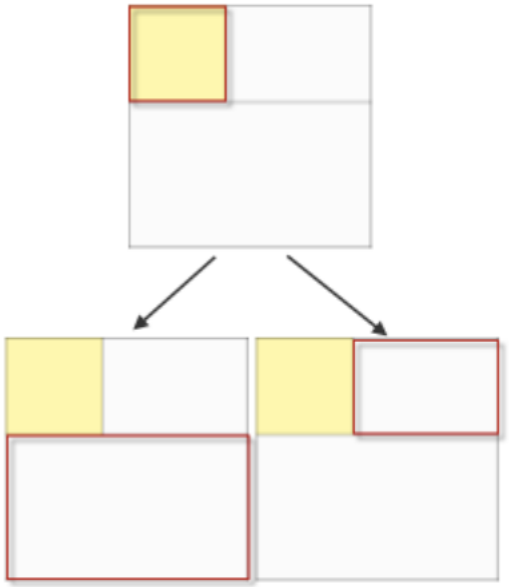
Yash Sunil Shindekar 2023CS10592

Software Assignment 1

Assuming some arbitrary fixed size, let's say 1024x768, how do we pack rectangular blocks in it?

I start by placing the first (largest) block in the top left corner, then you split that rectangle into 2 smaller rectangles that represent the remaining whitespace:



I did this recursively in the form of a binary tree and I end up with a packed image

Assuming the input is already sorted largest to smallest, is surprisingly simple.

```python
class Packer:
    def __init__(self, w, h):
        self.root = {'x': 0, 'y': 0, 'w': w, 'h': h}

    def fit(self, blocks):
        for block in blocks:
            node = self.find_node(self.root, block['w'], block['h'])
            if node:
                block['fit'] = self.split_node(node, block['w'], block['h'])

    def find_node(self, root, w, h):
        if 'used' in root:
            return self.find_node(root.get('right', {}), w, h) or self.find_node(root.get('down', {}), w, h)
        elif w <= root['w'] and h <= root['h']:
            return root
        else:
            return None

    def split_node(self, node, w, h):
        node['used'] = True
        node['down'] = {'x': node['x'], 'y': node['y'] + h, 'w': node['w'], 'h': node['h'] - h}
        node['right'] = {'x': node['x'] + w, 'y': node['y'], 'w': node['w'] - w, 'h': h}
        return node
```

Choosing Minimum Width and Height

We can now use a binary tree for packing small blocks into a fixed size rectangle. But what size should we choose to ensure that all our block fit in as optimal way as possible?

I considered several heuristics. One such example might be to take the average width and average height and multiply by sqrt(n) to try to generate a square:

- width: avg(width) * sqrt(n)
- height: avg(height) * sqrt(n)

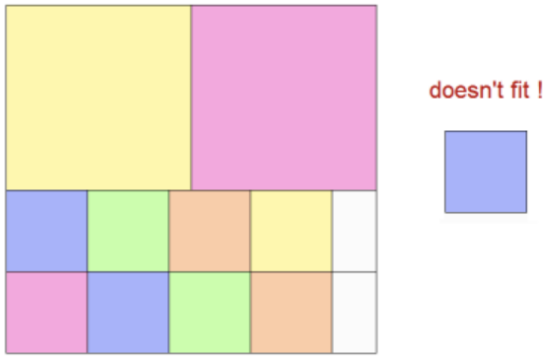
But any heuristic involving avg is easy to break with a single huge or tiny image and can break down pretty quickly.

Other heuristics are also problematic. So, can we take a different approach?

Packing Blocks into a Growing Rectangle

Instead of trying to guess the optimal width and height for a rectangle to pack all of our blocks into, we can start with a small target, just big enough for the first block, and then grow the target as needed whenever there is not enough room for the next block.

Consider a semi packed rectangle where the next block to be packed does not fit



We have 2 choices, we can grow the sprite down, or we can grow to the right

This can be done by adding a couple of new lines to the fit method from the original algorithm:

```
def fit(self, blocks):
    self.root = {'x': 0, 'y': 0, 'w': blocks[0]['w'], 'h': blocks[0]['h']}
    for block in blocks:
        node = self.find_node(self.root, block['w'], block['h'])
        if node:
            block['fit'] = self.split_node(node, block['w'], block['h'])
        else:
            block['fit'] = self.grow_node(block['w'], block['h'])
```

- ensure the root is initialized to the same size as block[0]
- call a new method growNode whenever findNode returns null

Actually, implementing the growNode method involves some more decisions:

```
def grow_node(self, w, h):
    can_grow_down = (w <= self.root['w'])
    can_grow_right = (h <= self.root['h'])

    should_grow_right = can_grow_right and (self.root['h'] >= (self.root['w'] + w))  # Keep square-ish by growing right
    should_grow_down = can_grow_down and (self.root['w'] >= (self.root['h'] + h))  # Keep square-ish by growing down

    if should_grow_right:
        return self.grow_right(w, h)
    elif should_grow_down:
        return self.grow_down(w, h)
    elif can_grow_right:
        return self.grow_right(w, h)
    elif can_grow_down:
        return self.grow_down(w, h)
    else:
        return None  # Ensure sensible root starting size to avoid this happening
```

NOTE: While it's technically possible to support growing both right and down simultaneously, the added complexity isn't justified. We can sidestep this issue by sorting our inputs beforehand, especially since performance isn't our top priority.

A couple of other notes on the code (2):

- If the target height is greater than its width plus the block width, then grow right in order to try to keep square-ish
- If the target width is greater than its height plus the block height, then grow down in order to try to keep square-ish

This prevents us from continuously growing to the right and creating a long horizontal strip. It also prevents us from continuously growing down and creating a narrow vertical strip. It keeps the result roughly square-ish.

I'm pretty happy with the results. Using the automatic growth algorithm described above, combined with a sort order based on max(width, height) gives me fairly good

results for a variety of examples. Where ‘fairly good’ means that the result is roughly square (not long and skinny) and has a minimal amount of whitespace.

Note: In my code I am taking input of number of gates input, scaling factor, then all the gates and then outputting my answer in text file and also showing the grid formed.