Advanced Mathematical Programming

Branch-and-Price for the Bin-Packing Problem

April 26, 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 instead of using a classical LP solver is presented. Furthermore, some heuristics to be proceeded before and during the tree exploration are provided. In the first section, the main mathematical background of a BnP method suited for the Bin-Packing problem is presented. In the second section, the main structure of the BnP algorithm is presented with some details about the simplest parts. The third section presents the node processing method for the Ryan & Foster and the Generic branching scheme. Then, the heuristics used before and during the exploration of the tree are presented. Some other method not implemented but which could have improve the algorithm are presented in the fifth section. Finally, some details about the implementation and 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 *size* in *bins* with equal *capacity* while minimizing the number of bin used. 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. Thus, the Bin-Packing Problem can be written under the following form :

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

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

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

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. Thus, the solution is a set of patterns and each patter correspond to a bin. The constraint ensure that each item is packed in a pattern.

On thing to notice is that $|P|$ (the number of variables) is huge. 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 ($\mathbb{RMBPP}$) will be the same as the optimal solution of ($\mathbb{SCBPP}$). The problem is to choose the right patterns to include in $\mathcal{P}'$. As $\mathcal{P}'$ will 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 will lead to an supra-optimal solution comparing to the integer form 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 the pattern 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 $\mathcal{P}' \cup \mathbf{y}$ will be 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 easy to solve. The key mechanism of the Branch-and-Price algorithm will be to solve sequentially ($\mathbb{RMBPP}$) and ($\mathbb{SPBPP}$) until a solution $z_{sp}{}^\star \geq 0$ is obtained. At this stage, ($\mathbb{RMBPP}$) will be solved at optimality without the integrity constraint on α^p. 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 solution. 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 will select fractional variables in the solution to branch on. Child nodes are created under the node just solved with new constraints in order to obtain solutions where the selected branching variable 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 will 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 will impose that $w_{ij} \geq 1$ (items i and j are always together) and in the other branch, we will 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 ($\mathbb{BPP}$) 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 allow to keep a single subproblem at each node but doesn't preserve the knapsack structure of the pricing problem. The subproblems will rather be knapsack with conflicts problems, harder to solve than the usual knapsack problem. The second branching rule which 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 will 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 explained at the end of this report but was not implemented. For the Generic branching method, the pricing problem is solved either with a LP solver or with a dynamic programming algorithm able to solve knapsack problems. 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. Finally, we will see two different way to add the nodes to the queue : FIFO and LIFO. We will see if the queueing methods affects the running time or the number of nodes explored during the BnP algorithm.

2.2 BnP structure

The structure of the branch of price algorithm is the same whatever branching-rule, queueing method, heuristics and other parameters are chosen. It can be resumed by the following algorithm :

```
Algorithm 1: Branch and Price
  Input: The items and their size $s_1, \ldots, s_N$, the bin capacity $C$, an upper bound on
         the number of bins $B$, a precision $\epsilon$
  // 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 to 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 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 will have a dedicated section to be explained.

2.2.1 Column management

The column pool contains all the columns which are created by the subproblems. We need to add an artificial column which allow 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 will 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. In order to save speed and memory space, the column pool is global to all nodes and each node have its proper node pool which is 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 at least assume that the total number of bin used will be 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 set $B = N$. The integer relaxation of ($\mathbb{BPP}$) gives also a lower bound for the integer ($\mathbb{BPP}$) : we can initially set $LB = \left\lceil \sum s_i/C \right\rceil$ [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 the 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. The LIFO method allows to find quickly good upper bounds but it will cut branches more 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 primal 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 has been 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 allow to see how close the algorithm is from the solution by bounding the optimal solution by LB and UB at each node processing. When a node in infeasible, the node 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 solution provided by the node (see 4.2).

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 correspond 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 will allow to keep a single subproblem per node but it will not be as easy to solve as a classical knapsack problem.

3.1.1 Branching rules

When a node is solved to optimality and that two items i and j are found to create a branch, we have to create one child where $w_{ij} \leq 0$ (the down branch) and one child where $w_{ij} \geq 1$ (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 created pattern 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 with i and j in it). The subproblem handle itself whether to generate patters with (resp. without) i and j because the reduced cost will be too high if too many pattern with (resp. without) i and j have already been be created.

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

3.1.2 Subproblem resolution

The subproblem will always have to initial structure presented in 1.2.3 with the new node branching constraints. 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 is very hard to implement. Despite having managed to implement this second method, we present it in 5.1.

3.2 Generic method

The generic branching method introduce 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.

3.2.1 Branching rules

When a node is solved to optimality and that two items i and j are found to create a branch, we have to create one child where $w_{ij} \leq 0$ (the down branch) and one child where $w_{ij} \geq 1$ (up branch). For the down branch, we will create a subproblem generating patterns with $x_i = 0$ and 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 $x_i = 1, x_j = 0$ so we impose that

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

in order to reduce the solution space. For the up branch, we create one subproblem generating patterns with $x_i = x_j = 0$ and patterns with $x_i = x_j = 1$. Thus, the solution will be composed of on 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 will also impose that

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

in order to reduce the solution space. For up and down branching rules, we can notice that the variables x_i and x_j are fixed so the subproblem keep its knapsack structure if a preprocess is made. After fixing the branching variable, a knapsack problem will remained to be solved (see 3.2.2).

The difficulty lies in the method to process several branches in a row in the tree. 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,

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

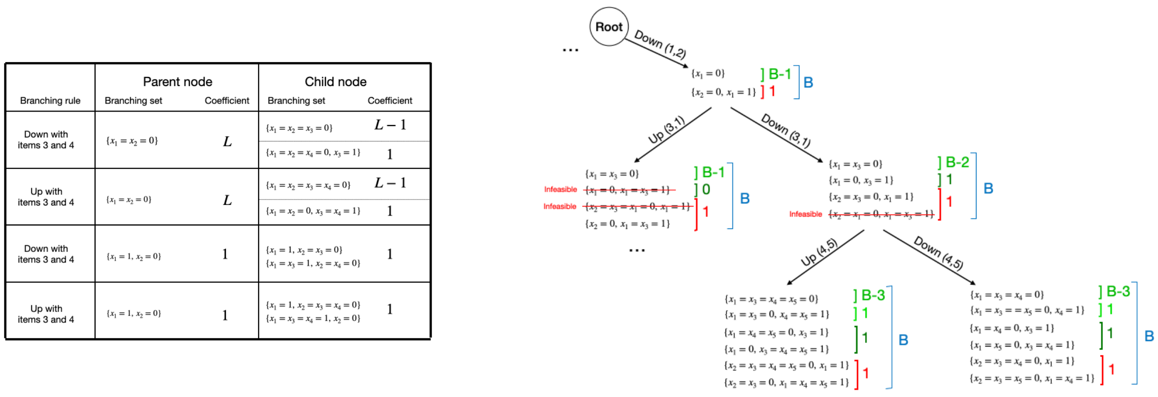
is a branching set with branching rules $\{x_1 = 0, x_2 = 1\}$ and $\{x_3 = 0, x_4 = 1\}$ and with coefficient 1 and

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

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 set is $\mathbf{L > 1}$:**
  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 set are identical to the parent branching set with a little modification. In the first, the item i is added to the variable set to 0 by 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 by the branching rules and the coefficient is set to **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 set are identical to the parent branching set with a little modification. In the first, the item i and j are added to the variable set to 0 by the branching rules and the coefficient is set to $\mathbf{L-1}$. In the second, the item i and j are added to the variables set to 1 by the branching rules and the coefficient is set to **1**.
- **The coefficient of the parent node 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 will 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 two new rule to the child branching set containing the rule of the parent branching set and either $x_i = 0$ or $x_i = 1, x_j = 0$. This new branching set has a coefficient equal to **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 two new rule to the child branching set containing the rule of the parent branching set and either $x_i = x_j = 0$ or $x_i = x_j = 1$. This new branching set has a coefficient equal to **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 each branching set add to B in order to satisfy the constraint $\sum_{p\in\mathcal{P}'} \alpha^p = B$ of ($\mathbb{RMBPP}$) for 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 rules 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.



	Parent node		Child node	
Branching rule	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

Root
...
Down (1,2)
{x1 = 0}] B-1
{x2 = 0, x1 = 1}] 1
B
Up (3,1)
Down (3,1)
{x1 = x3 = 0}] B-1
Infeasible ~~{x1 = 0, x1 = x3 = 1}~~] 0
Infeasible ~~{x2 = x3 = x1 = 0, x1 = 1}~~
{x2 = 0, x1 = x3 = 1}] 1
B
...
{x1 = x3 = 0}] B-2
{x1 = 0, x3 = 1}] 1
{x2 = x3 = 0, x1 = 1}
Infeasible ~~{x2 = x1 = 0, x1 = x3 = 1}~~] 1
B
Up (4,5)
Down (4,5)
{x1 = x3 = x4 = x5 = 0}] B-3
{x1 = x3 = 0, x4 = x5 = 1}] 1
{x1 = x4 = x5 = 0, x3 = 1}
{x1 = 0, x3 = x4 = x5 = 1}] 1
{x2 = x3 = x4 = x5 = 0, x1 = 1}
{x2 = x3 = 0, x1 = x4 = x5 = 1}] 1
B
{x1 = x3 = x4 = 0}] B-3
{x1 = x3 = = x5 = 0, x4 = 1}] 1
{x1 = x4 = 0, x3 = 1}
{x1 = x5 = 0, x3 = x4 = 1}] 1
{x2 = x3 = x4 = 0, x1 = 1}
{x2 = x3 = x5 = 0, x1 = x4 = 1}] 1
B

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 in the case of 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 to solve the subproblems. 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.

The first step to do before the dynamic programming process is to preprocess the data and treat the variables already fixed by 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 plus the preprocess cost and the items in the pattern are the items used in the knapsack 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 $i = 1, \dots, I^{knap}$, we have to introduce $t_{i,c}^{knap}$, the maximal cost 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 to find the optimal cost of the knapsack and the items involved in this optimal cost :

Algorithm 2: Knapsack problem

```
for $i = 1, \dots, I^{knap}$ do
    for $c = 1, \dots, C^{knap}$ 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 $i = I^{knap}, \dots, 1$ 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 a solver, some problems 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 won't be solved in the same amount of time. A problem with few items but with a huge capacity can be slower to solve than a problem with more items. 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 method. A node always contains some branching sets but in the case of the Ryan & Foster method, it only has one branching set with no rule 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 ($\mathbb{RMBPP}$) and σ are the constraints associated to each branching subset (only one in the case of Ryan & Foster method). Whatever the method set to process the node, the algorithm is always the same :

Algorithm 4: Node processing

```
// Initialization
nodePool ← filterColumnPool()
$\pi, \sigma$, solution, value ← solveMaster()
nodelb ← $\sum \pi_i$
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 subsets do
        for $r$ in branching rules of $s$ do
            reducedCost, column ← solveSubproblem($\pi, \sigma_s$)
            if reducedCost $< \infty$ then
                minReducedCost = min(minReducedCost ; reducedCost)
                if reducedCost < 0 then
                    Add column to the node pool and the column pool
                    Update the master problem data with the new column
                end
                nodelb = nodelb + reducedCost
            end
        end
        if all columns have reducedCost $= \infty$ for the subset $s$ then
            Node is infeasible
            break
        end
    end
    if $|nodelb - nodelb| < \epsilon$ then
        Return the solution of the master problem (node is infeasible if it
         contains the artificial column)
        break
    end
    $\pi, \sigma$, 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 subset s, then it is impossible to construct a feasible solution as at least one of the pattern needed to construct a solution will be missing.

4 Heuristics

Heuristics are proceeded during the algorithm in order to speed-up the resolution time by finding feasible solutions so as 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 will be 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 the object. The heuristics differs on the method to choose the bin. The general Decreasing Order Heuristic can be resumed 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* (FFD), the *Best-Fit* (BFD) and the *Worst-Fit* (WFD) algorithm 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 of the items (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 result 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. There exist many other heuristics which can be found at the Wikipedia page of the BPP.

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 of the ($\mathbb{BPP}$) [DIM16]. The second type of heuristics are described in [WG96] and are base 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 ($\mathbb{BPP}$) 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 ($\mathbb{SCBPP}$), 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 ($\mathbb{SCBPP}$) 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 ($\mathbb{RMBPP}$) at each node and compute the MIRUP bound. Then, the restricted master problem is solved 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 allow to reduce drastically the search space for an integer solution and ensure that UB and LB will have only one unit of difference, leading to a very tight GAP. However, the solving process can still be long as we solve an integer problem instead of a relaxed problem.

4.2.2 Rounding strategies

In 1995, Wäscher and Gau have presented several rounding strategies grouped in three different categories. This 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, the methods of this groups aims 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. A lot of methods are presented in this paper but only the following methods have been implemented.

Procedure BRUSIM The simplest procedure consists of simultaneously rounding-up any non-integer component of the ($\mathbb{RMBPP}$) 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 feasible to remove some patterns without violating the packaging constraint. Neumann and Morlock [Ban00] suggest 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 ($\mathbb{RMBPP}$) 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 ($\mathbb{RMBPP}$) with integrity constraints can be quite costly in time. The BRUSUC address to this problem by first fixing integer variable in the solution of ($\mathbb{RMBPP}$). Then, ($\mathbb{RMBPP}$) 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.

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 will remain a residual problem to be solved as removing cutting pattern 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 able 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. In the following, 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

5.2 Column generation based primal heuristics

6 Numerical applications

6.1 Implementation details

The code can be found at https://github.com/TheoGuyard/BnP-BinPacking. It is also presented in detail how to run the code. In the main directory, the file `README.md` list 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 subdirectory `data/` contains the instances used to test the BnP. The two different Bin-Packing type of datasets 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.
- 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 in the folder `src/`.

- `main.jl` : Parameter initialization and 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 solving 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` : Run benchmarks for a given dataset directory
- `display.jl` : Display algorithm results
- `data.jl` : Read data from 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()`. Finally, it is 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.

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 datasets with 60 items. 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.

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	23.0	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 2: 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 and Hybrid queueing method and a precision of 10^{-6}.

[TODO : Anayse]

Dataset	n_{expl}^{rf}	n_{expl}^{gg}	n_{expl}^{gd}	t^{rf}	t^{gg}	t^{gd}
`Falkenauer_t60_00`	2	2	2	2.775	3.039	**1.879**
`Falkenauer_t60_01`	**6**	15	15	2.588	2.943	**1.069**
`Falkenauer_t60_02`	2	2	2	0.985	1.283	**0.435**
`Falkenauer_t60_03`	**11**	17	17	2.121	3.465	**0.999**
`Falkenauer_t60_04`	2	2	2	1.099	1.174	**0.308**
`Falkenauer_t60_05`	1	1	1	1.088	1.161	**0.305**
`Falkenauer_t60_06`	8	**6**	**6**	1.482	1.523	**0.340**
`Falkenauer_t60_07`	2	2	2	1.020	1.212	**0.319**
`Falkenauer_t60_08`	1	1	1	0.912	1.038	**0.296**
`Falkenauer_t60_09`	1	1	1	1.972	1.202	**0.303**

Figure 3: 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 a precision of 10^{-6}.

6.2.2 Branching rule and subproblem method comparison

We can also compare the different branching rules and the subproblem resolution method. Two good criteria which can be used are the number of nodes explored and the total running time. rf denotes the Ryan & Foster branching rule with Gurobi to solve the subproblems, gg and gd denotes respectively the generic branching rule for a subproblem resolution using Gurobi and dynamic programming.
[TODO : Analyse]

6.2.3 Root-heuristic comparison

The following table shows the root-heuristic performance. For each dataset size, we note the number of time each method gave the best result among the three different root-heuristics.

Datasets	Number of datasets	FFD	BFD	WFD
`Falkenauer_u120_xx`	20	20	20	18
`Falkenauer_u250_xx`	20	20	19	11
`Falkenauer_u500_xx`	20	20	20	11
`Scholl_1`	540	441	441	540

Figure 4: Comparison of the root-heuristic on the Falkenauer's datasets

6.2.4 Tree-heuristic comparison

6.2.5 Queuing method comparison

References

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

[Tot80] Paolo Toth. “Dynamic programming algorithms for the zero-one knapsack problem”. In: *Computing* 25.1 (1980), pp. 29–45.

[RF81] DM Ryan and EA Foster. “Rn 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.

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

[BHB09] AK Bhatia, M Hazra, and SK Basu. “Better-fit heuristic for one-dimensional bin-packing problem”. In: *2009 IEEE International Advance Computing Conference*. IEEE. 2009, pp. 193–196.

[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 Fran**c**cois 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.