MIP, CP, SAT, metaheuristiques avec Ant Colony Optimization et GRASP) et également dans des communautés IA/planning. Toutes ces techniques présentent des bases communes et de nombreuses techniques peuvent être transférées d'une communauté à une autre. Les résultats préliminaires indiquent que ces techniques ont toute leur place dans la boite a outils des méthodes les plus performantes en recherche opérationnelle.

Dans ces travaux, nous dressons un état de l'art et une classification de différentes techniques de recherche arborescente que l'on retrouve dans les metaheuristiques, dans les méthodes exactes et en IA/planning. Nous développons un framework générique qui permet l'élaboration rapide d'algorithmes de recherche arborescente. Enfin, nous utilisons ces techniques pour proposer des méthodes compétitives avec les metaheuristiques généralement utilisées en recherche opérationnelle. Nous présentons de nouvelles méthodes de recherche arborescente pour plusieurs problèmes d'optimisation combinatoire ainsi que de nouvelles meilleures solutions connues.

Messages principaux :

- Les recherches arborescentes anytime peuvent être compétitifs comparé aux metaheuristiques classiques sur des instances de grande taille grâce aux recherches arborescentes anytime.
- La combinaison de composants algorithmiques habituellement utilisés par différentes communautés (IA, CP, meta-heuristiques, branch-and-bound) ainsi que l'étude des contributions de ces composants permet de produire des algorithmes nouveaux, compétitifs, et parfois plus simples que les méthodes état de l'art issues de chacune de ces communautés.

Introduction

While facing a hard combinatorial optimization problem, two types of solutions are usually considered:

- *Exact methods* allow to find optimal solutions at the price of potentially very long computation time. In this category, we find Mixed Integer Programming methods, Constraint Programming *etc.* This kind of methods generally use tree search techniques that usually rely on strong bounds and cuts like the cutting-plane algorithm or the branch-and-price algorithm *etc.*

- *Meta-heuristics* that allow to find near-optimal solutions. These methods usually rely on fast operators (neighborhoods, crossovers, mutations, *etc.*) and various search strategies (Simulated Annealing, Tabu Search, Evolutionary Algorithms, Ant Colony Optimization).

Tree search is mainly found in exact methods. In this specific context, *Depth First Search* or *Best First Search* seem to be the most suited methods. The first one for its simplicity, limited memory usage, and backtrack-friendliness. The second for its ability to improve bounds quickly and prove optimality in a (relatively) small amount of nodes[1]. However, on large instances, these methods do not usually obtain good quality solutions. Indeed, *Depth First Search* is prone to choose a bad branch early in the tree search and is not able to escape from it. *Best First Search* is prone to find solutions late in the search (thus not finding any solutions within hours of computation). We also note that work has been done to allow these methods to find good solutions fast (using a restarting strategy for *Depth First Search* and diving for *Best First Search*). But these improvements are usually insufficient to compete with classical meta-heuristics.

This is why it is usually admitted that tree search algorithms are not suited to solve large-scale or industrial instances.

[1]We note that this hypothesis is sometimes questioned [STDC18, TDCE04]

From Mathematical Programming Solver based on Local Search ([GBD$^+$14]):

> “ Tree search approaches like branch-and-bound are in essence designed to prove optimality [...] Moreover, tree search has an exponential behavior which makes it not scalable faced with real-world combinatorial problems inducing millions of binary decisions. ”

It may be interesting to re-evaluate the inefficiency of tree search algorithms to find near-optimal large-scale/industrial combinatorial optimization problems. Indeed, there are other tree search techniques originally proposed since the sixties in AI or planning conferences. To cite a few, we find *Beam Search* [OM88], well-known for its success to solve scheduling or packing problems, *Limited Discrepancy Search* [HG95], used intensively in Constraint Programming solvers, *Anytime Column Search* [VGAC12] *etc.* In this context, tree search can be used to find solutions quickly and continuously improve them similarly to the behavior of classical meta-heuristics (until they reach a stopping criterion or prove optimality by depleting the search tree). We qualify search methods with this property as *Anytime algorithms* (in our case *Anytime Tree Search algorithms*).

In this thesis, we show that, on several problems, it is possible to obtain competitive (and state of the art) methods only based on tree search algorithms (thus not using any local search or population-based strategies). We describe a general methodology to build efficient tree search techniques that are competitive with classical meta-heuristics. Our first main contribution presents an anytime tree search for the EURO/ROADEF challenge 2018 where our method was ranked first among 64 registered participants. Our second main contribution is also a simple anytime tree search algorithm for the *Sequential Ordering Problem* [Esc88] (Asymmetrical Traveling Salesman Problem with precedence constraints). This problem has been studied intensively during the last 30 years and a large variety of methods have been developed to solve it. It is especially well-known because it has instances with less than 50 cities that are still open. We propose a simple anytime tree search algorithm (approximately 200 lines of C++ code) that results from a study over many Branch-and-Bound algorithmic components. This method was able to improve best-known solutions on 6 over 7 open instances from the SOPLIB. During the design of these two methods, it appeared that the best combinations of ideas were often counter-intuitive, thus, it seems that a rigorous study of the influence of each tree search idea is required. To this extent, we developed a generic anytime tree search framework that allows us to rapidly prototype tree search techniques and integrate a large variety of components (we named it Combinator-based Anytime Tree Search framework). We used it to develop and benchmark our ideas and algorithms. This framework is our third (and last) main contribution in this thesis.

Outline of the thesis

Chapter 1 presents an introduction and a survey of anytime tree search algorithms existing in exact methods, Meta-heuristics, AI, Planning. In this chapter, our goal is to provide unified notations and try to integrate the best of each community (with a slight bias towards *Operations Research* due to our background). Some components were designed multiple times by different communities, some were unique to each community and we show that integrating these components can lead to significant performance improvement.

Chapter 2 presents the *Combinator-based Anytime Tree Search framework* that we developed. While designing anytime tree search algorithms it appeared that the best combinations were often counter-intuitive. A strong benchmarking of algorithmic components was needed. This framework allows rapid prototyping of tree search ideas. We present its general architecture, the main features, and a quick usage example in Section **??**.

Chapter 3 presents the algorithm we designed for the EURO/ROADEF 2018 challenge, we present the challenge subject, its industrial context, the problem specific parts. We provide an analysis of the impact of some tree search components applied to this problem. This chapter is inspired by an article we submitted [LF20b]. This Chapter and Chapter 4 are independent, thus, the later may be read before this one.

Chapter 4 presents the algorithm we designed for the Sequential Ordering Problem (SOP). We present a comprehensive study of several tree search components. We show that a simple combination of these leads to a state of the art algorithm that can find new best-known solutions on the largest SOPLIB instances in a few seconds where previous state of the art more complex methods (namely Lin-Kernighan-Held and Ant Colony System combined with Simulated Annealing) needed thousands of seconds. This chapter is inspired by an article accepted at ECAI2020 [LBCJ20].

Appendix A presents an article we submitted with *Florian Fontan* on an academic/industrial continuation of the EURO/ROADEF 2018 challenge [FL20a]. We investigate various guillotine *Cutting & Packing* problems and evaluate the performance of the ideas presented in Chapter 3.

About my research

My Ph.D. topic was originally focused on organ on chip approaches in neuroscience using microfluidic devices (partnership with NETRI[2]). For reasons that fall beyond the scope of this manuscript, we decided to focus on my work on anytime tree search algorithms. The following contributions will not be included in this manuscript:

Microfluidic chip automatic design: Microfluidic chip devices for neurology are a (relatively) new way to study the impact of neurodegenerative diseases (for instance, Alzheimer's disease, Parkinson's disease *etc.*) [TJ10]. They are more precise, less intrusive, and may replace some animal testing. However, such devices are difficult to design because of multiple microfluidic constraints (maximal axon lengths, fluid speed while inseminating the device *etc.*) and fabrication constraints (3D printing resolution, mechanical constraints on the polymer *etc.*). Such constraints make the chip design especially tedious. It sometimes takes months in order to design one chip (and we do not even discuss trial and errors caused by the fact that the neuron behavior is not fully understood yet!). In this research direction, we developed algorithms that aim at facilitating the design of chips, based on the neurofluidic design rules previously established. This research direction involves lots of modelization to obtain a realistic (yet solvable) model, some graph theory (as one part of the problem is to find outer-planar embeddings of a graph) and some algorithmic geometry. The algorithms have been packaged in a chip designer open-source software (written in javascript to be easily usable).

Triangle width: We investigated a simplified version of an embedded-vision problem that involves an input and an output processor. The input processor has to process a set of tasks X and the output processor, a set of tasks Y. Some tasks in Y may have predecessors in X. We show that this problem is $\mathcal{NP}$-hard. We also show that this problem is related to some scheduling problems, and, (surprisingly) is also related to some graph theory problems (graph visit) and matrix visualization. We built a web interface to visualize and solve small instances by hand `http://librallu.gitlab.io/hypergraph-viz/` and submitted an article with *Florian Fontan, Khadija Hadj Salem, Vincent Jost and Frédéric Maffray* [LJS+19].

Balanced words: We present new results and conjectures on N letters balanced words. We published an article in *Theorical Computer Science* with *Nadia Brauner, Yves Crama, Etienne Delaporte and Vincent Jost* [BCD+19]. This is based on work I did during an earlier internship with *Vincent Jost.*

Summer school Meta-heuristic competition: I took part in the Meta-heuristic Summer School 2018 competition (MESS18). The competition was about solving the balanced TSP (Given a graph $G = (V, E)$ and some target value T, find a tour whose edge sum is closest to T). It happened that the instances had some interesting properties (small amount of edges and edge cost could be decomposed in two lexicographic objectives). This allowed to design a MIP model that solved to optimality all instances of the dataset. A report is available online [Lib20].

[2]Neuro Engineering Technologies Research Institute (`https://netri.fr`)

Contents

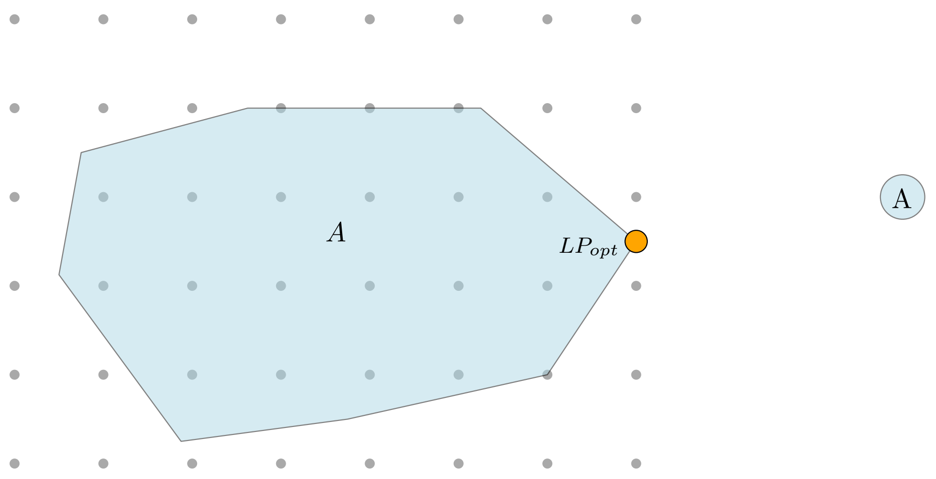
1 Anytime tree search algorithms – An overview

This chapter presents an overview of various anytime tree search techniques. We present anytime tree search algorithms in a unified language inspired from the vocabulary present within Operations Research, AI, planning and meta-heuristics.

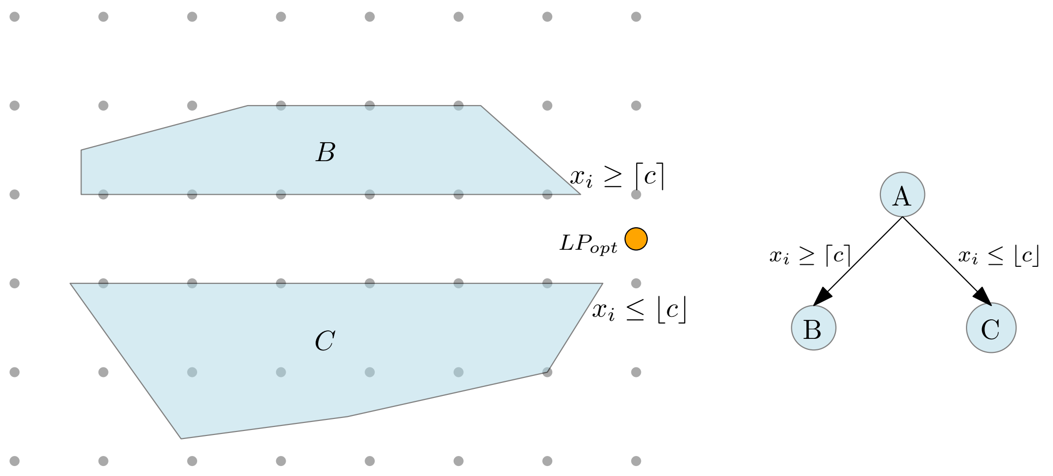
Contents

1.1 Tree search algorithms in operations research

Tree search algorithms are omnipresent within Operations Research. Indeed, most proposed methods to solve a given discrete optimization problem in practice rely on Mixed Integer Formulations, Constraint Programming or SAT. These problems are usually solved by means of a Branch-and-bound.



(a) The root node (A) consists in a polyhedron A where we want to find the best integer solution. Most algorithms use the simplex algorithm. It allows to solve the linear relaxation that has proven to be useful in many circumstances.



B
$x_i \geq \lceil c \rceil$
LP_{opt}
$x_i \leq \lfloor c \rfloor$
C
A
$x_i \geq \lceil c \rceil$
$x_i \leq \lfloor c \rfloor$
B
C

(b) The Branch-and-bound uses the simplex result (LP_{opt}) to decide where to branch (*i.e.* divide the polyhedron into two parts). It generates two children B and C. Then, each of these children will be explored as it is usually done within *divide-and-conquer* schemes.

Figure 1.1: Example of a Mixed Integer Programming Branch-and-Bound resolution.

For instance, Figure 1.1 presents an example of a MIP-based Branch-and-Bound. This algorithm decomposes a polyhedron into two separate polyhedrons. At each iteration, each node (defined as a polyhedron) is terminal (*i.e.* the linear relaxation is an integer or infeasible) or generates two children. Many different search strategies, choices on the variable to branch *etc.* have been studied. We refer the reader to [Ach09] for more information about Mixed-Integer-Programming based Branch-and-bounds.

A second well-known example of tree search algorithms is *Constraint Programming (CP) models*. They consist of a definition of the problem using variable domains and constraints that reduce the variable domains. The algorithm applies a succession of variable domain reduction algorithms. Once it is not able to prune additional values of the domains (we say a fixed-point is reached), the algorithm branches by adding some new constraints (usually of the form $x = c$ and $x \neq c$). Figure 1.2 presents an example of a very simple CP tree search where the goal is to find a 3-coloring of a map. Each variable (country) can be assigned 3 colors (red, green, blue). Two adjacent countries should not have the same colors (constraints)[1].

A final well-known example of tree search in Operations Research is *dedicated Branch-and-bounds*. It consists in a “smart” enumeration of all possible feasible solutions. They take advantage of some problem-specific properties. In the worst case, they enumerate all the solutions as a brute-force algorithm would do. Figure 1.3 presents an example of a dedicated branch-and-bound for the *Asymmetric Traveling Salesman Problem (ATSP)*. In the leftmost branch, the branch-and-bound found a feasible solution which has cost 6 (a, b, d, c, e). On the branch a, b, c the branch-and-bound stops because it cannot continue (the only arcs going out of c are c, b but b is already taken in the partial solution and c, e but e cannot be selected since it should be the last one to be selected (and d is not selected yet). Finally, branches a, c, b, a, d, b, c and a, d, c, b (with costs 7, 7 and 8 respectively) are not explored further because the cost of the partial solution they provide is larger than the feasible solution at cost 6. Thus, they would not provide a better solution than 6.

1.2 Fundamental tree search algorithms

Before introducing anytime tree search algorithms, we first describe three fundamental tree search techniques (*Breadth First Search*, *Depth First Search* and *A**) as almost all anytime tree search techniques are based on these. We briefly present these 3 fundamental algorithms, then introduce their respective variants.

First, it is worth noticing that all algorithms presented in the previous subsection share some similarities. Indeed, they intrinsically define a search tree. This tree is defined by the following elements.

- A root node (in MIPs the original polyhedron ; in CP, the original formulation with no additional constraint due to branching).

[1] We may note that one can build a more efficient model that uses more inference, for instance by adding AllDifferent constraints to some cliques and some symmetry breaking strategies.

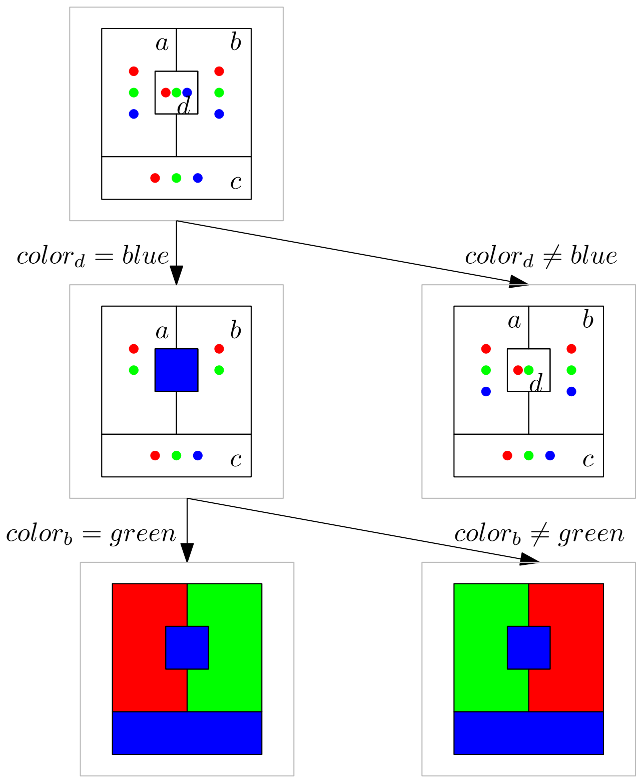
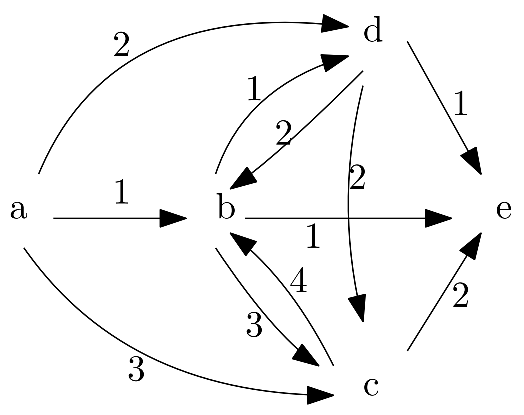
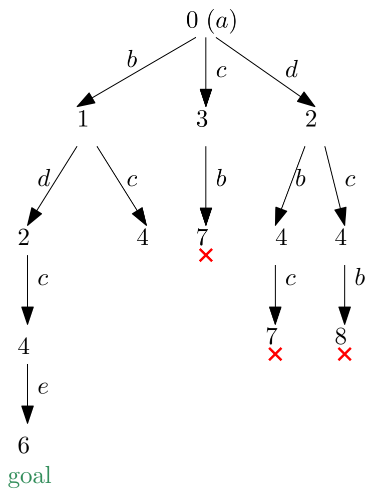


Figure 1.2: Example of a Constraint Programming tree exploration. The root node describes the original problem. In this node, no domain reduction can be done. The algorithm chooses then to branch (on this example on the country d and color blue). The leftmost child has d assigned color blue. Also, it can be deduced from the constraints that a and b cannot be assigned color blue. Finally, the algorithm branches again (here on b with color green). It can be deduced that a cannot be assigned color green (nor color blue from the previous node reasoning). The only remaining available color for a is red. Thus, a is assigned color red. Finally, by the same reasoning, c can be assigned color blue. The same process goes on for the other nodes. As we can see in this example, Constraint Programming takes advantage of computations done within each node to prune domains and thus be able to explore a smaller tree.



(a) Asymmetric Traveling Salesman example. A traveling salesman wants to go from a to e while minimizing the traveling cost (indicated on the arcs)



(b) Tree resulting from a dedicated Branch-and-Bound computation.

Figure 1.3: Dedicated Branch-and-bound example performed on an instance of the Asymmetric Traveling Salesman Problem.

- A way to generate children from a given node (in MIPs, splitting the polyhedron, in CP fixing a variable to a value or forbidding it to take this value).
- Possibly some bounds (LP relaxation in MIPs)
- A way to tell that a given node encodes a feasible solution (in MIPs, an integer LP solution, and in CP a node where all domains contain a single value).

Formally, a generic tree search algorithm relies on a few problem specific parts:

- How to define the root node. Usually, a root node is given as a parameter for the search procedure.
- From a given node, how to generate its children. As we develop later in this thesis, either all the children are expanded at the same time (EXPANDCHILDREN procedure), which is common among tree search algorithms present in the literature (all but one algorithms presented in this thesis use this procedure). There also exists another way: expanding children one by one. When a child of a given node n is extracted, n stays active (*i.e.* in memory) until it does not have children anymore. In this case, we use the primitives HASNEXTCHILDREN and EXPANDNEXTCHILDREN). These procedures are required by SMA* (see later in this chapter).
- A procedure that tells whether the node is a solution or not. We call this procedure ISGOAL as it is usually done in planning algorithms.
- An optimistic estimate of the best possible solution reachable below the node n (lower bound for minimization problems). It allows to prune dominated nodes that have a bound worse than the best solution found during earlier stages of the search. We call

it $f(n)$. Later in this thesis we consider it as a sum of two estimates and show that this decomposition can be useful to define more advanced tree search components.

- Possibly a non-optimistic estimate of the best possible solution reachable below the node n. We call it $f'(n)$. We name it the *guide* function. In some cases, as we discuss in Chapter 3, guiding the search using a bound can lead to poor-quality solutions.

In the following examples of tree search algorithms, light blue nodes are *active* nodes. They exist in the tree search memory and will be eventually explored or pruned before the end of the search. Dark gray nodes are *inactive* nodes. They were explored and are not explicitly maintained within the tree search memory[2]. Also, the isGoal condition is tested at each opened node and keeps the best goal node in memory, we do not include this part in the pseudo-codes to make them clearer. Prunings due to bounds are also performed in each Branch-and-bound algorithm presented in this thesis and not shown in pseudo-codes for the same reason unless we explicitly tell that bounding induced prunings are disabled.

1.2.1 Breadth First Search (BrFS)

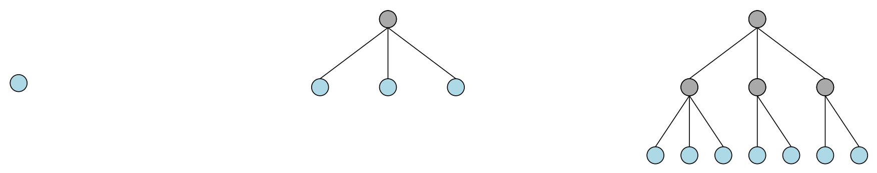
Breadth First Search is mostly known for its use within many graph algorithms. We refer to [CLRS09] for more information about such graph algorithms. Figure 1.4 presents an example of Breadth First Search. Note that Breadth First Search is usually referred as BFS or BrFS. In this thesis, we choose to use the BrFS notation as BFS sometimes stands for Best First Search (regarding Best First Search, we refer to it as Best First to avoid potential confusions).

Algorithm 1.1 presents a Breadth First Search pseudo-code. The algorithm maintains the current level. At the beginning (line 1) the first level corresponds to the root. Each iteration (lines 2 to 8) explores a new level by expanding all children of the current level. At the end of the iteration, the next level replaces the current level (line 7).

Algorithm 1.1: Breadth First Search pseudo-code

```
Input: root node: $root$
1 level ← {$root$}
2 while level $\neq \emptyset$ do
3     nextLevel ← $\emptyset$
4     foreach $n \in level$ do
5         nextLevel ← nextLevel ∪ expandChildren($n$)
6     end
7     level ← nextLevel
8 end
```

[2]Some algorithms explicitly maintain all the partial solution within each node, so they do not have to keep in memory inactive nodes. Some other algorithms only use information relative to the last decision taken within each node. Thus, inactive nodes are required to be able to retrieve solutions. The last one usually consumes less memory but is harder to implement. During our experiments, we did not observe a significant performance difference between the two strategies. Thus, we only consider in this document the former one as it is simpler.



(a) Initial State, only the root node is considered

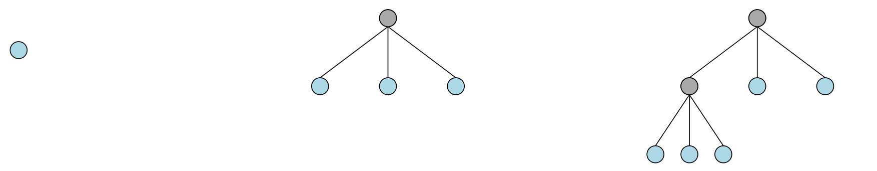
(b) The algorithm expands the root node. At the end of the iteration, its children are in memory.

(c) The algorithm expands all the nodes at level 1. At the end of the iteration, all level 2 nodes are in memory.

Figure 1.4: Breadth First Search iterations. Each iteration explores fully a level of the tree.

1.2.2 Depth First Search (DFS)

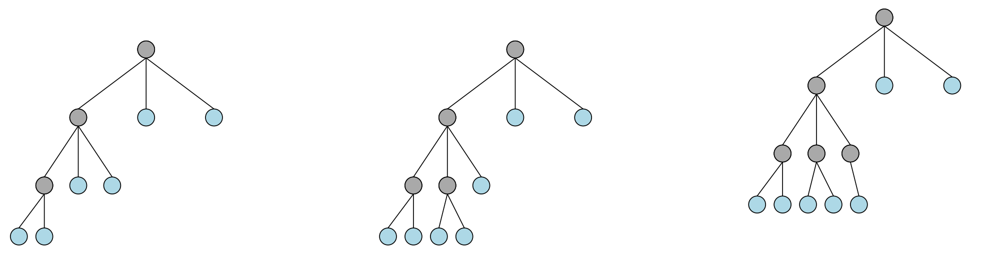
As Breadth First Search, *Depth First Search* is very well-known for its use in diverse graph algorithms. Because of its simplicity, its ability to obtain goal nodes quickly (valid solutions), and its backtracking friendliness, it was (and still is) used extensively in many optimization algorithms. Figure 1.5 presents an example of a tree exploration.



(a) initialization

(b) all root children are expanded

(c) the first child of the root is expanded



(d) some goal nodes are found

(e) the algorithm backtracks

(f) some more goal nodes are found

Figure 1.5: Depth First Search iterations.

Algorithm 1.2 presents a Depth First Search pseudo-code. The algorithm starts by initializing a stack data-structure containing only the root node (line 1). The algorithm starts expanding the last node added to the stack until it is empty (lines 2-6).

Algorithm 1.2: Depth First Search pseudo-code

```
   Input: root node: $root$
1  stack ← {$root$}
2  while stack ≠ ∅ do
3      $n$ ← stack.pop()
4      foreach $c \in$ expandChildren($n$) do
5          stack.push($c$)
6      end
7  end
```

1.2.3 A* / Best First Search

We now present the A* search algorithm [HNR68]. In the context of minimization problems, it consists in selecting first a node with the minimal lower bound. Using this strategy, when A* reaches a goal node (feasible solution), it is guaranteed to be optimal and the search stops. Indeed, let n be the first goal node. Since A* opens the node with the minimal lower bound, opening the node n indicates that the value of the solution encoded by n is smaller or equal to any other remaining node lower bound. Thus, n encodes an optimal solution.

The lower bound $f(n)$ used by A* takes its inspiration from shortest paths problems from s to t. It is usually divided into two parts: $g(n)$ (the distance from s to n) and $h(n)$ (the "as the crow flies" estimate of the distance from n to t) where $f(n) = g(n) + h(n)$:

- $g(n)$ which indicates the *prefix cost*. It corresponds to the cost of all decisions taken in order to reach the node n.
- $h(n)$ which indicates the *suffix cost*. It corresponds to an optimistic estimate of the cost to reach the best possible goal node below n.

The $f(n) = g(n) + h(n)$ bound decomposition is widely used in the AI/planning[3] and seldom used in Operations Research. This decomposition is useful to design new guides for tree search variants (see the next section) and dominance-breaking schemes (see the next chapter).

[3]In the AI/planning communities, $h(n)$ is called as an admissible heuristic, heuristic (in Greek: to find, discover), because it helps to find solutions, and admissible, because it never overestimates a cost (*i.e.* a lower bound). By contrast, non-admissible heuristics can overestimate costs. We believe that this notation is counter-intuitive in operations research since heuristics usually indicate functions that can overestimate costs. In this thesis, we use the terminology "bounds" instead of admissible heuristics and "guides" instead of non-admissible heuristics. For instance, it is common in AI/planning to describe A* (an exact/complete search) as "heuristic search"

Best First Search, (a generalization of A*) is guided by some function $f'(n)$ (not necessarily a bound). Technically, A* is a particular case of *Best First Search* since bounds can also be (and usually are) used as guides. However, in this thesis, we use the term “A*” to designate Best First Search algorithms guided by a bound and use the term “Best First” to designate algorithms that are not guided by a bound.

Figure 1.6 presents an example of a A*/Best First Search. The bounds/guides are represented by the numbers next to the nodes.

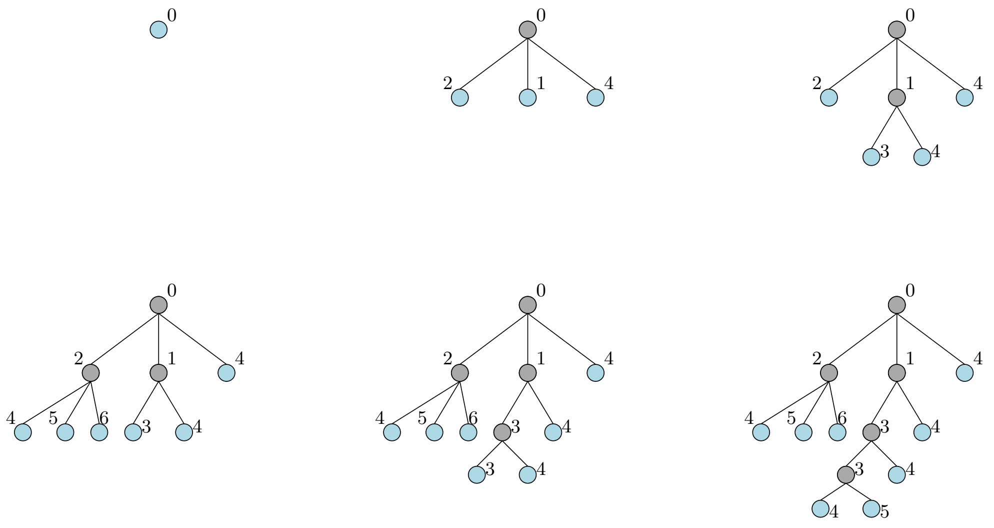


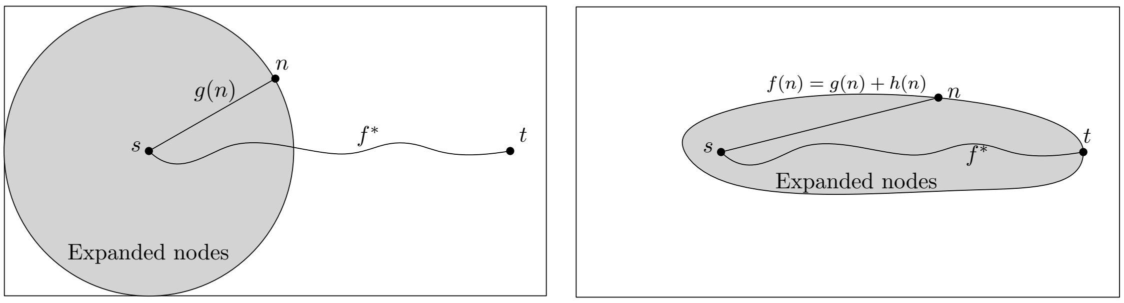
Figure 1.6: A*/best first search iterations

Algorithm 1.3 presents a Best First Search pseudo-code. The algorithm starts by initializing a priority-queue data-structure containing only the root node (line 1). The algorithm starts by expanding the best node added to the stack until it is empty (lines 2-6). If the algorithm is A*, we may stop the algorithm as soon as it finds a goal node (feasible solution).

Algorithm 1.3: Best First Search pseudo-code

```
Input: root node: $root$
1 priorityQueue ← {$root$}
2 while priorityQueue ≠ ∅ do
3     $n$ ← priorityQueue.extractMin()
4     foreach $c \in$ EXPANDCHILDREN($n$) do
5         priorityQueue.push($c$)
6     end
7 end
```

Parallel with shortest paths The primary use of A* was as a shortest path algorithm [HNR68]. Shortest path algorithms are fundamentally based on tree search algorithms where the root is the start vertex. The children of a given node are all unvisited neighbors of the node n. Dijkstra's algorithm[4] can be seen as a variant of Best First Search guided by $g(n)$ which is the distance from s to n. The A* algorithm uses an additional information $h(n)$ that is a lower bound on the distance from n to t. In the case of path-finding, it is usually the distance as the crow flies from n to t.



(a) Illustration of Dijkstra's algorithm. In this example, all costs are unitary thus Dijkstra's algorithm, Breadth First Search and a Best First algorithm guided by $g(n)$ are equivalent.

(b) Illustration of A* algorithm. It uses an additional lower bound that underestimates the cost to go from n to t (in this case the distance between n and t)

Figure 1.7: Shortest path algorithms examples. We show that the use of the "suffix" bound $h(n)$ helps to dramatically reduce the number of expanded nodes in this example.

1.2.4 Greedy algorithms

Surprisingly, greedy algorithms can be seen as another (fundamental and very simple) tree search. At each iteration, only the best child is selected for further expansion (the other children are discarded) until a solution is found. It is used in many contexts (from algorithms like Kruskal's algorithm [CLRS09] where the matroid theory guarantees that the solution obtained by the greedy algorithm is also optimal, to local search or population-based algorithms where it is used as a way to quickly obtain a solution as an initialization step). Such inclusion of greedy algorithms within tree search may seem surprising as they are usually considered as distinct concepts. We believe that it is inclusion is useful while considering anytime tree search algorithms. One of the main reasons is that many tree search algorithms we discuss later in this chapter can behave like a greedy algorithm. For instance, Beam Search, with a beam width of 1, is equivalent to a greedy algorithm. But if Beam Search has a very large beam width, it behaves like a branch and bound. Also, as we define an algorithm as tree search if it explores the tree, it becomes natural to classify greedy algorithms as tree search.

1.3 Anytime tree search algorithms from AI/planning

All the fundamental algorithms presented in the previous sections (Breadth First Search, Depth First Search, A*) have qualities, but also some drawbacks. The algorithms we present in this section are based on these 3 algorithms and aim to correct some of their drawbacks.

[4] In the tree search vocabulary, this kind of algorithms (guided by $g(n)$) is refered as *Uniform Cost Search*. In order to limit the number of notations used in this thesis, we will refer these algorithms as A* guided by $g(n)$ and $h(n) = 0$

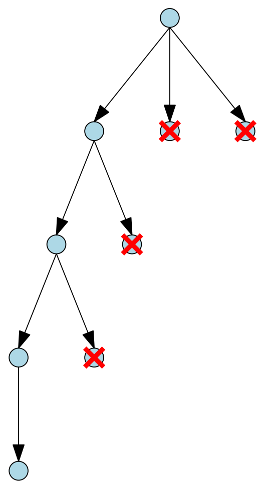


Figure 1.8: Example of a greedy execution. The best child of each node (the leftmost one) is chosen where the others are discarded.

Breadth First Search and Best First Search are not anytime (*i.e.* they are not able to provide solutions in a reasonable amount of time). On large instances, they may not be able to provide any solution within the time limit and the available memory. The main quality of A* is that it opens no node that has a lower bound worse than the best-so-far solution. Depth First Search is anytime (since it dives to the first child of a node before exploring its siblings, usually leading it to find a feasible solution quickly). It is also memory bounded (it contains at most db nodes where d is the maximum depth of the search tree and b the maximum number of children for a given node). However, it suffers from early bad decisions taken in the search tree. Indeed, on large instances, if DFS picks a child that does not lead to an optimal solution (which is likely to happen since guidance is often considered as imprecise close to the root), it has to explore the whole sub-tree before being able to overcome from a bad decision given at a given branch (that is virtually impossible to do because of the exponential nature of the search tree). Thus, DFS usually does not yield competitive results compared to meta-heuristics on large instances. Table 1.1 summarises the pros and cons of the fundamental tree search algorithms using the following (informal) criteria usually considered in the AI/planning communities:

Anytime: Ability to provide feasible solutions quickly and tries to improve them in the later stages of the search.

Memory bounded: performs the search using a polynomial amount of memory. Algorithms that do not have this property may exceed the available amount of memory.

Not sensitive to initial solutions: Ability to perform the search efficiently without requiring an initial solution.

Complete: Ability to detect when the search tree is depleted. This property is crucial for exact methods.

	DFS	**A*/BFS**	**BrFS**	**Greedy**
Anytime	✓			
Memory bounded	✓			✓
not sensitive to initial solutions		✓	✓	✓
Complete	✓	✓	✓	

Table 1.1: Advantages (and Drawbacks if a case is not checked) of fundamental tree search algorithms

As said before, the anytime algorithms presented below aim to correct some of the drawbacks of the fundamental algorithms. Figure 1.9 presents different inspirations from Breadth First Search, Depth First Search and A*. For instance, Beam Search (BS) can be seen as a truncated BrFS, LDS as an incomplete DFS that allows to overcome bad early decisions *etc.* We believe that many Operations Research practitioners are unaware of most of the existing tree search techniques in the AI/planning literature. We present in this section some of the most useful (in our opinion) tree search algorithms from AI/planning for discrete optimization.

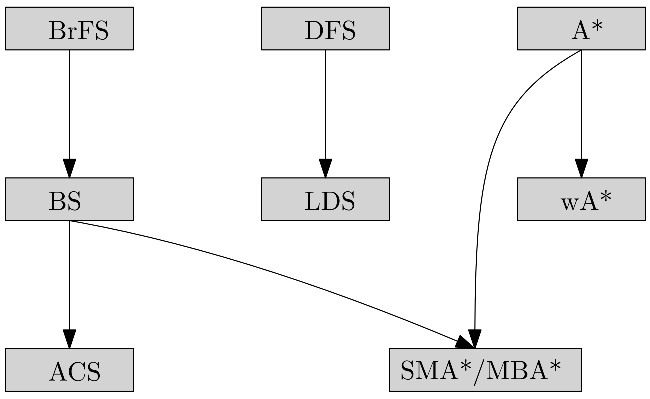


Figure 1.9: Anytime Tree Search Inspirations from the 3 fundamental tree search algorithms.

1.3.1 Limited Discrepancy Search (LDS)

Limited Discrepancy Search [HG95] aims to improve DFS behaviour by limiting the work done on the first child, and allowing it to explore other children. Ideally a tree search should explore a search tree as presented in Figure 1.10. Compared to DFS, it should not wait to have totally depleted the sub-tree below the first explored child to explore others.

Given a maximum number of allowed discrepancies d, an iteration of LDS explores all nodes that have at most d deviations from the best child (*i.e.* chosing a child not prefered by the heuristic). Each node stores an allowed number of discrepancies. The root node starts with d, its first best child also with d, its second best with $d-1$ and so on. Nodes with negative discrepancies are not considered and pruned. This allows to explore the most promising branches of the tree while still giving a quick look at the other branches.

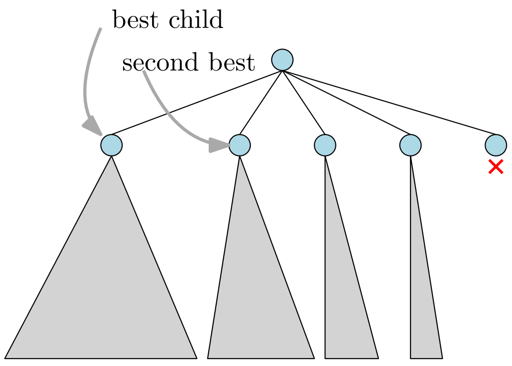


Figure 1.10: Ideal behaviour of a modified DFS. The first child is explored relatively intensively, the second best a bit less, the third one even less. Possibly, some of the last children are not explored.

It usually gives better solutions than DFS but also takes more time to deplete the search tree. If $d = 1$, LDS behaves like a greedy algorithm. If $d = \infty$, LDS behaves like a DFS.

Algorithm 1.4 shows the pseudo-code of an iterative LDS algorithm. A succession of LDS iterations (starting at 1 (line 1), then 2, then 3, *etc.*) are performed. Each LDS iteration initializes at each node an allowed discrepancy limit (line 4 for the root) and adds to the stack all children that have their quality rank in the brotherhood smaller than the number of allowed discrepancies of the current node (lines 6-15).

Algorithm 1.4: LDS algorithm

```
Input: root node
1  $D \leftarrow 1$
2  while stopping criterion not met do
3      $root.D \leftarrow D$
4      Candidates $\leftarrow$ root
5      while Candidates $\neq \emptyset$ do
6          $n \leftarrow$ Candidates.pop()
7          $i \leftarrow 0$
8          for $c \in$ sortedChildren($n$) do
9              $c.D \leftarrow n.D - i$
10             Candidates.push($c$)
11             if $c.D = 0$ then
12                 break
13             end
14             $i \leftarrow i + 1$
15         end
16     end
17     $D \leftarrow D + 1$
18 end
```

Figure 1.11 presents an example of a LDS iteration. We may note the tree shape similarity from the ideal modified Behaviour in Figure 1.10.

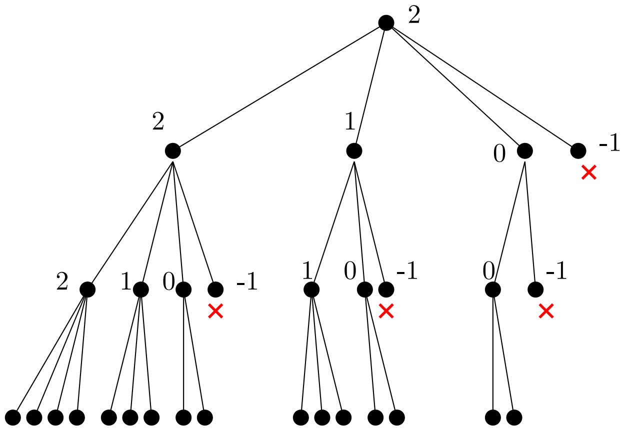


Figure 1.11: Example of a LDS iteration where $D = 2$

1.3.2 Beam Search (BS)

In LDS, nodes are selected depending on a comparison with their siblings and not depending on their absolute quality. We now present *Beam Search (BS)* that aims to explore a subset of a tree that only keeps the best nodes at a given level. Beam Search has been used successfully to solve many scheduling problems [OM88, SB99]. Beam Search is a tree search algorithm that uses a parameter called the beam size (D). Beam Search behaves like a truncated *Breadth First Search (BrFS).* It only considers the best D nodes on a given level. The other nodes are discarded. Usually, we use the bound of a node to choose the most promising nodes. It generalizes both a greedy algorithm (if $D = 1$) and a BrFS (if $D = \infty$).

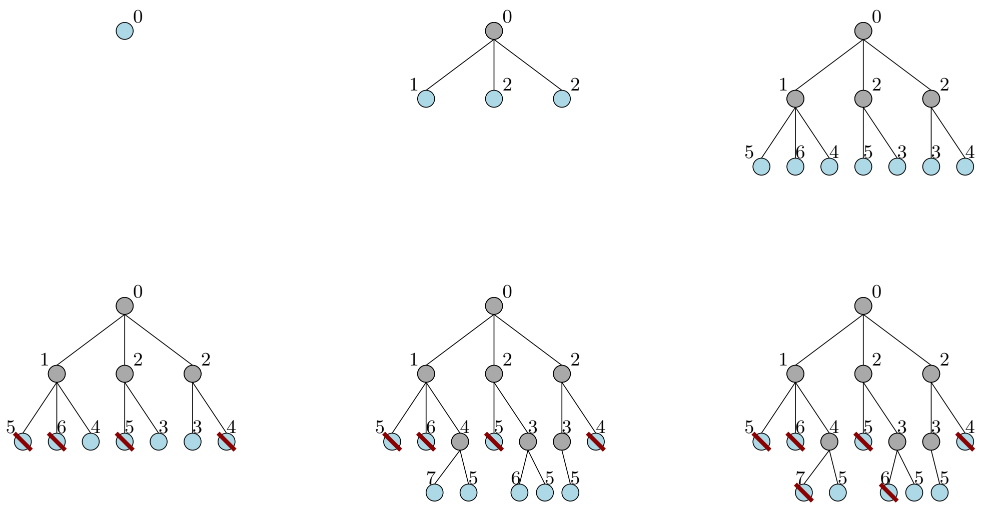


Figure 1.12: Beam Search Iterations with a beam width $D = 3$

Figure 1.12 presents an example of beam search execution with a beam width $D = 3$.

Beam Search was originally proposed in [R$^+$77] and used in speech recognition. It is an incomplete (*i.e.* performing a partial tree exploration and can miss optimal solutions) tree search parametrized by the beam width D. Thus, it is not an anytime algorithm. The parameter D allows to control the quality of the solutions and the execution time. The larger D is, the longer it will take to reach feasible solutions, and the better these solutions will be.

Beam Search was later improved to become *Complete Anytime Beam Search* to make it anytime. The idea is to perform a series of beam searches with a heuristic pruning rule that weakens as the iterations go [Zha98]. They prune a node n' if its bound exceeds by some constant the bound of its parent n (*i.e.* n' is pruned if $f(n') > f(n) + c$ with c increasing as iterations go). In this variant the beam is not limited to a beam width D, thus, tuning the parameter c is crucial. This variant is called *Iterative-weakening Beam Search.* Since many algorithms were later designed to make the beam search complete[5] (for instance *Anytime Column Search* or *Beam Stack Search* both presented later in this thesis), thus also being complete anytime beam searches, we refer this algorithm as an iterative beam search to avoid potential confusions.

Another iterative beam-search variant increases the beam width at each restart. It consists in performing a series of beam searches with increasing D. To the best of our knowledge, such approaches have not been much studied in the literature. We may cite its use on the car sequencing problem [GRB15] that consists of beam search runs of sizes $\{5, 10, 25, 50, 100, 500, 1000, 1500, \infty\}$. As the last iteration is not limited in width, this iterative version is complete.

It is worth noticing that Iterative Beam Search may reopen many nodes. Another possible beam increasing scheme could be to start by a beam of $D = 1$ (greedy algorithm). Then when the current search finishes, multiply D by some constant (for instance, a geometric growth of 2), thus running a beam of size 2, then 4, and so on. Such a scheme appears to be efficient in practice (and guarantees that not too many nodes are re-opened). To the best of our knowledge (and our surprise), such a variant does not seem to have been studied before. In this thesis, we will use the terminology "iterative beam search" this geometric-growth variant.

Algorithm 1.5 shows the pseudo-code of an iterative beam search. The algorithm runs multiple beam searches starting with $D = 1$ (line 1) and increases the beam size (line 8) geometrically. Each run explores the tree with the given parameter D. At the end of the time limit, we report the best solution found so far (line 10). In the pseudo-code, we increase geometrically the beam size by 2. This parameter can be tuned, however, we did not notice a significant variation in the performance while adjusting this parameter. This parameter (that can be a real number) should be strictly larger than 1 (for the beam to expand) and should not be too large, say less than 3 or 5 (otherwise, the beam grows too fast and when time limit is reached, most of the computational time was possibly wasted in the last incomplete beam, without providing any solution).

[5]A complete search can deplete the search tree and detect it did. Exact methods rely on complete search.

Algorithm 1.5: Iterative Beam Search algorithm

```
Input: root node
1 $D \leftarrow 1$
2 while stopping criterion not met do
3     Candidates $\leftarrow$ {root}
4     while Candidates $\neq \emptyset$ do
5         nextLevel $\leftarrow \bigcup_{n \in \text{Candidates}} \text{children}(n)$
6         Candidates $\leftarrow$ best $D$ nodes among nextLevel
7     end
8     $D \leftarrow D \times 2$
9 end
```

This geometric-growth variant appears to be a competitive algorithm on various problems in practice. Moreover, it appears that the average number of times a node is reopened is constant (or close to it in most cases). If opening a node can be done efficiently (for instance in $O(1)$), the iterative beam search can be more effective than a variant that stores all visited nodes (the storage usually costs $O(\ln n)$).

In Proposition 1.3.1, we present an argument why the iterative beam search opens a (close to) constant amount of nodes. We assume that each tree level is large enough to completely fill the largest beam width. Thus making this Proposition only valid on large search trees (as the ones we consider in this thesis). We may also keep in mind that it is an approximation as the very first tree levels usually contain less nodes than the beam width. In practice, this effect is so small that we belive we can safely neglect it.

Proposition 1.3.1. *An iterative beam search with a growth factor $k > 1$ opens at most $\frac{k}{k-1}$ times more nodes than the total number of nodes it explores.*

Proof. Given the n-th iteration of the iterative beam search, in the worst case, k^n nodes are opened at least once. The average number of openings of a given node is:

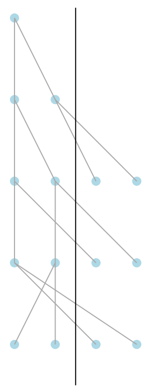
$$\frac{\sum_{i=0}^{n} k^i}{k^n} = \sum_{i=0}^{n} \frac{1}{k^i} \leq \sum_{i=0}^{\infty} \left(\frac{1}{k}\right)^i = \frac{1}{1-\frac{1}{k}} = \frac{k}{k-1}$$

□

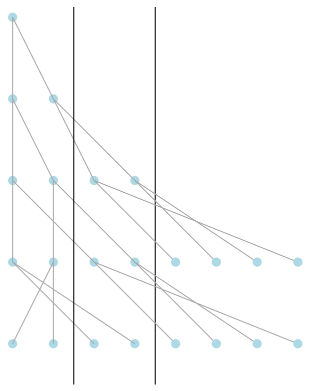
With $k = 2$ as described above, the iterative beam search opens at most twice the number of nodes. With $k = 3$, it opens on average 1.5 times a node. With $k = 1.5$, it opens in average 3 times a node. The parameter k allows controling the number of reopenings at the expense of the ability to provide often solutions. Decreasing it would allow providing more solutions at the expense of the number of node reopenings.

1.3.3 Weighted A* (WA*)

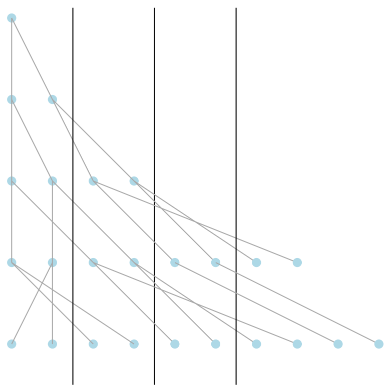
Weighted A* [Poh70] consists in modifying the A* guide as follows: $f'(n) = g(n) + w.h(n)$ where $w \geq 1$. It relaxes the optimality of the first solution by a factor of $w > 1$. The first solution found by Weighted A* is at most w times worse than the optimal one. The bigger



(a) First iteration of Anytime Column Search. It is similar to a beam search of width $D = 2$ but the nodes that could be heuristically pruned are stored in memory.



(b) Another beam search is performed starting by the higher remaining nodes in the tree.



(c) Another beam search is performed

Figure 1.13: Anytime Column Search iterations examples with $D = 2$.

w, the faster Weighted A* will find solutions. In the specific case of $w = 1$, we get the A* algorithm. If $w = \infty$ the first obtained solution is guided only by $h(n)$. It can be seen as minimizing the remaining work to do to obtain a solution.

1.3.4 Anytime Column Search (ACS)

Anytime Column Search [VGAC12] consists in extending a beam search by storing nodes that would have been heuristically pruned[6]. It keeps a priority queue for each level in the tree. It takes as a parameter an integer D (similar to the beam width). For each level, it expands the best D open nodes and goes to the next level. Once all nodes on a given level are explored, it starts again on level 0 and continues until the search is ended. This algorithm aims to make a complete anytime search based on Beam Search.

Algorithm 1.6 presents the pseudo-code of the Anytime Column Search procedure.

Algorithm 1.6: Anytime Column Search Pseudo-code

```
1 levels ← [ EMPTYPRIORITYQUEUE for each level $l \in \{0 \dots \text{maxDepth}\}$ ]
2 levels[0].add(root)
3 while some nodes are not explored do
4     for $i \in \{0 \dots \text{maxDepth}\}$ do
5         expand the $D$ best nodes at level $i$ if they exist
6         insert every child at levels[$i + 1$]
7     end
8 end
```

Figure 1.13 presents an example of an Anytime Column Search run with a width $D = 2$.

[6]We may note that a variant of Anytime Column Search was also independently discovered in Operations Research and is called *Cyclic Best First Search* [MSZ+17]. It corresponds to an Anytime Column Search with $D = 1$.

1.3.5 Simplified Memory-bounded A* (SMA*)

*Simplified Memory-bounded A** [Rus92] consists in exploring the search tree as A*, opening the node n with the smallest lower bound. It adds to the fringe the child of n which has the smallest $f(n)$, and updates the lower bound of n if needed. If n does not have any other child, n is discarded, otherwise, it is added again to the fringe with an updated lower bound. Like beam search, it considers a maximum fringe size D. If the fringe contains more than D nodes, SMA* discards the nodes which have the largest lower bound. As beam search or LDS, SMA* generalizes the greedy algorithm if $D = 1$, and SMA* generalizes A* if $D = \infty$.

Figure 1.14 presents an example of SMA* iterations. Algorithm 1.7 presents the SMA* pseudo-code.

Algorithm 1.7: Simplified Memory-bounded A* (SMA*)

```
1  fringe $\leftarrow$ {root}
2  while fringe $\neq \emptyset \wedge$ (time < time limit) do
3      $n \leftarrow$ extractBest(fringe)
4      fringe $\leftarrow$ fringe $\setminus \{n\}$
5      if hasChildren($n$) then
6          fringe $\leftarrow$ fringe $\cup$ {nextChildren($n$)}
7          fringe $\leftarrow$ fringe $\cup \{n\}$
8      end
9      while |fringe| > $D$ do
10         $n \leftarrow$ extractWorst(fringe)
11         fringe $\leftarrow$ fringe $\setminus \{n\}$
12     end
13 end
```

SMA* mainly differs from other algorithms presented so far by exploiting a new primitive: extracting a child of a given node and possibly updating its bound (using the HasNextChild and getNextChild procedures described in Chapter 1). During the EURO/ROADEF 2018 challenge (presented in Chapter 3), we independently discovered a variant of SMA* where we use the "classical" getChildren instead of the procedures used by SMA*. We named it *Memory Bounded A* (MBA*)*. In this thesis, we use the terminology "MBA*" to designate the variant using getChildren and SMA* the variant using HasNextChild and getNextChild.

1.3.6 Other tree search algorithms

In the previous section, we discussed some of the simpler and most efficient (in our opinion) anytime tree search algorithms from AI/planning for combinatorial optimization. However, there exists many more algorithms that could be applied to combinatorial optimization problems. This section aims to quickly present them, their underlying ideas, and strengths (without pretending to be exhaustive).

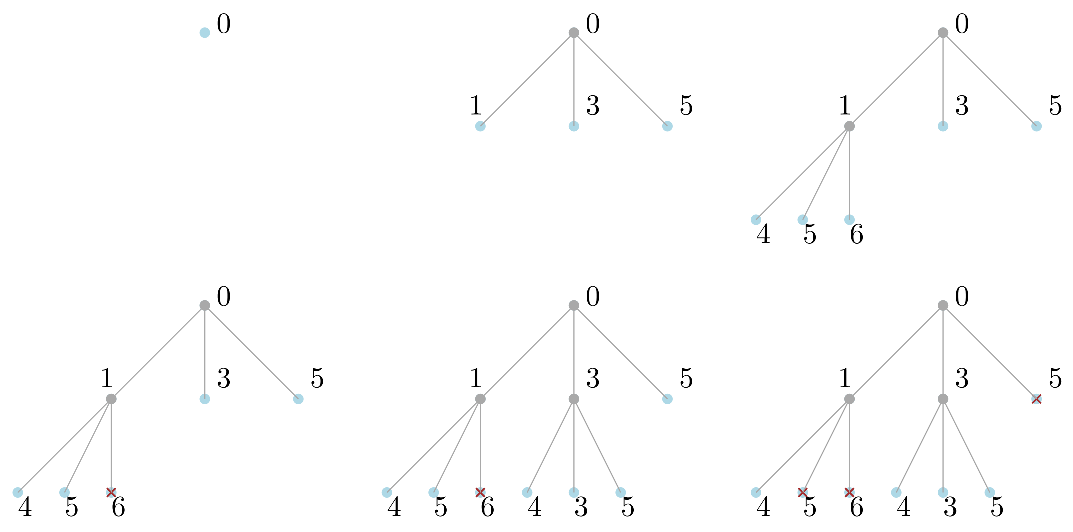


Figure 1.14: MBA*/SMA* iterations using standard children computation with width $D = 3$

Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search [CBSS08] obtained recent successes in game-playing [Cou06, BPW$^+$12, GS11, GKS$^+$12]. It is a method that is guided by experiments (randomized greedy starting at a given node). It usually uses the *Upper Confidence-bound on Trees (UCT)* formula that allows us to make an exploration/exploitation trade-off where promising nodes are more attractive, but also, nodes that were little explored are attractive. MCTS has many qualities (it is anytime since the experiments allow to find good solutions fast, it is complete since it explores the tree in a best-first search manner, and, it does not require any bound or guide). However, little success has been achieved in combinatorial optimization. We believe that it is due to the fact that it is crucial to rely on a-priori knowledge (*i.e.* guides and bounds) while solving a combinatorial optimization problem, thus, MCTS would take a very long time to be as competitive as a simple greedy algorithm on a hard combinatorial optimization problem. However, we advocate more work on the integration of MCTS for combinatorial optimization by adding classical guides within MCTS. We expect MCTS to be able to overcome some bias that a-priori knowledge may have on some instances.

Iterative Sampling

[Lan92] consists in following random paths. It is similar to greedy random constructive algorithms where only the random part is considered. *Iterative Sampling* is a special case of a greedy random algorithm.

Greedy Best-First Search

[BG01] considers a priority queue ordering using $h(n)$ as a selection criterion. It provides good solutions very fast. It is a best-first algorithm that aims to provide a feasible solution as fast as possible.

Improved Limited Discrepancy Search

[Kor96] improves the limited discrepancy search by only considering nodes that have exactly k discrepancies and not $\leq$ discrepancies. It only explores leaf nodes that are at discrepancy k and not all nodes below k. This makes LDS a valid exploration strategy as DFS. It prunes nodes that have a remaining depth below their number of allowed discrepancies. This modification allows LDS to be as efficient as DFS for proving optimality.

Recursive Best-First Search (RBFS)

[Kor93] improves the Best First algorithm by making it linearly bounded ($O(d)$ where d is the maximum depth of a node in the search tree). It adds to the nodes a bound f_{min} that is the remaining value that remains to be explored below a node. After exploring a node, it backtracks in the tree in order to reach the next node with f_{min}. Basically, it trades computation time (by reopening ancestors of the last node explored) for memory (since a linear number of nodes need to be stored). We may also cite some variants of RBFS like weighted RBFS that improves WA* [HZ07].

Beam Stack Search

[ZH05] extends the beam search by allowing it to continue exploring the tree. It implements a data structure called a *beam stack* and allows to integrate beam search within an exhaustive depth first search.

Beam Search Using Limited Discrepancy Backtracking (BULB)

[FK05] combines the Beam Search and the Limited Discrepancy Search to allow the beam search to backtrack following an LDS scheme. It is in some sense very similar to Beam Stack Search.

Anytime Pack Search (APS)

[VAC16] is a variant inspired from Anytime Column Search. Instead of selecting the starting nodes at the same level, it allows the beam search to select nodes at different levels at the same time according to their attractiveness.

Informed Backtracking Beam Search

[Wil10] modifies the Beam Search to make it complete. Instead of deleting nodes within the beam, we run the beam search, either storing new nodes in the beam or storing them in another data structure. When the beam finishes, it extracts the next most promising node in the storage and runs a beam search again.

Restarting WA* (RWA*)

[RTR10] consists in restarting a weighted A* and decreasing the weight w.

Anytime Nonparametric A* (ANA*)

[VDBSHG11] uses a guide function defined as follows (the algorithm opens the node with the maximal value first in this case):

$$f(n) = \frac{G - g(n)}{h(n)}$$

where G is the value of the best solution found so far, initially fixed at a large value. Opening a node using this criterion corresponds to the greediest possible way to improve the current solution[7].

Anytime Window A* (AWA*)

[ACK07] considers a window of size w. The exploration is done by opening only nodes at depth $w - l$ to w where l is the depth of the window. When all nodes within this range are explored, the window slides down. The nodes not in the window range are frozen. When the window reaches the bottom of the search tree, if it left no frozen nodes in the tree, the algorithm is optimal and stops, otherwise, the search is done again with a larger window (usually increased by 1). In the case where $w = 1$, it performs a DFS. When w is large, the algorithm behaves like A*.

Tree search algorithms not included in this thesis

While some tree search techniques can be imported from AI/planning to Operations Research, some are more specific to other problems types. In order to give a broad picture of the tree search algorithms in AI/planning, we present some (very popular) of them and why we do not discuss them in this thesis.

- Incremental heuristic search methods. For instance *D* lite* [KL02]. This kind of method is applicable when one performs a large number of repeated searches that are similar (re-planning methods mostly found in robotics). They take advantage of reusing previous searches to find solutions faster than without this information. Since we usually only perform one search in the paradigm presented in this thesis, we chose to not consider incremental heuristic searches.

- Iterative Deepening search methods. For instance, the well-known Iterative Deepening Depth First Search [Kor85] and many of its variants inspired from the algorithms presented in this chapter like Depth-Bounded Discrepancy Search [Wal97], Iterative deepening A* [RM94]. It consists in a succession of depth bounded tree search algorithms (where nodes below a given depth limit are pruned). If the search does not produce any result, it is restarted with a deeper depth bound. This technique has proven to be effective in puzzle-solving. Indeed, in such contexts, the goal is to find a solution that uses the smallest possible number of steps. Thus, the best goal nodes are (relatively) close to the root in a puzzle-solving context. In most discrete optimization problems, goal nodes (feasible solutions) are mainly found at the very bottom of the search tree. Thus making this kind of methods dominated by

[7]Source code for the original ANA* implementation can be found here: `https://github.com/sbpl/sbpl/blob/master/src/planners/ANAplanner.cpp`.

a simple Depth First Search or Best First like strategy. For this reason, we did not investigated these methods.

1.4 Constructive meta-heuristics seen as tree search

Even if the meta-heuristics literature mainly focuses on local-search-based or population-based methods, some meta-heuristics (for instance, GRASP and ACO) have a constructive nature, thus can be seen as anytime tree search algorithms. In this section, we present both of them from a tree search perspective.

1.4.1 Greedy Random Adaptive Search Strategy (GRASP)

GRASP [MSS99] consists in a Greedy algorithm that performs a "weak" inference at each node (hence the "adaptive" keyword indicating that the choices made at a given node depend on the sequence of previous choices[8]). The algorithm is greedy randomized (*i.e.* the next node is selected depending on its a-priori attractiveness and a random part). Finally, GRASP algorithms usually perform a local search step in order to further improve the obtained solutions[9]. For additional resources about GRASP, we invite the reader to consult [FR02] and [RR16].

1.4.2 Ant Colony Optimization (ACO)

Ant Colony Optimization [DMC91] uses the biological metaphor of ants, seeking food. Ants put pheromones along the path they take. The more food they bring, the more pheromones they will put. At the beginning, ants move at random and as time goes, ants only select the shortest path to food.

Some work has been done to consider ant colony optimization as tree search methods [Blu05b, Man99]. Each ant can be seen as an agent that explores the tree in a greedy manner. At each node, the ant is guided by two informations: η_{ij} the guide indicating the quality of the move $i \to j$ and τ_{ij} the amount of pheromones between states i and j. As search goes, τ_{ij} is updated in order to explore solutions similar to the ones that achieved the best performance and to avoid those that did not performed well. More formally, ACO can be seen as an iterative greedy where an online learning is performed to better guide the search.

In its simpler version, the ant chooses the decision $i \to j$ using the following formula:

$$p(ij|i) = \frac{\tau_{ij}^{\alpha}.\eta_{ij}^{\beta}}{\sum_{k \in \mathcal{N}(i)} \tau_{ik}^{\alpha}.\eta_{ik}^{\beta}}$$

[8]For instance in the TSP, the greedy algorithm may not select vertices already visited, in this sense, it is adaptive.

[9]If good local-search operators are available, we invite to combine anytime tree search algorithms with local search as much as possible (since they are highly complementary). In order to maintain the main message clear (*i.e.* anytime tree search can be enough to design state-of-the-art algorithms in some situations) we purposefully do not discuss nor implement local search algorithms.

Where $p(ij|i)$ indicates the probability of choosing j at node i. $\mathcal{N}(i)$ indicates the possible children of node i and α, β are hyper-parameters of the algorithm. Those hyper-parameters are used to adjust the exploitation of the a-priori guide or the online-learned guide.

Each ant finding some solution updates the pheromones as follows:

$$\tau_{ij} = (1-\rho)\tau_{ij} + \rho F(S)$$

where ρ is the learning rate. if ρ is close to 0, almost no learning will be done, and if ρ is close to 1, pheromones will be reset at every iteration. $F(S)$ is a function of a solution and increases pheromone trails if S is a good solution.

Also, for each decision taken by every ant, a pheromone evaporation is performed as follows:

$$\tau_{ij} = (1-\phi)\tau_{ij} + \phi\tau_{0_{ij}}$$

where ϕ is the evaporation parameter.

Conceptually, ant colony optimization can be seen as a memory management in the tree search. Indeed, after an iteration, pheromones are updated and the next iteration will explore parts of the tree where better solutions were found. This mechanism helps the tree search to adapt its guide depending on where the best solutions were found. This principle helps to overcome cases where the guide function does not provide good overall guidance.

In the classical Ant Colony Optimization algorithm, each ant is an agent that explores the tree by doing a greedy random selection at each node. The probability of selecting a node depends on the a-priori attractiveness and the pheromone trail. The pheromones can be seen as a data structure that maps states and decisions to an online-learned attractiveness represented as a real value. Usually, pheromones are put on decisions that are taken. For instance, for TSP-like problems, pheromones are put on edges.

ACO variants

On its own, the original ACO algorithm (called Ant System (AS)) is not competitive with other approaches. Some competitive variants were presented and aim to improve the Ant System scheme.

Elitist AS: [DMC$^+$96] consists in giving an additional weight of the best solution found so far. In practice, we "add" the best-known solution to the pool of newly found solutions.

Rank-based AS: [BHS97] sorts solutions according to their quality and only the w best ants are able to deposit pheromones and the best ants deposit more pheromones ($w - r$ where r is the rank of the ant).

MMAS: $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System [SH98] considers a trail minimum (resp. maximum) denoted $\tau_{\min}$ (resp. $\tau_{\max}$). If $\tau_{\min}$ is strictly greater than 0, each solution part can be chosen with a non-zero probability. At the beginning, each pheromone value is assigned to $\tau_{\max}$. Moreover, only the best ant is able to increase the pheromones values. At each round, each pheromone value is decreased and only the one within the best solutions may maintain the maximum level of pheromones.

ANTS: *Anytime Non-deterministic Tree Search* [Man99] obtained promising results on the *Quadratic Assignment Problem.* It is the first algorithm, to the best of our knowledge, to consider Ant Colony as a form of tree search. It performs bounding prunings and reports an estimate of the quality of the solution found, to be used to update pheromones.

ACS: *Ant Colony System* [DG97] introduces a parameter q_0 which gives the probability for a given ant to move deterministically by choosing the next node $j = \arg\max_{j \in \mathcal{N}_i}(\tau_{ij}^{\alpha}.\eta_{ij}^{\beta})$

Ant-Q: [GD95] was intended to create a link between Ant Colony Optimization and reinforcement learning. It was replaced by ACS since the pheromone updates rules where computationally equivalent and simpler.

Beam-ACO: [Blu05b] replaces the greedy random behaviour of ants by a beam search. Doing so, each ant reports many solutions instead of only one. This ACO variant obtained excellent results on the open shop problem.

Hyper-Cube framework: [BD04] restricts the pheromones within a [0,1] range. This algorithm allows to handle automatically the scaling of objective functions.

Moving beyond the metaphor

We propose to integrate Ant Colony Optimization within a more general tree search framework. Some work has been done in order to examine the contribution of each part of the ant colony optimization and try to explain its success [LISD16]. We study further ant colony optimization, from a tree search point of view and try to combine tree search techniques within the Ant Colony Optimization framework. As we will discuss, classical ACO techniques can be integrated within a tree search framework. This new view may lead to new ideas of hybridizations between ACO and AI-style tree search and Reinforcement Learning.

Let us start by generalizing ACO. Algorithm 1.8 presents a generic ACO algorithm. It starts by initializing an online-guide (for instance, all the pheromones set-up to 1). Then, it runs several tree search algorithms partially guided by the online-guide (an ant iteration) and regularly updates it to integrate new acquired knowledge into future search algorithms (pheromone update).

Algorithm 1.8: Generic ACO algorithm

```
1 Initialize online-guide
2 while stopping criterion not reached do
3     Run nbAnts anytime non-deterministic tree search algorithms
4     Update online-guide
5 end
```

An Ant Colony Optimization is made of several parts:

- A pheromone (or policy) store that maintains the policy learned during the previous search iterations. It associates *decisions* to an online-learned guide value (usually a simple real number).
- A series of tree search algorithms partly guided by the pheromones. A randomized greedy for classical ACO and a probabilistic beam search for Beam-ACO.

We now discuss possible modifications for each of this parts:

Anytime tree search algorithms The easier modification would be to update the anytime tree-search algorithm (ant iteration). Adaptive greedy random tree search algorithms are usually used in Ant Colony Optimization. It consists in a fast algorithm that diversifies the search by randomization. However, it suffers from a lack of performance compared to Beam Search like algorithms that are able to explore more promising solutions if their width are sufficiently large. Beam-ACO [Blu05b] aims to replace the greedy randomized algorithm by a beam search. This method was for several years the state of the art for the open-shop problem. Using a Beam Search allows to intensify more the search space at the price of less learning of good solution parts. As we consider the pheromones as a tree-search component that builds upon some guide, it may help to improve other existing tree search algorithms as well. For instance, one could think of a LDS-ACO or a MBA*-ACO.

Pheromones and decisions Originally, Ant colony Optimization was mostly used on graphs, for instance in TSP-like problems. Decisions were edges of the graph. However, we can think of several other decisions (and as we will see later, combining them within a single algorithm). For instance, on a TSP-like problem, one can use as decisions, edges of a graph but also city j being at position i, or city j being scheduled after city k.

Some work combined multiple pheromones structures within a single Ant Colony Optimization algorithm and reported an efficient algorithm on a variant of the 2005 ROADEF Challenge [Sol08]. More pheromone structures may lead to a better learning and may be considered. However, one needs to set different weights on each pheromone structure if some are more useful than others.

Links with Reinforcement Learning This online-learning view of ACO is very similar to some Reinforcement Learning techniques. Indeed, most Reinforcement Learning techniques consider the following hypothesis:

1. No supervisor: The agent only obtains rewards (positive if it did well, and negative if it did bad)
2. Delayed feedback: Rewards are obtained after possibly many steps
3. Over sequential processes: There is a strong correlation between an action and the previous action. The agent is influencing the data it gets through its actions.

A learning procedure in tree-search uses the same hypothesis. Indeed, rewards (solution quality) are obtained only after a series of decisions (delayed feedback). Moreover, the solution quality depends on, and is strongly correlated to the quality of each decision taken during the tree-search iteration. Reinforcement Learning (RL) offers, to the best of our knowledge, new ways to learn within the ACO framework. Indeed, most of RL algorithms consider two numbers (the number of times the agent selected a decision, and the average reward) whereas ACO only considers one (the pheromones). Pheromones values usually use a real number to encode possibly many information like the probability to find an improving solution, the number of times we used a given solution part without finding any good solution, or solution parts not used frequently during the search. The nature-inspired pheromone value serves all of these purposes at once but struggles to express every criterion within a single real value. One possible variant could be to integrate some Reinforcement Learning algorithm that uses a different information structure.

For instance, *Upper Confidence Bound (UCB)* considers the number of times a decision has been taken and the (weighted) average it gave for each decision. Such a UCT-ACO variant would be able to better control the compromise between exploration and exploitation. For more details about Reinforcement Learning, we invite the reader to consider [SB18].

A deterministic tree search for ACO? Non-deterministic tree search algorithms are said to be efficient in order to build a large diversity in solutions. However, many tree search algorithms, like LDS or Beam Search are known to be able to diversify the search in a deterministic fashion (even being able to be complete). Combined with a UCB-based pheromone system, it could be possible to consider a deterministic variant of Ant Colony Optimization.

Chapter conclusion

In this chapter, we investigated several tree search proposed by multiple communities, namely Operations Research exact methods with MIPs and CP, anytime tree search algorithms from AI/planning and some meta-heuristics that can be seen as tree search algorithms. As we have seen, there exist many algorithms. Also, many “tree search algorithmic components” like the online-guide of Ant Colony Optimization, specific branch-and-bound techniques like strong branching, dominance and symmetry breaking, or even the probing/diving strategies could be added to each tree search discussed above. This would make a huge number of potential algorithms. The next chapter aims to provide a unified tree search framework that can combine each tree search “building block” in order to obtain all of these combinations. We present in the last chapters of this thesis some success stories using this framework and this decomposed tree search vision and show that:

- Anytime tree search algorithms presented in this chapter can be competitive with classical meta-heuristics on some problems.
- Decomposing tree search based on a set of building blocks allows to better analyze the importance and synergy of each of them leading to a better overall understanding of why such methods work (as proposed for other meta-heuristics [Sör15]).

2 Combinator-based Anytime Tree Search framework (CATS)

This chapter presents *Combinator-based Anytime Tree Search framework* (CATS). It enables to rapidly prototype tree search and compare them in a fair environment. It also shows good performance. Indeed, tree search algorithms presented in the next chapters use CATS to implement their algorithms and are competitive compared to the state-of-the-art. The CATS framework aims to generalize the ideas we designed with *Florian Fontan* during the EURO/ROADEF2018-2019 challenge (presented in Chapter 3). We co-wrote many combinators with *Abdel-Malik Bouhassoun* during his masters thesis.

Contents

2.1 Why a generic search framework?

While we were designing and implementing methods to solve discrete optimization problems, it appeared clear that such a generic framework was required. This section aims to provide a few insights on why an OR practitioner should care about developing and using such a generic code.

Easier implementation The first (obvious) aim for a generic framework is to provide an easier implementation. Indeed, many algorithms take days, sometimes weeks to be implemented[1]. Re-using such algorithms is much more time-efficient and allows us to perform more diverse trials while designing an optimization algorithm.

A better description of algorithms Designing a generic piece of code imposes to separate generic concepts from problem specific ones. As we developed the CATS framework we noticed that our vocabulary changed. We started to express (and see) tree-search algorithms as a combination of elementary concepts. For instance, "a forward search with prefix bound using a beam search strategy for the SOP"[2] instead of yet another "novel state-of-the-art constructive meta-heuristic for the SOP". Such precise terminology conveys much more information about the algorithm.

A fair comparison of algorithms While comparing the impact of various search algorithms, it is important to compare them in a fair context. Indeed, while comparing two algorithmic components, if they rely on parts implemented differently, the performance difference becomes less obvious. It is commonly admitted in AI/planning that many algorithmic ideas are sometimes put aside because of a combination of programming "tricks" often omitted in research papers. Such improvements can produce a 30 times improvement in search codes [BHLR12]. Such a generic framework allows us to provide at the same time a fair comparison and make these improvements widely accessible.

A simple implementation A generic framework also provides several (generic) examples of implementations for various search algorithms. Such may serve as a simple starting point to build more intricate algorithms that have been extensively tested. We may acknowledge the Coin-OR [Sal02], MiniSAT [SE05] and Mini CP [MSVH18] libraries/solvers that aim to provide simple (yet efficient) optimization methods.

A perspective to automatic algorithm building Many works develop methods to automatically tune algorithm hyper-parameters given a set of instances [HHLBS09]. Such methods can also be applied to build algorithms that provided a standardized search framework and many components (for instance, many bounds or guides). For instance, SATzilla [XHHLB08] has been produced using many algorithmic parts from many SAT contest participants. The resulting method outmatched other dedicated algorithms during a SAT competition. One may hope, probably in a not-so-near future, to find collaborative state-of-the-art algorithms, where researchers around the world contribute by providing components to a standard algorithmic component library. Although we are far from here with this framework, we believe that it is a small (yet indispensable) step towards this ideal.

2.1.1 Related Work

Many optimization frameworks and libraries exist (both for branch-and-bound or meta-heuristics). However, to the best of our knowledge, no existing framework proposes anytime

[1]And we are not even talking about bugs that are likely to happen. In the best case, it would only extend the development time if caught early. In the worst case, it can even divert a researcher from an efficient method (it nearly happened to us both on the EURO/ROADEF challenge and for the SOP). We discovered (too many) bugs in our specific implementations while making them generic.

[2]We are describing here the algorithm that enabled us to obtain new-best-so-far solutions on the SOPLIB. See Chapter 4 for more details about it.

tree search. We still notice that some efforts have been done in this direction as a generic branch-and-bound and recovering beam search have been implemented [Ter04]. We show in the next chapters that this framework allows us to quickly design and prototype some competitive ideas, and, sometimes improving state-of-the-art algorithms. This subsection presents some frameworks that facilitate the prototyping of both exact-methods or meta-heuristics.

Some generic exact-method frameworks exist. We may cite for instance Bob++ [DLCCR06] to implement parallel algorithms built on classical MIP solvers, BapCod [VST05] to implement branch-and-price algorithms and more recently, Coluna.jl that aims to re-implement BapCod in Julia. Many aim to provide a parallel branch-and-bound as PUBB (Parallelization Utility for Branch-and-Bound algorithms)[SFIH98], ALPS [XRLS05], PICO [EPH01], PPBB-lib [TP95], FATCOP [CFL01], ZRAM [Mar98], MW framework [GL06] to perform branch-and-bounds on grids and Symphony [RG05]. While the search parallelization is crucial and can prove to be a difficult task (which justifies the need of many frameworks as listed above), there is, up to our knowledge no tree search framework that allows to experiment various search strategies (for instance Beam Search, LDS, wA* *etc.*). CATS framework aims to fill this gap.

Many meta-heuristic generic frameworks exist, we refer the reader to [PRCLF12]. Most of them implement local-search based algorithms and evolutionary algorithms. Some of them consider constructive algorithms (namely GRASP and ACO that are implemented in FOM [PRG$^+$03], OAT [B$^+$07], MALLBA [AAB$^+$02]). We may also note that ParadiseEO [CMT04] is particularely used in meta-heuristics communities.

2.1.2 CATS vision

Our Combinator-based Anytime Tree Search Framework is built along with the following principles:

Standardization As anytime tree search formalism for discrete optimization is rather new, a lot of effort has been put to provide a simple and standard interface to build tree search algorithms. We chose terminologies so that they can be understood by researchers from meta-heuristics, AI/planning, and Operations Research while keeping ideas as simple as possible.

Extensible The framework is built in such a way that it is easy to add a new search tree or a new algorithm. Moreover, the combinator paradigm enables to even alter existing algorithms easily.

Performance Originally, the framework is an attempt to reproduce and generalize the code that enabled us to win the final phase EURO/ROADEF 2018 challenge. Thus, we kept a similar architecture in order to enumerate solutions fast and possibly parallelize algorithms on multiple CPUs. With simple enumeration schemes, the framework can open more than 1 million nodes per second. Moreover, we provide some data structures (for instance sets of integers) that are optimized for tree search. We aim to complete this list by the integration of various data structures found in the AI/planning literature. For instance we may cite weak-heapsort [ES02], QuickXsort [EW14], and various data structures found in the heuristic search textbook [ES11].

2.2 Towards a generic tree search

We discuss in this section a "standard" usage of the CATS framework. It consists of using pre-defined tree search strategies to solve a specific problem.

As discussed in Chapter 1, tree search components can be divided into two types of components:

Problem-specific parts: bounds, fathomings, guides, branching schemes (*i.e.* how to define the root node and children from a given node)

Generic parts: the search strategy (*e.g.* DFS, A* . . .), and also generic transformations (for instance a new guide using the problem-specific one and a greedy run from the node, or even generic dynamic-programming-like cuts)

The Combinator-based Anytime Tree Search framework is based on this principle. It defines a generic class (abstract in C++, and can be seen as an interface in Java), called Node, that acts as a contract between the problem specific parts and the generic parts. Problem specific parts inherit from Node and implement several standardized methods (getChildren, isGoal . . .). Generic parts rely on implemented Node methods to perform computations. The more one implements Node methods, the more generic parts become available.

For instance, if one only defines the root node (Node constructor) and how to generate children, only uninformed search methods would become available (namely DFS, BrFS). For instance, A* would not be available since it also requires some bounds. We discuss in Chapter 4 a usage example of the CATS framework.

2.3 Generic tree search modifications – the combinators

While providing several tree search strategies, the CATS framework is also designed to easily prototype tree search strategies and additional components (like generic dominance or symmetry breaking strategies). Consider the following scenario: You design several tree search algorithms (for instance DFS, LDS, Beam Search and several others). Then, you notice that you want to record some information about the search like the number of opened nodes, the average number of children per node, *etc.* The classical way to do so would be to integrate within each tree search algorithm recordings and possibly a boolean to indicate if we want to record information or not. This solution has some major drawbacks:

- It results in many code duplicates as it is likely that all these modifications would be relatively similar from one tree search algorithm to another. Even when using a finely designed inheriting scheme, there will still be some code duplicates making the code maintenance difficult.

- When designing a new modification, it will require to alter many other tree search algorithms (this amount of repetitive work would be more and more important as the number of algorithms grow). The same goes when someone wants to design a new tree search algorithm. It would require to integrate all already existing modifications which can become a tedious task as the framework expands.

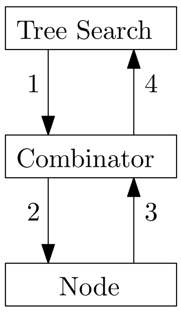


Figure 2.1: An example of how the combinator inserts between the tree search and another node.

As many algorithms found in the literature rely on generic modifications of a given generic scheme, it is important to provide them with a generic framework. We address the code maintenance issue by introducing the concept of combinators (terminology inspired from λ-calculus) that allows to define generic tree search modifications. A combinator (let us call it X) takes as a parameter a NODE object (it can be another combinator or a problem specific NODE), and mimics it while slightly altering its behavior. Consider the following cases:

- In this example, we call N a simple node without any combinator.
- $X(N)$ corresponds to N on which the combinator X is applied.
- $Y(X(N))$ corresponds to N where the combinators X and Y are applied successively.

Each combinator can alter every method of a tree search, including the constructor, bounds, the way it generates children . . .

Figure 2.1 illustrates how the combinator acts as an interface between a problem specific node and a generic tree search.

1. The tree search requests some node method.
2. The combinator "transfers" this request to the underlying node.
3. The underlying node responds.
4. The combinator responds to the tree search by possibly modifying the value the node provided.

This scheme allows us to design modular and extensible search components. Many combinators aim to be easily combined and offer a large degree of freedom. The next part of this chapter presents the different combinators we designed and implemented within the *Combinator-based Anytime Tree Search framework*. We believe that such an approach to building optimization algorithms is promising, and, to the best of our knowledge, new.

2.3.1 Limited Discrepancy Combinator

Limited Discrepancy Search (LDS) [HG95] can be seen as a DFS in which some nodes are fathomed if they deviate too much from the search guide. One can build an LDS-combinator(N, d) where N is a node (combinator or not) and d the maximum number of allowed discrepancies. It behaves as follows:

- Its constructor remembers the number of allowed discrepancies.
- While generating children, the combinator sorts children generated by N and wrap them within an LDS-combinator with an updated d value depending on their ranks. It cuts children with $d < 0$.

In the CATS framework, LDS is implemented by a DFS where an LDS-combinator is applied at the root node.

2.3.2 Dominance Combinator

In tree search, we define as "prefix of node N" the set of decisions taken from the root to obtain the node N. We define as "suffix of node N" the subtree rooted in N (set of nodes below N).

We define as "equivalent", two prefixes p_1, p_2 such that their suffixes are isomorphic (same structure, same bounds for each node in the suffix $\dots$).

It can be seen as a form of node merging and the resulting branch-and-bound explores a directed acyclic graph instead of a tree (it might be possible to find disjoint paths to go from the root to N).

Dominance prunings are a way to eliminate symmetries and dominated partial-solutions. It can be seen as a form of dynamic programming integrated within a tree search algorithm. It stores all explored sub-states. Each node compares its prefix subset and last vertex to existing entries in the database. If it is dominated, the node is pruned. This strategy has been used in a large variety of methods. For instance, memorization in branch and bounds [SJ15, STDC18].

Our implementation of generic dominances consists in altering the behavior of the branch and bound as follows: Each time a node n is opened, the prefix of n is compared to what exists in the database. If the subset of vertices spanned by n does not exist in the database it is added to it, otherwise it is compared to the best equivalent prefix found so far. If the subset has a prefix cost worse than the one in the database, the node n is pruned.

We implement the database using a hash table. In our numeric experiments, we notice that a branch and bound using the prefix equivalence opens on average 4 to 5 times less nodes on the largest instances than its equivalent version without prefix equivalence.

In some versions of the Prefix Equivalence (for instance the one found in history cuts [SJ15] or in branch-and-bounds-and-remember [STDC18] called memoization), nodes are pruned if their prefix matches an existing entry in the database even if their cost is equal. Notice that we restart tree search algorithms (*i.e.* in Iterative Beam Search and Limited Discrepancy), which perform heuristic prunings (they prune nodes to avoid saturating the memory and to ensure reaching feasible solutions). To allow our algorithms to close an instance (*i.e.* to prove the optimality of the best solution it found), we prune nodes only if they are *strictly* dominated by the best equivalent recorded in the database. The reason for doing so is that, although the value recorded in the database corresponds to a node that has been already explored, this exploration might have been partial and we need to ensure that the search does not perform any heuristic pruning to provide proof of optimality. For instance, running a beam search with parameter $d = 1$ (similar to a greedy algorithm) registers an entry in the database for the root node. At the second iteration, the entry would be used and prune the root node as it would have the same prefix value. That alleviates this scenario, we only prune a node if it is strictly dominated.

The Dominance-combinator(N, store) behaves as follows:

- Its constructor takes as a parameter a pointer (in C++ a reference) to a dominance-database.
- Before generating children, the dominance-combinator checks in the database if there exist an equivalent and better prefix. If there exists one, the current node is fathomed (*i.e.* no child is generated). If the current prefix is strictly better, the combinator updates the database to integrate the new better prefix.

Note: In this work, we assume that entries would not saturate the memory (which holds only for small runtimes). As we discuss In Chapter 4, small runtimes are enough to obtain new-best-known solution on the Sequential Ordering Problem (less than 600 seconds). Research has been done to investigate this issue [STDC18].

2.3.3 Probings

In anytime tree search, guide functions appear to be a key component. In some cases, this guide can be dramatically improved by executing a greedy algorithm and use the solution it found. Such additional guidance is costly (as it runs a greedy at each node), however many existing algorithms use it to improve the solutions. We may find for instance, the Pilot method [VFD05] which is a meta-heuristics that uses probings (called pilot methods) to guide the search. The Branch and Greed algorithm that performs a greedy algorithm and uses probing as guidance [SC99], is similar to the Pilot method. We may also recall that, various probings also appear in MIP-based branch-and-bounds or branch-and-price methods [Ach09, SVP$^+$19].

We define the Probing-Combinator(N, greedy) as follows:

- Its constructor takes as a parameter the underlying node N, and a search algorithm (usually a simple greedy algorithm, however, we may use any other anytime tree search as a probing strategy provided that it terminates relatively quickly)

Branch-and-Greed/pilot method extensions As Branch-and-Greed and the Pilot method are commonly used and known to perform well in various situations, one can propose several variants by altering the "pilot" method (*i.e.* the greedy algorithm ran at each node). The CATS framework allows rapid and native prototyping of such methods by

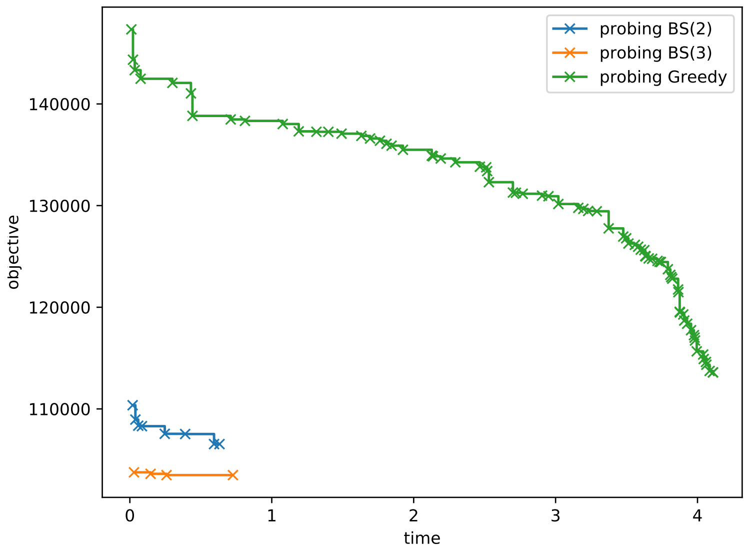


Figure 2.2: Comparison between the branch and greed algorithm (greedy + probing using a greedy algorithm) with (greedy + probing using a Beam Search of width 2 and 3)

using another tree search than the greedy. We tried such an algorithm and present quick results on a hard instance of the SOP (more on this problem in Chapter 4) in Figure 2.2. It appears that on this specific case, such variant of branch-and-greed using a beam search of width 2 or 3 performs much better than the original branch-and-greed/pilot method. Although this result is somewhat biased, as we prove in Chapter 4 that Beam Search outperforms existing state-of-the-art for the SOPLIB, it exhibits an interesting question about the efficiency of such variations on situations where classical pilot method/branch-and-greed are used.

2.3.4 Statistics Combinator

The previous combinators altered the search behavior and were used to build other existing (or sometimes new) algorithms. We now present a different combinator that aims to provide useful information and feedback about the search without altering its behavior from a search point of view. The Statistics Combinator does not perform any action other than recording events. For instance, at each time GETCHILDREN is called, it updates the average branching factor depending on the number of children generated. It records multiple search events in a SEARCHSTATSSTORE. Using a combinator for statistics recording allows us to keep the manager and the tree search simpler. Also, it is easy to remove it when seeking optimal performance (However, we advise to use it as it hardly costs 20% of performance and yields useful insights).

In its current version, it records the following information:

- Number of expanded nodes (*i.e.* number of nodes whose children were generated).
- Number of generated nodes (*i.e.* number of nodes generated, expanded or not).

- Number of bounds and guides calls.
- Average number of children.
- Performance profiles (For instance as Figure 2.2). It registers the best solution known at a given search time.
- Node opening events. While activating this option, the StatsCombinator registers all node opening events. Such information can later be used to visualize the search tree.

2.4 Implemented algorithms

So far, the following algorithms have been implemented:

- A* [HNR68]
- Ant Colony Optimization [DMC91] (AS, EAS, RAS, MMAS, ACS variants)
- Anytime Column Search [VGAC12]
- Anytime Pack Search [VAC16]
- Beam Search [OM88] / Iterative Beam Search (version presented in Chapter 1)
- BrFS
- Branch and Greed [SC99]
- BULB [FK05]
- DFS
- Greedy
- GreedyRandom (proportionnality rule)
- MBA* / Iterative MBA*
- LDS [HG95]
- wA* [Poh70]

3 A tree search for the EURO/ROADEF 2018 challenge

This chapter presents the first application of anytime tree search of this thesis. We applied tree search algorithms on the EURO/ROADEF glass cutting challenge and the resulting method was ranked first among 64 participants. We further generalized this approach to a large variety of cutting & packing problems and obtained competitive results on most of them. Sometimes even obtaining new best-so-far solutions. This chapter is a collaboration with *Florian Fontan*. This chapter is inspired from an article about the algorithm we designed for the challenge [LF20a].

Contents

3.1 Introduction

We propose an anytime tree search with some simple bounds, pseudo-dominance properties, and symmetry breaking rules. We introduce some new guidance strategy that allows the algorithm to perform significantly better than if it was guided by a bound as in classical branch-and-bound methods. We may note that this is unusual as almost all top-ranked methods in previous editions of the challenge mainly rely on local search or mathematical programming techniques. However, it is worth noticing that tree search algorithms are popular in the *Cutting & Packing* literature. Thus, it is common to find various beam-search techniques to solve this kind of problem [AHM09a, BS10, BCPT14]. Such remarks make anytime tree search algorithms an apriori good bet, while aiming to provide an efficient method for the EURO/ROADEF 2018 challenge, and more generally on *Cutting & Packing* problems.

The search strategy can be roughly described as follows. It is a restarting strategy that starts its first iteration by performing very aggressive heuristic prunings. At the second iteration, it performs less aggressive heuristic prunings, taking more time than the previous iteration, but finding better solutions. If the algorithm runs long enough, some iteration may perform no heuristic pruning, thus the method will be able to guarantee optimality. The resulting method obtained the best results compared to the other submitted approaches during the final phase. We named it *Memory Bounded A** as it performs a series of A* with heuristic prunings which guarantee no-more than a given amount of nodes active at the same time.

We also highlight a general methodology that can be applied to other complex problems (and with other tree search algorithms). Indeed, the method can be divided into two parts: the *Branching Scheme*, usually problem-specific, which is a definition of the implicit search tree (*i.e.* root node, how to generate children of a given node, lower bounds, dominance rules, *etc.*); and a strategy, usually generic, to explore the tree. This decomposition allows rapid prototyping of both search tree definitions and tree search algorithms as many generic parts can be reused within other algorithms. It also helps to draw insights about the contribution of each component to the resulting search algorithm.

This chapter is structured as follows. In Section 3.2, we state the problem constraints and objective. In Section 3.3, we give some notations and definitions. In Section 3.4, we describe the branching scheme and in Section 3.5, the tree search algorithm we designed. Finally, in Section 3.6, we show the numerical results we obtained.

3.2 Problem description

The 2018 ROADEF/EURO challenge was dedicated to an industrial cutting problem from the French company *Saint-Gobain*. The challenge consists in packing rectangular glass items into standardized bins of dimensions $W \times H$ (6m $\times$ 3.21m).

The cutting plan needs to satisfy the following constraints:

- All items need to be produced
- Item rotation is allowed
- Cuts must be of guillotine type. Figure 3.1 illustrates two examples of non-guillotine and guillotine patterns. Furthermore, the number of stages (levels of cuts) is lim-

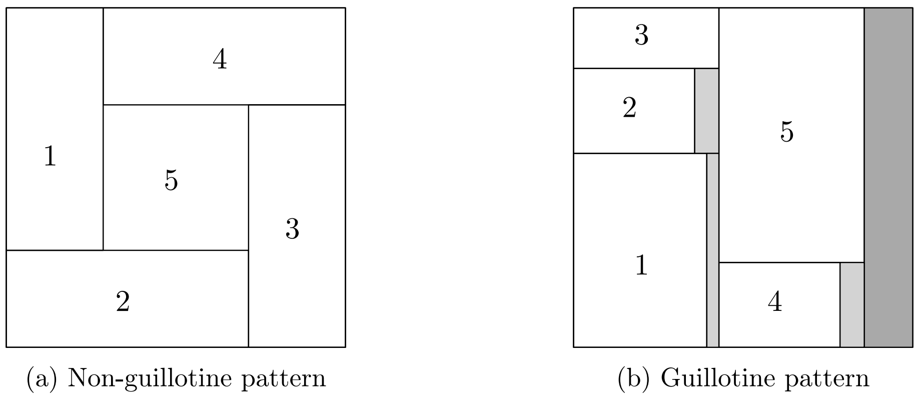


Figure 3.1: Illustration of a non-guillotine pattern (a) and a guillotine one (b)

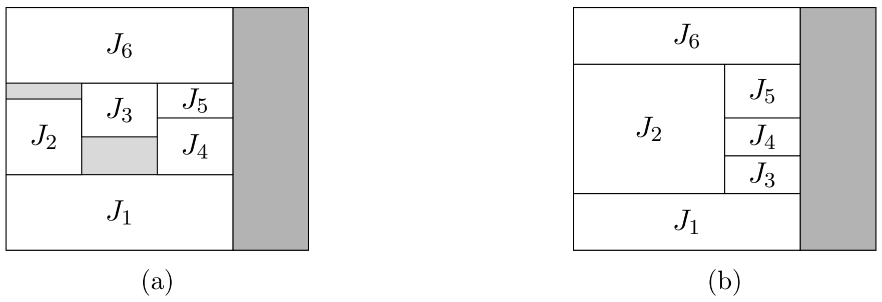


Figure 3.2: Only one 4-cut is allowed. Therefore, pattern (a) is feasible but pattern (b) is not

ited to four, with only one 4-cut allowed on a sub-plate obtained after 3-cuts. This configuration is close to classical three-staged non-exact guillotine patterns, but differs in that a sub-plate obtained after 3-cuts may contain two items as illustrated in Figure 3.2.

- Items are subject to chain precedence constraints. The extraction order is as follows: rightmost first level sub-plates first; within a first level sub-plate, bottommost second level sub-plates first; within a second level sub-plate, rightmost items first; and within a third level sub-plate, bottommost item first. Most instances have a dozen chains, three instances have 2 chains and five instances are not subject to precedence constraints.

- Bins contain defects (between 0 and 8 rectangles about a few centimeters high and wide). Items must be defect-free and it is forbidden to cut through a defect. Even if the bins have the same dimensions, the presence of defects makes the set of bins heterogenous. It is important to note that bins must be used in the order they are given.

- Depending on their level, sub-plates are subject to minimum and maximum size constraints. The width of first level sub-plates must lie between $w^1_{\text{min}} = 100$ and $w^1_{\text{max}} = 3500$, except for wastes. The height of second-level sub-plates must be at least $w^2_{\text{min}} = 100$, except for wastes. Finally, the width and the height of any waste

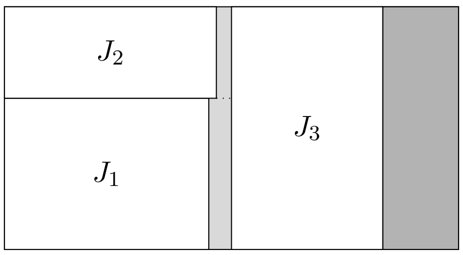


Figure 3.3: Optimal solution of the case containing the following three items with the chain precedence constraint $J_1 \to J_2 \to J_3$. Additional waste must be added before the first 1-cut. Otherwise either the waste area to the right of J_1 or the waste area to the right of J_2 would violate the minimum waste constraint.

area must be at least $w_{\min} = 20$. This last constraint has an unusual consequence as illustrated in Figure 3.3.

The objective is to minimize the total waste area. It differs from classical Bin Packing Problems in that the remaining part of the last bin is not counted as waste. This objective is known in the packing literature as Bin Packing with Leftovers. It can be formulated as:

$$\min \quad nHW - Hw - \sum_{i \in \mathcal{I}} w_i h_i$$

where n is the number of bins used; W and H are respectively the standardized width and height of the bins; w is the position of the last 1-cut; $\mathcal{I}$ is the set of produced items; and w_i and h_i are respectively the width and the height of item $i \in \mathcal{I}$.

3.3 Definitions and notations

We use the following vocabulary: a k-cut is a cut performed in the k-th stage. Cuts separate bins or sub-plates in k-th level sub-plates. For example, 1-cuts separate the bin in several first level sub-plates. S denotes a solution or a node in the search tree.

We call the last first level sub-plate, the rightmost one containing an item; the last second level sub-plate, the topmost one containing an item in the last first level sub-plate; and the last third level sub-plate the rightmost one containing an item in the last second level sub-plate. $x_1^{\text{prev}}(S)$ and $x_1^{\text{curr}}(S)$ are the left and right coordinates of the last first level sub-plate; $y_2^{\text{prev}}(S)$ and $y_2^{\text{curr}}(S)$ are the bottom and top coordinates of the last second level sub-plate; and $x_3^{\text{prev}}(S)$ and $x_3^{\text{curr}}(S)$ are the left and right coordinates of the last third level sub-plate. Figure 3.4 presents a usage example of these definitions. We define the area and the waste of a solution S as follows:

To compute area(S) we distinguish two cases

- if S contains all items:
$$\text{area}(S) = x_1^{\text{curr}}(S)h$$
- and otherwise:
$$\begin{aligned}\text{area}(S) = A \quad &+ \quad x_1^{\text{prev}}(S)h \\ &+ \quad (x_1^{\text{curr}}(S) - x_1^{\text{prev}}(S))y_2^{\text{prev}}(S) \\ &+ \quad (x_3^{\text{curr}}(S) - x_1^{\text{prev}}(S))(y_2^{\text{curr}}(S) - y_2^{\text{prev}}(S))\end{aligned}$$

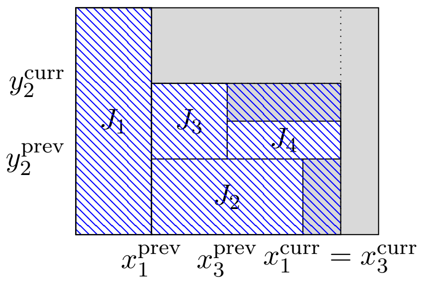


Figure 3.4: Last bin of a solution which does not contain all items. The area is the whole hatched part and the waste in the grey hatched part.

We compute the waste of a partial solution as follows:

$$\text{waste}(S) = \text{area}(S) - \text{item\_area}(S)$$

with A the sum of the areas of all but the last bin, h the height of the last bin and item_area(S) the sum of the area of the items of S. Area and waste are illustrated in Figure 3.4.

3.4 Branching scheme

3.4.1 General scheme

Two kinds of packing strategies are used in the packing literature: item-based and block-based. In item-based strategies, only one item is inserted at each step, whereas in block-based strategies, multiple items are inserted. Although several researchers highlighted the benefits of block-based approaches [BJ12, WTZL14, LMP17], we chose an item-based strategy. Two reasons support this choice. First, the problem has more constraints than classical packing problems from the literature. Thus, generating feasible solutions is already challenging and block-based approaches add even more complexity. Second, the benefits of the block-based approaches might be compensated by a more powerful tree search algorithm.

However, our strategy is not purely item-based: instead of packing one item at each step, we pack the next third level sub-plate. This comes from the observation that because only one 4-cut is allowed in a third level sub-plate, a third level sub-plate has only five possible configurations; it may contain:

1. exactly one item, without waste
2. exactly one item with some waste above
3. exactly one item with some waste below
4. exactly two items, without waste
5. no item, only waste

These configurations are illustrated in Figure 3.5. The sub-plates containing J_1 and J_2 respectively follow configurations 1 and 2. These are the *standard* configurations. Placing

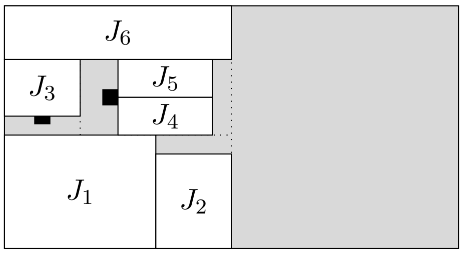


Figure 3.5: Illustration of third level sub-plate possible configurations. Black rectangles are defects.

an item on top of the sub-plate as in configuration 3 may be necessary to reach an optimal solution (by a combination of a defect and a min-waste constraint). Similarly, inserting only waste (configuration 5) may also be necessary if the region contains a defect as the sub-plate containing the second defect. We do not allow directly inserting only waste in a region containing no defects. Such sub-plate may appear in a solution, as the third-level sub-plate to the right of J_4 and J_5, but it is implicitly generated when J_6 is inserted. Finally, the sub-plate containing items J_4 and J_5 corresponds to configuration 4.

Third level sub-plates are inserted in the order they are extracted. In Figure 3.5, this follows the numbering of the items. This ensures to never violate the precedence constraints. All items are candidates if their insertion does not lead to a precedence constraint violation.

Then, a third level sub-plate can be inserted at several depths:

- depth 0: in a new bin,
- depth 1: in a new first level sub-plate to the right of the current one,
- depth 2: in a new second-level sub-plate above the current one,
- depth 3: in the current second-level sub-plate already, to the right of the last third-level sub-plate

To reduce the size of the tree, we apply some simple pruning rules:

- if a third-level sub-plate can be inserted in the current bin, we do not consider insertions in a new bin; and if a third level sub-plate can be inserted in the current first (resp. second) level sub-plate without increasing the position of its left 1-cut (resp. top 2-cut), we do not consider insertions in a new first (resp. second) level sub-plate;
- If the last insertion is an empty sub-plate at depth d, then the next insertion must also happen at depth d;
- If the last insertion is a 2-item insertion at depth $d \neq 3$, then the next insertion must be at depth 3.

With this branching scheme, item rotation and minimum and maximum distances between cuts constraints are easy to take into account.

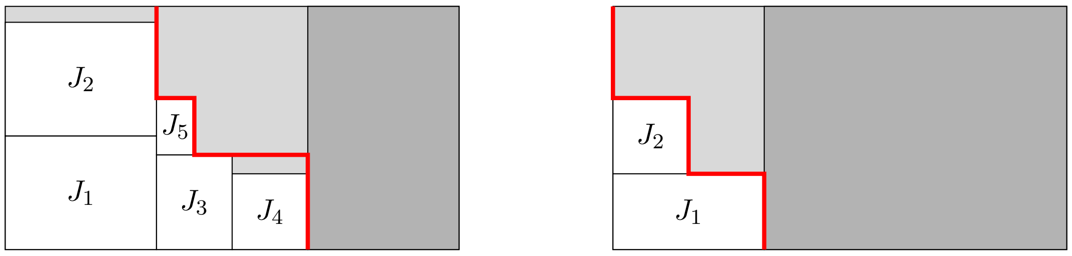


Figure 3.6: Illustration of the front of two partial solutions

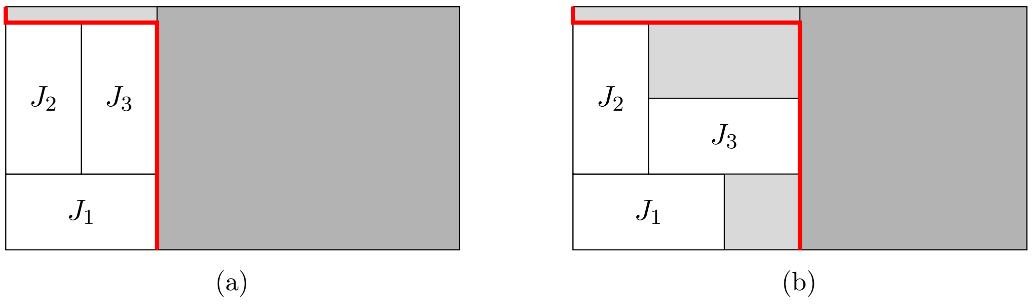


Figure 3.7: Solution (a) dominates solution (b)

3.4.2 Pseudo-dominance rule

In this section, we describe a more sophisticated heuristic dominance rule. For a (partial) solution, we define its *front* as the polygonal chain

$$((x_1^{\text{curr}}, 0), (x_1^{\text{curr}}, y_2^{\text{prev}}), (x_3^{\text{curr}}, y_2^{\text{prev}}),$$
$$(x_3^{\text{curr}}, y_2^{\text{curr}}), (x_1^{\text{curr}}, y_2^{\text{curr}}), (x_1^{\text{curr}}, h))$$

Figure 3.6 shows two examples of solution fronts.

Then we say that solution S_1 dominates solution S_2 iff they contain the same items and the front of S_1 is *before* the front of S_2. (see Figure 3.7).

If the number of possible subsets of items is small, then for a given subset, we can memorize the best front currently seen during the search and prune any new dominated node encountered. This situation occurs in instances with strong precedence constraints (*i.e.* two chains) and this is the strategy of the DPA* algorithm presented afterward. However, for most instances, the number of possible subsets is too large and we only use the pseudo-dominance rule among the children of a node. To compensate, an additional symmetry breaking strategy is introduced.

3.4.3 Symmetry breaking strategy

We designed the following symmetry breaking strategy: if they do not contain defects and can be exchanged without violating the precedence constraints, a k-level sub-plate is forbidden to contain an item with a smaller index than the previous k level sub-plate of the same $(k-1)$-level sub-plate.

Preliminary experiments showed that applying the strategy for $k = 2$ and $k = 3$ yield the best results. The symmetry breaking strategy is illustrated in Figure 3.8.

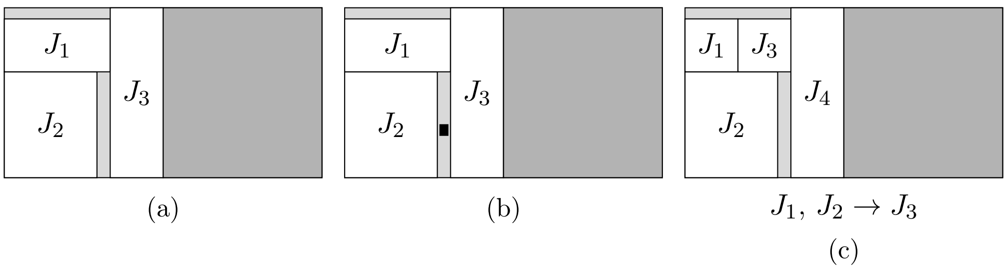


Figure 3.8: Illustration of the symmetry breaking strategy: pattern (a) is forbidden because the second-level sub-plates containing J_1 and J_2 can be exchanged without a feasibility issue. However, pattern (b) is allowed because of the defect and pattern (c) is also allowed because if the second-level sub-plates are exchanged, then the precedence constraint between J_2 and J_3 is violated.

It should be noted that the branching scheme is not dominant, *i.e.* for some instances, it may not contain an optimal solution. Likewise, the pseudo-dominance rule considers that solution S_1 dominates solution S_2 whereas no optimal solution can be reached from S_1 but one can be from S_2. More details about this are given by [Fon19].

3.5 Tree search

During our initial work on the challenge, we first explored the classical *"Operations Research"* optimization algorithms (local-search, evolutionary algorithms and branch and bounds). However, it seemed difficult for us to find efficient local-search or evolutionary moves, while it felt relatively natural to design constructive methods. We implemented several classical constructive algorithms: a greedy algorithm quickly providing solutions but with limited quality; a Best First (A*) algorithm returning the "optimal" one (relatively to the branching scheme) on small instances; and a Depth First struggling to improve the greedy solution.

At each iteration, the *best* node is extracted from the fringe and its children are added to the fringe. As written above, our implementation of A* can find the "optimal" solutions for very small instances but fails to provide one in an acceptable time and runs out of memory quickly. Therefore, we decided to heuristically prune nodes to bound the required memory. This "heuristic" algorithm performed beyond expectations and provided excellent solutions. However, it depended on the amount of memory allowed for the fringe. If this parameter is too small, the search ends quickly and does not benefit from the remaining available time. If too big, the search takes more time and does not provide any solution within the time limit. To get rid of this parameter, we chose to use a restart strategy where we geometrically increase the allowed memory at each restart. The new parameter to calibrate becomes the growth factor, but we found that any value between 1.25 and 3 provided similar results. This simple approach provided good solutions.

Motivated by this simple but yet efficient algorithm, we investigated other anytime tree search algorithms such as beam search [OM88] and beam stack search [ZH05]. We implemented and compared them on the challenge problem. To our surprise, they did not perform as well as the previously described approach. To the best of our knowledge, this approach has not been used in the Operations Research literature before. We describe it in more detail in the next section.

3.5.1 Memory Bounded A* (MBA*)

A* is known to minimize the cost estimate on nodes it opens. However, it suffers from a large memory requirement since it has to store a large number of nodes in the fringe. We propose a simple but yet powerful heuristic variant of A* that cuts less promising nodes if the size of the fringe goes over a parameter D. We call this tree search algorithm *Memory Bounded A* (MBA*)*. If $D = 1$, it generalizes a greedy algorithm and if $D = \infty$, it generalizes A*.

We recall the MBA* pseudo-code in Algorithm 3.1. Only lines 8 to 11 are added compared to the A* algorithm. MBA* starts with a fringe containing only the root node (line 1). At each iteration, the *best* node is extracted from the queue (lines 3 and 4) and its children are added to the queue (lines 5 to 7). If the size of the queue goes over D, the *worst* nodes are discarded (lines 8 to 11).

Algorithm 3.1: Memory Bounded A* (MBA*)

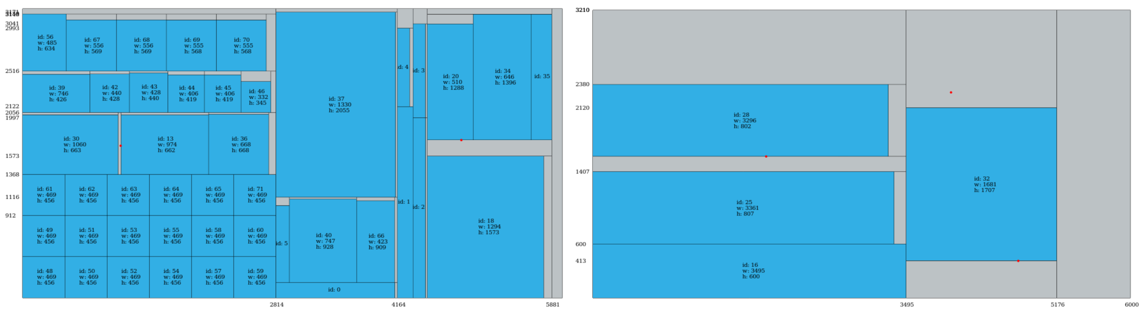
```
1  fringe ← {root};
2  while fringe ≠ ∅ and time < timelimit do
3      $n \leftarrow extractBest(\text{fringe})$;
4      fringe ← fringe \ {$n$};
5      forall $v \in neighbours(n)$ do
6          fringe ← fringe ∪ {$v$};
7      end
8      while |fringe| > $D$ do
9          $n \leftarrow extractWorst(\text{fringe})$;
10         fringe ← fringe \ {$n$};
11     end
12 end
```

3.5.2 Guide functions

Tree search methods are dependent on well-crafted guide functions which define the meaning of *best* and *worst* nodes. Using a lower bound is common in the tree search literature. Indeed, if the objective is to prove optimality, using a lower bound as a guide function will minimize the number of opened nodes. Therefore, we first tried this approach and used the waste as a guide function. We noticed that the resulting solutions packed small items on the first plates and big items on the last ones, thus generating little waste at the beginning but a lot at the end. Globally: the solution quality was not satisfactory as illustrated in Figure 3.9a.

Taking this into account, we designed new guides to balance the cost of inserting small items at the beginning of the solutions:

waste percentage (= waste / (waste + item area)):
compared to waste only, the waste has less impact if the solution contains larger items.


(a) waste only guide – first and last plate

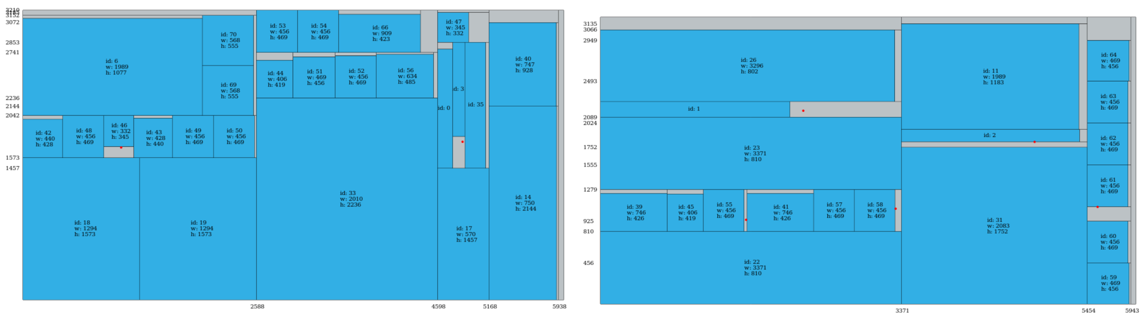

(b) waste / average size guide – first and last plate

Figure 3.9: 3.9a shows a solution obtained using the waste as guide function. Notice that at the beginning of the solution, small items are omnipresent whereas in later plates, only large items are present, thus globally generating more waste. 3.9b shows the effect of the guide biased by the item average size on a solution of the same instance. We observe that small and large items are better mixed and significantly less waste is generated at the end of the solution.

waste percentage / average surface of packed items:
this guide function directly adds a reward to solutions containing large items. Indeed, the average surface of packed items directly favors partial solutions with big items first.

The benefit of these guides is illustrated in Figure 3.9b.

3.5.3 DPA*: solving instances with strong precedence constraints

Three instances of the challenge contain only 2 precedence chains. If we denote by n_1 (resp. n_2) the length of the first (resp. second) chain, then the number of possible subsets of items packed in a partial solution of the branching scheme becomes $n_1 n_2$. Since the number of items in an instance is less than 700 (this information was given in the challenge description), it becomes possible to store the non-dominated fronts encountered for each possible subset without overcoming the memory limitation, compare the front of each opened node with the non-dominated fronts from all the previously encountered nodes and prune the dominated ones.

Therefore, we developed a dedicated algorithm for these instances named Dynamic Programming A* (DPA*). DPA* is an A* algorithm implementing the scheme described in the previous paragraph. DPA* does not bound the size of the queue as MBA* does, and uses the waste as a guide function. Therefore, if it terminates, it returns the "optimal" solution (relatively to the branching scheme and the pseudo-dominance rule).

We may note that for a given subset of items, there could be an exponential number of non-dominated fronts, which could degrade DPA* performances. This is at least not an issue for the concerned instances from the challenge.

3.5.4 Global algorithm

For the competition, we distinguished the case where the instance has two chains or less. In this case, we run DPA*.

If it has strictly more than two chains, we do not use DPA* since it would overcome the memory limitation. Since the processor used to evaluate the participant submissions had 4 physical cores, we run 4 threads, each one running a restarting MBA* with a given growth factor and a given guide function. Each MBA* is initially executed with a fringe maximal size of 2, and each time one terminates, it is restarted with a maximal fringe size multiplied by its growth factor. If the growth factor is 2, the maximal size doubles at each iteration. All the threads share the information of the best solution found. If one finds a better solution, the others take advantage of it to perform more cuts and globally perform better together than alone. The threads run the following algorithms:

- MBA*, waste percentage guide, growth factor 1.33
- MBA*, waste percentage guide, growth factor 1.5
- MBA*, waste percentage / average size guide, growth factor 1.33
- MBA*, waste percentage / average size guide, growth factor 1.5

3.6 Numerical results

In this section, we first evaluate the contribution of the components we described in the previous sections in the main algorithm. Then, we show the benefits of using DPA* on instances with only two precedence chains. Finally, we provide computational results with the challenge setting. Instances generally have between 300 and 600 items and 10 to 15 chains. They are available online[1].

3.6.1 Contribution of the components

In this section, computational experiments have been performed on a personal computer with an Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz with 16GB RAM.

We consider the 12 possible combinations of the components we designed, namely MBA* to be compared with an Iterative Beam Search [Zha98]; with or without the symmetry breaking strategy; and with waste (w), waste percentage (p), or waste percentage/average size (a) guide. We run each pair of instance-algorithm for 100 seconds.

[1] `https://www.roadef.org/challenge/2018/en/instances.php`

Combination	# best	# only best
BS+no-sym+w	2	0
BS+no-sym+p	2	0
BS+no-sym+a	2	0
BS+sym+w	5	1
BS+sym+p	9	4
BS+sym+a	9	4
MBA*+no-sym+w	2	0
MBA*+no-sym+p	3	0
MBA*+no-sym+a	4	0
MBA*+sym+w	3	0
MBA*+sym+p	23	17
MBA*+sym+a	22	18

Table 3.1: Comparison over all possible algorithms using the proposed algorithmic components. "# best" indicate the number of times the algorithm was able to find the best solution on given instances compared to the 11 other algorithms. "# only best" indicates the number of times the algorithm was the only one to find the best solution.

In Table 3.1, we show a summary of the performances of each variant. The goal is to help us selecting the best combinations to use. MBA* with the symmetry breaking strategy and guided by the waste percentage (*MBA*+sym+p*) and MBA* with the symmetry breaking strategy and guided by the waste percentage / average size (*MBA*+sym+a*) clearly outperform all other combinations. That is why these are the two combinations that we use. Since the processor used to evaluate participant submissions has 4 physical cores, we dedicate two threads for each combination with different growth factor (1.33 and 1.5) to add some robustness.

Instance	MBA* (4 threads, 3600s)	DPA*
B5	88 590 815	72 155 615 (2.06s)
X8	24 875 331	22 265 601 (59.12s)

Table 3.2: DPA* vs MBA*

Table 3.3 presents an analysis of the contribution of each component individually. Each column corresponds to the best result per instance obtained by a subset of algorithms that uses a given component. For instance *best IBS* corresponds to a subset of algorithms using Iterative Beam Search, thus excluding MBA* (6 algorithms). MBA* variants outperform the Iterative Beam Search variants (producing 12% less waste). It finds 41/50 best solutions and 36 best solutions that the Beam Search variants were not able to obtain. The algorithms using the symmetry breaking strategy clearly produce better results than the one without (13% less waste and 46 best solutions not attainable by the variants without the symmetry breaking strategy) showing that integrating state-space reductions can greatly benefit to anytime tree search algorithms and probably even to constructive metaheuristics. Finally, as expected, the waste (lower bound) guide provides the worst results

Instance	best IBS	best MBA*	best no sym	best with sym	best w	best p	best a
A1	**425 486**	**425 486**	**425 486**	**425 486**	**425 486**	**425 486**	**425 486**
A2	10 514 609	**9 676 799**	10 537 079	**9 676 799**	10 659 059	10 418 309	**9 676 799**
A3	**2 651 880**	**2 651 880**	3 441 540	**2 651 880**	3 056 340	**2 651 880**	**2 651 880**
A4	3 242 520	**3 220 050**	3 306 720	**3 220 050**	3 505 740	**3 220 050**	3 306 720
A5	**3 033 273**	3 566 133	4 856 553	**3 033 273**	3 736 263	**3 033 273**	3 566 133
A6	**3 225 930**	3 572 610	3 652 860	**3 225 930**	3 800 520	**3 225 930**	3 460 260
A7	5 063 280	**4 800 060**	5 194 890	**4 800 060**	5 933 190	**4 800 060**	4 938 090
A8	**9 187 874**	10 077 044	12 218 114	**9 187 874**	12 568 004	10 077 044	**9 187 874**
A9	**2 930 706**	2 985 276	3 550 236	**2 930 706**	3 929 016	**2 930 706**	2 985 276
A10	4 097 221	**4 084 381**	5 272 081	**4 084 381**	4 797 001	**4 084 381**	4 122 901
A11	**4 718 449**	4 978 459	6 076 279	**4 718 449**	6 711 859	5 251 309	**4 718 449**
A12	**2 050 084**	2 245 894	2 342 194	**2 050 084**	**2 050 084**	2 104 654	2 194 534
A13	15 096 453	**12 133 623**	14 865 333	**12 133 623**	15 099 663	12 197 823	**12 133 623**
A14	14 363 778	**12 097 518**	14 793 918	**12 097 518**	14 363 778	**12 097 518**	13 490 658
A15	15 277 961	**13 185 041**	15 168 821	**13 185 041**	16 029 101	**13 185 041**	15 014 741
A16	**3 380 333**	**3 380 333**	**3 380 333**	**3 380 333**	**3 380 333**	**3 380 333**	**3 380 333**
A17	**3 617 251**	**3 617 251**	**3 617 251**	**3 617 251**	**3 617 251**	**3 617 251**	**3 617 251**
A18	5 898 468	**5 535 738**	5 763 648	**5 535 738**	7 737 798	5 596 728	**5 535 738**
A19	**3 323 744**	3 654 374	4 187 234	**3 323 744**	4 620 584	**3 323 744**	3 965 744
A20	**1 467 925**	**1 467 925**	1 493 605	**1 467 925**	**1 467 925**	**1 467 925**	**1 467 925**
B1	4 173 228	**3 633 948**	4 150 758	**3 633 948**	4 012 728	4 324 098	**3 633 948**
B2	15 715 685	**15 359 375**	18 466 655	**15 359 375**	20 155 115	**15 359 375**	15 715 685
B3	32 668 193	**21 253 433**	23 365 613	**21 253 433**	41 315 933	24 890 363	**21 253 433**
B4	8 885 365	**8 862 895**	11 238 295	**8 862 895**	9 222 415	**8 862 895**	8 920 675
B5	92 433 185	**88 590 815**	**88 590 815**	**88 590 815**	103 719 545	**88 590 815**	**88 590 815**
B6	**13 371 637**	13 480 777	15 653 947	**13 371 637**	17 509 327	14 113 147	**13 371 637**
B7	14 576 799	**11 434 209**	12 801 669	**11 434 209**	14 319 999	**11 434 209**	12 801 669
B8	24 490 999	**19 512 289**	24 121 849	**19 512 289**	24 490 999	**19 512 289**	20 048 359
B9	20 511 607	**20 046 157**	25 085 857	**20 046 157**	46 721 257	39 071 827	**20 046 157**
B10	28 012 013	**27 344 333**	29 225 393	**27 344 333**	35 815 523	31 055 093	**27 344 333**
B11	38 143 250	**29 113 520**	34 175 690	**29 113 520**	41 523 380	32 589 950	**29 113 520**
B12	18 122 077	**16 086 937**	19 929 307	**16 086 937**	18 122 077	**16 086 937**	16 314 847
B13	31 138 545	**29 674 785**	33 716 175	**29 674 785**	31 138 545	32 213 895	**29 674 785**
B14	10 482 820	**10 043 050**	11 619 160	**10 043 050**	12 046 090	10 434 670	**10 043 050**
B15	41 533 241	**28 372 241**	34 143 821	**28 372 241**	41 533 241	**28 372 241**	31 466 681
X1	21 022 877	**17 299 277**	17 970 167	**17 299 277**	29 911 367	17 803 247	**17 299 277**
X2	11 459 837	**8 583 677**	8 923 937	**8 583 677**	9 318 767	9 206 417	**8 583 677**
X3	9 424 756	**8 712 136**	9 842 056	**8 712 136**	9 578 836	**8 712 136**	8 927 206
X4	19 035 422	**15 976 292**	19 305 062	**15 976 292**	19 035 422	**15 976 292**	16 772 372
X5	**5 383 037**	5 620 577	7 029 767	**5 383 037**	6 728 027	5 623 787	**5 383 037**
X6	14 443 523	**12 167 633**	14 488 463	**12 167 633**	14 443 523	13 024 703	**12 167 633**
X7	30 327 120	**26 170 170**	27 146 010	**26 170 170**	31 328 640	29 161 890	**26 170 170**
X8	27 693 711	**27 109 491**	27 693 711	**27 109 491**	27 693 711	27 494 691	**27 109 491**
X9	33 431 655	**23 599 425**	33 919 575	**23 599 425**	33 370 665	**23 599 425**	26 716 335
X10	23 400 722	**19 901 822**	23 522 702	**19 901 822**	23 673 572	23 975 312	**19 901 822**
X11	14 349 972	**14 247 252**	16 102 632	**14 247 252**	14 349 972	**14 247 252**	14 921 352
X12	14 775 805	**12 422 875**	14 576 785	**12 422 875**	14 775 805	**12 422 875**	12 589 795
X13	19 208 322	**14 624 442**	18 900 162	**14 624 442**	20 007 612	16 271 172	**14 624 442**
X14	11 075 552	**9 730 562**	11 916 572	**9 730 562**	11 004 932	**9 730 562**	10 128 602
X15	16 301 394	**13 540 794**	17 261 184	**13 540 794**	16 461 894	13 990 194	**13 540 794**
total waste	779 159 574	679 871 064	779 027 964	676 914 654	870 817 914	725 241 204	693 016 014
nb best	14	41	4	50	5	27	28
nb only best	9	36	0	46	1	21	22

Table 3.3: Analysis of the contribution of each introduced algorithmic component

among the 3 considered guides (16% more waste than the waste percentage guide and 20% more waste than the waste percentage / average size guide and only 5 best solutions on 50 instances). However, the waste percentage guide and the waste percentage / average size guide provided similar results (with a slight advantage on the latest as it produces 5% less waste and finds one best solution more). These results are interesting as they show that both guides are complementary. Indeed, they produce 21 (resp. 22) best solutions where they are the only one to obtain them. Thus it is worth using both.

3.6.2 DPA*

Table 3.2 shows the benefits of using DPA* on instances with only two precedence chains. There is one such instance in each dataset, but the one in dataset A is trivial to solve, therefore we only consider instances B5 and X8. Unlike MBA*, DPA* is not anytime and terminates long before the 3600 seconds time limit. Furthermore, the solutions it returns are significantly better and are even the best-known solutions for both instances.

3.6.3 Final results

Table 3.4 sums up the challenge final results. Computational experiments have been performed on a computer with an Intel Core i7-4790 CPU @ 3.60 GHz $\times$ 8 processor with 31.3 Go of RAM. This configuration is similar to the one of the challenge. Since the challenge, a few adjustments have been made. Therefore, the results presented here slightly differ from the results obtained during the final phase. Compared to the challenge version, the current version performs better: the total waste on dataset B and X decreases from $493,600,549$ for the challenge version to $469,910,749$ for the current one. Columns *Final phase best 180s* and *Final phase best 3600s* contain the best solutions found during the final phase. Results annotated with a star indicate that this solution was found by our algorithm during the final phase of the challenge. The *Best known* column contains the best solution up to our knowledge. They may have been found during the development of the algorithm, with execution times exceeding 3600 seconds or by other teams. Finally, even if it is not indicated in the table, on most of the instances, if the algorithm is run longer, for example, 2 hours, the solution will still be improved.

Instance	Comments	Final phase best 180s	MBA*/DPA* 180s	Final phase best 3600s	MBA*/DPA* 3600s	Best known
A1	Trivial	-	425 486	-	425 486	425 486
A2	No prec	-	9 506 669	-	4 383 509	4 383 509
A3		-	2 651 880	-	2 651 880	2 651 880
A4		-	3 024 240	-	2 924 730	2 924 730
A5		-	2 924 730	-	3 283 653	3 017 223
A6		-	3 389 640	-	3 225 930	3 188 646
A7		-	4 703 760	-	4 334 610	3 920 520
A8		-	9 691 844	-	8 378 954	8 378 954
A9		-	2 664 276	-	2 664 276	2 664 276
A10		-	4 084 381	-	4 084 381	4 084 381
A11		-	4 660 669	-	4 622 149	4 358 929
A12		-	2 056 504	-	1 879 954	1 879 954
A13		-	10 226 883	-	9 440 433	9 331 293
A14		-	11 686 638	-	10 383 378	10 383 378
A15		-	12 918 611	-	11 108 171	10 828 901
A16	Trivial	-	3 380 333	-	3 380 333	3 380 333
A17	2 chains	-	3 617 251	-	3 617 251	3 617 251
A18		-	5 596 728	-	4 983 618	4 983 618
A19		-	3 654 374	-	3 323 744	3 323 744
A20	Trivial	-	1 467 925	-	1 467 925	1 467 925
B1	No prec	3 232 698	3 765 558	***2 661 318**	3 136 398	2 661 318
B2		*15 635 435	14 312 915	*13 674 125	13 398 065	11 931 095
B3		20 540 813	19 786 463	18 191 093	17 093 273	15 786 803
B4		*8 269 045	8 323 615	*8 269 045	7 973 725	7 315 675
B5	2 chains	72 155 615	72 155 615	**72 155 615**	**72 155 615**	72 155 615
B6		*12 116 527	12 488 887	*11 195 257	11 089 327	10 800 427
B7	No prec	9 601 299	9 177 579	*8 355 819	7 678 509	6 628 839
B8		*17 865 559	17 152 939	16 067 959	15 840 049	14 398 759
B9		18 502 147	19 969 117	17 484 577	17 474 947	16 495 897
B10		26 012 183	26 904 563	***21 951 533**	23 065 403	21 951 533
B11		25 251 890	27 312 710	22 584 380	23 820 230	20 626 280
B12		*15 868 657	13 734 007	*13 958 707	13 120 897	12 514 207
B13		*28 349 055	27 360 375	*24 471 375	23 078 235	22 657 725
B14		*9 346 480	9 442 780	*8 656 330	8 377 060	8 023 960
B15		*27 794 441	24 568 391	*24 517 031	23 088 581	22 619 921
X1		*15 508 097	15 302 657	*14 127 797	14 127 797	13 720 127
X2	No prec	6 034 937	6 083 087	*5 434 667	4 879 337	4 795 877
X3		*8 285 206	7 649 626	*7 473 076	7 180 966	6 837 496
X4		12 182 072	15 488 372	**11 405 252**	13 366 562	11 405 252
X5		5 081 297	4 988 207	4 712 147	4 715 357	4 522 757
X6		12 565 673	11 031 293	*10 363 613	9 496 913	9 365 303
X7		*22 443 360	22 876 710	21 127 260	21 191 460	20 568 720
X8	2 chains	*24 788 661	22 265 601	*24 788 661	**22 265 601**	22 265 601
X9		*22 251 225	22 312 215	20 167 935	20 479 305	20 039 535
X10		*20 110 472	18 778 322	*17 824 952	17 186 162	16 865 162
X11		*13 489 692	12 802 752	*12 417 552	11 676 042	11 011 572
X12		*11 963 845	12 358 675	*10 583 545	10 503 295	10 246 495
X13		15 950 172	14 345 172	*13 533 042	13 125 372	12 130 272
X14		*8 889 542	8 591 012	*8 013 212	7 644 062	7 422 572
X15		13 990 194	13 710 924	11 682 204	11 682 204	10 882 914

Table 3.4: Computational experiments comparing the proposed approach compared to other contestants

3.7 Conclusion and perspectives

In this chapter, we presented a new anytime tree search algorithm called MBA* for the 2018 ROADEF/EURO challenge glass cutting problem. It performs successive iterations, restarting when its heuristic search tree exploration is completed. During the first iterations, it performs aggressive prunings and behaves like a greedy algorithm. As iterations go, the algorithm performs fewer heuristic prunings, and thus gets access to better solutions (at the cost of an increased computation time of each iteration). If enough time and memory are available, the algorithm ends up performing an iteration with no heuristic pruning, finding the best solution regarding the branching scheme.

We proposed two guides (waste percentage, and waste percentage / average item size). These guides can find significantly better solutions than using a lower bound (the waste), which is what is usually used in branch and bounds. We also presented a symmetry breaking strategy and showed that it significantly improves the quality of the solutions returned by the algorithm.

Also, we designed another algorithm, DPA*, for instances with only two precedence chains. This algorithm returns the best-knowns solutions on these instances within short times.

This result shows that anytime tree search algorithms from the AI/planning communities, and branch-and-bounds from the Operations Research community can benefit from each other, leading to algorithms competitive with classical meta-heuristics (even on a competitive industrial challenge).

Motivated by the success of MBA* on this glass cutting application, we adapted it for classical guillotine cutting problems from the literature and showed that even on more fundamental Cutting & Packing problems, it is still competitive with the other dedicated algorithms from the literature [FL20b]. We investigated the following guillotine variants: knapsack variants (maximizing the weighted sum of packed items), bin-packing variants (minimizing the number of used jumbos) and strip-packing (given an original plate with infinite length, minimize the last horizontal cut position). We investigated 9 datasets from the literature and compared with many existing algorithms. In many situations, MBA* proved to be competitive with dedicated methods, and, in some cases even could find better solutions than state-of-the-art dedicated methods.

4

Tree search algorithms for the Sequential Ordering Problem (SOP)

In this chapter, we study several generic tree search techniques applied to the *Sequential Ordering Problem*. This study enables us to propose a simple yet competitive tree search. It consists of an iterative beam search that favors search over inference and integrates prunings that are inspired by dynamic programming. The resulting method proves optimality on half of the SOPLIB instances, 10 to 100 times faster than other existing methods. Furthermore, it finds new best-known solutions on 6 among 7 open instances of the benchmark in a small amount of time. These results highlight that there is a category of problems (containing at least SOP) where an anytime tree search is extremely efficient (compared to classical meta-heuristics) but was underestimated. Indeed, to the best of our knowledge, it is the first pure constructive method proposed for the SOP. Moreover, despite its simplicity (approximately 250 C++ lines) it is competitive with state-of-the-art more intricate approaches. The source code and solutions can be downloaded at `https://gitlab.com/librallu/cats-ts-sop`.

This chapter is inspired by a paper we presented at ECAI2020 [LBCJ20] and is joint work with *Abdel-Malik Bouhassoun*, *Hadrien Cambazard* and *Vincent Jost*.

Contents

4.1 Introduction

We consider a well-studied operations research problem with a published benchmark (the SOP and the SOPLIB) where a large variety of intricate and advanced approaches have been applied for more than 30 years [Esc88]. We show that a simple anytime tree search algorithm is competitive against classical meta-heuristics. The resulting algorithm performed beyond expectations. It finds better solutions than the currently best-known ones on 6 among 7 open instances of the SOPLIB in a short amount of time (less than 600 seconds on a laptop computer compared to days for some other methods). Furthermore, this algorithm reports optimality proofs on large (very constrained) instances about 10 to 100 times faster than the existing exact approaches. This method is so simple that it could be considered as a baseline algorithm in AI/planning. Indeed, it is an iterative beam search with a closed-list mechanism using no heuristic information other than the prefix cost. This study shows that anytime tree search is a crucial component (at least on the SOPLIB) and might deserve a greater consideration while designing optimization algorithms.

This chapter is structured as follows: Section 4.1 presents the Sequential Ordering Problem and a quick survey of existing methods. Section 4.2 presents the SOP specific bounds we use and compare (namely prefix bound, ingoing/outgoing bound, MST bound). Finally, Section 4.4 presents numerical results on the impact of the search strategy and a comparison with the existing state-of-the-art algorithms.

4.1.1 SOP formal definition

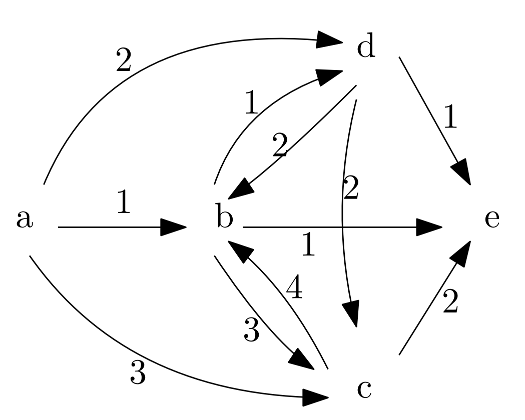
Sequential Ordering Problem (SOP) is an Asymmetrical Traveling Salesman Problem with precedence constraints. See Figure 4.4 for an illustrative example.

An instance of SOP consists of a directed graph $G = (V, A)$, arc weights $w : A \to \mathbb{R}$, a set of precedence constraints $C \subseteq V \times V$ modeled as another graph, a start vertex $s \in V$, and a destination vertex $t \in V$. G is complete except for edges (u, v) where $(v, u) \in C$.

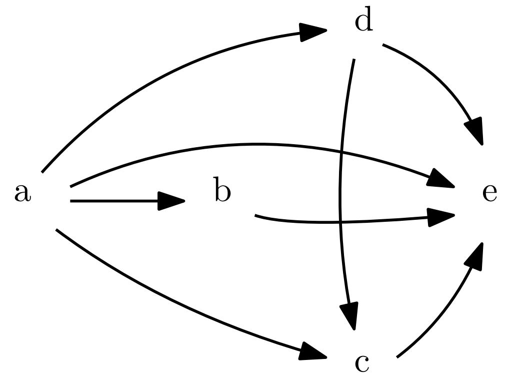
We search for a permutation of vertices that starts with s, ends with t, satisfies the precedence constraints (*i.e.* for each precedence constraint $(a, b) \in C$, vertex a must be visited before vertex b) and that minimizes the sum of weights the arcs joining the vertices in the permutation.

4.1.2 Literature review

SOP was originally presented alongside some exact algorithms based on a mathematical programming model [Esc88]. It has been extensively studied in the past 30 years, and



(a) graph representation – arc weights



(b) graph representation – precedence constraints

Figure 4.1: Example of a SOP instance with 5 vertices and 1 precedence constraint where a is the start vertex and e the end vertex. Permutation a, d, c, b, e is a feasible (since d is visited before c) and has cost $2 + 2 + 4 + 2 = 10$. Permutations a, b, c, d, e and a, c, b, d, e are not feasible. Permutation a, b, d, c, e is optimal with cost $1 + 1 + 2 + 2 = 6$.

many applications and resolution methods have been considered. SOP generalizes several combinatorial problems: Relaxing the precedence constraints gives the Asymmetric Traveling Salesman Problem (ATSP) [LO95]. If, moreover, arc lengths are symmetric, we get the symmetric TSP. We present in this section the most common applications and algorithms for SOP.

SOP arises in many industrial applications. On stacker crane trajectory optimization [Asc96], one has to fulfill transportation jobs as fast as possible. This problem can be modeled using SOP where vertices represent jobs and arc weights represent the time needed to go from a job to another. In automotive paint shops [SGV04] where the goal is to minimize the set-up cost of a paint job (flushing old paint, retrieving new color *etc.*). Also, since car lanes relative order cannot be changed during retrieval, precedence constraints need to be taken into account. SOP also occurs in the switching energy minimization of compilers [SJ15]. While compiling a program, the compiler has to visit operations so that the switching cost is minimized. Since some operations require other operations to be done before starting, precedence constraints also need to be considered. One can also note the use of SOP in freight transportation [EGM94], flexible manufacturing systems [Asc96], and helicopter visiting [FTP92].

Many exact approaches have been proposed to solve the Sequential Ordering Problem. As we discuss in this section, most of the literature focuses on finding strong lower bounds. Earlier approaches to SOP include cutting planes [AEGS93] and Lagrangian relax and cut algorithm [EGM94]. A mathematical programming model solved with a branch-and-bound in which the branching is performed in order to decompose the problem as much as possible was also studied [MMDC$^+$12]. The uncapacitated m-PDTSP, which is a generalization of SOP, led to competitive results on SOP using a branch-and-cut algorithm combined with a generalized variable neighborhood search [GR15]. Also, decision diagrams made a huge impact by generating automatically good quality bounds [CvH13, Her04]. In 2015, a dedicated branch-and-bound has been proposed [SJ15], it combines quick and elementary bounds (prefix, ingoing/outgoing degrees, and MST) with a technique inspired from TSP dynamic programming called *History Cuts* that allows pruning dominated partial solutions. Despite the simplicity of its bounds, the later method obtained excellent numerical results. It, therefore, inspired us to study further the impact of the branch-and-bound components. This algorithm has been further improved by the integration of a custom assignment bound and a local-search at each node of the search tree [JSP$^+$17].

In meta-heuristics, numerous works focus on a local-search move called SOP-3-exchange and combine it with various search algorithms. It is a 3-OPT move optimized to take into account precedence constraints and asymmetrical arc weights. This SOP-3-exchange procedure is presented alongside an Ant Colony Optimization algorithm [GD97]. It has also been used within a particle swarm optimization algorithm [AMPG11], by a hybrid genetic algorithm using a new crossover operator referred to as *Voronoi Quantized Crossover* [SM03], as well as a bee colony optimization [WWKT14], and a parallel roll-out algorithm [GM03].

Since the hybrid ant colony algorithm HAS-SOP [GD97] obtained excellent numerical results, a considerable amount of work has been done to improve it. First, by the integration of a better data structure called the *don't push stack* [GMW12]. HAS-SOP was again improved by the integration of a Simulated Annealing scheme [Ski17]. Recently, the LKH heuristic was improved to be able to solve SOP instances [Hel17]. These two last methods obtained the best solutions on large instances of the SOPLIB.

According to the literature review on the Sequential Ordering Problem, the existing works seem to consider as a working hypothesis that local-search is a key feature to obtain state-of-the-art solutions on large instances and that strong lower bounds are the key components of branch-and-bound algorithms. Moreover, since 2006, every work based on meta-heuristics use the SOPLIB as a standard benchmark, as we do here [AMPG11, CvH13, GMW12, GR15, GM03, Hel17, JSP$^+$17, MMDC$^+$12, SJ15, Ski17, WWKT14]. In the next sections of this paper, we investigate different branch-and-bound components and show that specific combinations can build very efficient methods that provide new best-known solutions on large and constrained SOPLIB instances and prove optimality on half of them (which is not possible with most local-search strategies).

4.2 A search tree for the SOP

When designing a tree search algorithm, it is common to divide it into two parts. The search tree (problem-specific part, *i.e.* how to branch, bounds, pruning, *etc.*) and the generic parts (a search strategy, such as DFS, Beam Search *etc.* or generic prunings, in our case domination prunings). This section presents the problem specific parts and the next section presents the generic parts.

During the implementation of the search trees, we focused on fast bounds ($O(1)$ for ingoing/outgoing bounds and $O(|E|)$ for the Minimum Spanning Tree bound). The key idea is to favor search over bounding/filtering. In the specific case of the SOPLIB, we show that using stronger bounds dramatically affects the performance of the method, even if the resulting branch-and-bound explores a smaller tree and has better guidance.

We branch as follows: The root node contains the start vertex s. Each child of a given node corresponds to each possible next vertex to be visited (vertices not already added to the prefix and whose predecessors have all been added already to the prefix).

4.2.1 Definition and computation of lower bounds

We define our bounds as it is usually done in AI Planning. For a given node n we define the lower bound as follows: $f(n) = g(n) + h(n)$

where:

- $g(n)$ is the prefix bound (*i.e.* cost of arcs between already selected vertices)
- $h(n)$ is the suffix bound (*i.e.* an optimistic estimate of the remaining work to be done). The three bounds we develop in this section only differ on this criterion.

4.2.2 Prefix bound

The prefix bound consists in setting $f(n) = g(n)$ for any node of the search tree. That is $h(n) = 0$.

This bound (*i.e.* $g(n)$) can be computed in $O(1)$ along a branch of the search tree, simply by accessing, when adding vertex b to a prefix that ended with vertex a, the cost w_{ab} from the input. Surprisingly, our computational experiments show that the bound guide, is the best among the three bounds considered.

4.2.3 Ingoing/Outgoing bound

For the Ingoing/Outgoing (or I/O) bound, we keep the prefix bound and add a lower bound on the suffix.

Consider a node n of the search tree. Let $\text{prefix}(n) = v_1 \dots v_{k-1}$ be an ordered set of already visited vertices excluding the last added vertex v_k, and $\text{suffix}(n)$ the set of remaining vertices to add. We remind that s denotes the start vertex and t the end vertex of the SOP instance.

We design the optimistic estimate of remaining cost $h(n) = \max(h_{in}(n), h_{out}(n))$ where:

$$h_{in}(n) = \sum_{v \in \text{suffix(n)}} \min_{u \in V, uv \in A} w_{uv}$$

$$h_{out}(n) = \sum_{u \in (\text{suffix(n)} \cup \{v_k\}) \setminus \{t\}} \min_{v \in V, uv \in A} w_{uv}$$

This bound can be computed in $O(1)$ along a branch of the search tree. Indeed, one can precompute the sum of ingoing arcs at the root node. When adding a vertex v to the prefix, this sum can be updated in constant time by removing the minimum ingoing arc for v. The same algorithm can be applied for outgoing arcs. Note that arcs considered in this bound can have one of their endpoints in the prefix. One can consider implementing a stronger bound that removes such arcs to improve its value.

4.2.4 Minimum spanning-tree bound

For the MST bound, we keep the prefix bound and add a lower bound on the suffix.

Let $w_{ab} = +\infty$ if b must be visited before a. Define $w'_{ab} = \min(w_{ab}, w_{ba})$. The suffix cost $h(n)$ is then computed using Prim's algorithm on the graph spanned by the vertices not yet visited, with edge costs w'.

A key analysis, on the instances used for this paper, revealed that it would be pointless to try to speed-up the implementation of the MST bound, because, even if it could be computed as fast as the prefix bound, it would not lead to better solutions than the algorithms using this weaker bound. We ran an algorithm with MST within the time limit. Then ran algorithms with cheaper bounds restricting the number of nodes to the number opened by the MST based algorithm.

4.3 Dominance pruning

Dominance prunings are a way to eliminate symmetries and dominated partial-solutions. It can be seen as a form of dynamic programming integrated within a tree search algorithm. It stores all explored sub-states. Each node compares its prefix subset and last vertex to existing entries in the database. If it is dominated, the node is pruned. This strategy has been used in a large variety of methods. For instance, memorization in branch-and-bounds ([SJ15, STDC18]).

A dominance pruning for the Sequential Ordering Problem can be defined as follows (inspired from TSP dynamic programming [CGP12], history cuts [SJ15] and the call-based dynamic programming [BJJ14]):

Two solution prefixes n_1, n_2 are called *equivalent* if they cover the same subset $S \subseteq V$ of vertices and end with the same last vertex v. If the prefix cost $g(n_1)$ (*i.e.* the sum of selected arcs between vertices from $S \cup \{v\}$) is (strictly) greater than $g(n_2)$, then n_1 is (strictly) dominated by n_2 and thus can be pruned.

In other words, the dominance prunings can be seen as a form of dynamic programming where the formulation can be described as follows where $\text{pred}(S, j)$ indicates that j is not a predecessor of any vertex in S:

$$f^*(S, i) = \min_{j \in S \wedge \text{pred}(S,j)}(f^*(S \setminus \{j\}, j) + w_{ji})$$

Our implementation of dominance prunings consists in altering the behavior of the branch-and-bound as follows: Each time a node n is opened, the prefix of n is compared to what exists in the database. If the subset of vertices spanned by n does not exist in the database it is added to it, otherwise, it is compared to the best equivalent prefix found so far. If the subset has a prefix cost worst than the one in the database, the node n is pruned.

We implement the database using a hash table. In our numeric experiments, we notice that a branch-and-bound using the prefix equivalence opens on average 4 to 5 times fewer nodes than its equivalent version without prefix equivalence.

In some versions of the Prefix Equivalence (for instance the one found in history cuts [SJ15]), nodes are pruned if their prefix matches an existing entry in the database even if their cost is equal. Notice that we restart tree search algorithms (*i.e.* Iterative Beam Search and Limited Discrepancy), which perform heuristic prunings (they prune nodes to avoid saturating the memory and to ensure reaching feasible solutions). To allow our algorithms to close an instance (*i.e.* to prove the optimality of the best solution it found), we prune nodes only if they are *strictly* dominated by the best equivalent recorded in the database. The reason for doing so is that, although the value recorded in the database corresponds to a node that has been already explored, this exploration might have been partial and we need to ensure that the search does not perform any heuristic pruning to provide proof of optimality.

4.4 Computational results

Results were obtained from a Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz with 8GB RAM. We run each pair of instance-algorithm for 600 seconds. Instances come from the SOPLIB benchmark available here

`http://www.idsia.ch/~roberto/SOPLIB06.zip`

The Instances are randomly generated and their names contain 3 numbers indicating: the number of nodes (from 200 to 700), the range of the cost drawn uniformly (either between 0 and 100 or between 0 and 1000), and the percentage of precedence constraints. Notice that the Instance $R.200.100.60$ is ill-defined as its costs are drawn between 0 and 1000.

Best known bounds and solutions are an aggregation of results coming from [Ski17, GR15, MMDC$^+$12, JSP$^+$17, Hel17]. Note that the Lin Kernighan Helsgaun 3 Algorithm [Hel17] was run on each instance for 100.000 seconds. The Enhanced Ant Colony System with Simulated Annealing [Ski17] was run 30 times per instance for 600 seconds so 18.000 seconds per instance. The time limit of 600s used in the present paper is therefore considerably smaller.

Performance of tree search components We ran 18 different tree search algorithms (DFS, LDS and Beam Search) with and without Prefix equivalence using the prefix, Ingoing/Outgoing degrees or the MST bound for 600 seconds. It turns out that there are two clear winners out of these methods (Beam Search + Prefix Equivalence + Prefix or Ingoing/Outgoing bound). Since the results of the two best methods are very similar, we choose to emphasize the simplest one (*i.e.* Beam Search + Prefix Equivalence + Prefix bound). We show in Table 4.1 that any deviation of search strategy, prunings or bounds lead to a performance drop (except for the Beam Search + Prefix Equivalence + Ingoing/Outgoing degree bound). For the sake of clarity, we only show deviations of BS+PE+P and not the 18 algorithms.

Discussion As expected, tree-search performs better (even more with prefix-equivalence prunings) on most-constrained instances (proving optimality on the 60% and 30% and competitive results on the 15%) while obtaining poor results on loosely constrained ones (1%)

The minimum-spanning-tree-based tree search algorithms open fewer nodes (1.000 to 10.000 times less than the ingoing/outgoing bound). As a result, fewer solutions are found (sometimes none within the time limit) and are less efficient. It appears that on medium-size instances, the MST bound does not provide a significant guide improvement (and thus harms performance since it is more expensive to compute than the Prefix or Ingoing/Outgoing bound). One might wonder whether a possible incremental evaluation of the MST bound, that is, computation of it along a branch of the search tree taking advantage of the similarity between the MST for a node and the MST for one of its child would make a difference. For the benchmark we used, it would make absolutely no difference. In the best scenario, we would end up with a third algorithm equivalent to our other two champions. We do not report numerical results on this issue here, but restricting the algorithms by the number of nodes, and not by the time limit, we observed that the MST bound did not improve the results overall.

We remark that the search strategy also plays an important role while finding good solutions or closing instances within the time limit. Globally, the Beam Search strategy finds better solutions than LDS that finds better solutions than DFS. Although DFS can find the optimal solution and to prove optimality on some instances, it doesn't match the quality of the solutions of either Beam Search or LDS. The main advantage of DFS is that it does not reopen any node, its main drawback is that it struggles to provide good quality solutions fast. In comparison, Beam Search reopens nodes, but by finding very good solutions fast, it can prune more nodes and thus, close more instances. In this study, the beam search strategy (using prefix equivalence) appears to be the best, in terms of both proving optimality and finding the best solutions within the time limit.

We compare the *Beam Search + Prefix Equivalence + Prefix or Ingoing/Outgoing bound* against the best solutions reported in the literature by other state-of-the-art algorithms. Our method finds new best-known solutions on 6 among 7 open instances of the SOPLIB in a much shorter time than the other algorithms. It also proves optimality quickly on all instances with 30 and 60 percent precedence (about 10 to 100 times faster than the DFS+prefix equivalence+stronger bounds+local-search [JSP+17]). We remark that the proposed method fails to provide good solutions for 1% precedence due to the poor quality of bounds on these instances that are close to ATSP.

Finally, we may study the performance profiles of all the algorithms we presented and compare them to the state-of-the-art. We notice that even if DFS is an anytime algorithm and improves regularly the quality of its solutions found over time, it is largely dominated by the other methods and struggle to compete with other methods. LDS aims to improve the anytime behavior of DFS, and indeed, does. However, it is still not enough to compete with the state-of-the-art. Finally, Beam Search can compete. Its version without Prefix Equivalence dominances obtains similar quality of solutions over time than EACS+SA, and its version with dominances even improves the best-known solution in the literature in approximately 50 seconds. We notice that in all algorithms, the dominances allow performing a little bit better.

4.5 Conclusions and future works

In this chapter, we discussed the importance of considering anytime tree search while designing algorithms for large-scale instances. In this (striking) example, we showed that even a simple tree-search algorithm (that could also be considered as a baseline algorithm due to its simplicity) could outperform state-of-the-art operations research intricate and advanced meta-heuristics on a well-studied benchmark.

The search algorithms usually considered in operations research (namely DFS and LDS) proved to be dominated by the beam search on the SOPLIB. Beam search outperformed

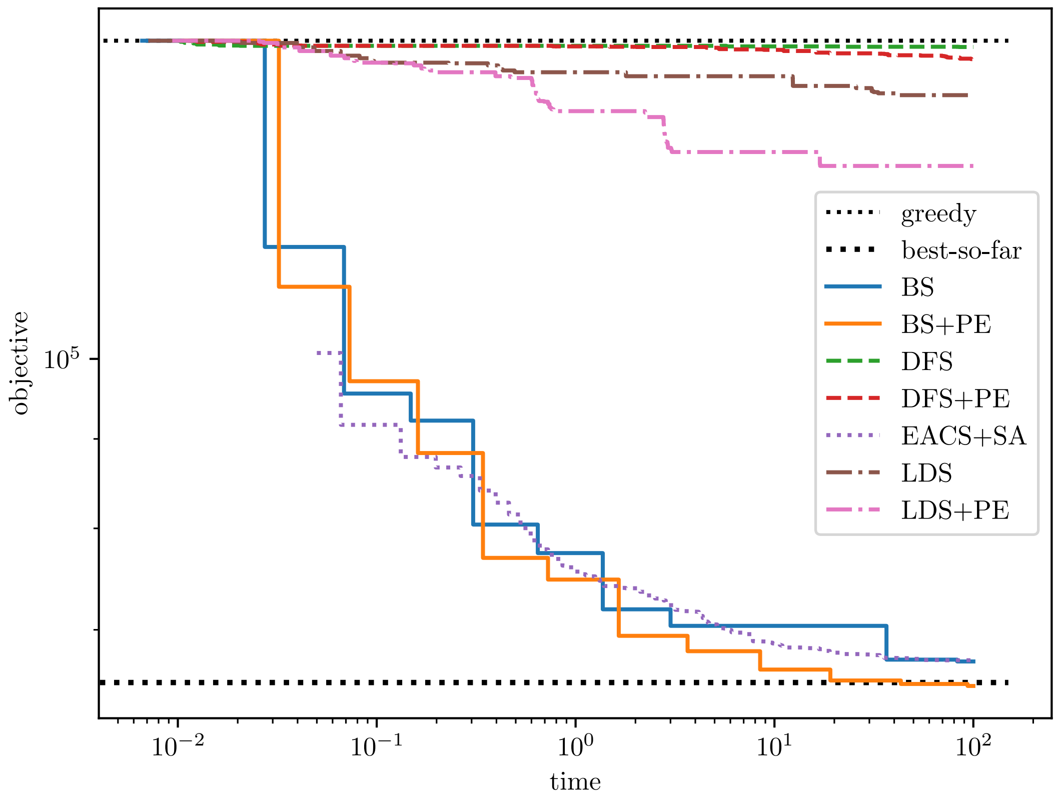


Figure 4.2: Performance profile comparison of the current state of the art (EACS+SA) with the different algorithms developed in this chapter. The solution obtained by a nearest-neighbour greedy and the best-so-far are showed to better indicate the contribution of each component.

DFS by proving optimality on 25 instances (DFS proved optimality only on 17 instances) in less than 600 seconds. It outperformed existing algorithms in finding new best-known solutions on 6 among 7 open instances in a short amount of time. We also demonstrated the importance of deconstructing optimization algorithms (in this case a tree-search) and analyze the contribution of each separate building block. Indeed, neither beam search alone, nor the prefix equivalence prunings with DFS/LDS alone were able to outperform state-of-the-art , but together, they did.

While aiming to implement an efficient all-purpose SOP solver, one (in our opinion) should integrate a Beam Search and prefix equivalence as it proves itself to be a very efficient and complementary approach on large and constrained instances. However, we note that such tree-search methods are not as efficient on loosely constrained instances as LP-based branch-and-bounds or local-search, thus the importance of hybridizing them together in such all-purpose solver.

The existing SOPLIB contains mostly very constrained instances (at the exception of the 1% precedencies). It was probably designed in such a way that local-search and LP-based approaches struggle to find good solutions. As tree-search outperforms these approaches on 15% instances and is outperformed on 1% instances, it is an interesting question to update the benchmark with intermediate densities (5%, 10% *etc.*) as they would probably be harder considering both classes of algorithms.

This paper only considers the Sequential Ordering Problem. However, a similar decomposition methodology of complicated algorithms into simple building blocks and the assessment of their contributions, computational costs, and ideally synergies, can be applied to other combinatorial optimization problems. For instance, anytime tree search has been successfully applied on various hard combinatorial optimization problems such as "simple assembly line balancing problem" [Blu08], "Longest Palindromic Common Sub-sequence" [DRB19]. We believe that similar conclusions (anytime tree-search are underestimated by operations research practitioners) can be drawn on several other problems.

Moreover, we limited this study to DFS, LDS and Beam Search as search strategies. Many more exist (like Beam stack search [ZH05], BULB [FK05], Anytime Focal Search [CGM+18] *etc.*). Also, one can study other branch-and-bound components like performing a local-search within each node [JSP+17, DCGT04] or the probing strategy (starting a greedy algorithm at each node to obtain a better quality estimate). This would probably lead to better insights on when and how using anytime tree search while facing operations research problems.

To try and evaluate the contribution of such building blocks and ideas on various problems, we started developing a framework for generic Tree-Search, which allows us to address a combinatorial problem as a branching scheme, specific prunings, LP cuts, and bounds and then to apply generic Tree Search strategies. This might lead to a new standard way (aside Mathematical Programming, Constraint Programming, local-search [GBD+14], Satisfiability of Boolean Clauses, *etc.*) to define and solve combinatorial problems.

Table 4.1 legend: BKLB (resp. BKUB) refers to the Best Known Lower Bound (resp. Upper Bound) from our literature review.
BS,PE,P refers to the combination of Beam Search, Prefix Equivalence and Prefix bound.
BS,PE,IO refers to Beam Search, Prefix Equivalence and Ingoing/Outgoing bound.
BS,PE,MST refers to the Beam Search with Prefix Equivalence and MST bound.
BS,P refers to the Beam Search with the Prefix bound and without Prefix Equivalence.
DFS,PE,P refers to Depth First Search with Prefix Equivalence and Prefix bound.
LDS,PE,P refers to limited Discrepancy Search with Prefix Equivalence and Prefix bound.
"T record" indicates the time required by BS,PE,P to reach a solution of value BKUB or lower.
"T opt" indicates the time required by BS,PE,P to close the instance.

Bold instances are still open.
"-" (in column BS,PE,MST) indicate that no solution has been found by the method within 600 seconds.
"-" (in columns record (resp. opt)) indicate that no new record (resp. proof of optimality) was found by any of our methods.
Bold objective values indicate when the method was able to close the instance within the time limit.
Underlined objective values indicate an improvement of best-known solutions.

Instance	BKLB	BKUB	BS,PE,P	BS,PE,IO	BS,PE,MST	BS,P	DFS,PE,P	LDS,PE,P	T record (s)	T opt (s)
R.200.100.1	61	61	189	189	299	189	283	192	-	-
R.200.100.15	1.792	1.792	1.792	1.792	1.887	1.796	3.740	2.325	19.5	-
R.200.100.30	4.216	4.216	**4.216**	**4.216**	**4.216**	4.249	**4.216**	4.216	0.1	0.6
R.200.100.60	71.749	71.749	**71.749**	**71.749**	**71.749**	71.749	**71.749**	71.749	0.0	0.0
R.200.1000.1	1.404	1.404	2.554	2.554	3.398	2.554	3.448	2.684	-	-
R.200.1000.15	20.481	20.481	**20.481**	**20.481**	20.952	20.517	34.982	25.592	16.3	547.7
R.200.1000.30	41.196	41.196	**41.196**	**41.196**	**41.196**	41.728	**41.196**	41.196	0.1	0.4
R.200.1000.60	71.556	71.556	**71.556**	**71.556**	**71.556**	71.556	**71.556**	71.556	0.0	0.0
R.300.100.1	26	26	214	214	406	204	265	225	-	-
R.300.100.15	3.152	3.152	3.152	3.152	3.458	3.201	5.355	4.081	178.9	-
R.300.100.30	6.120	6.120	**6.120**	**6.120**	6.330	6.200	**6.120**	6.120	2.2	7.9
R.300.100.60	9.726	9.726	**9.726**	**9.726**	**9.726**	9.726	**9.726**	9.726	0.0	0.0
R.300.1000.1	1.294	1.294	3.080	3.080	4.784	2.888	3.551	3.012	-	-
R.300.1000.15	29.006	29.006	29.006	29.006	33.885	29.481	51.152	43.597	220.0	-
R.300.1000.30	54.147	54.147	**54.147**	**54.147**	54.491	54.533	55.791	54.147	0.4	3.6
R.300.1000.60	109.471	109.471	**109.471**	**109.471**	**109.471**	109.471	**109.471**	109.471	0.0	0.0
R.400.100.1	13	13	191	191	-	175	295	203	-	-
R.400.100.15	3.879	3.879	3.879	3.879	5.011	3.961	8.103	6.584	176.7	-
R.400.100.30	8.165	8.165	**8.165**	**8.165**	8.210	8.183	**8.165**	8.165	0.4	2.1
R.400.100.60	15.228	15.228	**15.228**	**15.228**	**15.228**	15.228	**15.228**	15.228	0.0	0.0
R.400.1000.1	1.343	1.343	3.247	3.247	-	3.247	4.466	3.525	-	-
R.400.1000.15	35.364	38.963	38.963	38.963	53.789	39.722	76.463	69.342	157.2	-
R.400.1000.30	85.128	85.128	**85.128**	**85.128**	87.698	85.720	**85.128**	85.128	0.5	1.8
R.400.1000.60	140.816	140.816	**140.816**	**140.816**	**140.816**	140.816	**140.816**	140.816	0.0	0.0
R.500.100.1	4	4	267	275	-	202	272	232	-	-
R.500.100.15	4.628	5.284	5.261	5.261	7.593	5.411	9.917	9.610	206.5	-
R.500.100.30	9.665	9.665	**9.665**	**9.665**	10.388	9.778	10.999	9.665	1.4	6.2
R.500.100.60	18.240	18.240	**18.240**	**18.240**	**18.240**	18.257	**18.240**	18.240	0.0	0.1
R.500.1000.1	1.316	1.316	4.079	4.079	-	3.541	4.703	3.717	-	-
R.500.1000.15	43.134	49.504	49.366	49.366	71.888	50.624	103.985	94.625	120.8	-
R.500.1000.30	98.987	98.987	**98.987**	**98.987**	115.074	99.492	114.544	98.987	1.7	3.8
R.500.1000.60	178.212	178.212	**178.212**	**178.212**	**178.212**	178.355	**178.212**	178.212	0.0	0.0
R.600.100.1	1	1	289	289	-	289	306	246	-	-
R.600.100.15	4.803	5.472	5.469	5.469	9.329	5.695	13.007	10.939	160.5	-
R.600.100.30	12.465	12.465	**12.465**	**12.465**	12.929	12.475	13.899	12.465	3.1	10.3
R.600.100.60	23.293	23.293	**23.293**	**23.293**	**23.293**	23.324	**23.293**	23.293	0.0	0.0
R.600.1000.1	1.337	1.337	4.030	4.030	-	3.853	4.814	4.074	-	-
R.600.1000.15	47.042	55.213	54.994	54.994	90.977	55.748	115.295	108.164	183.6	-
R.600.1000.30	126.789	126.789	**126.798**	**126.798**	158.425	128.761	145.672	126.798	1.6	7.2
R.600.1000.60	214.608	214.608	**214.608**	**214.608**	214.608	214.608	**214.608**	214.608	0.1	0.1
R.700.100.1	1	1	186	250	-	186	281	258	-	-
R.700.100.15	5.946	7.021	7.020	7.020	11.392	7.220	13.778	13.206	386.9	-
R.700.100.30	14.510	14.510	**14.510**	**14.510**	17.125	14.632	19.655	14.510	4.2	21.6
R.700.100.60	24.102	24.102	**24.102**	**24.102**	24.848	24.102	**24.102**	24.102	0.2	0.5
R.700.1000.1	1.231	1.231	4.403	4.403	-	4.042	4.629	4.028	-	-
R.700.1000.15	54.351	65.305	64.777	64.777	108.627	65.775	141.544	121.189	25.5	-
R.700.1000.30	134.474	134.474	**134.474**	**134.474**	158.327	136.073	158.613	134.474	1.3	4.8
R.700.1000.60	245.589	245.589	**245.589**	**245.589**	245.688	245.752	**245.589**	245.589	0.1	0.1
nb closed			25	25	11	0	17	0		

Table 4.1: Tree search components performance.

```
WORKSPACE/
├── cats-framework/
│   └── [...]
└── cats-ex-sop/
    ├── src/
    │   ├── Instance.hpp
    │   ├── NodeForward.hpp
    │   ├── main.cpp
    │   └── checker.hpp
    └── insts/
        ├── R.700.1000.15
        └── [...]
```

Figure 4.3: example project directory overview

4.6 A CATS-framework application example

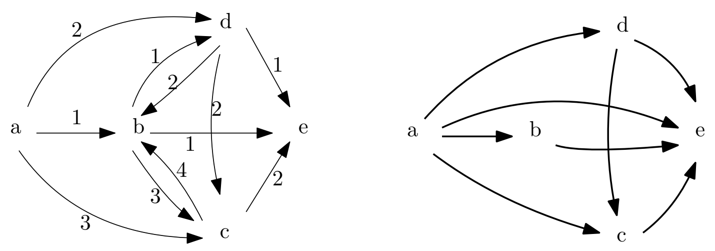
We believe that the code we developed during the SOP study (Chapter 4), is relatively simple (around 250 lines of code) and provides good results (new best-so-far solutions on 6 over 7 open instances of the SOPLIB). For these reasons, we believe that it is a good example of an application of the CATS framework and allows a simple usage example and tutorial.

The complete example source code is available at `https://gitlab.com/librallu/cats-ex-sop`
We use the version 0.3 of the cats-framework for this example.

4.6.1 Project Architecture

The project is structured as follows: the cats-framework (version 0.3) directory, and the *cats-ex-sop* directory (our working directory) are at the same level. The *cats-ex-sop* directory contains a source sub-directory (src), an instance directory (insts) where we put the SOPLIB instances. For the sake of simplicity, we do not discuss the files used for compilation (CMakeLists.txt, etc.). Figure 4.3 presents an illustration of the workspace.

4.6.2 Problem description – instances & solutions



(a) arc weights (b) precedence constraints

	a	b	c	d	e
a	0	1	3	2	∞
b	-1	0	3	1	2
c	-1	4	0	-1	2
d	-1	2	2	0	1
e	-1	-1	-1	-1	0

(c) matrix representation

Figure 4.4: Example of a SOP instance with 5 vertices and 1 precedence constraint where a is the start vertex and e the end vertex.

Consider the example given in Figure 4.4. An instance is defined by two graphs: the arc weight graph that indicates the cost of selecting an edge and the precedence graph that indicates the precedence constraints (*i.e.* if there is an arc from u to v in the precedence graph, u must be scheduled before v).

It is also possible to encode an SOP instance within a matrix (see Figure 4.4c). If $M_{ij} \geq 0$, the edge (i, j) exists and costs M_{ij}. If $M_{ij} = -1$, it indicates that j must precede i (and thus, arc i, j does not exist). Finally, the arc from the start vertex going to the end vertex is infeasible and has weight ∞ (in practice a very large weight).

4.6.3 Instance format

The instance presented in Figure 4.4 is described by the following file. The first line is the number of vertices n (here $n = 5$). Then, each line represents each vertex (a, then b, *etc.*). Each -1 represents a precedence constraint. In the example, we can see that b, c, d, e cannot be placed before a, e cannot be placed before a, b, c, d and c cannot be placed before d.

ex.sop

```
5
0   1   3   2   999
-1  0   3   1   2
-1  4   0   -1  2
-1  2   2   0   1
-1  -1  -1  -1  0
```

4.6.4 Solution format

The solution $< a, b, d, c, e >$ can be described in the following format where each vertex is represented by an id starting by 0 (a is represented by 0, b by 1, *etc.*).

opt.sol

```
0 1 3 2 4
```

4.6.5 Reading the instance & pre-processings

We first parse the instance files and preprocess some information that will be used extensively during the search. Preprocessing information as much as we can is usually efficient since these computations will be possibly executed millions of times during the search. All of this information is stored in the *Instance* class. For the sake of simplicity, we omit in this document the details of the file-parsing code[1].

We first define the following C++ types (not mandatory but is good practice):

- **NodeId** that represents the node id (as an integer)
- **Weight** that represents the cost of selecting an arc (and the cost of solutions)
- **Precedence** that is a couple of two vertices (u, v) indicates that u must precede v

We then build the instance class that provides the following information:

[1]this code is available in the companion git repository

- n: instance size, *get_ n* procedure
- $w_{ij} = M_{ij}$: cost of selecting arc (i, j), *get_ weight* procedure
- possible successors of u: all vertices except u and its predecessors in the precedence graph. We use this procedure to enumerate all next vertices of a given partial solution. *get_ possible_ successors* procedure.
- predecessors of u: all predecessors of u in the precedence graph. We use this procedure to check that all dependencies of u are satisfied before adding it. *get_ predecessors* procedure.
- start and end vertices: return the start vertex id (resp. the end vertex id). *get_ start_ vertex* and *get_ end_ vertex* procedures.

the following C++ code shows the Instance getters defined above.

Instance.hpp - getters

```
NodeId get_n();
Weight get_weight(NodeId i, NodeId j);
std::vector<NodeId>& get_possible_successors(NodeId i);
std::vector<NodeId>& get_predecessors(NodeId i);
std::vector<Precedence>& get_precedences();
NodeId get_start_vertex() const;
NodeId get_end_vertex() const;
```

4.6.6 Search tree definition

We now define the search tree that we will use in our tree search algorithm. A search tree definition is implemented as a class called *Node* (it can be seen as a form of an interface). Possibly, one can define a *PrefixEquivalenceNode* that allows defining an equivalence class for a node. It can be seen as a form of dynamic programming integration that allows pruning dominated or symmetric nodes. This class will contain as a constructor, the root definition, a *getChildren* method that returns all the node children, an *isGoal* method that returns true if the node is a feasible solution and bounds and guides function (resp. *evaluate* and *guide*) that will be used to prune nodes and guide the search. We call this class *NodeForward* as it performs a forward search (*i.e.* starting by the starting vertex and constructing the partial solutions by appending a new vertex at the end of it).

Each node will be copied multiple times. Thus, the less attributes in the node, the better (both for computation time and memory usage). In this example, we use 4 attributes:

- *inst* a reference to the instance. A C++ reference acts similarly as pointers, thus only uses a few bytes.
- *prefix* an array containing all the vertices selected in the partial solution. We use this information to reconstitute the final solution.
- *cost_ prefix* an integer used to compute the cost of the partial solution.
- *added_ subset* A subset of vertices corresponding to the set of already added vertices. It is implemented in an efficient way using bitsets to limit the memory usage and can add/access a vertex in $O(1)$.

NodeForward.hpp - Attributes

```
class NodeForward : public PrefixEquivalenceNode<PE_SOP> {
 private:
    Instance& inst_;
    std::vector<NodeId> prefix_;
    Weight cost_prefix_;
    SubsetInt added_subset_;
```

With this attributes, the root definition initializes the prefix to the start vertex s, the cost to 0 and the added subset to $\{s\}$. We also define a copy constructor that allows to copy nodes. It will be used during the search and to create children.

NodeForward.hpp - Constructors

```
explicit NodeForward(Instance& inst):
Node(), inst_(inst), cost_prefix_(0), added_subset_() {
    prefix_.push_back(inst_.get_start_vertex());
    added_subset_.add(inst_.get_start_vertex());
}

explicit NodeForward(const NodeForward& s): Node(s), inst_(s.inst_),
prefix_(s.prefix_), cost_prefix_(scost_prefix_), added_subset_(s.added_subset_) {}

inline NodePtr copy() const override { return NodePtr(new NodeForward(*this)); }
```

We first design two helper methods:

- *lastCity* : that returns the last city added within the node
- *addCity* : that adds a city to the node (used in addition to a copy, to create a child)

We now define the *getChildren* method. For every possible successor (called neigh), we check that it is not already added, and that all its predecessors are added. If so, we add it to the children list.

NodeForward.hpp - Children

```
inline std::vector<NodePtr> getChildren() override {
    std::vector<NodePtr> res;
    NodeId last_city_ = lastCity();
    // for each vertex, check if we can add it
    for ( NodeId neigh : inst_.get_possible_successors(last_city_) ) {
        // check that it is not already added and preds satisfied
        bool to_add = !added_subset_.contains(neigh);
        // check that precedences are satisfied
        if ( to_add ) {
            for ( NodeId u : inst_.get_predecessors(neigh) ) {
                if ( !added_subset_.contains(u) ) {
                    to_add = false;
                    break;
                }
            }
        }
        if ( to_add ) {  // generate children and add it to the result
            NodeForward* child = new NodeForward(*this);
            child->addCity(neigh);
            res.push_back(NodePtr(child));
        }
    }
    return res;
}


inline NodeId lastCity() const { return prefix_.back(); }


inline void addCity(NodeId j) {
    NodeId i = lastCity();
    added_subset_.add(j);
    prefix_.push_back(j);
    cost_prefix_ += inst_.get_weight(i, j);
}
```

We define our last mandatory function, the goal condition. A node is a goal if it contains every vertex of the instance (and ends by vertex t). The framework also allows defining a *handleNewBest* method. This method is called when a node is a goal and found a best-so-far solution. We use it to call a checker (to validate the correctness of our implementation and solutions) and to write the solution in a file.

NodeForward.hpp - Goal

```
inline bool isGoal() const override {
    return inst_.get_n() == static_cast<int>(prefix_.size());
}

void handleNewBest() override {
    double check_val = checker(inst_, prefix_);
    assert(check_val == this->evaluate());
    // write solution file
    [...]
}
```

Dominances and the dominance combinator The code presented above would be enough to implement a simple beam search for the SOP. As we show in this chapter, this algorithm would be competitive with the state-of-the-art. However, we can integrate some dominances to further improve the algorithm. The prefix equivalence dominance can be roughly described as follows: For each node in the search tree, we store the set of visited cities, the last city, and the cost of the partial solution. If, during our search, we find another node with the same set of visited cities and last city with a larger cost, we may discard this node as it is "dominated" by another node. To implement this component, we store all entries in a data structure and perform queries efficiently. To this extent, hash-tables seem to be a good candidate. Using the framework, we have to define a structure defining a prefix equivalence for SOP (called *PE_SOP*) and a hash function (called *nodeEqhash*) for *PE_SOP*. Once these definitions are done, all the other parts of the dominance (prefix equivalence store and domination combinator) are implemented in the framework.

NodeForward.hpp - Prefix Equivalence

```
/**
 * SOP prefix equivalence
 */
```

```
struct PE_SOP {
    SubsetInt subset;
    int last;

    PE_SOP(const PE_SOP& n) :
        subset(n.subset), last(n.last) {}
    PE_SOP(const SubsetInt sub, int l) : subset(sub), last(l) {}

    bool operator==(const PE_SOP& a) const {
        if ( last != a.last ) return false;
        return subset == a.subset;
    }
};

/**
 * hash function taking a nodeEquivalenceSOP as a parameter
 */
struct nodeEqHash {
    size_t operator()(const PE_SOP& n) const noexcept {
        size_t seed = n.subset.hash();
        boost::hash_combine(seed, static_cast<size_t>(n.last));
        return seed;
    }
};
```

To finalize the implementation of the prefix equivalence dominance, we now add the *getPrefixEquivalence* attribute to the NodeForward class using the type defined above.

NodeForward.hpp - Prefix Equivalence in NodeForward

```
inline PE_SOP getPrefixEquivalence() const override {
    return PE_SOP(added_subset_, lastCity());
}
```

4.6.7 Wrapping everything together

In the previous subsections, we defined the instance and the search tree. We can now wrap everything together and build the tree search algorithm. We start by parsing the user input. The user will provide the instance name and the execution time (time limit).

main.cpp - Parsing the program input

```
if ( argc < 3 ) {
    std::cout << "\n[ERROR] USAGE: " << argv[0] << " INSTANCE TIME" << std::endl;
    return 1;
}

// parse user input
Instance inst(argv[1]);
int time_limit = std::stoi(argv[2]);
```

We define static objects. The first one, the search manager, maintains the best-so-far solution, the best-known upper and lower bounds and displays information during the search. The prefix equivalence store maintains for each node equivalence class the best-so-far cost.

main.cpp - Static objects definition

```
// defines the search manager and the prefix equivalence store
SearchManager search_manager = SearchManager();
GenericPEStoreHash<PE_SOP, nodeEqHash, DominanceInfos2> prefix_equivalence_store;
```

We define the root of the node. It is first defined using the *NodeForward* class. Then, we add a statistics combinator. This combinator will give us feedback on the number of opened nodes, the number of bound evaluations, *etc.* Finally, we add a dominance combinator.

main.cpp - Search tree definition

```
// define root of the tree
NodePtr root_ptr = NodePtr(new NodeForward(inst));

// measure some node openings, etc.
root_ptr = NodePtr(new StatsCombinator<PE_SOP>(
    root_ptr, search_manager, search_manager.getSearchStats(), false
));

// add the prefix equivalence domination combinator
root_ptr = NodePtr(new DominanceCombinator2<PE_SOP>(
    root_ptr, search_manager, prefix_equivalence_store
));
```

We now define the tree search. We choose to use an iterative beam search with a starting width $D = 1$ and a growth factor of 2. Finally, we tell the search manager the search started (it starts the time measurement) and run the tree search algorithm.

main.cpp - Tree search algorithm definition

```
// construct the tree search algorithm
TreeSearchParameters ts_params = {.root = root_ptr, .manager = search_manager, .id = 0};
auto ts = IterativeBeamSearch(ts_params, 1, 2, true);

// start the search
search_manager.start();
ts.run(time_limit);
```

After the search ends (after the time limit or the tree search exhausted the search tree), we can print diverse information about the search. In this case, we print the different search statistics, the dominance informations and the performance profile informations in a json format.

main.cpp - getting search statistics

```
// display statistics and write performance profile file
search_manager.printStats();
prefix_equivalence_store.printStats();
search_manager.writeAnytimeCurve("perfprofile_ibs.json", "Iterative Beam Search");
```

Finally, we can compile and execute our program and observe the following output:

program output

```
./main.exe insts/R.700.1000.15.sop 60

sol nb      time   nb nodes    nodes/s  objective         algorithm    node name
```

```
     1     0.005        2 K       575 K    151.331        BS(id:0,1)    D2((forward))
     2     0.027        9 K       334 K    109.960        BS(id:0,2)    D2((forward))
     3     0.145       21 K       149 K     97.104        BS(id:0,4)    D2((forward))
     4     0.256       47 K       184 K     88.343        BS(id:0,8)    D2((forward))
     5     0.341      101 K       296 K     76.951       BS(id:0,16)    D2((forward))
     6     0.510      207 K       407 K     74.786       BS(id:0,32)    D2((forward))
     7     0.856      420 K       491 K     69.446       BS(id:0,64)    D2((forward))
     8     1.558      834 K       535 K     68.056      BS(id:0,128)    D2((forward))
     9     3.077        1 M       531 K     66.430      BS(id:0,256)    D2((forward))
    10     6.308        3 M       502 K     65.482      BS(id:0,512)    D2((forward))
    11    13.274        6 M       456 K     65.182     BS(id:0,1024)    D2((forward))
    12    29.564       11 M       385 K     65.011     BS(id:0,2048)    D2((forward))
========== Search statistics ===========
nb generated                20.748.134
nb expanded                 3.699.873
nb trashed                  0
nb pruned                   0
avg branching factor        5.61
nb evaluations              193
nb guide calls              463.707.272
searched                    60.0397 s
generated nodes / s         345.573
Primal                      65.011
==== Prefix Equivalence statistics =====
nb elements in store        1.657.577
nb prefix eq cuts           1.872.070
nb prefix eq stores         1.657.577
nb prefix eq accesses       3.914.366
nb PE exist tests           5.571.943
```

This example presented the workflow we currently use to solve combinatorial problems using the combinator-based framework. In this example, we discussed the preprocessing steps we performed and the way we defined the search tree. This (relatively) simple program contains approximately 200-300 lines of code and is able to obtain state-of-the-art performance (see Chapter 4 for more details about the state-of-the-art). Finally, we believe that this framework allows building a very effective tree search algorithm in a few days on many optimization problems. Moreover, it allows to quickly prototype various versions with different components. Lets us suppose we want to perform a comparison of various tree search algorithms (namely DFS, IBS, LDS, MBA*). We can simply replace the tree search procedure by another and run the search.

program output

```
// auto ts = DFS(ts_params);
auto ts = IterativeBeamSearch(ts_params, 1, 2, true);
// auto ts = IterativeMBAStar(ts_params, 1, 2);
// auto ts = LDS(ts_params);
```

We can then use a script provided in the framework that parses the performance profile json files for every tree search algorithm and display an image of it (Figure 4.5).

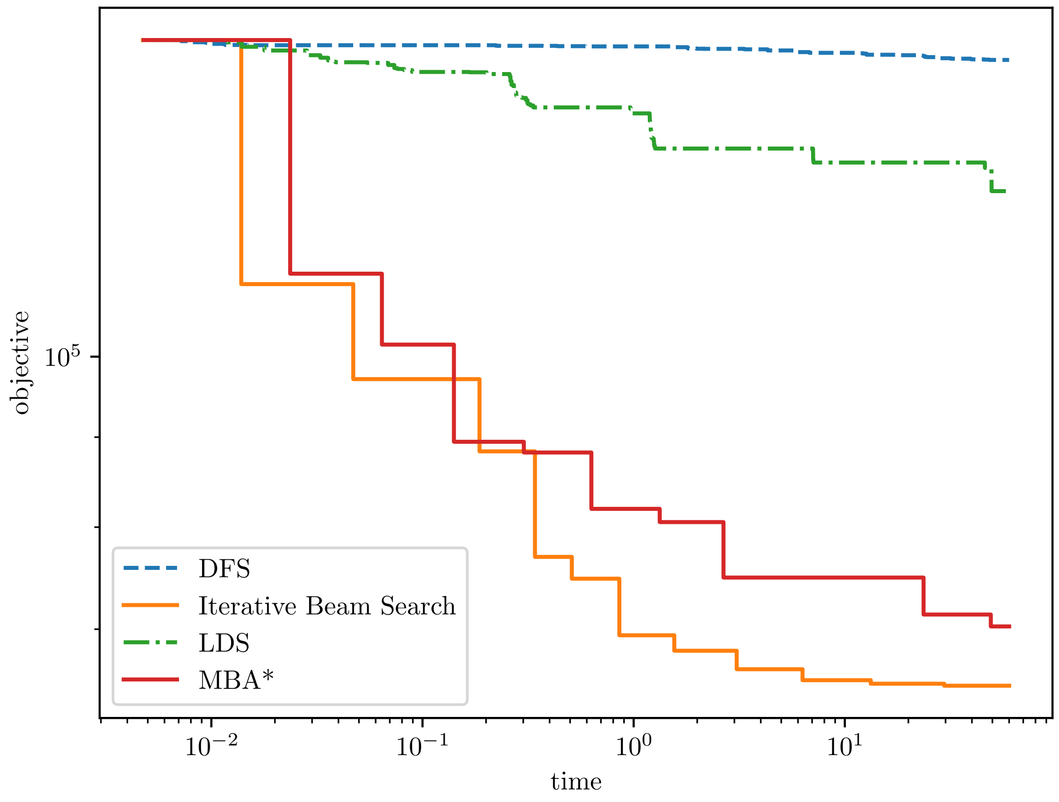


Figure 4.5: Tree search comparison on the R.700.1000.15 SOP instance

5 Conclusions & Perspectives

5.1 Main conclusions

In this thesis, we discussed a generic methodology to build anytime tree-search algorithms that aim to provide near-optimal solutions to large-scale instances. Such algorithms are built using ideas from classical branch-and-bound components from exact methods (using bounds, symmetry breaking and dominance schemes), integrating guidance strategies from constructive meta-heuristics and search strategies from AI/planning. Chapter 1 presented the different concepts found in classical branch-and-bounds, meta-heuristics and AI/Planning. Chapter 2 presented our generic implementation of anytime tree search algorithms and how one can integrate all of these concepts. Chapters 3 and 4 presented situations where anytime tree search algorithms proved to be efficient. Namely on the EURO/ROADEF challenge, an industrial Cutting & Packing problem and the sequential ordering problem, a well-known academic transportation problem[1].

"*A metaheuristic is a high-level* ***problem-independent*** *algorithmic framework that provides a set of guidelines or strategies to develop* ***heuristic optimization algorithms***." [SG13, SSG17].

Considering the meta-heuristic definitions in the literature (see an example above), two key points appear: They consist of a problem-independent methodology that could be applied to many optimization problems. They also are heuristic algorithms, that aim to provide near-optimal solutions in a reasonable time.

According to these points, anytime tree search algorithms are meta-heuristics (while also being branch-and-bounds). Indeed, it is a problem-independent formalism (where the search space is modeled as a tree) and aims to provide near-optimal solutions fast. Such a remark allows considering anytime tree search algorithms as branch-and-bounds and at the same time as meta-heuristics, two categories that are usually considered disjoint. They can easily benefit from both guidance functions found in constructive meta-heuristics and search-space reductions from exact methods (bounds, dominances *etc.*). As the studies presented in this thesis suggest, all these components play a major role in the algorithm performance.

[1]We may also note that similar algorithms have been applied to scheduling problems [STDC18, GMMT20].

We believe this representation of anytime tree search algorithms as meta-heuristics is rather new. We also believe that the representation of anytime tree search algorithms in the Operations Research literature does not reflect the benefits of applying such methods on complex optimization problems. To the best of our knowledge, many of them remain unexplored such as Beam Stack Search [ZH05], SMA* [Rus92] or Anytime Column Search [CGM$^+$18].

In this thesis, we discussed several problems in which tree search algorithms obtained state-of-the-art performance. We aim in this paragraph to draw general conclusions on when such algorithms are interesting and what would be the apriori most interesting algorithmic components. We may keep in mind that such general conclusions are meant to be a relatively good starting point to solve a combinatorial problem or if anytime tree search algorithms can be suited to tackle a problem. We advocate more research to validate (or invalidate) these conclusions on other problems. We already started to challenge these working hypotheses by implementing tree search algorithms on other combinatorial optimization problems. Namely the Longest Common Subsequence problem where we obtained competitive results and even some new best-so-far solutions, and on the Permutation flowshop.

The first (obvious) property that makes an anytime tree search good is the search-space size (*i.e.* the size of the tree). It is the average branching factor (average number of children per node) and the average depth of the tree. Search-space reductions techniques from exact methods can lead to a significant decrease in tree size as prefix equivalence dominances in Chapter 4 has proven to be a key component. However, as it is usually described in the Constraint Programming literature, a compromise has to be found regarding the node inference (reasoning work done within each node). The more reasoning, the smaller the search space, but also the fewer nodes opened. While considering anytime tree search algorithms, it seems that the computational-cost/search-space-reduction balance favors the fastest reasonings to improve the algorithm performance (computing a minimum spanning tree is already too costly on SOP). Some problems can be heavily constrained. For instance, SOPLIB: on open instances, our search tree nodes have 5 to 7 children on average even on the largest instances. The more the constraints, the more efficient the tree search algorithm will be (in contrast to local-search-based or classical MIP-based methods that tend to struggle on heavily constrained instances). However, on loosely constrained instances, the search tree becomes too large to maintain a good performance while local search gets much better solutions (thus the complementarity of both approaches) and MIP-based approaches have a very good guidance and search-tree-reduction thus being able to prove optimality on loosely constrained instances. However, this "search-tree size condition" is not sufficient to explain the success of tree search approaches.

It is common to use a bound to guide anytime branch-and-bound algorithms. This bound also allows to perform prunings. However, as we demonstrated in Chapter 3, using a bound to guide the search may lead to biases in the search (for instance, small items first where good solutions have some regularity within the average size) that leads to poor-quality solutions. In the other hand, many constructive meta-heuristics (for instance GRASP or ACO) use a heuristic guidance strategy. This guide helps the search to find better solutions, but cannot be used easily for pruning. Indeed, as constructive meta-heuristics usually perform local-search on newly-obtained solutions, a not-so-good solution may lead to an excellent one after local-search. Considering both classes of algorithms, it may be useful to distinguish two distinct mechanisms within branch-and-bounds. The bounds and the guides. Bounds are used to fathom nodes that are dominated by the best-so-far solution and the guidance used to help focusing on the most interesting region of the search-space. Doing so, excellent solutions can be obtained, even if the search-tree is relatively large. For instance, in many situations, the algorithm we designed for the EURO/ROADEF glass-cutting challenge obtained the best solutions on loosely constrained instances (with sometimes more than 100 children in average).

Another factor that could explain/predict the efficiency of an anytime tree search algorithm is the proportion of "local constraints". A local constraint can directly prune children nodes that violate the constraint, thus benefiting the algorithm by limiting the tree size. If the other hand, in some other cases, a constraint can be violated, but the algorithm would be able to tell only after many decisions. Thus, it could result in a large proportion of "useless" explored nodes, thus harming the ability to find feasible solutions. The algorithm we designed in Chapter 3 uses some symmetry-breaking schemes that prunes many nodes that are symmetric to some other nodes (some are more aggressive than others). It turned out that the most aggressive ones struggled to find good solutions because the tree search procedure spent too much time in parts of the tree that would be pruned by the symmetry-breaking scheme. The best option was to perform a less aggressive scheme that still reduces the search tree while not spending too much time on to-be-pruned parts of the tree. We may note that it is possible to integrate some no-good recording strategy to "learn" parts of the tree that would be pruned, thus improving the algorithm performance while using these symmetry-breaking schemes.

Finally, the search strategy also plays an important role. As we discussed in Chapter 4, while being an anytime algorithm, DFS does not often compete with classical meta-heuristics. However, just by replacing it by an iterative beam search, the resulting branch-and-bound can reach similar results as the Ant Colony Optimization + Simulated Annealing hybrid that found the previous best-known results on the SOPLIB. It is unlikely that there exists a universal anytime tree search algorithm that outperforms all the others. However, according to our experiments, iterative beam search appears to be a first good candidate as it obtains good results on many scenarios. To date, it is the best one for the SOP (Chapter 4) and is relatively performant on the glass-cutting challenge (Chapter 3), slightly behind iterative MBA*[2]. Even if iterative beam search may reopen nodes, it is guaranteed not to open too many nodes. In the worst case, it would reopen each node only once in average (see Chapter 1). Both methods we proposed only perform a very light inference and are able to open millions of nodes per second. Thus, reopening a node is inexpensive compared to the overhead induced by the data-structure (for instance, a binary heap) that maintains a memory of the opened nodes (usually $O(\ln n)$). In a context where opening a node costs much more, it may be interesting to consider variants of the iterative beam search that do not reopen nodes (for instance beam stack search [ZH05], or anytime column search [VGAC12]).

[2]As a side remark, MBA* seems to be less efficient than iterative beam search on many problems (except the glass-cutting challenge). The reasons why MBA* performs so well in this specific case is, for now, an open question.

5.2 Perspective and future research

In this thesis, we focused on anytime tree search algorithms to better assess their performance. They seem to obtain excellent results on some classes of problems and greatly complement perturbation-based methods where the instance contains many precedence constraints. The two types of methods can be combined. For instance, we may cite Recovering Beam Search [GP05, DCT02] that integrates a local search step on each node of the search tree. This local-search iteration allows us to “recover” from bad decisions taken before. This technique has shown to be efficient on various scheduling problems and generalizes the classical local-search executed on solutions found in constructive meta-heuristics as GRASP and Ant Colony Optimization.

A second combination with meta-heuristics could be to add an online-learning procedure within an anytime tree search. Indeed, as discussed in Chapter 1, Ant Colony Optimization performs a series of greedy randomized runs with a learning mechanism to take into account the information provided by the previous iterations. Such learning components can be integrated within another anytime tree search. For instance, Beam ACO [Blu05a] replaces the classical greedy randomized by a randomized Beam Search. This method allowed to obtain state-of-the-art algorithms on various optimization problems [Blu05a, Blu08, LIB10]. The combination of the learning part from ACO and anytime tree search can lead to new competitive methods. In the future, we may, for instance, see a new LDS-ACO or MBA*-ACO.

Another combination, this time with exact methods, could be to develop further interactions between anytime tree search and exact approaches. Indeed, while exact methods bounds are usually stronger and more expensive to compute, they provide a good space reduction. Thus, modifying the search strategy of MIP-based branch-and-bounds or Constraint Programming may build another efficient heuristics. Some steps have already been done in this direction. Indeed, an ant optimization framework has been used to solve constraint programming problems [Sol10]. More recently, LDS-based divings have been used combined within a branch & price to replace their greedy equivalents [SVP^{+}19] and using column-generation based heuristic procedures become more and more popular [CSVV19].

Regarding the Combinator-based Anytime Tree Search framework (Chapter 2), we believe that it could be an interesting tool to quickly prototype and develop anytime tree search algorithms. One obvious direction could be to integrate more components. We already implemented some Ant Colony Optimization algorithms within the framework to add a online-learning component to the algorithms we presented. The next step would be to evaluate the performance of the combination of Beam Search, MBA*, LDS, *etc.* with the ACO-combinator. Furthermore, in this thesis, we focused our implementations on anytime tree search algorithms. However, they can benefit from the hybridization with classical meta-heuristics. For instance by the combination of a genetic algorithm and local search [RLT12]. To this extent, we may develop some simple local-search or genetic algorithms within the framework or build wrappers for an existing framework (EasyLocal++ [DGS03] seems to be a good candidate as written in C++ and with a relatively close architecture). Moreover, implementing a specific tree search algorithm for a given problem takes time and expertise. Constraint Programming provides tools to easily model a problem and the resolution techniques are rather similar as they explore a tree. Thus, simply replacing the CP solver search strategy by an anytime tree search procedure would allow to get a relatively efficient algorithm. To this extent, Gecode [SLT06] seems to be a good candidate as it is written in C++ and uses copying, thus allowing good flexibility in the search. Finally, the concept of combinator may be used for other classes of algorithms. We may use combinators to modify the behavior of local-search algorithms. For instance, a *guided local search* combinator that allows to penalize solutions that were previously seen.

Bibliography

[AAB$^+$02] Enrique Alba, Francisco Almeida, M Blesa, J Cabeza, Carlos Cotta, Manuel Díaz, Isabel Dorta, Joaquim Gabarró, Coromoto León, J Luna, et al. Mallba: A library of skeletons for combinatorial optimisation. In *European Conference on Parallel Processing*, pages 927–932. Springer, 2002.

[Ach09] Tobias Achterberg. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[ACK07] Sandip Aine, PP Chakrabarti, and Rajeev Kumar. Awa*-a window constrained anytime heuristic search algorithm. In *IJCAI*, pages 2250–2255, 2007.

[ACV$^+$09] Filipe Alvelos, T. M. Chan, Paulo Vilaça, Tiago Gomes, Elsa Silva, and J. M. Valério de Carvalho. Sequence based heuristics for two-dimensional bin packing problems. *Engineering Optimization*, 41(8):773–791, August 2009.

[AEGS93] Norbert Ascheuer, Laureano F Escudero, Martin Grötschel, and Mechthild Stoer. A cutting plane approach to the sequential ordering problem (with applications to job scheduling in manufacturing). *SIAM Journal on Optimization*, 3(1):25–42, 1993.

[AHM09a] Hakim Akeb, Mhand Hifi, and Rym M'Hallah. A beam search algorithm for the circular packing problem. *Computers & Operations Research*, 36(5):1513–1528, 2009.

[AHM09b] Hakim Akeb, Mhand Hifi, and Rym M'Hallah. A beam search algorithm for the circular packing problem. *Computers & Operations Research*, 36(5):1513–1528, May 2009.

[AHM10] Hakim Akeb, Mhand Hifi, and Rym M'Hallah. Adaptive beam search lookahead algorithms for the circular packing problem. *International Transactions in Operational Research*, 17(5):553–575, 2010.

[AHN11] Hakim Akeb, Mhand Hifi, and Stéphane Negre. An augmented beam search-based algorithm for the circular open dimension problem. *Computers & Industrial Engineering*, 61(2):373–381, September 2011.

[AMPG11] Davide Anghinolfi, Roberto Montemanni, Massimo Paolucci, and Luca Maria Gambardella. A hybrid particle swarm optimization approach for the sequential ordering problem. *Computers & Operations Research*, 38(7):1076–1085, 2011.

[AMS20] Ignacio Araya, Mauricio Moyano, and Cristobal Sanchez. A beam search algorithm for the biobjective container loading problem. *European Journal of Operational Research*, March 2020.

[AR14] I. Araya and M. C. Riff. A beam search approach to the container loading problem. *Computers & Operations Research*, 43:100–107, March 2014.

[Asc96] Norbert Ascheuer. Hamiltonian path problems in the on-line optimization of flexible manufacturing systems. *Technical Report TR 96–3*, 1996.

[ASdC14] Filipe Alvelos, Elsa Silva, and José Manuel Valério de Carvalho. A Hybrid Heuristic Based on Column Generation for Two- and Three- Stage Bin Packing Problems. In Beniamino Murgante, Sanjay Misra, Ana Maria A. C. Rocha, Carmelo Torre, Jorge Gustavo Rocha, Maria Irene Falcão, David Taniar, Bernady O. Apduhan, and Osvaldo Gervasi, editors, *Computational Science and Its Applications – ICCSA 2014*, Lecture Notes in Computer Science, pages 211–226, Cham, 2014. Springer International Publishing.

[AVMTP07] Ramón Alvarez-Valdes, Rafael Martí, Jose M. Tamarit, and Antonio Parajón. GRASP and Path Relinking for the Two-Dimensional Two-Stage Cutting-Stock Problem. *INFORMS Journal on Computing*, 19(2):261–272, May 2007.

[AVPT02] Ramón Alvarez-Valdés, Antonio Parajón, and Jose'e Manuel Tamarit. A tabu search algorithm for large-scale guillotine (un)constrained two-dimensional cutting problems. *Computers & Operations Research*, 29(7):925–947, June 2002.

[B$^+$07] Jason Brownlee et al. Oat: The optimization algorithm toolkit. *Complex Intelligent Systems Laboratory (CIS), Centre for Information Technology Research (CITR), Faculty of Information and Communication Technologies (ICT), Swinburne University of Technology, Victoria, Australia, Technical Report A*, 20071220, 2007.

[BCD$^+$19] Nadia Brauner, Yves Crama, Etienne Delaporte, Vincent Jost, and Luc Libralesso. Do balanced words have a short period? *Theoretical Computer Science*, 793:169–180, 2019.

[BCMS18] J. A. Bennell, M. Cabo, and A. Martínez-Sykora. A beam search approach to solve the convex irregular bin packing problem with guillotine cuts. *European Journal of Operational Research*, 270(1):89–102, October 2018.

[BCPT14] Mauro Maria Baldi, Teodor Gabriel Crainic, Guido Perboli, and Roberto Tadei. Branch-and-price and beam search algorithms for the variable cost and size bin packing problem with optional items. *Annals of Operations Research*, 222(1):125–141, 2014.

[BD04] Christian Blum and Marco Dorigo. The hyper-cube framework for ant colony optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(2):1161–1172, 2004.

[Bea85] J. E. Beasley. Algorithms for Unconstrained Two-Dimensional Guillotine Cutting. *Journal of the Operational Research Society*, 36(4):297–306, April 1985.

[BG01] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[BHLR12] Ethan Andrew Burns, Matthew Hatem, Michael J Leighton, and Wheeler Ruml. Implementing fast heuristic search code. In *Fifth Annual Symposium on Combinatorial Search*, 2012.

[BHS97] Bernd Bullnheimer, Richard F Hartl, and Christine Strauss. A new rank based version of the ant system. a computational study. *Central European Journal of Operations Research*, 1997.

[BJ12] Andreas Bortfeldt and Sabine Jungmann. A tree search algorithm for solving the multi-dimensional strip packing problem with guillotine cutting constraint. *Annals of Operations Research*, 196(1):53–71, July 2012.

[BJJ14] Thierry Benoist, Antoine Jeanjean, and Vincent Jost. Call-based dynamic programming for the precedence constrained line traveling salesman. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 1–14. Springer, 2014.

[Blu05a] Christian Blum. Ant colony optimization: Introduction and recent trends. *Physics of Life reviews*, 2(4):353–373, 2005.

[Blu05b] Christian Blum. Beam-aco—hybridizing ant colony optimization with beam search: An application to open shop scheduling. *Computers & Operations Research*, 32(6):1565–1591, 2005.

[Blu08] Christian Blum. Beam-aco for simple assembly line balancing. *INFORMS Journal on Computing*, 20(4):618–627, 2008.

[BPW$^+$12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[BS10] Julia A Bennell and Xiang Song. A beam search implementation for the irregular shape packing problem. *Journal of Heuristics*, 16(2):167–188, 2010.

[BW87] J. O. Berkey and P. Y. Wang. Two-Dimensional Finite Bin-Packing Algorithms. *Journal of the Operational Research Society*, 38(5):423–429, May 1987.

[BW09] Andreas Bortfeldt and Tobias Winter. A genetic algorithm for the two-dimensional knapsack problem with rectangular pieces. *International Transactions in Operational Research*, 16(6):685–713, 2009.

[CBSS08] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008.

[CCTH15] Yi-Ping Cui, Yaodong Cui, Tianbing Tang, and Wei Hu. Heuristic for constrained two-dimensional three-staged patterns. *Journal of the Operational Research Society*, 66(4):647–656, April 2015.

[CF11] Christoforos Charalambous and Krzysztof Fleszar. A constructive bin-oriented heuristic for the two-dimensional bin packing problem with guillotine cuts. *Computers & Operations Research*, 38(10):1443–1451, October 2011.

[CFL01] Qun Chen, Michael C Ferris, and Jeff Linderoth. Fatcop 2.0: Advanced features in an opportunistic mixed integer programming solver. *Annals of Operations Research*, 103(1-4):17–32, 2001.

[CGH08] Y. Cui, T. Gu, and W. Hu. An algorithm for the constrained two-dimensional rectangular multiple identical large object placement problem. *Optimization Methods and Software*, 23(3):375–393, June 2008.

[CGM$^+$18] Liron Cohen, Matias Greco, Hang Ma, Carlos Hernandez, Ariel Felner, TK Satish Kumar, and Sven Koenig. Anytime focal search with applications. In *IJCAI*, pages 1434–1441, 2018.

[CGP12] Chetan Chauhan, Ravindra Gupta, and Kshitij Pathak. Survey of methods of solving tsp along with its implementation using dynamic programming approach. *International journal of computer applications*, 52(4), 2012.

[CHC00] V.-D. Cung, M. Hifi, and B. Le Cun. Constrained two-dimensional cutting stock problems a best-first branch-and-bound algorithm. *International Transactions in Operational Research*, 7(3):185–210, 2000.

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms.* MIT press, 2009.

[CMT04] Sébastien Cahon, Nordine Melab, and E-G Talbi. Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of heuristics*, 10(3):357–380, 2004.

[Cou06] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[CSVV19] François Clautiaux, Ruslan Sadykov, François Vanderbeck, and Quentin Viaud. Pattern-based diving heuristics for a two-dimensional guillotine cutting-stock problem with leftovers. *EURO Journal on Computational Optimization*, 7(3):265–297, 2019.

[CvH13] Andre A Cire and Willem-Jan van Hoeve. Multivalued decision diagrams for sequencing problems. *Operations Research*, 61(6):1411–1428, 2013.

[CW77] Nicos Christofides and Charles Whitlock. An Algorithm for Two-Dimensional Cutting Problems. *Operations Research*, 25(1):30–44, February 1977.

[CYC13] Yaodong Cui, Liu Yang, and Qiulian Chen. Heuristic for the rectangular strip packing problem with rotation of items. *Computers & Operations Research*, 40(4):1094–1099, April 2013.

[CYC16] Yaodong Cui, Yi Yao, and Yi-Ping Cui. Hybrid approach for the two-dimensional bin packing problem with two-staged patterns. *International Transactions in Operational Research*, 23(3):539–549, 2016.

[CYZ18] Yi-Ping Cui, Yi Yao, and Defu Zhang. Applying triple-block patterns in solving the two-dimensional bin packing problem. *Journal of the Operational Research Society*, 69(3):402–415, March 2018.

[CZ13] Yaodong Cui and Zhigang Zhao. Heuristic for the rectangular two-dimensional single stock size cutting stock problem with two-staged patterns. *European Journal of Operational Research*, 231(2):288–298, December 2013.

[CZC17] Yi-Ping Cui, Yongwu Zhou, and Yaodong Cui. Triple-solution approach for the strip packing problem with two-staged patterns. *Journal of Combinatorial Optimization*, 34(2):588–604, August 2017.

[DCGT04] Federico Della Croce, Marco Ghirardi, and Roberto Tadei. Recovering beam search: Enhancing the beam search approach for combinatorial optimization problems. *Journal of Heuristics*, 10(1):89–104, 2004.

[DCT02] Federico Della Croce and Vincent T'kindt. A recovering beam search algorithm for the one-machine dynamic total completion time scheduling problem. *Journal of the Operational Research Society*, 53(11):1275–1280, 2002.

[DG97] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation*, 1(1):53–66, 1997.

[DGS03] Luca Di Gaspero and Andrea Schaerf. Writing local search algorithms using easylocal++. In *Optimization Software Class Libraries*, pages 155–175. Springer, 2003.

[DLCCR06] A Djerrah, Bertrand Le Cun, V-D Cung, and Catherine Roucairol. Bob++: Framework for solving optimization problems with branch-and-bound methods. In *2006 15th IEEE International Conference on High Performance Distributed Computing*, pages 369–370. IEEE, 2006.

[DLM12] Mohammad Dolatabadi, Andrea Lodi, and Michele Monaci. Exact algorithms for the two-dimensional guillotine knapsack. *Computers & Operations Research*, 39(1):48–53, January 2012.

[DMC91] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The ant system: An autocatalytic optimizing process. *Technical Report*, 1991.

[DMC$^+$96] Marco Dorigo, Vittorio Maniezzo, Alberto Colorni, et al. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, man, and cybernetics, Part B: Cybernetics*, 26(1):29–41, 1996.

[dNdQJ19] Oliviana Xavier do Nascimento, Thiago Alves de Queiroz, and Leonardo Junqueira. A MIP-CP based approach for two- and three-dimensional cutting problems with staged guillotine cuts. *Annals of Operations Research*, November 2019.

[DRB19] Marko Djukanovic, Günther R Raidl, and Christian Blum. Anytime algorithms for the longest common palindromic subsequence problem. *Computers & Operations Research*, page 104827, 2019.

[EGM94] Laureano F Escudero, Monique Guignard, and Kavindra Malik. A lagrangian relax-and-cut approach for the sequential ordering problem with precedence relationships. *Annals of Operations Research*, 50(1):219–237, 1994.

[EPH01] Jonathan Eckstein, Cynthia A Phillips, and William E Hart. Pico: An object-oriented framework for parallel branch and bound. In *Studies in Computational Mathematics*, volume 8, pages 219–265. Elsevier, 2001.

[ES02] Stefan Edelkamp and Patrick Stiegeler. Implementing heapsort with (n log n-0.9 n) and quicksort with (n log n+ 0.2 n) comparisons. *Journal of Experimental Algorithmics (JEA)*, 7:5, 2002.

[ES11] Stefan Edelkamp and Stefan Schroedl. *Heuristic search: theory and applications*. Elsevier, 2011.

[Esc88] Laureano F Escudero. An inexact algorithm for the sequential ordering problem. *European Journal of Operational Research*, 37(2):236–249, 1988.

[EW14] Stefan Edelkamp and Armin Weiß. Quickxsort: Efficient sorting with n logn- 1.399n+ o (n) comparisons on average. In *International Computer Science Symposium in Russia*, pages 139–152. Springer, 2014.

[FHZ98] D. Fayard, M. Hifi, and V. Zissimopoulos. An efficient approach for large-scale two-dimensional guillotine cutting stock problems. *Journal of the Operational Research Society*, 49(12):1270–1277, December 1998.

[FK05] David Furcy and Sven Koenig. Limited discrepancy beam search. In *IJCAI*, pages 125–131, 2005.

[FL20a] Florian Fontan and Luc Libralesso. PackingSolver: a solver for Packing Problems. *arXiv:2004.02603 [cs]*, April 2020. arXiv: 2004.02603.

[FL20b] Florian Fontan and Luc Libralesso. PackingSolver: a tree search-based solver for two-dimensional two-and three-staged guillotine packing problems. working paper or preprint, April 2020.

[Fle13] Krzysztof Fleszar. Three insertion heuristics and a justification improvement heuristic for two-dimensional bin packing with guillotine cuts. *Computers & Operations Research*, 40(1):463–474, January 2013.

[FMT16] Fabio Furini, Enrico Malaguti, and Dimitri Thomopulos. Modeling Two-Dimensional Guillotine Cutting Problems via Integer Programming. *INFORMS Journal on Computing*, 28(4):736–751, October 2016.

[Fon19] Florian Fontan. *Theoretical and practical contributions to star observation scheduling problems.* PhD thesis, November 2019. Publication Title: http://www.theses.fr.

[FR02] Paola Festa and Mauricio GC Resende. Grasp: An annotated bibliography. In *Essays and surveys in metaheuristics*, pages 325–367. Springer, 2002.

[FS97] Sándor P. Fekete and Jörg Schepers. A new exact algorithm for general orthogonal d-dimensional knapsack problems. In Rainer Burkard and Gerhard Woeginger, editors, *Algorithms — ESA '97*, Lecture Notes in Computer Science, pages 144–156, Berlin, Heidelberg, 1997. Springer.

[FTP92] Marie T Fiala Timlin and William R Pulleyblank. Precedence constrained routing and helicopter scheduling: heuristic design. *Interfaces*, 22(3):100–111, 1992.

[GBD$^+$14] Frédéric Gardi, Thierry Benoist, Julien Darlay, Bertrand Estellon, and Romain Megel. *Mathematical programming solver based on local search.* Wiley Online Library, 2014.

[GD95] Luca M Gambardella and Marco Dorigo. Ant-q: A reinforcement learning approach to the traveling salesman problem. In *Machine Learning Proceedings 1995*, pages 252–260. Elsevier, 1995.

[GD97] Luca Maria Gambardella and Marco Dorigo. Has-sop: Hybrid ant system for the sequential ordering problem. *Technical Report IDSIA 11-97*, 1997.

[GG65] P. C. Gilmore and R. E. Gomory. Multistage Cutting Stock Problems of Two and More Dimensions. *Operations Research*, 13(1):94–120, February 1965.

[GKS$^+$12] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michele Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer go: Monte carlo tree search and extensions. *Communications of the ACM*, 55(3):106–113, 2012.

[GL06] Wasu Glankwamdee and JT Linderoth. Mw: A software framework for combinatorial optimization on computational grids. *Parallel Combinatorial Optimization*, 58:239, 2006.

[GM03] Francesca Guerriero and Marco Mancini. A cooperative parallel rollout algorithm for the sequential ordering problem. *Parallel Computing*, 29(5):663–677, 2003.

[GMMT20] Jan Gmys, Mohand Mezmaz, Nouredine Melab, and Daniel Tuyttens. A computationally efficient branch-and-bound algorithm for the permutation flow-shop scheduling problem. *European Journal of Operational Research*, 2020.

[GMW12] Luca Maria Gambardella, Roberto Montemanni, and Dennis Weyland. An enhanced ant colony system for the sequential ordering problem. In *Operations Research Proceedings 2011*, pages 355–360. Springer, 2012.

[GP05] Marco Ghirardi and Chris N Potts. Makespan minimization for scheduling unrelated parallel machines: A recovering beam search approach. *European Journal of Operational Research*, 165(2):457–467, 2005.

[GR15] Luis Gouveia and Mario Ruthmair. Load-dependent and precedence-based models for pickup and delivery problems. *Computers & Operations Research*, 63:56–71, 2015.

[GRB15] Uli Golle, Franz Rothlauf, and Nils Boysen. Iterative beam search for car sequencing. *Annals of Operations Research*, 226(1):239–254, 2015.

[GS11] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[Hel17] Keld Helsgaun. An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems. *Roskilde: Roskilde University*, 2017.

[Her04] István T Hernádvölgyi. Solving the sequential ordering problem with automatically generated lower bounds. In *Operations Research Proceedings 2003*, pages 355–362. Springer, 2004.

[HG95] William D Harvey and Matthew L Ginsberg. Limited discrepancy search. In *IJCAI (1)*, pages 607–615, 1995.

[HHLBS09] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[Hif97] Mhand Hifi. An improvement of viswanathan and bagchi's exact algorithm for constrained two-dimensional cutting stock. *Computers & Operations Research*, 24(8):727–736, August 1997.

[HM09] Mhand Hifi and Rym M'Hallah. Beam search and non-linear programming tools for the circular packing problem. *International Journal of Mathematics in Operational Research*, 1(4):476–503, January 2009.

[HMS08] Mhand Hifi, Rym M'Hallah, and Toufik Saadi. Algorithms for the Constrained Two-Staged Two-Dimensional Cutting Problem. *INFORMS Journal on Computing*, 20(2):212–221, January 2008.

[HNOS12] Mhand Hifi, Stephane Negre, Rachid Ouafi, and Toufik Saadi. A parallel algorithm for constrained two-staged two-dimensional cutting problems. *Computers & Industrial Engineering*, 62(1):177–189, February 2012.

[HNR68] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Hop00] Eva Hopper. *Two-dimensional packing utilising evolutionary algorithms and other meta-heuristic methods.* PhD Thesis, University of Wales. Cardiff, 2000.

[HR01] Mhand Hifi and Catherine Roucairol. Approximate and Exact Algorithms for Constrained (Un) Weighted Two-dimensional Two-staged Cutting Stock Problems. *Journal of Combinatorial Optimization*, 5(4):465–494, December 2001.

[HT01] E Hopper and B. C. H Turton. An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem. *European Journal of Operational Research*, 128(1):34–57, January 2001.

[HZ07] Eric A Hansen and Rong Zhou. Anytime heuristic search. *Journal of Artificial Intelligence Research*, 28:267–297, 2007.

[JSP$^+$17] J Jamal, G Shobaki, Vassilis Papapanagiotou, Luca Maria Gambardella, and Roberto Montemanni. Solving the sequential ordering problem using branch and bound. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–9. IEEE, 2017.

[KL02] Sven Koenig and Maxim Likhachev. D^* lite. *Aaai/iaai*, 15, 2002.

[Kor85] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[Kor93] Richard E Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.

[Kor96] Richard E Korf. Improved limited discrepancy search. In *AAAI/IAAI, Vol. 1*, pages 286–291, 1996.

[Krö95] Berthold Kröger. Guillotineable bin packing: A genetic approach. *European Journal of Operational Research*, 84(3):645–661, 1995.

[Lan92] Pat Langley. Systematic and nonsystematic search strategies. In *Artificial Intelligence Planning Systems*, pages 145–152. Elsevier, 1992.

[LBCJ20] Luc Libralesso, Abdel-Malik Bouhassoun, Hadrien Cambazard, and Vincent Jost. Tree searches for the Sequential Ordering Problem. working paper or preprint, January 2020.

[LF20a] Luc Libralesso and Florian Fontan. An anytime tree search algorithm for the 2018 ROADEF/EURO challenge glass cutting problem. working paper or preprint, April 2020.

[LF20b] Luc Libralesso and Florian Fontan. An anytime tree search algorithm for the 2018 ROADEF/EURO challenge glass cutting problem. *arXiv:2004.00963 [cs]*, April 2020. arXiv: 2004.00963.

[LIB10] Manuel López-Ibáñez and Christian Blum. Beam-aco for the travelling salesman problem with time windows. *Computers & operations research*, 37(9):1570–1583, 2010.

[Lib20] Luc Libralesso. Mixed Integer Programming formulations for the balanced Traveling Salesman Problem with a lexicographic objective. working paper or preprint, May 2020.

[LISD16] Manuel López-Ibáñez, Thomas Stützle, and Marco Dorigo. Ant colony optimization: A component-wise overview. *Handbook of heuristics*, pages 1–37, 2016.

[LJS$^+$19] Luc Libralesso, Vincent Jost, Khadija Hadj Salem, Florian Fontan, and Frédéric Maffray. Study on partial flexible job-shop scheduling problem under tooling constraints: complexity and related problems. 2019.

[LM03] Andrea Lodi and Michele Monaci. Integer linear programming models for 2-staged two-dimensional Knapsack problems. *Mathematical Programming*, 94(2):257–278, January 2003.

[LMP17] Andrea Lodi, Michele Monaci, and Enrico Pietrobuoni. Partial enumeration algorithms for Two-Dimensional Bin Packing Problem with guillotine constraints. *Discrete Applied Mathematics*, 217:40–47, January 2017.

[LMV04] Andrea Lodi, Silvano Martello, and Daniele Vigo. Models and Bounds for Two-Dimensional Level Packing Problems. *Journal of Combinatorial Optimization*, 8(3):363–379, September 2004.

[LO95] Gilbert Laporte and Ibrahim H Osman. Routing problems: A bibliography. *Annals of Operations Research*, 61(1):227–262, 1995.

[Man99] Vittorio Maniezzo. Exact and approximate nondeterministic tree-search procedures for the quadratic assignment problem. *INFORMS journal on computing*, 11(4):358–369, 1999.

[Mar98] Ambros Marzetta. *ZRAM: A library of parallel search algorithms and its use in enumeration and combinatorial optimization.* Citeseer, 1998.

[MMDC$^+$12] Marco Mojana, Roberto Montemanni, Gianni Di Caro, Luca M Gambardella, and P Luangpaiboon. A branch and bound approach for the sequential ordering problem. *Lecture Notes in Management Science*, 4:266–273, 2012.

[MP10] Reinaldo Morabito and Vitória Pureza. A heuristic approach based on dynamic programming and and/or-graph search for the constrained two-dimensional guillotine cutting problem. *Annals of Operations Research*, 179(1):297–315, September 2010.

[MSS99] Joao P Marques-Silva and Karem A Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[MSVH18] Laurent Michel, Pierre Schaus, and Pascal Van Hentenryck. Minicp: A lightweight solver for constraint programming, 2018.

[MSZ$^+$17] David R Morrison, Jason J Sauppe, Wenda Zhang, Sheldon H Jacobson, and Edward C Sewell. Cyclic best first search: Using contours to guide branch-and-bound algorithms. *Naval Research Logistics (NRL)*, 64(1):64–82, 2017.

[MV98] Silvano Martello and Daniele Vigo. Exact Solution of the Two-Dimensional Finite Bin Packing Problem. *Management Science*, 44(3):388–399, March 1998.

[NPC08] Napoleão Nepomuceno, Plácido Pinheiro, and André L. V. Coelho. A Hybrid Optimization Framework for Cutting and Packing Problems. In Carlos Cotta and Jano van Hemert, editors, *Recent Advances in Evolutionary Computation for Combinatorial Optimization*, Studies in Computational Intelligence, pages 87–99. Springer, Berlin, Heidelberg, 2008.

[OF90] JoséFernando Oliveira and JoséSoeiro Ferreira. An improved version of Wang's algorithm for two-dimensional cutting problems. *European Journal of Operational Research*, 44(2):256–266, January 1990.

[OM88] Peng Si Ow and Thomas E Morton. Filtered beam search in scheduling. *The International Journal Of Production Research*, 26(1):35–62, 1988.

[Poh70] Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial intelligence*, 1(3-4):193–204, 1970.

[PR07] Jakob Puchinger and Günther R. Raidl. Models and algorithms for three-stage two-dimensional bin packing. *European Journal of Operational Research*, 183(3):1304–1327, December 2007.

[PRCLF12] José Antonio Parejo, Antonio Ruiz-Cortés, Sebastián Lozano, and Pablo Fernandez. Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing*, 16(3):527–561, 2012.

[PRG$^+$03] José Antonio Parejo, Jesús Racero, Fernando Guerrero, Terence Kwok, and Kate A Smith. Fom: A framework for metaheuristic optimization. In *International Conference on Computational Science*, pages 886–895. Springer, 2003.

[R$^+$77] D Raj Reddy et al. Speech understanding systems: A summary of results of the five-year research effort. department of computer science, 1977.

[RG05] Ted K Ralphs and Menal Güzelsoy. The symphony callable library for mixed integer programming. In *The next wave in computing, optimization, and decision technologies*, pages 61–76. Springer, 2005.

[RLT12] Mohamed Ali Rakrouki, Talel Ladhari, and Vincent T'kindt. Coupling genetic local search and recovering beam search algorithms for minimizing the total completion time in the single machine scheduling problem subject to release dates. *Computers & Operations Research*, 39(6):1257–1264, 2012.

[RM94] Alexander Reinefeld and T. Anthony Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.

[RR16] Mauricio GC Resende and Celso C Ribeiro. *Optimization by GRASP*. Springer, 2016.

[RTR10] Silvia Richter, Jordan Tyler Thayer, and Wheeler Ruml. The joy of forgetting: Faster anytime search via restarting. In *ICAPS*, pages 137–144, 2010.

[Rus92] Stuart Russell. Efficient memory-bounded search methods. *ECAI-1992, Vienna, Austria*, 1992.

[Sal02] Matthew J Saltzman. Coin-or: an open-source library for optimization. In *Programming languages and systems in computational economics and finance*, pages 3–32. Springer, 2002.

[SAVdC10] Elsa Silva, Filipe Alvelos, and J. M. Valério de Carvalho. An integer programming model for two- and three-stage two-dimensional cutting stock problems. *European Journal of Operational Research*, 205(3):699–708, September 2010.

[SB99] Ihsan Sabuncuoglu and M Bayiz. Job shop scheduling with beam search. *European Journal of Operational Research*, 118(2):390–412, 1999.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SC99] Francis Sourd and Philippe Chrétienne. Fiber-to-object assignment heuristics. *European Journal of Operational Research*, 117(1):1–14, 1999.

[SE05] Niklas Sorensson and Niklas Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005(53):1–2, 2005.

[SFIH98] Yuji Shinano, Tetsuya Fujie, Yoshiko Ikebe, and Ryuichi Hirabayashi. Solving the maximum clique problem using pubb. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 326–332. IEEE, 1998.

[SG13] Kenneth Sörensen and Fred Glover. Metaheuristics. *Encyclopedia of operations research and management science*, 62:960–970, 2013.

[SGV04] Sven Spieckermann, Kai Gutenschwager, and Stefan Voß. A sequential ordering problem in automotive paint shops. *International journal of production research*, 42(9):1865–1878, 2004.

[SH98] Thomas Stützle and Holger Hoos. Improvements on the ant-system: Introducing the max-min ant system. In *Artificial neural nets and genetic algorithms*, pages 245–249. Springer, 1998.

[SJ15] Ghassan Shobaki and Jafar Jamal. An exact algorithm for the sequential ordering problem and its application to switching energy minimization in compilers. *Computational Optimization and Applications*, 61(2):343–372, 2015.

[Ski17] Rafał Skinderowicz. An improved ant colony system for the sequential ordering problem. *Computers & Operations Research*, 86:1–17, 2017.

[SLT06] Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode, 2006.

[SM03] Dong-Il Seo and Byung-Ro Moon. A hybrid genetic algorithm based on complete graph representation for the sequential ordering problem. In *Genetic and Evolutionary Computation Conference*, pages 669–680. Springer, 2003.

[Sol08] Christine Solnon. Combining two pheromone structures for solving the car sequencing problem with ant colony optimization. *European Journal of Operational Research*, 191(3):1043–1055, 2008.

[Sol10] Christine Solnon. *Ant colony optimization and constraint programming*. Wiley Online Library, 2010.

[Sör15] Kenneth Sörensen. Metaheuristics—the metaphor exposed. *International Transactions in Operational Research*, 22(1):3–18, 2015.

[SSG17] Kenneth Sorensen, Marc Sevaux, and Fred Glover. A history of metaheuristics. *arXiv preprint arXiv:1704.00853*, 2017.

[STDC18] Lei Shang, Vincent T'Kindt, and Federico Della Croce. The memorization paradigm: Branch & memorize algorithms for the efficient solution of sequencing problems. *preprint*, 2018.

[SVP$^+$19] Ruslan Sadykov, François Vanderbeck, Artur Pessoa, Issam Tahiri, and Eduardo Uchoa. Primal heuristics for branch and price: The assets of diving methods. *INFORMS Journal on Computing*, 31(2):251–267, 2019.

[TDCE04] Vincent T'kindt, Federico Della Croce, and Carl Esswein. Revisiting branch and bound search strategies for machine scheduling problems. *Journal of Scheduling*, 7(6):429–440, 2004.

[Ter04] Fabrice Tercinet. *Méthodes arborescentes pour la résolution des problèmes d'ordonnancement, conception d'un outil d'aide au développement.* PhD thesis, Tours, 2004.

[TH95] Stefan Tschöke and Norbert Holthöfer. A new parallel approach to the constrained two-dimensional cutting stock problem. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems*, Lecture Notes in Computer Science, pages 285–300, Berlin, Heidelberg, 1995. Springer.

[TJ10] Anne M Taylor and Noo Li Jeon. Micro-scale and microfluidic devices for neurobiology. *Current opinion in neurobiology*, 20(5):640–647, 2010.

[TP95] Stefan Tschöke and Thomas Polzer. Portable parallel branch-and-bound library: User manual. *Technical Report*, 500, 1995.

[VAC16] Satya Gautam Vadlamudi, Sandip Aine, and Partha Pratim Chakrabarti. Anytime pack search. *Natural Computing*, 15(3):395–414, 2016.

[VDBSHG11] Jur Van Den Berg, Rajat Shah, Arthur Huang, and Ken Goldberg. Anytime nonparametric a. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[VFD05] Stefan Voßs, Andreas Fink, and Cees Duin. Looking ahead with the pilot method. *Annals of Operations Research*, 136(1):285–302, 2005.

[VGAC12] Satya Gautam Vadlamudi, Piyush Gaurav, Sandip Aine, and Partha Pratim Chakrabarti. Anytime column search. In *Australasian Joint Conference on Artificial Intelligence*, pages 254–265. Springer, 2012.

[VST05] François Vanderbeck, R Sadykov, and I Tahiri. Bapcod–a generic branch-and-price code. *See http://wiki. bordeaux. inria. fr/realopt*, 2005.

[VU19] André Soares Velasco and Eduardo Uchoa. Improved state space relaxation for constrained two-dimensional guillotine cutting problems. *European Journal of Operational Research*, 272(1):106–120, January 2019.

[Wal97] Toby Walsh. Depth-bounded discrepancy search. In *IJCAI*, volume 97, pages 1388–1393, 1997.

[Wan83] P. Y. Wang. Two Algorithms for Constrained Two-Dimensional Cutting Stock Problems. *Operations Research*, 31(3):573–586, June 1983.

[Wil10] Christopher Wilt. Informed backtracking beam search. *Conference proceedings*, 2010.

[WL15] Lijun Wei and Andrew Lim. A bidirectional building approach for the 2D constrained guillotine knapsack packing problem. *European Journal of Operational Research*, 242(1):63–71, April 2015.

[WLZ13] Ning Wang, Andrew Lim, and Wenbin Zhu. A multi-round partial beam search approach for the single container loading problem with shipment priority. *International Journal of Production Economics*, 145(2):531–540, October 2013.

[WTZL14] Lijun Wei, Tian Tian, Wenbin Zhu, and Andrew Lim. A block-based layer building approach for the 2D guillotine strip packing problem. *European Journal of Operational Research*, 239(1):58–69, November 2014.

[WWKT14] Moon Hong Wun, Li-Pei WongT, Ahamad Tajudin Khader, and Tien-Ping Tan. A bee colony optimization with automated parameter tuning for sequential ordering problem. In *2014 4th World Congress on Information and Communication Technologies (WICT 2014)*, pages 314–319. IEEE, 2014.

[XHHLB08] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008.

[XRLS05] Yan Xu, Ted K Ralphs, Laszlo Ladányi, and Matthew J Saltzman. Alps: A framework for implementing parallel tree search algorithms. In *The next wave in computing, optimization, and decision technologies*, pages 319–334. Springer, 2005.

[ZH05] Rong Zhou and Eric A Hansen. Beam-stack search: Integrating backtracking with beam search. In *ICAPS*, pages 90–98, 2005.

[Zha98] Weixiong Zhang. Complete anytime beam search. In *AAAI/IAAI*, pages 425–430, 1998.

A

PackingSolver: a solver for guillotine packing problems

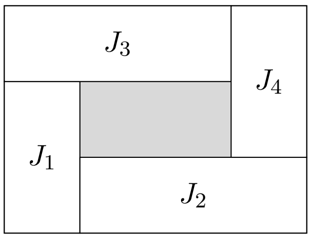
This chapter presents a submitted article on an academic/industrial continuation of the EURO/ROADEF 2018 challenge with *Florian Fontan.* We investigate various guillotine *Cutting & Packing* problems and evaluate the performance of the ideas presented in Chapter 3.

In Chapter 3, we proposed an anytime tree search algorithm for the 2018 ROADEF/EURO challenge glass cutting problem[1]. The resulting program was ranked first among 64 participants. In this article, we generalize it and show that it is not only effective for the specific problem it was originally designed for, but is also very competitive and even returns state-of-the-art solutions on a large variety of Cutting and Packing problems from the literature. We adapted the algorithm for two-dimensional Bin Packing, Multiple Knapsack, and Strip Packing Problems, with two- or three-staged exact or non-exact guillotine cuts, the orientation of the first cut being imposed or not, and with or without item rotation. The combination of efficiency, ability to provide good solutions fast, simplicity and versatility makes it particularly suited for industrial applications, which require quickly developing algorithms implementing several business-specific constraints. The algorithm is implemented in a new software package called PackingSolver.

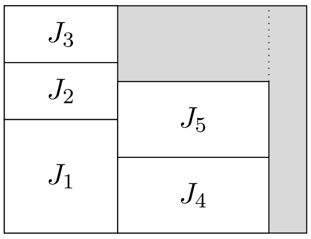
The 2018 ROADEF/EURO challenge featured an industrial glass cutting problem arising at the French company Saint Gobain. We proposed an anytime tree search algorithm that was ranked first in the final phase of the challenge. They showed that the algorithm performs very well on this specific variant with the specific instances considered. Indeed, some of the industrial constraints of the problem seem to favor this kind of constructive approach. In particular, the problem includes precedence constraints, which highly penalize other approaches such as local search, dynamic programming, mixed-integer linear programming or column generation. Therefore, it was not obvious *a priori* whether the algorithm would be competitive on other variants. In this article, we show that even on pure Packing Problems from the literature, it is competitive compared to the other dedicated algorithms, and is even able to return state-of-the-art solutions on several variants.

Even though most of the new constraints taken into account integrate naturally within the algorithm, several improvements need to be made to make it efficient on the large variety of problems and instances from the literature: two new guide functions are proposed to deal with instances with different item distributions; an additional guide is designed for

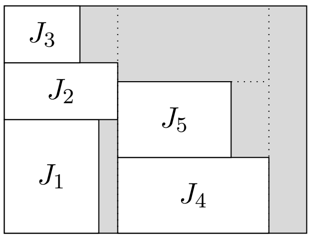
[1]`https://www.roadef.org/challenge/2018/en/index.php`



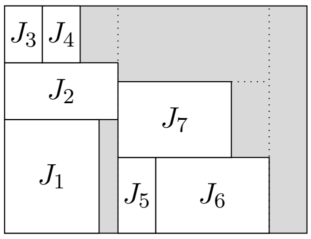
(a) Non-guillotine pattern



(b) Two-staged exact guillotine pattern, first stage vertical



(c) Two-staged non-exact guillotine pattern, first stage vertical



(d) Three-staged exact guillotine pattern, first stage vertical

Figure A.1: Pattern type examples

the Knapsack objective; and some flexibility has been introduced in the symmetry breaking strategy.

We proposed an efficient algorithm for a specific problem with specific constraints and instances. Here, we propose an efficient approach which should be useful for almost any (guillotine for now) Packing Problem. Also, as discussed in Section A.5, experimenting on all these variants greatly improved our understanding of the effectiveness of MBA* and other tree search algorithms.

A.1 Introduction

We consider two-dimensional guillotine Packing Problems: one has to pack rectangles of various sizes into larger bins while only edge-to-edge cuts are allowed. In a solution, guillotine cuts can be partitioned into stages, *i.e.* series of parallel cuts, and it is common to limit the number of allowed stages. Here, we restrict to two- or three-staged guillotine patterns. In both cases, we consider both exact and non-exact variants. In the non-exact variant, an additional cut is allowed to separate items from waste. Figure A.1 illustrates the different pattern types.

We consider the three main objectives studied in the literature: Bin Packing, Knapsack and Strip Packing. In Bin Packing and Strip Packing Problems, all items need to be produced. In Bin Packing Problems, the number of used bins is minimized, while in Strip Packing Problems, there is only one container with one infinite dimension and the objective is to minimize the length used in this dimension. In Knapsack Problems, the number of containers is limited, every item has a profit and the total profit of the packed items is maximized.

Finally, for each variant, we consider the oriented case where item rotation is not allowed and non-oriented case where it is.

Throughout the article, the different variants are named following our notations illustrated with the following examples:

- BPP-O: (non-guillotine) Bin Packing Problem, Oriented
- G-BPP-R: Guillotine cuts, Bin Packing Problem, Rotation
- 2G-KP-O: 2-staged exact guillotine cuts, first cut horizontal or vertical, Knapsack Problem, Oriented
- 3NEGH-SPP-O: 3-staged non-exact guillotine cuts, first cut horizontal, Strip Packing Problem, Oriented

We also use the following vocabulary: a k-cut is a cut performed in the k-th stage. Cuts separate bins into k-th level sub-plates. For example, 1-cuts separate the bin in several first level sub-plates. S denotes a solution or a node in the search tree (a partial solution).

The following definitions are given for the case where the first cut in the last bin is vertical, but naturally, adapt to the case where it is horizontal. We call the last first level sub-plate, the rightmost one containing an item; the last second level sub-plate, the topmost one containing an item in the last first level sub-plate; and the last third level sub-plate the rightmost one containing an item in the last second level sub-plate. $x_1^{\text{prev}}(S)$ and $x_1^{\text{curr}}(S)$ are the left and right coordinates of the last first level sub-plate; $y_2^{\text{prev}}(S)$ and $y_2^{\text{curr}}(S)$ are the bottom and top coordinates of the last second level sub-plate; and $x_3^{\text{prev}}(S)$ and $x_3^{\text{curr}}(S)$ are the left and right coordinates of the last third level sub-plate. Figure A.2 presents a usage example of these definitions. We define the area and the waste of a solution S as follows:

$$
\begin{aligned}
\text{area}(S) &= \begin{cases} A + x_1^{\text{curr}}(S)h & \text{if } S \text{ contains all items} \\ A + x_1^{\text{prev}}(S)h \\ \quad + (x_1^{\text{curr}}(S) - x_1^{\text{prev}}(S))y_2^{\text{prev}}(S) \\ \quad + (x_3^{\text{curr}}(S) - x_1^{\text{prev}}(S))(y_2^{\text{curr}}(S) - y_2^{\text{prev}}(S)) & \text{otherwise} \end{cases} \\
\text{waste}(S) &= \text{area}(S) - \text{item\_area}(S)
\end{aligned}
$$

with A the sum of the areas of all but the last bin, h the height of the last bin and item_area(S) the sum of the area of the items of S. Area and waste are illustrated in Figure A.2.

A.2 Literature review

Two-dimensional guillotine Packing Problems have been introduced by [GG65] and have received a lot of attention since. Researchers usually focus on one specific variant or only on a few ones.

Algorithms are sometimes adapted for both the oriented and the non-oriented cases. [VU19] developed a heuristic for G-KP-O and G-KP-R, [WTZL14] for G-SPP-O and G-SPP-R, [CF11], [Fle13] and [CYZ18] for G-BPP-O and G-BPP-R, [LM03] an exact algorithm for 2NEGH-KP-O and 2NEGH-KP-R.

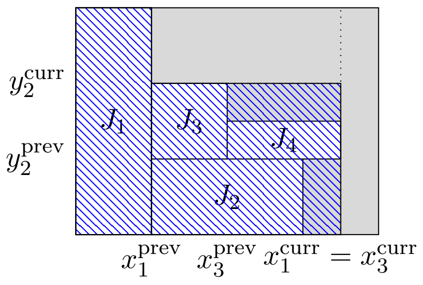


Figure A.2: Last bin of a solution which does not contain all items. The area is the whole hatched part and the waste in the grey hatched part.

Some methods have been designed to work on more variants. [dNdQJ19] developed an exact algorithm for G-KP-O, 3NEGH-KP-O, 2NEGH-KP-O and the three-dimensional variants, [BW09] developed a genetic algorithm for G-KP-O, G-KP-R, and the non-guillotine variants. [ACV$^+$09] and [SAVdC10] respectively developed a heuristic and an exact algorithm for 3NEGH-BPP-O, 3GH-BPP-O, 2NEGH-BPP-O and 2GH-BPP-O, and the non-oriented cases. [FMT16] introduced a model for G-KP-O and G-SPP-O. [LMV04] proposed a unified tabu search for two- and three-dimensional Packing Problems. They provide computational experiments for BPP-O and the three-dimensional variant. They also describe how to adapt the algorithm for several variants such as Strip Packing or Multiple Knapsack. However, adapting the algorithm requires to provide a heuristic procedure, on which the efficiency of the algorithm highly relies. We did not find any use of their tabu search in the subsequent literature. Also, a framework has been proposed by [NPC08]; unfortunately, it has only been implemented for BPP-O and we did not find any use of their framework in the subsequent literature either.

Regarding our methodology, even though tree search algorithms have been widely used to solve Packing Problems, the search algorithm that we implemented does not seem to have been proposed before. We may notice that many packing algorithms rely on Beam Search which is relatively close, as discussed in Section A.5. [AHM09b], [HM09], [AHM10] and [AHN11] implemented it for Circular Packing Problems; [BS10] and [BCMS18] for Irregular Packing Problems; [WLZ13], [AR14] and [AMS20] for three-dimensional Packing Problems; and [HNOS12] for 2NEGH-KP-O. However, these Beam Search implementations significantly differ from our tree search implementation. Most of them do not use a restart strategy, are block-based approaches and use probing (filling partial solutions with a greedy heuristic) to evaluate the quality of nodes. Furthermore, they are globally more complex than our tree search implementation, suggesting that we better captured the key ideas that make tree search algorithms efficient for Packing Problems.

A.3 Algorithm description

We propose an anytime tree search algorithm.

Anytime is a terminology usually found in automated planning and scheduling (AI planning) communities. It means that the algorithm can be stopped at any time and still provides good solutions. In other words, it produces feasible solutions quickly and improves them over time (as classical meta-heuristics do).

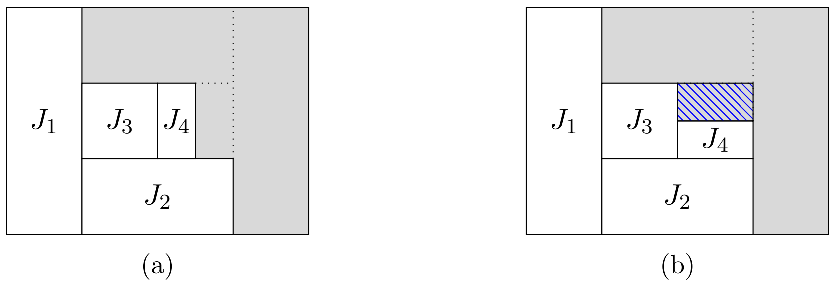


Figure A.3: Solution (a) dominates solution (b) because the hatched area will not be used

Tree search algorithms represent the solution space as an implicit tree called *branching scheme* and explore it completely in the case of exact methods or partially in the case of heuristic methods. The branching scheme is described in Section A.3.1 and the tree search algorithm in Section A.3.2.

A.3.1 Branching scheme

We describe the branching scheme for the 3-staged cases with vertical cuts in the first stage. For the 2-staged cases, we merely impose the position of the first cut to be at the end of the bin and adjust the computation of parameters accordingly; and when the cuts in the first stage are horizontal, we simply adapt the computation of coordinates.

The branching scheme is rather straightforward. The root node is the empty solution without any items, and at each stage, a new item is added. All items that do not belong to the current node are considered. However, items in a solution are inserted according to the following order: rightmost first level sub-plates first; within a first level sub-plate, bottommost second level sub-plates first; and within a second level sub-plate, rightmost items first. Thus, a new item can be inserted in a new bin; in a new first level sub-plate to the right of the current one; in a new second-level sub-plate above the current one; in a new third-level sub-plate, to the right of the last added item. If the cuts of the first stage can be vertical or horizontal, then two different insertions in a new bin are considered: an insertion in a new bin with vertical cuts in the first stage, and an insertion in a new bin with horizontal cuts in the first stage.

To handle exact guillotine cuts, we simply fix the position of the 2-cut above an item inserted in a new bin, first or second level sub-plate, *i.e.* the next items inserted in the same second level sub-plate will only be those of the same height.

Item rotation or not is naturally handled in the branching scheme.

To reduce the size of the tree, we apply some simple dominance rules.

First, if an item can be inserted in the current bin, we do not consider insertions in a new bin; and if an item can be inserted in the current first (resp. second) level sub-plate without increasing the position of its left 1-cut (resp. top 2-cut), we do not consider insertions in a new first (resp. second) level sub-plate.

Then, if item rotation is allowed, some insertions can be discarded as illustrated in Figure A.3.

We also impose an order on identical items.

Finally, we add the following symmetry breaking strategy: a k-level sub-plate is forbidden to contain an item with a smaller index than the previous k level sub-plate of the same

$(k-1)$-level sub-plate. The symmetry breaking strategy is controlled with a parameter s, $1 \leq s \leq 4$. If $s = k$, then the symmetry breaking strategy is only used with k' level sub-plates, $k' \geq k$. For example, if $s = 4$, no symmetry breaking strategy is used. The choice of the value of s is discussed in Section A.5.

A.3.2 Tree search algorithm

The tree described in the previous section is too large to be entirely explored. Therefore, we use a tree search algorithm that we called Memory Bounded A* (MBA*) to explore the most interesting parts in priority. The pseudo-code is given in Algorithm A.1. MBA* starts with a queue containing only the root node. At each iteration, the *best* node is extracted from the queue and its children are added to the queue. If the size of the queue goes over a pre-defined threshold value, the *worst* nodes are discarded. We start with a threshold of 2, and each time the queue becomes empty, we start over with a threshold multiplied by the growth factor f. We choose $f = 1.5$ as discussed in Section A.5.

Algorithm A.1: Memory Bounded A* (MBA*)

```
fringe ← {root};
while fringe ≠ ∅ and time < timelimit do
    $n \leftarrow extractBest(\text{fringe})$;
    fringe ← fringe \ {$n$};
    forall $v \in neighbours(n)$ do
        fringe ← fringe ∪ {$v$};
    end
    while |fringe| > $D$ do
        $n \leftarrow extractWorst(\text{fringe})$;
        fringe ← fringe \ {$n$};
    end
end
```

The function used to define *better* and *worse* is called a guide. The lower the value of the guide function is, the better the solution. For Bin Packing and Strip Packing Problems, we designed the following guide functions:

$$c_0(S) = \text{waste\_percentage}(S)$$

$$c_1(S) = \frac{\text{waste\_percentage}(S)}{\text{mean\_item\_area}(S)}$$

$$c_2(S) = \frac{0.1 + \text{waste\_percentage}(S)}{\text{mean\_item\_area}(S)}$$

$$c_3(S) = \frac{0.1 + \text{waste\_percentage}(S)}{\text{mean\_squared\_item\_area}(S)}$$

with

- waste_percentage(S) = waste(S)/area(S);
- mean_item_area(S) the mean area of the items of S;

- mean_squared_item_area(S) the mean squared area of the items of S.

For Knapsack Problems, we use the following guide function:

$$c_4(S) = \frac{\text{area}(S)}{\text{profit}(S)}$$

with profit(S) the sum of profit of the items of S.

The importance and design of these guide functions are discussed in Section A.5.

A.4 Computational experiments

The algorithm has been implemented in C++ in a new software package called PackingSolver. The code is available online[2]. The repository also contains all the scripts used to conduct the experiments so that results can be reproduced. The results presented above have been obtained with PackingSolver 0.2[3] running on a personal computer with an Intel Core i5-8500 CPU @ 3.00GHz × 6. We allow running up to 3 threads with different settings in parallel. The settings have been chosen following the observations given in Section A.5. Better settings may exist, we try to reproduce the results one would obtain in a practical situation where the global characteristics of the instances are known.

We compare the performances of our algorithm with the best algorithms from the literature for each variant. Due to a large number of problems, we only provide a synthesis of the results here. However, detailed results are available online[4] and the interested reader is encouraged to have a look at them.

Results are summarized in Tables A.1, A.2 and A.3. The first column of the tables indicates the article from which the results have been extracted or the parameters we used for our algorithm. c_a^b indicates a thread with guide function c_a and symmetry breaking parameter b. TL stands for *time limit*. The time limit has been chosen to yield a good compromise between computation time and the best solution value. We only indicate the frequencies of the processors used to evaluate the other algorithms when they significantly differ from ours, *i.e.* below 2GHz.

For Bin Packing Problems, the second column contains the total number of bins used in Table A.1a and the average of the average percentage of waste of each sub-dataset in Table A.1b. For Knapsack and Strip Packing Problems, it contains the average gap to the best-known solutions. The third one indicates the average time to best when available, or the average computation time.

Dataset *hifi* is a dataset composed of instances from [CW77], [Wan83], [OF90], [TH95], [FS97], [FHZ98], [Hif97] and [CHC00]. Researchers usually test their algorithms on a subset of these instances, but often not the same. Dataset *bwmv* refers to datasets from [BW87] and [MV98] which are usually used together.

Other datasets are

- *beasley1985* from [Bea85]
- *fayard1998* from [FHZ98]

[2]`https://github.com/fontanf/packingsolver`
[3]`https://github.com/fontanf/packingsolver/releases/tag/0.2`
[4]`https://github.com/fontanf/packingsolver/blob/0.2/results_rectangleguillotine.ods`

- *kroger1995* from [Krö95]
- *hopper2000* from [Hop00]
- *hopper2001* from [HT01]
- *alvarez2002* from [AVPT02]
- *morabito2010* from [MP10]
- *hifi2012* from [HNOS12]
- *velasco2019* from [VU19]

A.4.1 Bin Packing Problems

Results for Bin Packing Problems are summarized in Table A.1. On 2NEGH-BPP-O and 2NEGH-BPP-R, PackingSolver respectively needs fewer bins than the algorithms from [CZ13] and [CYC16] for the considered datasets. Furthermore, the average time to best is of the order of a second, which is significantly smaller than the average time reported for the other algorithms. On 3NEGH-BPP-O, 3GH-BPP-O, and 2NEGH-BPP-O, the average of the average percentage of waste of PackingSolver is smaller than the one of the algorithms from [ACV$^+$09]. However, on 2GH-BPP-O, it is greater. Finally, compared to the algorithms from [PR07] and [ASdC14], it needs more bins, but the average time to best is two orders of magnitude smaller than the average time reported for those algorithms. We also note that PackingSolver respectively needs significantly fewer bins on 3NEGH-BPP-O and 3GH-BPP-O compared to the algorithms from [PR07] and [ASdC14] for 3GH-BPP-O and 2NEGH-BPP-O,

A.4.2 Knapsack Problems

Results for Knapsack Problems are summarized in Table A.2. We include comparisons with algorithms designed for the non-staged variants. In these cases, PackingSolver usually fails to find the best solutions. It seems likely that they often cannot be reached with only 3 stages. However, its average gap to best is generally less than 1% and on datasets *velasco2019* it is even better than the recent algorithm from [VU19]. The same happens on dataset *fayard1998* for G-KP-R, but the algorithm developed by [BW09] seems to perform significantly worse than more recent algorithms and none of them has been tested on this dataset.

On 3NEGV-KP-O, the average gap to best of PackingSolver is better than [CCTH15], but at the expense of longer computation times. For 2NEGH-KP-O, as [AVMTP07], it finds all the best solutions, but faster. Compared to the algorithm from [HMS08], it performs slightly worse on dataset *alvarez2002* (even if the average gap is 0.0, it fails to find the best solution on two instances) but better on dataset *hifi2012*.

On variants 2NEG-KP-R, 2G-KP-O, 2GH-KP-O, and 2GV-KP-O for which [LM03] and [HR01] developed exact algorithms, PackingSolver finds all optimal solutions in reasonable computation times.

Note that, to the best of our knowledge, only [CGH08] proposed an algorithm for a variant of a Multiple Knapsack Problem. However, they consider homogenous T-shaped patterns which we do not consider in this article.

Article / Parameters	Total	Time (s)
3NEGH-BPP-O, *bwmv*		
PS, $c_0^2 c_2^2 c_3^3$, TL 60s	7278	0.790
3GH-BPP-O, *bwmv*		
[PR07]	7325	160.68
PS, $c_0^2 c_2^2 c_3^3$, TL 60s	7344	0.808
2NEGH-BPP-O, *bwmv*		
[ASdC14]	7372	29.42
[ASdC14]	7364	84.04
PS, $c_0^2 c_2^2 c_3^3$, TL 60s	7391	0.814
2NEGH-BPP-O, *hifi*		
[CZ13]	260	0.19
PS, $c_2^3 c_3^3 c_3^4$, TL 10s	255	0.106
2NEGH-BPP-O, *alvarez2002*		
[CZ13]	219	9.5
PS, $c_2^3 c_3^3 c_3^4$, TL 10s	218	0.346
2NEGH-BPP-R, *bwmv*		
[CYC16]	7034	20.72
PS, $c_0^2 c_2^2 c_3^3$, TL 60s	7029	0.590

(a)

Article / Parameters	Waste	Time (s)
3NEGH-BPP-O, *bwmv*		
[ACV+09]	26.52	
PS, $c_0^2 c_2^2 c_3^3$, TL 60s	20.93	0.790
3GH-BPP-O, *bwmv*		
[ACV+09]	26.29	
PS, $c_0^2 c_2^2 c_3^3$, TL 60s	22.34	0.808
2NEGH-BPP-O, *bwmv*		
[ACV+09]	26.12	
PS, $c_0^2 c_2^2 c_3^3$, TL 60s	23.21	0.807
2GH-BPP-O, *bwmv*		
[ACV+09]	49.06	
PS, $c_0^3 c_2^3 c_3^4$, TL 60s	49.45	0.181

(b)

Table A.1: Results on Bin Packing Problems

A.4.3 Strip Packing Problems

Not many variants of guillotine Strip Packing Problems have been studied in the literature; only G-SPP-O, G-SPP-R, and 2NEGH-SPP-O. This makes comparisons with PackingSolver difficult since it is limited to three-staged patterns, and 2NEGH-SPP-O has several specific structural properties that dedicated algorithms can exploit, but not a more generic one. We still provide computational experiments for these variants in Table A.3. As expected, PackingSolver does not perform as well. Still, on dataset *bwmv*, it returns strictly better average solutions on 16 out of 50 groups of instances for G-SPP-O and on 14 out of 50 groups of instances for G-SPP-R than the algorithm from [WTZL14]. To highlight a bit more the contribution of our algorithm for Strip Packing Problems, we provide a comparison of the solutions from [LMV04] and from [CZC17] for 2NEGH-SPP-O with the solutions returned by PackingSolver for 2NEGH-SPP-R, *i.e.* when item rotation is allowed. The average solutions returned by PackingSolver are strictly better on each of the 50 groups of instances of dataset *bwmv*.

A.5 Discussion

In this section, we discuss some items related to the algorithm.

Growth factor of the queue size threshold: In Section A.3.1, we indicated that we set the growth factor of the queue size threshold to 1.5. The greater the threshold, the better the solutions will be, but the longer MBA* will take to terminate. Furthermore, for

Article / Parameters	Gap	Time (s)
G-KP-O, *fayard1998*		
[VU19]	0.00	0.06
PS, 3NEG-KP-O, $c_4^2c_4^3$, TL 10s	0.16	0.182
G-KP-O, *alvarez2002*		
[WL15]	0.02	21.987
[VU19]	0.00	93.681
PS, 3NEG-KP-O, $c_4^2c_4^3$, TL 60s	0.48	13.264
G-KP-O, *hopper2001*		
[WL15]	0.31	22.214
PS, 3NEG-KP-O, $c_4^2c_4^3$, TL 10s	4.69	1.283
G-KP-O, *morabito2010*		
[VU19]	0.01	19.57
PS, 3NEG-KP-O, $c_4^2c_4^3$, TL 10s	0.17	0.332
G-KP-O, *beasley1985*		
[DLM12]	0.00	1397.738
[WL15]	0.44	20.923
PS, 3NEG-KP-O, $c_4^2c_4^3$, TL 10s	0.56	0.204
G-KP-O, *velasco2019*		
[VU19]	1.42	165.618
PS, 3NEG-KP-O, $c_4^2c_4^3$, TL 120s	0.47	34.682
G-KP-R, *hopper2001*		
[WL15]	0.00	5.04
PS, 3NEG-KP-R, $c_4^2c_4^3$, TL 30s	1.71	8.049
G-KP-R, *fayard1998*		
[BW09]	1.57	
PS, 3NEG-KP-R, $c_4^2c_4^3$, TL 30s	0.00	2.578
G-KP-R, *velasco2019*		
[VU19]	1.05	170.20
PS, 3NEG-KP-R, $c_4^2c_4^3$, TL 120s	0.51	38.590
3NEGV-KP-O, *alvarez2002*		
[CCTH15]	0.09	2.06
PS, $c_4^1c_4^2c_4^3$, TL 60s	0.01	11.879

Article / Parameters	Gap	Time (s)
2NEGH-KP-O, *hifi*		
[AVMTP07]	0.00	0.5
PS, $c_4^2c_4^3$, TL 3s	0.00	0.032
2NEGH-KP-O, *alvarez2002*		
[HMS08]	0.00	0.2
PS, $c_4^2c_4^3$, TL 10s	0.00	0.410
2NEGV-KP-O, *alvarez2002*		
[HMS08]	0.00	0.2
PS, $c_4^2c_4^3$, TL 10s	0.00	0.382
2NEGH-KP-O, *hifi2012*		
[HMS08]	0.26	368.365
PS, $c_4^2c_4^3$, TL 300s	0.12	138.742
2NEGV-KP-O, *hifi2012*		
[HMS08]	0.24	310.105
PS, $c_4^2c_4^3$, TL 300s	0.00	121.014
2NEGH-KP-R, *hifi*		
[LM03] (533 MHz)	0.00	34.348
PS, $c_4^2c_4^3$, TL 3s	0.00	0.161
2G-KP-O, *hifi*		
[HR01] (250 Mhz)	0.00	1.253
PS, $c_4^2c_4^3$, TL 1s	0.00	0.003
2GH-KP-O, *hifi*		
[HR01] (250 Mhz)	0.00	1.145
PS, $c_4^2c_4^3$, TL 1s	0.00	0.002
2GV-KP-O, *hifi*		
[HR01] (250 Mhz)	0.00	1.147
PS, $c_4^2c_4^3$, TL 1s	0.00	0.005

Table A.2: Results on Knapsack Problems

Bin Packing and Strip Packing Problems, full solutions are usually found shortly before it terminates. Therefore, by choosing a too large value for the growth factor, we take the risk to reach the time limit having to spend a lot of time with a given threshold without obtaining any solutions from it. On the other hand, if the growth factor is too small, then only small thresholds value will be explored and no good solutions will be found. In our experiments, 1.5 proved to be a good compromise.

Choice of guide functions: The effectiveness of MBA* highly relies on the definition of its guide function. For MBA*, the guide function should be relevant to compare two nodes at different stages of the tree. Therefore, the waste-percentage c_0 appears much more relevant than the waste alone for Bin Packing and Strip Packing variants. Guide function c_1 is adapted from c_0, but it favours solutions containing larger items. This helps to avoid situations where all small items are packed in the first bins and the last bins get all

Article / Parameters	Gap	Time (s)
G-SPP-O, *kroger1995*		
[WTZL14]	0.27	22.67
PS, 3NEGH-SPP-O, $c_0^2 c_0^3 c_0^4$, TL 30s	3.65	10.416
G-SPP-O, *hopper2001*		
[WTZL14]	0.00	6.267
PS, 3NEGH-SPP-O, $c_0^2 c_0^3 c_0^4$, TL 30s	6.75	4.364
G-SPP-O, *hopper2000*		
[WTZL14]	0.00	20.647
PS, 3NEGH-SPP-O, $c_0^2 c_0^3 c_0^4$, TL 30s	8.72	5.899
G-SPP-O, *bwmv*		
[WTZL14]	0.15	17.736
PS, 3NEGH-SPP-O, $c_0^2 c_5^2 c_6^3$, TL 60s	1.10	12.831
G-SPP-R, *kroger1995*		
[CYC13]	0.00	56
PS, 3NEGH-SPP-R, $c_0^2 c_0^3 c_0^4$, TL 30s	1.84	9.716
G-SPP-R, *hopper2001*		
[WTZL14]	0.00	13.466
3NEGH-SPP-R, $c_0^2 c_0^3 c_0^4$, TL 30s	3.00	4.153
G-SPP-R, *hopper2000*		
[WTZL14]	0.00	13.465
PS, 3NEGH-SPP-R, $c_0^2 c_0^3 c_0^4$, TL 30s	3.30	10.7
G-SPP-R, *bwmv*		
[WTZL14]	0.13	18.253
PS, 3NEGH-SPP-R, $c_0^2 c_5^2 c_6^3$, TL 30s	0.58	12.592

Article / Parameters	Gap	Time (s)
2NEGH-SPP-O, *alvarez2002*		
[CYC13]	0.02	4.78
PS, $c_4^1 c_4^2 c_4^3$, TL 30s	1.13	3.726
2NEGH-SPP-O, *bwmv*		
[LMV04]	0.02	66.71
[CZC17]	0.13	1.77
PS, $c_0^2 c_5^2 c_6^3$, TL 30s	0.68	0.992
2NEGH-SPP-R, *bwmv*		
[LMV04]	7.96	66.71
[CZC17]	8.08	1.77
PS, $c_0^2 c_5^2 c_6^3$, TL 30s	0.00	1.773

Table A.3: Results on Strip Packing Problems

the large items, creating large waste areas. Guide function c_2 is adapted from c_1: indeed, even if c_1 favors large items first, solutions with no waste at all will always be extracted first, even if they contain only small items. The constant in c_2 aims at fixing this behavior and will lead to better solutions on instances in which optimal solutions contain significant waste (more than 10%). c_3 is adapted from c_2 and favours even more large items first. This guide function is useful for some instances containing several very large items. Finally, c_4 is a natural adaption of c_0 for Knapsack variants. An experimental comparison of several guide functions for the 2018 ROADEF/EURO challenge glass cutting problem is presented in Chapter 3.

Depth of the symmetry breaking strategy: In exact tree search algorithms, it is usually worth breaking symmetries. However, this is not the case when the tree is not meant to be explored completely. For example, consider two symmetrical nodes, the first one normally appearing in the queue, but the second one never being added to the queue because one of its ancestors has been removed to reduce the size of the queue. If the first one is not explored because the symmetry has been detected, then this solution will not be found during the search. How to determine the ideal depth of the symmetry breaking strategy for an instance is not clear yet. The relative size of the items compared to the bin might be an influential factor. For the experiments, we chose 2 or 3 as *standard* values. For some instances containing many items (more than 1000), only a value of 4 ensures finding a feasible solution quickly; in contrast, for some knapsack instances with few first-level sub-plates, a value of 1 gives access to better solutions. An experimental evaluation of the influence of the symmetry breaking strategy for the 2018 ROADEF/EURO challenge glass cutting problem is presented in Chapter 3.

MBA* vs Beam Search: Beam Search is another popular tree search algorithm in the packing literature. Beam Search also starts with a queue containing only the root node. However, at each iteration, all nodes of the queue are expanded, and as in MBA*, if the size of the queue goes over a pre-defined threshold, the worst nodes are discarded. Thus, at each iteration, the queue always contains nodes belonging to the same level of the tree. Beam Search seems therefore effective when the guide function is relevant to compare nodes belonging to the same level. This is for example generally not the case in Branch-and-Cut trees where branching consists in fixing a variable to 0 or 1. With our branching scheme for Packing Problems, it is easier to compare such solutions, but the guide functions we presented in Section A.3 make it even possible to compare nodes at different levels of the search tree. Thus, Beam Search expands many nodes which are not that much interesting, whereas MBA* always expands only the best current node. An experimental comparison of MBA* and Beam Search for the 2018 ROADEF/EURO challenge glass cutting problem is presented in Chapter 3. It shows that MBA* finds significantly better solutions than the equivalent Beam Search implementation, thus the integration within PackingSolver.

Higher staged guillotine cuts: Our branching scheme generates up to three-staged patterns. One could wonder whether it could be possible to adapt it for four-staged or non-staged guillotine patterns. However, if a similar branching scheme seems possible, it may significantly increase symmetry issues. We believe that this would be prohibitive.

MBA* might be used to solve these variants, but new branching schemes need to be designed.

Item-based vs block-based: Many researchers highlighted the benefits of using block-based approaches, *i.e.* inserting several items at each stage of the tree [BJ12, WTZL14, LMP17]. It is interesting to note that it is not what we implemented, yet our algorithm is competitive.

A.6 Conclusion and future work

We showed that the algorithm proposed by [LF20b] for the 2018 ROADEF/EURO challenge glass cutting problem is actually also very competitive compared to other dedicated algorithms for guillotine Packing Problems from the literature, and is even able to return state-of-the-art solutions on several variants. Its performances seem to rely on two key components: a branching scheme which limits symmetry issues; and a tree search algorithm fully exploiting guide functions which make it possible to compare nodes at different levels of the search tree.

In addition to effectiveness, the choice of a tree search algorithm makes the algorithm attractive for problems with additional side constraints. Indeed, new constraints are likely to reduce the size of the search tree.

The algorithm is implemented in a new software package intended for researchers in Packing Problems to develop new branching schemes for other variants, for researchers in Artificial Intelligence to experiment new tree search algorithms, and for OR practitioners to quickly develop efficient algorithms implementing several business-specific constraints.

Future research will focus on developing algorithms for Cutting Stock and Variable-sized Bin Packing Problems, as well as branching schemes to generate another kind of patterns such as non-guillotine or non-staged guillotine ones.