INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Advanced Mathematical Programming

Branch-and-Price for the Bin-Packing Problem

April 9, 2020

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

Contents

1 Introduction **2**
1.1 Bin-Packing Problem . . . 2
1.2 Branch-and-Price approach . . . 2
1.2.1 Set covering formulation . . . 2
1.2.2 Restricted master problem . . . 3
1.2.3 Subproblem . . . 3
1.2.4 Branching . . . 3
1.3 Resolution method . . . 4

2 Branch-and-Price outline **5**

3 Ryan-Foster branching rule **6**
3.1 Node processing . . . 6
3.2 Subproblem resolution . . . 6

4 Generic branching constraints **7**
4.1 Node processing . . . 7
4.2 Subproblem resolution . . . 7

5 Heuristics **8**
5.1 Root heuristics . . . 8
5.2 Tree heuristics . . . 8

6 Numerical applications **9**
6.1 Implementation detail . . . 9
6.2 Numerical results . . . 9
6.3 Result analysis . . . 9

1 Introduction

1.1 Bin-Packing Problem

In this report, we focus on the Bin-Packing Problem. It consists of putting objects 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 :

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

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 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 combination of items in 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 :

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

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. 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 the N items, namely $\sum_{n=1}^{P} \binom{P}{n}$. Even for small instances, the number of variable is not trackable. 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 $\mathcal{P}' \subset \mathcal{P}$ or cardinal P' containing only a fraction of the available patterns. We can solve ($\mathbb{SCBPP}$) on this restricted number of pattern, which is more tracktable. This leads to a sub-optimal bound. This new problem is called the restricted master problem :

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

If the set $\mathcal{P}'$ is well chosen and contained the optimal patterns, then the optimal solution of ($\mathbb{RMBPP}$) will be the same as the optimal solution of ($\mathbb{SCBPP}$) (and also the solution of ($\mathbb{BPP}$)). The problem is to choose the rights 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 $\alpha_p \in [0,1]\ \forall\, p = 1, \ldots, P'$. 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 ($\mathbb{BPP}$).

1.2.3 Subproblem

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

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

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 will penalize patterns which can not improve the set $\mathcal{P}'$ in the sense that if a pattern p has a high reduced cost, the solution of ($\mathbb{RMBPP}$) with $\mathcal{P}'$ or with $\mathcal{P}' \cup p$ will be the same. In fact, 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 $z_{sp}^\star < 0$.

On 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. As ($\mathbb{RMBPP}$) is a relaxation of ($\mathbb{SCBPP}$), then we have to introduce a new mechanism in order to obtain feasible solutions.

1.2.4 Branching

Now, let's remember that ($\mathbb{RMBPP}$) is a relaxation of ($\mathbb{SCBPP}$) so it gives a supra-optimal solution. In order to have the optimal integer solution, we will 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 with its local branching rules. When a node is solved to optimality (*i.e.* ($\mathbb{RMBPP}$) and ($\mathbb{SPBPP}$) are solved sequentially until optimality), we will select a fractional variables to branch on. Then, child node will be created under the node just solved with new constraints in order to have a non-fractional variable for the rest of the branch. The tree will be explored node after node until all the nodes are explored. The key idea will be 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 always together) and in the other branch, we will impose that $w_{ij} \leq 0$ (items i and j always separated).

1.3 Resolution method

To solve the Bin-Packing Problem, we will 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 problem. The subproblems will rather be knapsack with conflicts problems, harder to solve than the usual knapsack problem. The second branching rule which will 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 will have to be solved.

For the Ryan & Foster branching rule, two methods will be proposed for the subproblem resolution. The first one is to use a classical solver so as to model and solve the subproblem with the new branching constraints. We will also see how to solve the subproblems with dynamic programming [Tot80]. For the Generic branching rule, we will also propose to solve the subproblem either with a solver or with a dynamic programming algorithm [SV13].

In addition, we will see how to construct an heuristic solution before the tree exploration in order to set better initial bounds. Three variations of a decreasing-size-order packing algorithm will be proposed [BHB09].

2 Branch-and-Price outline

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 $s_1, \dots, s_N$, 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 $\leftarrow +\infty$
Initialize LB $\leftarrow -\infty$
Process an heuristic to find a better UB (see 5.1)
// Tree exploration
while the queue is non-empty do
    Process a tree heuristic to tighten UB (see 5.2)
    Pop the first node in the queue
    Proceed the node according to the branching rule set (see 3 or 4)
    Find the branching variable and add the two child nodes to the queue
    Update LB and UB with the solution found in the node
    Cut the branch that cannot improove the upper bound
    if $|LB - UB| < \epsilon$ then
        return the best solution associated to UB
    end
end
```

We will explain with more details some of the steps.

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 have a solution. This artificial column contains all the objects and have a very high cost. It the solution of a master problem contains this artificial column, we know that the master problem is infeasible because of the branching rules.

In the loop, we always choose the first node in the queue but the nodes won't always been stored in that same order in the queue. When adding the two child nodes to the queue, two method 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 is to put the new nodes at the end of the queue (FIFO) in order to proceed a Breadth-First-Search. The LIFO method will allow to find quickly good upper bounds but it will cut branches more deeply in the tree. The FIFO method will allow to cut branches at a high level in the tree but will be slower to get good bounds. The best exploration method depends on the structure of the problem and in general, there is no one better than the other.

After the node proceeding, it is possible that the solution given is integer. In this case, we can see if this solution improve the current UB and if it i 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.

Once the bounds are updated, we take a look at the nodes left in the queue. If a node has a lower bound greater than UB, then this node can't improve the current best solution. Thus, we can stop the exploration of the branch after this node.

When $UB \simeq LB$, 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.

The method to proceed a node, to run heuristics before and within the tree will be explained in the following sections. Then, we will evaluate the BnP with some performance criteria.

3 Ryan-Foster branching rule

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 Node processing

When a node is processing in the tree, it have to solve ($\mathbb{RMBPP}$) with the additional branching constraints.

3.2 Subproblem resolution

4 Generic branching constraints

4.1 Node processing

4.2 Subproblem resolution

5 Heuristics

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

5.1 Root heuristics

A first heuristic can be proceeded 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 choice of the bin to pack the object. The general Decreasing Order Heuristic can be proceeded as following.

Algorithm 2: Decreasing Order Heuristic

```
while The are items that have not been packed do
    Pick the biggest item
    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 (in $\mathcal{O}(N \log N)$) and to proceed a loop over each of the items (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 in the Wikipedia page of the BPP.

5.2 Tree heuristics

6 Numerical applications

6.1 Implementation detail

6.2 Numerical results

6.3 Result analysis

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.

[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.