INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Advanced Mathematical Programming

Branch-and-Price for the Bin-Packing Problem

May 15, 2020

Théo Guyard
theo.guyard@insa-rennes.fr
4GM, Mathematics Department, INSA Rennes

Abstract

This technical report presents a Branch-and-Price algorithm aiming to solve the Bin-Packing problem. For this algorithm, two different branching rules and three different exploration methods are implemented. A dynamic programming method able to solve the subproblems dynamically instead of using a classical LP solver is also provided. Furthermore, heuristics which can tighten upper and lower bounds can be proceeded before and during the branching tree exploration . In the first section, the mathematical background of a Branch and Price method suited for the Bin-Packing problem is presented. In the second section, the main structure of the algorithm is exposed with some details about its simplest parts. The third section presents the node processing method for the Ryan & Foster and the Generic branching scheme. Then, the heuristics implemented are detailed. Some other method not yet implemented but which could have improve the algorithm are presented in the fifth section. Finally, some details about the implementation and the computational results are given in the last section.

Contents

1 Introduction

1.1 Bin-Packing Problem

In this report, we focus on the Bin-Packing Problem. It consists of putting *items* of different *sizes* in a minimal number of *bins* with equal *capacity*. We consider that set if items is given by $I = \{1, \dots, N\}$, their size is given by $S = \{s_1, \dots, s_n\}$ and that B bins with a capacity C are available. The number of bin is supposed large enough to store all the N objects. The Bin-Packing problem can be written under the following linear program :

$$\begin{cases} \min & \displaystyle\sum_{b=1}^{B} y_b & & \\ \text{s.t.} & \displaystyle\sum_{i=1}^{N} s_i x_{ib} \leq C y_b & \forall\ b = 1, \dots, B & \\ & \displaystyle\sum_{b=1}^{N} x_{ib} = 1 & \forall\ i = 1, \dots, N & \\ & x_{ib} \in \{0,1\} & \forall\ i = 1, \dots, N & \forall\ b = 1, \dots, B \\ & y_b \in \{0,1\} & \forall\ b = 1, \dots, B & \end{cases} \tag{$\mathbb{BPP}$}$$

Where $x_{ib} = 1$ if the item i is packed in the bin number b ($x_{ib} = 0$ otherwise) and $y_b = 1$ if the bin number b is non-empty ($y_b = 0$ otherwise). The first constraint ensure that the bin capacity is not exceeded and the second constraint ensure that each item is packed in a unique bin. This problem is known to be NP-hard.

1.2 Branch-and-Price approach

To address the NP-hardness of the problem, we can solve this problem with a Branch-and-Price algorithm.

1.2.1 Set covering formulation

We choose an other formulation of the ($\mathbb{BPP}$). Let $\mathcal{P}$ the set of all combinations of items of I such that their cumulative size doesn't exceed the bin capacity C. Each element of $\mathcal{P}$ is called a *pattern*. Let $P = |\mathcal{P}|$, we have

$$\mathcal{P} = \left\{ \mathbf{x} \in \{0,1\}^{|N|}, \quad \sum_{i=1}^{N} x_i s_i \leq C \right\} = \left\{ \mathbf{x} = \sum_{p=1}^{P} \alpha^p \bar{x}^p, \quad \sum_{p=1}^{P} \alpha^p = 1, \alpha \in \{0,1\}^{|P|} \right\}$$

Here, $\bar{x}^p$ is a pattern and $\bar{x}^p_i = 1$ if the item i belongs to the pattern p. Using this Dantzig reformulation of the set $\mathcal{P}$, we can write ($\mathbb{BPP}$) under its *set covering formulation* :

$$\begin{cases} \min & \displaystyle\sum_{p=1}^{P} \alpha^p & & \\ \text{s.t.} & \displaystyle\sum_{p=1}^{P} x_i^p \alpha^p = 1 & \forall\ i = 1, \dots, N & \\ & \alpha^p \in \{0,1\} & \forall\ p = 1, \dots, P & \end{cases} \tag{$\mathbb{SCBPP}$}$$

For this formulation, $x_i^p = 1$ if the item i is within the pattern p and $\alpha^p = 1$ if the pattern p is used in the solution. The optimal solution is a set of patterns and each patter correspond to a bin. The constraint ensure that each item is packed in a unique pattern/bin.

One thing to notice is that $|P|$ (the number of variables) is huge and we have to know every possible patterns. At most, $|P|$ is the number of combination available with N items. Even for small instances, the number of variable is not tractable as it. Furthermore, it is needed to enumerate and store all the patterns possible in order to solve the problem which could involve a huge amount of memory.

1.2.2 Restricted master problem

To handle this problem, we introduce a new set $\mathcal{P}' \subset \mathcal{P}$ of cardinal P' containing only a fraction of the available patterns. We can solve ($\mathbb{SCBPP}$) on this restricted number of pattern, which is more tractable. This new problem is called the *restricted master problem* :

$$\begin{cases} \min & \displaystyle\sum_{p=1}^{P'} \alpha^p \\ \text{s.t.} & \displaystyle\sum_{p=1}^{P'} x_i^p \alpha^p = 1 \quad \forall\ i = 1, \dots, N \\ & \displaystyle\sum_{p=1}^{P'} \alpha^p \leq B \\ & \alpha^p \in [0,1] \qquad \forall\ p = 1, \dots, P' \end{cases} \tag{$\mathbb{RMBPP}$}$$

The solution of ($\mathbb{RMBPP}$) is a sub-optimal bound of ($\mathbb{SCBPP}$) as some of the optimal patterns may not be included in $\mathcal{P}'$. However, if the set $\mathcal{P}'$ is well chosen and contains the optimal patterns, then the optimal solution of ($\mathbb{RMBPP}$) is the same as the optimal solution of ($\mathbb{SCBPP}$). The problem is to choose the right patterns to include in $\mathcal{P}'$. As $\mathcal{P}'$ could also have a big cardinal, we rather solve a relaxation of the master problem and that is why we have that $\alpha_p \in [0,1]\ \forall\ p = 1, \dots, P'$. We also add a constraint on the number of pattern allowed as we know that there are only B bins available. If we have some additional informations about the problem, it is possible to tighten this bound. Solving the relaxation leads to a supra-optimal solution comparing to the integer formulation of ($\mathbb{RMBPP}$) (with $\alpha^p \in \{0,1\}$). We will see later how to obtain an integer solution in order to have the same solution as the optimal solution of ($\mathbb{SCBPP}$) but now, we focus on how to select right patterns to include in $\mathcal{P}'$.

1.2.3 Subproblem

We can write an other optimization problem called the *subproblem* (or *pricing problem*) which aims to generate "interesting" patterns to include in $\mathcal{P}'$. If we note π and σ the optimal dual variables corresponding to the first and second constraint of ($\mathbb{RMBPP}$), the pricing problem takes the following form :

$$\begin{cases} \min & 1 - \displaystyle\sum_{i=1}^{N} \pi_i y_i - \sigma \\ \text{s.t.} & \displaystyle\sum_{i=1}^{N} y_i s_i \leq C \\ & y_i \in \{0,1\} \qquad \forall\ i = 1, \dots, N \end{cases} \tag{$\mathbb{SPBPP}$}$$

where $y_i = 1$ if the item i is included in the pattern $\mathbf{y}$. The solution of the pricing problem is a feasible pattern (*i.e.* which doesn't exceed the bin capacity). The optimal cost is called the *reduced cost*. We can see that usually, the cost of a pattern is 1 (one pattern used correspond to one bin used) but here, the cost is penalized by a term containing π and σ. This second term penalizes patterns which can not improve the set $\mathcal{P}'$ in the sense that if a pattern $\mathbf{y}$ has a high reduced cost, the solution of ($\mathbb{RMBPP}$) with $\mathcal{P}'$ or with $\mathcal{P}' \cup \mathbf{y}$ is the same. If $z_{sp}^\star$ denotes the optimal cost and $\mathbf{y}^\star$ the associated patter of ($\mathbb{SPBPP}$) for a given π and σ, then $\mathbf{y}^\star$ can improve the solution of ($\mathbb{RMBPP}$) if and only if $z_{sp}^\star < 0$.

One thing very important to notice is that this pricing problem is equivalent to a knapsack problem with weights π, sizes s and capacity C. A knapsack problem is relatively easy to solve. The key mechanism of the Branch-and-Price algorithm is to solve sequentially ($\mathbb{RMBPP}$) and ($\mathbb{SPBPP}$), including the new pattern provided by the subproblem, until a solution ${z_{sp}}^\star \geq 0$ is obtained. At this stage, the continuous relaxation of ($\mathbb{RMBPP}$) is solved at optimality. We now focus on a method aiming to obtain integer solutions for ($\mathbb{RMBPP}$) in order to have the optimal solution of ($\mathbb{SCBPP}$). This method is called branching.

1.2.4 Branching

($\mathbb{RMBPP}$) is a relaxation of ($\mathbb{SCBPP}$) so it gives a supra-optimal solutions. In order to have the optimal integer solution, we can introduce *branching rules* (new constraints) to break the fractional property of the variables. Let's

consider a tree where each node correspond to a problem like ($\mathbb{RMBPP}$) with its local branching rules. When a node is solved to optimality (*i.e.* ($\mathbb{RMBPP}$) is solved sequentially until $z^\star_{sp} \geq 0$), we can select fractional variables in the solution. Child nodes are created under the node just solved with new constraints in order to obtain solutions where the selected branching variables remains integer for the rest of the branch. The tree is explored node after node until all the nodes are explored. The key idea is to cut non-improving branches during the explorations so as to explore only a part of the tree.

We can not branch only on the fractional α^p. Instead, we select a pair (i, j) of items such that the variable

$$w_{ij} = \sum_{p \in \mathcal{P}' | x_i^p = x_j^p = 1} \alpha^p$$

is fractional. In one branch, we impose that $w_{ij} \geq 1$ (items i and j are always together) and in the other branch, we impose that $w_{ij} \leq 0$ (items i and j are always separated).

2 Branch-and-Price outline

In this section, the main ideas of the BnP algorithm implemented to solve (𝔹ℙℙ) are presented.

2.1 Resolution method

To solve the Bin-Packing Problem, we can use two different branching rules. The first one was proposed by Ryan & Foster in 1981 [RF81]. This branching rule allows to keep a single subproblem at each node but doesn't preserve the knapsack structure of the pricing problem. The subproblems are rather knapsack with conflicts problems, harder to solve than the usual knapsack problem. The second branching rule which can be used is a generic branching scheme introduced by Vance in 1994 [Van+94]. This branching rule preserves the knapsack structure of the subproblems but at each node, multiple subproblems have to be solved. For the Ryan & Foster branching rule, the pricing problem is solved with a LP solver. A dynamic programming method is also presented at the end of this report but was not implemented successfully. For the Generic branching method, the pricing problem is solved either with a LP solver or with a dynamic programming algorithm suited for the knapsack problem. To improve the speed of the algorithm, several root-heuristics are proposed in order to obtain a better initial upper bound. Some tree-heuristics allowing to get feasible solutions throughout the tree exploration are also presented. Two different ways to add the nodes to the queue are granted : FIFO and LIFO.

2.2 BnP structure

The structure of the BnP algorithm is the same whatever the chosen branching-rule, queueing method, heuristics or parameters. It can be summarized by the following algorithm :

Algorithm 1: Branch and Price

Input: The items and their size $s_1, \dots, s_N$, the bin capacity C, an upper bound on the number of bins B, a precision ϵ

```
// Initialization
```
Initialize an empty tree
Initialize the queue with the root
Initialize the column pool with an artificial column
Initialize UB $\leftarrow B$
Initialize LB $\leftarrow \left\lceil \sum s_i / C \right\rceil$
Process a root heuristic to find a better UB (see 4.1)
```
// Tree exploration
```
while *the queue is non-empty* **do**
  Pop the first node in the queue
  Proceed the node according to the branching rule set (see 3.1, 3.2 and 3.3)
  Find the branching variable and add the two child nodes in the queue
  Process a tree heuristic to tighten UB (see 4.2)
  Update LB and UB with the solution found by the node or by the tree heuristic
  Cut branches that cannot improve the upper bound
  if $|UB - LB| < \epsilon$ **then**
    return the best solution associated to UB
  end
end

In the following, the simplest steps of the BnP are explained. The heuristics, the branching method and the node processing have a dedicated section.

2.2.1 Column management

The column pool contains all the columns which are created by the subproblem, whatever the node. We need to add an artificial column for the master problem to always be feasible. This artificial column contains all the items and have a very high cost. If the solution of a master problem contains this artificial column, we know that

the master problem is infeasible because of the branching rules. Before each node processing, we have to create a local node pool containing only the patterns satisfying the branching rules of the node in order to create a solution satisfying the current branching rules. To save speed and memory space, the column pool is global to all nodes and each node has its proper node pool which is basically a filter on the global column pool according to the node branching rules. When a new pattern is created in a subproblem, it is added both in the node pool and in the global column pool.

2.2.2 Initial bounds

We could have set the initial bound at $UB = +\infty$ and $LB = -\infty$ but better initial bound can be found. Indeed, we can assume that the total number of bin used is at most the number of items. Indeed, if at least one of the items cannot be packed in any bin, the problem is infeasible. Thus, we can initially set $UB = N$. The integer relaxation of ($\mathbb{BPP}$) gives also a lower bound for the integer ($\mathbb{BPP}$) : we can initially set $LB = \left\lceil \sum_i s_i/C \right\rceil$ [SV13]. In addition, a root-heuristic can be proceeded to obtain a better initial UB (see 4.1).

2.2.3 Queuing method

In the while loop, we always choose the first node in the queue but the nodes are not always stored in the same order in the queue. When adding the two child nodes to the queue, three methods can be used. The first one puts new nodes at the beginning of the queue (LIFO) in order to proceed a Deep-First-Search in the tree. The second method puts the new nodes at the end of the queue (FIFO) in order to proceed a Breadth-First-Search in the tree. The LIFO method allows to find quickly good upper bounds but cuts branches deeply in the tree. The FIFO method allows to cut branches at a high level in the tree, letting less nodes to be explored, but is slower to get good upper bounds. The best exploration method depends on the structure of the problem and in general, there is no one better than the other. An hybrid exploration of the tree is also provided. First, the LIFO method is set in order to find quickly a good upper-bound. Once this upper bound is found, the queuing strategy switches to FIFO in order to explore the tree with a Breadth-First-Search. This method aims to find quickly an upper bound and then cut branches high in the tree. If a root heuristic is computed, then the Hybrid method is just like the LIFO method.

2.2.4 End of node processing handling

After the node processing (see 3.1, 3.2 and 3.3), it is possible that the solution obtained is integer. In this case, we can see if this solution can improve the current UB and if it is the case, we can update the value of UB. We also choose the best lower bound among all those of the nodes in order to update the global LB. This allows to see how close the algorithm is from the solution by bounding the optimal solution by LB and UB at each node. When a node is infeasible, the branch processing returns a solution containing the artificial column. In this case, we know that it is useless to continue the exploration in the branch so the node is pruned by infeasibility. If the node is feasible, a branching variable is selected, two new nodes are added to the tree with new constraints and the process goes on. Just before adding the new nodes, an heuristic can be processed in order to created a feasible solution using the fractional solution provided by the node (see 4.2). This feasible solution can be used to tighten the current UB.

2.2.5 Tree pruning

Once the bounds are updated, we take a look at the nodes left in the queue. If a node has a lower bound larger than UB, then this node can't improve the current best solution. Thus, we can stop the exploration of the branch after this node. branches are also pruned when a node is infeasible. When $|UB - LB| < \epsilon$, then the current best solution can not be improved and corresponds to the optimal solution of ($\mathbb{BPP}$). If the queue becomes empty while $UB \neq LB$, this means that either the problem has no solution or that ϵ is too small regarding to numerical errors.

3 Node processing

Now that the core structure of the BnP has been presented, we can focus on the node processing where two methods can be used : the Ryan & Foster and the Generic method.

3.1 Ryan & Foster method

The Ryan & Foster branching rule allows to keep a single subproblem per node but it is not as easy to solve as a classical knapsack problem.

3.1.1 Branching rules

When a node is solved at optimality and that two items i and j have been found to create a branch, we have to create one child where $w_{ij} \leq 0$ (the down branch) and one child where $w_{ij} \geq 1$ (the up branch). For the down branch, the Ryan & Foster branching rule simply add the constraint $x_i + x_j \leq 1$ in the subproblem in order to create patterns with items i and j separated. For the up branch, the constraint $x_i = x_j$ is added to the subproblem in order to create patterns containing i and j and patterns containing neither i nor j (as it is impossible to create a solution only with patterns containing both i and j). The subproblem handle itself whether to generate patterns with (resp. without) i and j because the reduced cost is too high if too many pattern with (resp. without) i and j have already been be created.

While exploring the tree, the branching rules are added sequentially to the nodes. Thus, for nodes at the bottom of the tree, the subproblems have several branching constraints. It is possible that some combinations of constraints make the subproblem infeasible. In this case, the process of the node is stopped and the branch is cut as it is impossible to create solutions for the rest of the branch. Using such method for the node processing allows to keep a unique subproblem per node and it is simple to create the local node pool as we can only loop on the global column pool and keep only the columns satisfying the current branching rules.

3.1.2 Subproblem resolution

The subproblem always have the structure presented in 1.2.3 where the node branching constraints are added. Thus, it is possible to solve this problem using a solver like Gurobi. A dynamic programming method can also be used to solve the pricing problem. However, it wasn't implemented successfully. Despite failing to implement this second method, we present it in 5.1.

3.2 Generic method

The generic branching method is a way more complex branching scheme but allows to keep subproblems with the knapsack structure. As the knapsack problem can be solved with dynamic programming, it allows to have a faster resolution method for the subproblems. However, several subproblems needs to be solved.

3.2.1 Branching rules

When a node is solved at optimality and that two branching items i and j have been found, we have to create one child where $w_{ij} \leq 0$ (the down branch) and one child where $w_{ij} \geq 1$ (up branch). For the down branch, we create a subproblem generating patterns with $x_i = 0$ and one subproblem generating patterns patterns with $x_i = 1, x_j = 0$. Thus, the solution constructed includes patterns with items i and j separated. But in the solution, we only need one pattern with $x_i = 1, x_j = 0$. The idea is to split the constraint $\sum_{p=1}^{P'} \alpha^p \leq B$ of ($\mathbb{RMBPP}$) into the two following constraints :

$$\begin{cases} \sum\limits_{p \in \mathcal{P}' | x_i^p = 1, x_j^p = 0} \alpha^p \leq 1 \\ \sum\limits_{p \in \mathcal{P}' | x_i^p = 0} \alpha^p \leq B - 1 \end{cases} \tag{3.2.1}$$

These two constraints preserves the number of patterns created but specify the number of patterns of each type. For the up branch, we create one subproblem generating patterns with $x_i = x_j = 0$ and one subproblem generating patterns patterns with $x_i = x_j = 1$. Thus, the solution is composed of patterns with i and j together and of (and

patterns without i and j). But we only need one pattern with $x_i = x_j = 1$ so we can split the constraint into the following new constraints :

$$\begin{cases} \sum_{p \in \mathcal{P}' | x_i^p = x_j^p = 1} \alpha^p \leq 1 \\ \sum_{p \in \mathcal{P}' | x_i^p = x_j^p = 0} \alpha^p \leq B - 1 \end{cases} \tag{3.2.2}$$

These two constraints preserves the number of patterns created but specify the number of patterns of each type. For up and down branching rules, we can notice that the variables x_i and x_j are fixed so the subproblem keeps its knapsack structure if a preprocess is made. After fixing the branching variable, a knapsack problem remains to be solved (see 3.2.2).

The difficulty lies in the way to process several branches in a row. We have to introduce the notion of **branching set** which are composed of some **branching rules** (corresponding to the filter below the sum in (3.2.1) and (3.2.2)) and one **coefficient** (corresponding to the scalar 1 or $B-1$ in (3.2.1) and (3.2.2)). Each branching rule has several variables set to zero and several variables set to one. For example,

$$\sum_{p \in \mathcal{P}' | \{x_1 = 0, x_2 = 1\} \cup \{x_3 = 0, x_4 = 1\}} \alpha^p \leq 1$$

is a branching set with branching rules $\{x_1 = 0, x_2 = 1\}$ and $\{x_3 = 0, x_4 = 1\}$ and with coefficient 1 and

$$\sum_{p \in \mathcal{P}' | \{x_1 = x_2 = 0\}} \alpha^p \leq B - 1$$

is a branching set with branching rule $\{x_1 = x_2 = 0\}$ and with coefficient $B - 1$. When a new node is created, we loop over all the branching sets of the parent node and create new sets for the child node by adding the new branching rules. From each parent branching set, at most two branching sets can be created for the child node. Four cases can happen :

- **The coefficient of the parent node branching set is $\mathbf{L} > \mathbf{1}$:**
  In this case, the branching set have a unique rule by construction of the child nodes.
  - **We have to add a down branching rule with items i and j :** In this case, we create two branching sets for the child node. The two child branching sets are identical to the parent branching set with a little modification. In the first, the item i is added to the variables set to 0 in the branching rules and the coefficient is set to $\mathbf{L-1}$. In the second, the item i is added to the variables set to 1 in the branching rules and the coefficient is set to $\mathbf{1}$.
  - **We have to add an up branching rule with items i and j :** In this case, we create two branching sets for the child node. The two child branching sets are identical to the parent branching set with a little modification. In the first, the items i and j are added to the variables set to 0 in the branching rules and the coefficient is set to $\mathbf{L-1}$. In the second, the items i and j are added to the variables set to 1 in the branching rules and the coefficient is set to $\mathbf{1}$.
- **The coefficient of the parent node branching set is 1 :**
  In this case, the branching set can have a multiple rule by construction of the child nodes. Only one of the patterns allowed to be created by the branching set can be used in the solution.
  - **We have to add a down branching rule with items i and j :** Only one set is created in the child node. For each branching rule in the parent node branching set, we add either $x_i = 0$ or $x_i = 1, x_j = 0$ to the child branching set containing the rules of the parent branching set. This new branching set has a coefficient equal to $\mathbf{1}$.
  - **We have to add an up branching rule with items i and j :** Only one set is created in the child node. For each branching rule in the parent node branching set, we add either $x_i = x_j = 0$ or $x_i = x_j = 1$ to the child branching set containing the rule of the parent branching set. This new branching set has a coefficient equal to $\mathbf{1}$.

The way new constraints are added to the child nodes is a bit complicated as we have to handle multiple subproblems and we have to ensure that the coefficient of the branching sets add to B in order to preserve the initial constraint $\sum_{p\in\mathcal{P}'} \alpha^p \leq B$ of ($\mathbb{RMBPP}$) in each node. Of course, some of the rules created may be infeasible if one item is both in the set of variables set to 0 and 1. In this case, the rule is not included in the branching set. When the parent branching set is split into one feasible and one infeasible branching set, then we only keep the feasible branching set and the coefficient of the new set is equal to the coefficient of the parent branching set.

Branching rule	Parent node		Child node	
	Branching set	Coefficient	Branching set	Coefficient
Down with items 3 and 4	$\{x_1 = x_2 = 0\}$	L	$\{x_1 = x_2 = x_3 = 0\}$	$L-1$
			$\{x_1 = x_2 = x_4 = 0,\ x_3 = 1\}$	1
Up with items 3 and 4	$\{x_1 = x_2 = 0\}$	L	$\{x_1 = x_2 = x_3 = x_4 = 0\}$	$L-1$
			$\{x_1 = x_2 = 0,\ x_3 = x_4 = 1\}$	1
Down with items 3 and 4	$\{x_1 = 1,\ x_2 = 0\}$	1	$\{x_1 = 1,\ x_2 = x_3 = 0\}$ $\{x_1 = x_3 = 1,\ x_2 = x_4 = 0\}$	1
Up with items 3 and 4	$\{x_1 = 1,\ x_2 = 0\}$	1	$\{x_1 = 1,\ x_2 = x_3 = x_4 = 0\}$ $\{x_1 = x_3 = x_4 = 1,\ x_2 = 0\}$	1

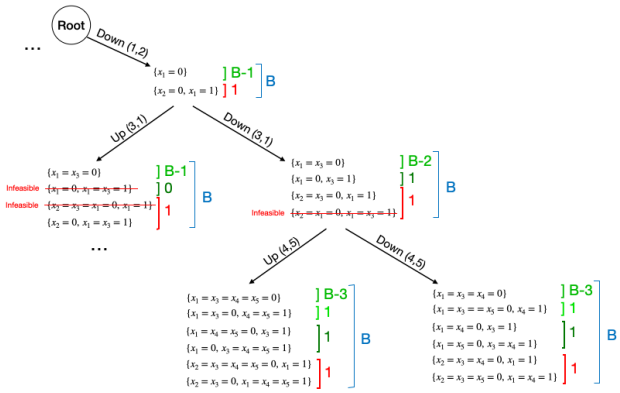

Figure 1: Left : The four cases presented above. Right : An example of how branching constraints are handled and how to proceed when a branching set is infeasible.

3.2.2 Subproblem resolution

In each node for the generic branching scheme, there are as many subproblems to solve as the total number of branching rules. Like for the Ryan & Foster branching method, it is possible to use a LP solver like Gurobi. It can preprocess the problem by setting the variables to 0 or 1 according to the branching schemes before the solving process. But as the knapsack structure is preserved for each subproblem, it is possible to use dynamic programming instead of a classical solver which is usually slower.

The first step before the dynamic programming process is to preprocess the data and treat the variables already fixed in the branching rules. For each variable set to 0, we simply remove this variable from the problem. For each variable set to 1, we remove the variable from the problem, decrease the knapsack capacity by the size of the item and add its cost to a preprocess cost. At the end of the preprocess, if the capacity of the knapsack is still positive, then the dynamic programming process can start. If the capacity is negative, then the knapsack problem is infeasible and the resolution is stopped here. At the end of the dynamic programming process, the total cost is the cost outputted by the dynamic program plus the preprocess cost. The items in the pattern are the items used in the dynamic program solution plus the items set to 1 during the preprocess step.

To solve dynamically the knapsack problem considering a capacity C^{knap}, items with sizes s_i^{knap} and costs p_i^{knap} for $i = 1, \ldots, I^{knap}$, we have to introduce $t_{i,c}^{knap}$, the maximal profit of a knapsack problem with capacity c and with i items in it among all the items. We can apply the following dynamic programming algorithm and backtracking method to find the optimal cost of the knapsack and the items involved in this optimal cost :

Algorithm 2: Knapsack problem

for $i = 1, \ldots, I^{knap}$ **do**
  for $c = 1, \ldots, C^{knap}$ **do**
    if $s_i^{knap} > c$ **then**
      $t_{i+1,c} \leftarrow t_{i,c}$
    else
      $t_{i+1,c} \leftarrow \max\{t_{i,c}\ ;\ p_i^{knap} + t_{i,c-s_i^{knap}}\}$
    end
  end
end

Algorithm 3: Knapsack problem, backtracking

Pattern $\leftarrow \emptyset$
$c = C^{knap}$
for $i = I^{knap}, \ldots, 1$ **do**
  if $t_{i,c} \neq t_{i-1,c}$ **then**
    Pattern $\leftarrow$ Pattern $\cup\ i$
    $c \leftarrow c - s_i^{knap}$
  end
end

The optimal cost is $t_{I^{knap},C^{knap}}$ and the variable $Pattern$ contains the items involved in the optimal cost.

We can observe that the complexity of the dynamic program is $o(I^{knap}C^{knap})$. Although it can be way better than solving the subproblem using an IP solver, some issues can happen when the capacity of the knapsack is large. The dynamic program is not only size-dependant but also data-dependant in the way that two problems with the same number of items won't be solved in the same amount of time for different capacities. A problem with few items but with a huge capacity can be slower to solve than a problem with more items and a smaller capacity. In practice, it could be interesting to measure the solving time per number of items while using the solver and the solving time per number of items and capacity using dynamic programming in order to chose the solving method. Furthermore, as the dynamic programming method is implemented "by hand" in this project while the solver has been developed and optimized by a specialized team, the solver can still be faster in practice than the dynamic program even if the theoretical complexity is worst.

3.3 Node processing algorithm

To make things simpler in the implementation, we only use one type of node for both branching methods. A node always contains some branching sets but in the case of the Ryan & Foster method, it only has one branching set with no rules and a coefficient equal to B as we only have one subproblem per node. A node also has a local lower bound and a list of all up and down branching items previously made in its branch in order to constructs the new constraints for the Ryan & Foster method (but also to debug the BnP when needed). In the following, π is the dual variable associated to the constraint of ($\mathbb{RMIBPP}$) and σ are the constraints associated to each branching subset (only one σ in the case of Ryan & Foster method). Whatever the method chosen to process the node, the algorithm is always the following :

Algorithm 4: Node processing

```
// Initialization
nodePool ← filterGlobalColumnPool()
π, σ, solution, value ← solveMaster()
nodelb ← ∑ π_i
nodeub ← value
minReducedCost ← 0
// Master problem and subproblems are sequentially solved
while true do
    if solution is integer then
        Update global UB if necessary
        Prune branches with a larger nodelb than UB
        break
    end
    for s in branching sets do
        for r in branching rules of s do
            reducedCost, column ← solveSubproblem(π, σ_s)
            if reducedCost < ∞ then
                minReducedCost = min(minReducedCost ; reducedCost)
                if reducedCost < 0 then
                    Add column to the node pool and the global column pool
                    Update the master problem data with the new column
                end
                nodelb = nodelb + reducedCost
            end
        end
        if all columns have reducedCost = ∞ for the subset s then
            Node is infeasible
            break
        end
    end
    if |nodelb − nodelb| < ϵ then
        Return the solution of the master problem (node is infeasible if it
         contains the artificial column)
        break
    end
    π, σ, solution, value ← solveMaster()
end
```

For the Ryan & Foster method, the loop on s and r are only passed once as the node for the Ryan & Foster method has only one branching set with an empty branching rule. At the beginning of the **while** loop, if the solution is integer then the node has been solved to optimality. We check is this new feasible solution can improve the current bound an if so, we check is some nodes in the queue can be pruned by bounds. At the end of each **for** loop on the variable s, we check if the node is infeasible. Indeed, if no pattern can be constructed for the branching set s, then it is impossible to construct a feasible solution as patterns of s are missing.

4 Heuristics

Heuristics are proceeded during the algorithm in order to speed-up the resolution time by finding feasible solutions allowing to cut more branches.

4.1 Root heuristics

A first heuristic can be processed at root. This is a very important if we want to have a fast algorithm because the higher a branch is cut, the less node are explored. If a very good bound is found at the root, it can prevent to explore a huge part of the tree.

The three heuristics which can be used at root are three variations of a more general heuristic called *Decreasing Order Heuristic* [Joh73]. The key idea is to sort the items is decreasing order and for each item, choose a bin to pack it. The heuristics differs on the way to choose the bin. The general Decreasing Order Heuristic can be summarized as following :

Algorithm 5: Decreasing Order Heuristic

```
while The are items that have not been packed do
    Pick the item with the largest size
    if The item can't be packed in any bin then
        Add a new bin
    end
    Choose a bin able to pack the item
    Put the item in the bin
end
```

The three heuristics implemented are the *First-Fit-Decreasing* (FFD), the *Best-Fit-Decreasing* (BFD) and the *Worst-Fit-Decreasing* (WFD) and they select the bin to pack the item as following :

- **FFD** : Chooses the first bin in which the item can be packed
- **BFD** : Chooses the bin with the least amount of free space in which the item can be packed
- **WFD** : Chooses the bin with the most amount of free space in which the item can be packed

As all these heuristics have to sort the items (complexity in $\mathcal{O}(N \log N)$) and to proceed a loop over each item (complexity in $\mathcal{O}(N)$), their overall complexity is $\mathcal{O}(N \log N)$. Furthermore, if $B^\star$ is the optimal number of bins, it was shown that the results in the worst case for these heuristics are

- **FFD** : $\lfloor 1.7B^\star \rfloor$ [DS13]
- **BFD** : $\lfloor 1.7B^\star \rfloor$ [DS14]
- **WFD** : $2B^\star - 2$ [Joh73]

We can see that these heuristics are very interesting both regarding to their complexity and their efficiency. It is very recommended to enable it when running the BnP algorithm.

4.2 Tree heuristics

Dual bounds found through the tree are proven to be quite good [SV13]. Thus, an efficient primal heuristic allowing to get good primal bounds can lead to a very successful algorithm. Two different types of heuristics are implemented. The first type relies on the MIRUP property conjectured for the ($\mathbb{BPP}$) [DIM16]. The second type of heuristics are described in [WG96] and are based on different rounding strategies, starting from the solution of the relaxed restricted master of a node. Despite having managed to implement diving heuristics, we present it in 5.2.

4.2.1 MIRUP-based strategy

The *MIRUP* property of the (𝔹ℙℙ) is a conjecture which is still open. It has been conjectured [ST95] that given $z^\star_{\mathbb{CR}}$ the solution of the continuous relaxation of (𝕊ℂ𝔹ℙℙ), the following inequality holds :

$$z^\star \leq \lceil z^\star_{\mathbb{CR}} \rceil + 1 \tag{4.2.1}$$

where $z^\star$ is the solution of (𝕊ℂ𝔹ℙℙ) with the integrity constraint. $\lceil z^\star_{\mathbb{CR}} \rceil + 1$ is called the MIRUP bound (Mixed Integer Round-Up Property). The idea of this primal heuristic is first to obtain the solution of (ℝℳ𝔹ℙℙ) at each node and compute the MIRUP bound. Then, the restricted master problem is solved a last time with integrity constraint and with the new constraint :

$$\sum_{p \in \mathcal{P}'} \alpha^p \leq \lceil z^\star_{\mathbb{CR}} \rceil + 1$$

If the solution is feasible and outperform the current best solution, then the best solution is updated. Adding this constraint allows to reduce drastically the search space for an integer solution and ensure that the further UB and LB will have only one unit of difference, leading to a very tight dual GAP. However, the solving process can be long as we solve an integer problem instead of a continuous problem.

4.2.2 Rounding strategies

In 1995, Wäscher and Gau [WG96] presented several rounding strategies grouped in three different categories. The first group is called the *Basic Pattern Approach* and relies directly on the solution of the relaxed master problem. Methods of this group have a name starting with **B**. The second group is called the *Residual Pattern Approach* and is based on the solution of the relaxed master problem where non-integer components have been rounded-down to zero. As this modification may lead to an infeasible solution, methods of this groups aim to construct a feasible solution by solving a *Residual Problem* (find new columns to add to the rounded-down solution to create a feasible solution). Methods of this group have a name starting with **R**. Finally, the third group is called the *Composite Approach* and is a mix of the two first groups. Methods of this group have a name starting with **C**. Several method are proposed in [WG96] and methods **RSUC** and **CSTAOPT** are shown to be the most effective. The following methods have been implemented.

Procedure BRUSIM The simplest procedure consists of simultaneously rounding-up any non-integer component of the (ℝℳ𝔹ℙℙ) to one. The advantages of this procedure are obvious : it is extremely fast and immediately result in a feasible solution. However, it produces very bad primal bounds.

Procedure BRURED Rounding-up the non-integer components simultaneously often creates an over-packaging of some items. Then it may be possible to remove some patterns without violating the packaging constraint. Neumann and Morlock [Ban00] suggested to check whether a pattern can be eliminated without causing a violation of the packaging constraints after having run a BRUSIM procedure.

Procedure BOPT This strategy simply solve (ℝℳ𝔹ℙℙ) on the node pool with integrity constraints on the α^p. This can be done by any optimal solving procedure. However, it may be very long to run this procedure as the problem to solve has integrity constrains. This method provides quite good upper bounds.

Procedure BRUSUC Solving directly (ℝℳ𝔹ℙℙ) with integrity constraints can be quite costly in time. The BRUSUC address to this problem by first fixing integer variable in the solution of (ℝℳ𝔹ℙℙ). Then, (ℝℳ𝔹ℙℙ) is re-optimized (still without integrity constraints) until all the variables are integer. Before each re-optimization, the variable with the higher fractional value is fixed to one. At most P' re-optimization are made.

Procedure CSTAOPT A *cutting pattern* is a pattern containing an item which is packed more than one. The CSTAOPT procedure starts by finding a feasible solution by applying the BRUSUC procedure. Then, it removes iteratively cutting patterns until there are no cutting pattern left. It remains a residual problem to solve as removing cutting patterns can lead to an infeasible solution. The residual pattern is solved by any optimal method.

5 Additional work not implemented

As mentioned above, a dynamic programming allowing to solve the pricing problems for Ryan & Foster method and diving heuristics have also been studied. However, due to their complexity, the have not been implemented successfully. In the following section, we present how it could have been possible to improve the BnP algorithm using these methods.

5.1 Dynamic programming for Ryan & Foster branching rule

As mentioned in 3.2, we can use dynamic programming to solve the subproblems for the generic branching method. Indeed, the branching rules simply fix variables so it is easy to preprocess the subproblem to put it under a classical knapsack problem. However, this is not possible for the Ryan & Foster branching method. The constraints of type $x_i = x_j$ can be handled by creating a "super-object" corresponding to both i and j, which ensure that the constraint is satisfied. But the constraints of type $x_i + x_j \leq 1$ cannot be handled easily. For this branching method, a knapsack problem with conflicts has to be solved. In [SV13], a dynamic programming method is given.

The first thing to do is to extract a *conflict graph* from the subproblem. The vertices correspond to the items in the problem and an edge is added between i and j as soon as there is a constraint $x_i + x_j \leq 1$. Each edge represents a conflict between two items. Once this graph is extracted, the method constructs the *interval representation* of the conflict graph. It is possible to construct this representation in $o(|V| + |E|)$ [COS98] for a conflict graph $G = (V, E)$ when such representation exists. In [SV13], a method is also provided when the interval representation cannot be constructed.

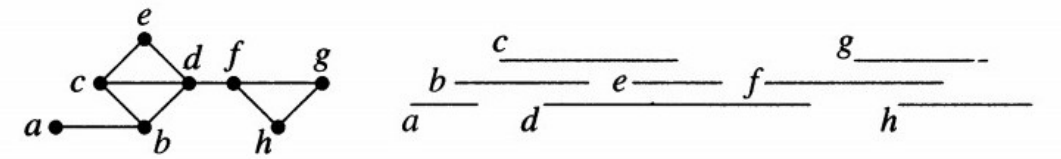

Figure 2: A conflict graph where each edge (i, j) corresponds to a conflict between i and j and its interval representation. It is possible to assign at each interval a coordinate for its left and right endpoints.

Given the conflict graph $G = (V, E)$, where the vertex are indexed in non-decreasing order of the right endpoints of the corresponding intervals $\{I_i = (a_i, b_i)\},\ i \in V$, associated to the subproblem. We can define

$$Q_i = \{j :\ j < i,\ (i, j) \notin E\} \qquad \text{and} \qquad prev_i = \begin{cases} \max\{j,\ j \in Q_i\} & \text{if } Q_i \neq 0 \\ 0 & \text{if } Q_i = 0 \end{cases}$$

for all $i \in V$. The algorithm relies on two observations. The first is that for every pair (i, j) such that $1 \leq j \leq prev_i$, i and j are not in conflict but if $prev_i < j < i$, then i and j are in conflict. If we note $P(i, c)$ the value of an optimal solution of the knapsack with conflict for the first i items and a knapsack capacity c, the second observation is the following

$$P(i, c) = \max\ \{P(prev_i, c - s_i) + p_i,\ P(i - 1, c)\} \tag{5.1.1}$$

where s_i and p_i are the size and the profit corresponding to the item i. Using (5.1.1), we can compute iteratively the values $P(i, c)$ and the optimal value of the knapsack with conflict is given by $P(N, C)$. Then, it is possible to process a backtracking step as in (5.1) to recover the items involved in the optimal cost.

5.2 Column generation based primal heuristics

The BnP algorithm provides very good dual bounds. A good primal heuristic is needed for the algorithm to be efficient. As mentioned in 4.2, several heuristics based on the MIRUP property of the BPP and on rounding strategies are implemented. They are efficient but more sophisticated heuristics could have been implemented. Such heuristics are diving heuristics that are particularly well suited for BnP algorithms. The idea is to explore in a "smart" way an enumeration tree to get good feasible solutions fast. They can be combined with the BnP algorithm in two ways.

The first option is to create a new enumeration tree at each node aiming to get a good feasible solution. For a given node, we consider the optimal solution of ($\mathbb{RMBPP}$) which can be fractional. Starting from this solution, an

enumeration tree is explored in the same way as for the BnP algorithm but the branches are directly created using the variables α^p instead of an item pair (i, j). At each node of this new enumeration tree, the most fractional α^p is fixed to 1 in the up branch and to 0 in the down branch and the master problem is re-optimized with the new constraint. The difference with the BRUSUC procedure is that new columns are generated to get better solutions.

The second option is to change completely the way branches are created in the BnP algorithm and to branch on the α^p. In this case, the whole BnP correspond to the heuristic. However, branching constraints created with the α^p are weaker. In this case, we only have one enumeration tree but it can be deeper as dual bounds are worst.

In [SV13], it is proposed to create a tabu list of the columns in order to diversify the search. The tabu list of columns at a branch-and-price node is the union of the tabu list of its ancestors and the columns chosen in previous child nodes of the ancestor. To get some feasible solutions faster, it is possible to use a maximum depth and a maximum discrepancy parameters in order to only explore a part of the enumeration tree. It can be very long to fully explore the tree and a good, but not necessary optimal, solution can be found using these two parameters.

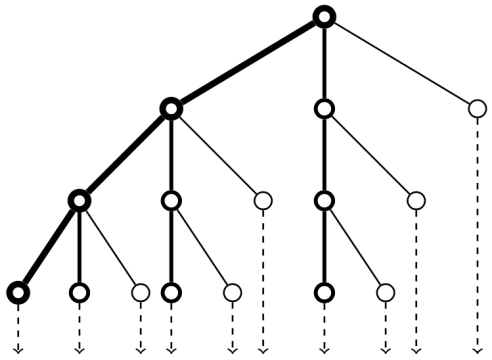

Figure 3: The enumeration tree of a diving heuristic with parameters maximum depth equal to 3 and maximum discrepancy equal to 2. The dotted lines are pure diving into the enumeration tree.

6 Numerical applications

6.1 Implementation details

The code can be found at https://github.com/TheoGuyard/BnP-BinPacking. It is presented in detail how to run the code. In the main directory, the file `README.md` lists most of the things details below. The directory `results` contains the results under `.csv` format when running the algorithm on one dataset (`results/single_dataset/`) or for a benchmark on a dataset directory (`results/benchmark/`). The directory `doc/` contains the papers used during the development of the BnP. The directory `src/` contains the implementation of the algorithm.

6.1.1 Datasets

The directory `data/` contains the instances used to test the BnP. The two Bin-Packing type of datasets used come from the BPPLIB :

- Falkenauer dataset : Divided into two classes of 80 instances each : the first class has uniformly distributed item sizes ('Falkenauer U') with N between 120 and 1000, and C = 150. The instances of the second class ('Falkenauer T') includes the so-called triplets, i.e., groups of three items (one large, two small) that need to be assigned to the same bin in any optimal packing, with N between 60 and 501, and C = 1000. The number of items is noted in the name of the dataset.
- Scholl dataset : Divided into three sets of 720, 480, and 10, respectively, uniformly distributed instances with N between 50 and 500. The capacity C is between 100 and 150 (set 'Scholl 1'), equal to 1000 (set 'Scholl 2'), and equal to 100 000 (set 'Scholl 3'), respectively.

If an other dataset is used, make sure that the file is under the `.bpp` format.

6.1.2 Code file description

File discribed below are under the folder `src/`.

- `main.jl` : Parameter initialization and main code entry
- `bnp.jl` : Branch-and-Price core structure
- `master.jl` : Restricted master problem resolution method
- `node.jl` : Core structure of node processing resolution
- `subproblem.jl` : Subproblem resolution method
- `knapsack.jl` : Knapsack problem methods
- `root_heurisitcs.jl` : Root heuristic algorithms used before the Branch-and-Price algorithm
- `tree_heurisitcs.jl` : Tree heuristic algorithms used within the Branch-and-Price algorithm
- `benchmarks.jl` : Tools to run benchmarks for a given dataset directory
- `display.jl` : Tools to display algorithm results
- `data.jl` : Tools to read data from a dataset file
- `typedef.jl` : Type definition used in the algorithm
- `mip.jl` : Mixed Integer Programming formulation to solve the problem with Gurobi

6.1.3 Running code with `main.jl`

In this file, it is possible to specify the parameters used for the algorithm. It to possible to run the BnP on a single dataset using the function `solve_BnP()` or to use Gurobi to solve this instance with the function `solve_MIP()`. It is also possible to run the BnP for all the instances located in the specified `benchmarksDirectory` with less than `maxItems` in it with a limit on the solving time using `maxTime` parameter. The parameters allowed are :

- `branching_rule` : `"ryan_foster"` or `"generic"`
- `subproblem_method` : `"gurobi"` or `"dynamic"`
- `root_heuristic` : `"FFD"`, `"BFD"`, `"WFD"` or `"None"`
- `tree_heuristic` : `"MIRUP"`, `"BRUSIM"`, `"BRURED"`, `"BOPT"`, `"BRUSUC"`, `"CSTAOPT"` or `"None"`
- `queueing_method` : `"FIFO"`, `"LIFO"` or `"Hybrid"`
- `verbose_level` : `1`, `2` or `3`
- ϵ : between 10^{-16} and 10^{-4}
- `maxTime` : Maximum solving time allowed in seconds

If `verbose_level=1`, the BnP doesn't display anything. If `verbose_level=2`, only the LB and UB found each 10 nodes are outputted. If `verbose_level=3`, all the details of the BnP algorithm are outputted. It is not possible to use both `branching_rule="ryan_foster"` and `subproblem_method="dynamic"` as only the dynamic programming algorithm for the generic branching is implemented.

6.2 Numerical results

In this section, we test the method for different instances. We also test how each parameter acts on the BnP algorithm and which parameters are better.

6.2.1 Dimension influence

The following table shows the BnP result on Falkenauer's and Scholl's datasets. The number of items in each tabular is respectively 60 and 120 for Falkenauer's datasets and 50 and 100 for Scholl's datasets. In the tables, n_{expl} is the total number of nodes explored, $z^\star$ is the solution found, GAP is the last dual-gap and t is the running time in seconds. A maximum running time have been set to 60 seconds.

Dataset	$z^\star$	GAP	n_{expl}	t
Falkenauer_t60_00	20.0	0%	1	0.2
Falkenauer_t60_01	20.0	0%	5	0.3
Falkenauer_t60_02	20.0	0%	4	0.0
Falkenauer_t60_03	20.0	0%	12	0.1
Falkenauer_t60_04	20.0	0%	2	0.0
Falkenauer_t60_05	20.0	0%	1	0.0
Falkenauer_t60_06	20.0	0%	3	0.1
Falkenauer_t60_07	20.0	0%	1	0.0
Falkenauer_t60_08	20.0	0%	2	0.0
Falkenauer_t60_09	20.0	0%	1	0.1
Falkenauer_t60_10	20.0	0%	27	0.2
Falkenauer_t60_11	20.0	0%	1	0.0
Falkenauer_t60_12	20.0	0%	12	0.1
Falkenauer_t60_13	20.0	0%	8	0.0
Falkenauer_t60_14	20.0	0%	1	0.9
Falkenauer_t60_15	20.0	0%	3	0.0
Falkenauer_t60_16	20.0	0%	2	0.0
Falkenauer_t60_17	20.0	0%	1	0.1
Falkenauer_t60_18	20.0	0%	3	0.0
Falkenauer_t60_19	20.0	0%	1	0.0

Dataset	$z^\star$	GAP	n_{expl}	t
Falkenauer_t120_00	42.0	4.76%	284	60
Falkenauer_t120_01	42.0	4.76%	312	60
Falkenauer_t120_02	42.0	4.76%	283	60
Falkenauer_t120_03	42.0	4.76%	289	60
Falkenauer_t120_04	42.0	4.76%	342	60
Falkenauer_t120_05	42.0	4.76%	311	60
Falkenauer_t120_06	42.0	4.76%	330	60
Falkenauer_t120_07	43.0	6.98%	328	60
Falkenauer_t120_08	41.0	2.44%	329	60
Falkenauer_t120_09	42.0	4.76%	387	60
Falkenauer_t120_10	42.0	4.76%	342	60
Falkenauer_t120_11	42.0	4.76%	277	60
Falkenauer_t120_12	42.0	4.76%	371	60
Falkenauer_t120_13	42.0	4.76%	314	60
Falkenauer_t120_14	42.0	4.76%	334	60
Falkenauer_t120_15	42.0	4.76%	354	60
Falkenauer_t120_16	42.0	4.76%	312	60
Falkenauer_t120_17	42.0	4.76%	271	60
Falkenauer_t120_18	42.0	4.76%	344	60
Falkenauer_t120_19	42.0	4.76%	322	60

Figure 4: BnP result on Falkenauer's datasets with 60 and 120 items using the generic branching scheme with dynamic programming, a FFD root-heuristic, a BRUSUC tree-heuristic, an Hybrid queueing method and $\epsilon = 10^{-6}$.

Dataset	$z^\star$	GAP	n_{expl}	t
Scholl_1/N1C1W1_A	25.0	1%	356	60
Scholl_1/N1C1W1_B	31.0	1.61%	353	60
Scholl_1/N1C1W1_C	20.0	0%	7	0.271
Scholl_1/N1C1W1_D	28.0	0%	40	1.069
Scholl_1/N1C1W1_E	26.0	0%	1	0.064
Scholl_1/N1C1W1_F	27.0	0%	47	1.085
Scholl_1/N1C1W1_G	25.0	0%	1	0.062
Scholl_1/N1C1W1_H	31.0	0%	1	0.041
Scholl_1/N1C1W1_I	25.0	3.71%	340	60
Scholl_1/N1C1W1_J	26.0	1.54%	385	60
Scholl_1/N1C1W1_K	26.0	2.56%	407	60
Scholl_1/N1C1W1_L	33.0	0%	58	2.558
Scholl_1/N1C1W1_M	30.0	1.67%	437	60
Scholl_1/N1C1W1_N	26.0	4.26%	296	60
Scholl_1/N1C1W1_O	32.0	0%	1	0.072
Scholl_1/N1C1W1_P	26.0	0%	272	35.751
Scholl_1/N1C1W1_Q	28.0	0%	1	0.08
Scholl_1/N1C1W1_R	25.0	2.8%	299	60
Scholl_1/N1C1W1_S	28.0	0%	1	0.138
Scholl_1/N1C1W1_T	28.0	0%	1	0.087

Dataset	z^*	GAP	n_{expl}	t
Scholl_1/N2C1W1_A	48.0	1.39%	286	60
Scholl_1/N2C1W1_B	49.0	1.02%	363	60
Scholl_1/N2C1W1_C	46.0	1.43%	344	60
Scholl_1/N2C1W1_D	50.0	1.5%	400	60
Scholl_1/N2C1W1_E	58.0	0.86%	358	60
Scholl_1/N2C1W1_F	50.0	1.56%	302	60
Scholl_1/N2C1W1_G	60.0	1.11%	335	60
Scholl_1/N2C1W1_H	52.0	1.44%	308	60
Scholl_1/N2C1W1_I	62.0	0.0%	1	0.166
Scholl_1/N2C1W1_J	59.0	0.0%	1	0.197
Scholl_1/N2C1W1_K	55.0	0.0%	1	0.22
Scholl_1/N2C1W1_L	55.0	0.0%	1	0.221
Scholl_1/N2C1W1_M	46.0	0.68%	278	60
Scholl_1/N2C1W1_N	48.0	0.0%	128	9.052
Scholl_1/N2C1W1_O	48.0	1.39%	423	60
Scholl_1/N2C1W1_P	54.0	0.0%	1	0.301
Scholl_1/N2C1W1_Q	46.0	0.54%	280	60
Scholl_1/N2C1W1_R	56.0	0.0%	1	0.226
Scholl_1/N2C1W1_S	45.0	0.95%	367	60
Scholl_1/N2C1W1_T	52.0	0.82%	356	60

Figure 5: BnP result on Scholl's datasets with 50 and 100 items using the generic branching scheme with dynamic programming, a FFD root-heuristic, a BRUSUC tree-heuristic, an Hybrid queueing method and $\epsilon = 10^{-6}$.

We can see that the Falkenauer's datasets with 60 item are solved in less than a second. For most of them, only one node have been explored. For these datasets, the root heuristic provides a first upper bound at a value near $\overline{z} \simeq 22$. Then, the columns generated at the roots are enough to obtain the optimal solution with the BRUSUC tree heuristic. For the datasets where several nodes are explored, the branches are cut quickly because of the Hybrid queueing method, making the BnP close the instance fast. However, for Falkenauer's datasets with 120 items, none were solved at optimality under a minute. For these datasets, the optimal solution is $z^* = 40$. Even though the optimal solution is not reached, the dual bound met the optimal solution in the very first nodes. The dual bounds created by the column generation are really strong. Though, the primal bound struggles to decrease to its minimal value. As mentioned in 5.2, a better primal heuristic such as diving heuristics could have improve the algorithm a lot. The primal heuristics used in the algorithm are not strong enough. The same conclusions can be drawn from results of Scholl's datasets. For these last datasets, we can also see the impact of the root heuristics as most of the solved instances only explored the root node before converging.

6.2.2 Branching rule and subproblem method comparison

We can also compare the different branching rules and the subproblem resolution methods. Two performance criteria are be used : the number of nodes explored and the total running time. In the following table, these two criteria are presented for the Ryan & Foster branching rule with Gurobi (rf), the generic branching rule with Gurobi (gg) or with dynamic programming (gd).

Dataset	n^{rf}_{expl}	n^{gg}_{expl}	n^{gd}_{expl}	t^{rf}	t^{gg}	t^{gd}
Falkenauer_t60_00	2	2	2	0.2775	0.3039	**0.1879**
Falkenauer_t60_01	**6**	15	15	0.2588	0.2943	**0.1069**
Falkenauer_t60_02	2	2	2	0.0985	0.1283	**0.0435**
Falkenauer_t60_03	**11**	17	17	0.2121	0.3465	**0.0999**
Falkenauer_t60_04	2	2	2	0.1099	0.1174	**0.0308**
Falkenauer_t60_05	1	1	1	0.1088	0.1161	**0.0305**
Falkenauer_t60_06	8	**6**	**6**	0.1482	0.1523	**0.0340**
Falkenauer_t60_07	2	2	2	0.1020	0.1212	**0.0319**
Falkenauer_t60_08	1	1	1	0.0912	0.1038	**0.0296**
Falkenauer_t60_09	1	1	1	0.1972	0.1202	**0.0303**

Figure 6: Comparison of the branching rule and the subproblem resolution method on the Falkenauer's datasets using FFD root-heuristic, BRUSUC tree-heuristic, Hybrid queueing method and $\epsilon = 10^{-6}$.

There is no clear difference for the number of nodes explored. No method is better than others. Nevertheless, we can note that in the case of the generic branching method, the resolution with Gurobi or with dynamic programming gives the same number of nodes explored. Even though a knapsack problem can have several optimal solutions, the columns generated with Gurobi and with dynamic programming seems to be similar. Regarding to the running time, there is no clear difference between the two branching methods when using Gurobi to solve the subproblems.

However, there is a significant amelioration when dynamic programming is used with the generic branching scheme. The dynamic resolution is faster than Gurobi as a IP solver is suited for general problems whereas the dynamic method is specific for the knapsack problem. As mentioned in 5.1, a dynamic programming method for the Ryan & Foster branching scheme could have improve a lot its running time.

6.2.3 Root-heuristic comparison

The following table shows the root-heuristics performance. For each Falkenauer dataset of a given size, we note the relative error of the root heuristic bound $|z^{root} - z^*|/z^*$ where z^* is the optimal value for the dataset and z^{root} is the primal bound provided by the heuristic. The value in the table is the average relative error on all the datasets of a given size.

Datasets	# datasets	FFD	BFD	WFD
Falkenauer_u120_xx	20	75.6%	75.6%	**74.6%**
Falkenauer_u250_xx	20	76.2%	76.3%	**75.9%**
Falkenauer_u500_xx	20	77.7%	77.7%	**77.5%**
Falkenauer_u1000_xx	20	78.4%	78.3%	**78.2%**
Falkenauer_t60_xx	20	84.0%	84.0%	84.0%
Falkenauer_t120_xx	20	85.5%	85.5%	85.5%
Falkenauer_t249_xx	20	85.5%	85.5%	85.5%
Falkenauer_t501_xx	20	86.2%	86.2%	86.2%

Figure 7: Comparison of the root-heuristic on the Falkenauer's datasets

The three heuristics give quite good upper bound to begin the tree exploration. Even if the WFD gives a better result in four cases, the results are not significantly different. These heuristics provides very good first bound regarding to their complexity so it would be unwise not to use it.

6.2.4 Queuing method comparison

Finally, we can compare the different queuing methods. We test all the Falkenauer datasets with 60 items for the tree different options. No root/tree heuristics are proceeded as we wan to retrieve only the impact of the queuing method. In the table, we note the number of nodes explored until convergence. If the optimal value is not met under 60 seconds, the algorithm is stopped and theses cases are represented by "-" in the tables.

Dataset	FIFO	LIFO	Hybrid
Falkenauer_t60_00	1	1	1
Falkenauer_t60_01	5	-	-
Falkenauer_t60_02	4	5	5
Falkenauer_t60_03	17	8	8
Falkenauer_t60_04	2	2	2
Falkenauer_t60_05	1	1	1
Falkenauer_t60_06	2	-	-
Falkenauer_t60_07	1	1	1
Falkenauer_t60_08	2	2	2
Falkenauer_t60_09	1	1	1

Dataset	FIFO	LIFO	Hybrid
Falkenauer_t60_10	24	-	-
Falkenauer_t60_11	1	1	1
Falkenauer_t60_12	18	12	12
Falkenauer_t60_13	8	-	-
Falkenauer_t60_14	1	1	1
Falkenauer_t60_15	2	12	12
Falkenauer_t60_16	2	2	2
Falkenauer_t60_17	1	1	1
Falkenauer_t60_18	4	4	4
Falkenauer_t60_19	1	1	1

Figure 8: Comparison of the root-heuristic on the Falkenauer's datasets with no tree/root heuristics and a generic branching method with dynamic programming.

We can notice that for the FIFO queueing method, all the instances are solved under a minute whereas it is not the case for the two other queueing methods. It seems that the LIFO and Hybrid method struggle to cut branches because no good upper bounds are found. The Hybrid method is first a FIFO and then a LIFO when a first upper bound is found. We can notice that this method is not very efficient and it seems that the first upper bound found is not good and doesn't help after to cut branches. The FIFO method looks like the best method when no heuristics are proceeded.

7 Conclusion

In this report, we try to solve the Bin Packing problem with a Branch and Price approach. We have seen that this is relevant as this problem has an easy and a hard constraint. These constraints can be handled using a column generation approach, by using patterns instead of raw items and by restricting the problem to a subset of the possible patterns. We also saw that the problem can be continuously relaxed in order to be solved quickly but we have to introduce branching rules to get the integer optimal solution. For theses branching rules, two methods were treated : Ryan & Foster method and a more generic branching method. The first method allows to keep only one subproblem per node of the enumeration tree and it can be solved with Gurobi. The seconds methods preserve some particular properties but several subproblem have to be solved. The conservation of these particular properties allows to solve the subproblems either with Gurobi or using a dynamic programming algorithm. We have also presented different queueing methods to explore the branching tree and different ways to get upper bounds at the root or within the tree. Results are quite good for small instances but the algorithms struggles a bit when there are too many items. Some amelioration can still be implemented to improve the algorithm such as diving heuristics or a dynamic programming resolution for the Ryan & Foster method.

References

[Joh73] David S Johnson. "Near-optimal bin packing algorithms". PhD thesis. Massachusetts Institute of Technology, 1973.

[RF81] DM Ryan and EA Foster. "An integer programming approach to scheduling". In: (1981).

[Van+94] Pamela H Vance et al. "Solving binary cutting stock problems by column generation and branch-and-bound". In: *Computational optimization and applications* 3.2 (1994), pp. 111–130.

[ST95] Guntram Scheithauer and Johannes Terno. "The modified integer round-up property of the one-dimensional cutting stock problem". In: *European Journal of Operational Research* 84.3 (1995), pp. 562–571.

[WG96] Gerhard Wäscher and Thomas Gau. "Heuristics for the integer one-dimensional cutting stock problem: A computational study". In: *Operations-Research-Spektrum* 18.3 (1996), pp. 131–144.

[COS98] Derek G Corneil, Stephan Olariu, and Lorna Stewart. "The ultimate interval graph recognition algorithm?" In: *SODA*. Vol. 98. 1998, pp. 175–180.

[Ban00] Udo Bankhofer. "Quantitative Unternehmensplanung 2". In: *Modulhandbuch* 3 (2000), p. 1976.

[DS13] Gyorgy Dosa and Jiri Sgall. "First Fit bin packing: A tight analysis". In: *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2013.

[SV13] Ruslan Sadykov and Franccois Vanderbeck. "Bin packing with conflicts: a generic branch-and-price algorithm". In: *INFORMS Journal on Computing* 25.2 (2013), pp. 244–255.

[DS14] Gyorgy Dosa and Jiri Sgall. "Optimal analysis of Best Fit bin packing". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2014, pp. 429–441.

[DIM16] Maxence Delorme, Manuel Iori, and Silvano Martello. "Bin packing and cutting stock problems: Mathematical models and exact algorithms". In: *European Journal of Operational Research* 255.1 (2016), pp. 1–20.