

ALGORITHMS AND COMPUTABILITY  
PROJECT REPORT  
THE 2D BIN PACKING PROBLEM

Igor Saluch  
Jianhao Luo

November 19, 2019

# 1 Introduction

It is a common problem of trying to pack a lot of items in limited space. For example, truck loaders deal with such problem every day – how to load a truck as efficiently as possible, but without overloading the vehicle. It is also sometimes called “a thief problem”, where a burglar tries to pack as many items as possible into their backpack.

In computer science, we also deal with this problem very often. Packing virtual machines on a server with limited memory is a problem set on completely different kind of resources, but still solved using the same principles. For example, we want to present the user their photos on the thumbnail view, without cropping, while retaining the sizes (or at least ratios).

## 2 Problem description

The *bin packing problem* is an combinatorial optimization problem. According to Wikipedia[1],

*[...] items of different volumes must be packed into a finite number of bins or containers each of a fixed given volume in a way that minimizes the number of bins used.*

The problem has many variations, it arises in various places where space resources are limited and there is a need to find the best packing.

In our case, the limited resource is a **two dimensional space**. We have a set of rectangles (blocks), each with its own **height** and **width**. In addition, each block has it's own **value**. We want to score as much value points as we can, by fitting in a fixed space as much blocks (of the best values), as we can.

## 3 Solution

### 3.1 Solution description

Before diving into details, let us start with a brief, step-by-step description of the algorithm.

1. Find all permutations of the input set, including all rotations.
2. For each permutation of the input set, do the following:
  - (a) Put the first item from the list in the top left corner,
  - (b) Split the remaining space into 2 rectangles, one on the right to the first item, another below,
  - (c) Repeat steps 2 and 3 recursively in the form of a binary tree until there is no more space left.
  - (d) Save the value score.
3. Compare scores and pick the best packing.

#### 3.1.1 Permutation

First we permutate our input set in order to get all possible packings. Then we can proceed to find the best fit for each permutation.

#### 3.1.2 Placing the block

This step is quite simple – we need to put the block somewhere, and one of the corners would let us have a two-rectangle split for the next step. Top left corner is an arbitrary choice – it can be any other corner as long as we are consistent with our choice. We store all whitespace rectangles in a binary tree.

#### 3.1.3 Splitting the remaining space

We divide the remaining whitespace in order to allow recursion. Each smaller part can be treated as a new rectangle for fitting blocks from our queue.

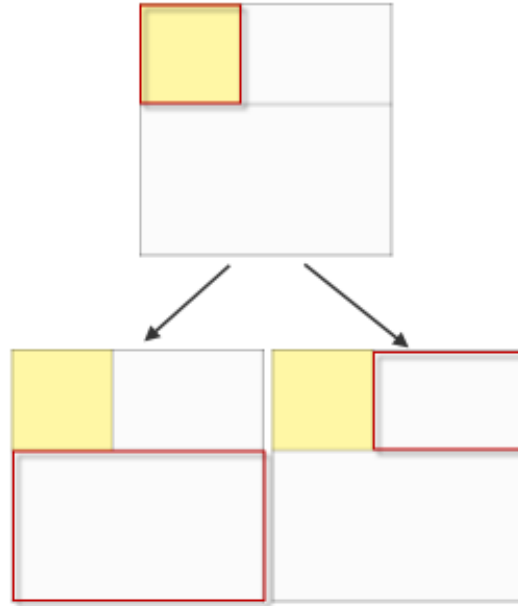


Figure 1: Visualisation of splitting the space and binary tree creation

#### 3.1.4 Recursion

Finally, we do the same thing again for each new item to pack, and for the each smaller whitespace rectangle. There is one caveat, though – we do it in a form of binary tree. It simply means that we traverse our tree to find the smallest free space in which our rectangle would fit. If there is no such space, we discard our block and proceed to the next one.

#### 3.1.5 Saving and comparing scores

After calculating all packings for each permutation, we can pick one of the best value.

## 3.2 Solution pseudocode

```
1 # defines the rectangle we want to pack
2 class Block:
3     width: int
4     height: int
5     value: int # the price of the rectangle
6     fit: Node or None
7
8
9 # defines the container and subcontainer
10 class Node:
11     right: Node
12     down: Node
13     position.x: int
14     position.y: int
15     width: int
16     height: int
17     used: bool
18
19 # Node Methods:
20 fit(self, blocks):
21     for each block in the blocks:
22         # try to find a node that suits current block
23         node = self.findNode(block.w, block.h)
24         if find a node:
25             # pack and split the Node
26             block.fit = node.splitNode(block.w, block.h)
27
28 findNode(self, w, h):
29     # if self is already occupied,
30     # we look for its neighbours recursively
31     if self.used:
32         return self.right.findNode(w, h) or self.down.findNode(w, h)
33     # if current fits then we return current Node
34     else if self.width >= w and self.height >= h:
35         return self
36     else:
37         # Nothings fits for the block
38         return 0
39
40 splitNode(self):
41     # now the node is used
42     self.used = True
43     self.right = Node(self.x+w, self.y, self.width-y, h)
44     self.down = Node(self.x, self.y+h, self.width, self.height-h)
45     return self
```

Listing 1: Python-style pseudocode

## 3.3 Pseudocode description

The `Node.insert` function traverses the tree looking for a place to insert the block. It returns the node the block can be placed into or `None` to say it could not fit.

The code that calls `Node.insert` can then use the rectangle from the returned node to figure out where to place the block in the space, then update the node's `block_id` to use as a handle for future use.

## 3.4 Time complexity calculation

### 3.4.1 Finding and splitting

Each time it traverses to find a suitable node is Depth First Traversal in binary search tree. Imagine we have  $N-1$  blocks already packed, we want to pack the  $N$  th rectangle The worst case of the nodes number will be  $N + 1$  So the time complexity of the worst finding is  $\mathcal{O}(N)$  To sum it up from 1 to  $N$  The time complexity will be  $\mathcal{O}(N^2)$  (we assume splitting is a constant timing consume also) So the worst case of finding and spliting for  $N$  blocks will be  $\mathcal{O}(N^2)$

### 3.4.2 Permutations and getting values

For  $N$  blocks we have  $N!$  permutations, for each permutation the worst finding and splitting time complexity will be  $\mathcal{O}(N^2)$  Our project time complexity will be

$$T_C = \mathcal{O}(N^2 * N!)$$

## 4 Conclusion

Although the algorithm is quite *brute-force* in some aspects, which may be especially cumbersome on larger input, it provides very good results, aside from being simple and easy to understand.

## References

[1] [https://en.wikipedia.org/wiki/Bin\\_packing\\_problem](https://en.wikipedia.org/wiki/Bin_packing_problem)