Algorithms and Computability Project Report

The 2D Bin Packing Problem

Igor Sałuch
Jianhao Luo

December 3, 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 combinational 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 and rotate our input set in order to get all possible packings for all rotations. 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.

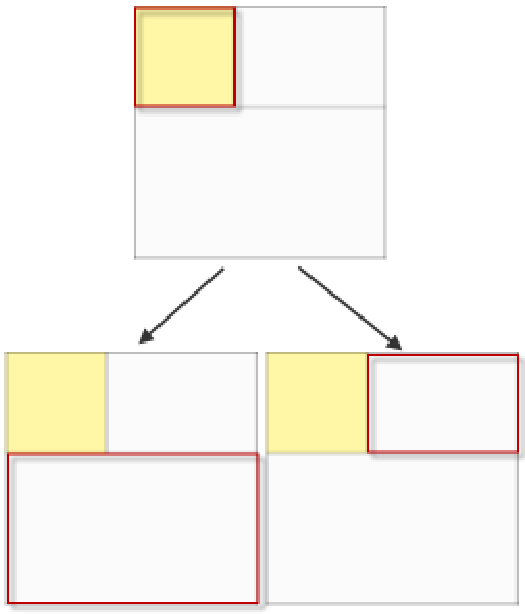

Figure 1: Visualisation of splitting the space and binary tree creation

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

```
# defines the rectangle we want to pack
class Block:
  width: int
  height: int
  value: int  # the price of the rectangle
  fit: Node or None


# defines the container and subcontainer
class Node:
  right: Node
  down: Node
  position.x: int
  position.y: int
  width: int
  height: int
  used: bool

  # Node Methods:
  def fit(self, blocks):
    for each block in the blocks:
      # try to find a node that suits current block
      node = self.findNode(block.w, block.h)
      if find a node:
        # pack and split the Node
        block.fit = node.splitNode(block.w, block.h)

  def findNode(self, w, h):
    # if self is already occupied,
    # we look for its neighbours recursively
    if self.used:
      return self.right.findNode(w, h) or self.down.findNode(w, h)
    # if current fits then we return current Node
    else if self.width >= w and self.height >= h:
      return self
    else:
      # Nothings fits for the block
      return 0

  def splitNode(self):
    # now the node is used
    self.used = True
    self.right = Node(self.x+w, self.y, self.width-y, h)
    self.down = Node(self.x, self.y+h, self.width, self.height-h)
    return self


def get_all_permutations(input):
  output = []
  rotated = get_all_rotations(input)
  for rotated in rotations:
    output.append(
      itertools.permutations(rotated)  # built-in permutation function
    )

def getValueBlocks(block_list, out_list):
    out_list.clear()
    S=0
    for block in block_list:
        if not block.fit is None:
            out_list.append((block.fit.x, block.fit.y, block.w, block.h))
            S+=block.value
    return S

def main():
  input = read_input_file()  # read the input file
  permutations = get_all_permutations(input)
  for permutation in permutations:
    root = Node(
      permutation[0], permutation[1], permutation[2], permutation[3]
    )
    root.fit(block_list_t)
    out_list_t = []
    value = getValueBlocks(block_list_t, out_list_t)
    if value>value_max:
        out_list=out_list_t
        value_max=value

  display_packing(out_list, value_max)
```

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, all the combinations of the rotation of rectangles is the number of all subsets of n rectangles, which is:

$$\sum_{0\le k\le n} \binom{n}{k} = 2^n$$

So we have $2^n * n!$ inputs. Next, for each input the worst finding and splitting time complexity will be $\mathcal{O}(N^2)$ Our project time complexity will be

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

3.5 Proof of correctness

For any correct input we receive an output. We are sure that the answer is correct (that is, the packing is the best possible), because we found all possible packings and picked the best one.

4 Conclusion

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

References

[1] `https://en.wikipedia.org/wiki/Bin_packing_problem`