Report on the two-dimensional packing problem

The task was completed by*Maxim Zhukov*
phone: +7(985)942-21-72
email: max.7896@mail.ru
telegram: mazhukov

1. Introduction

The task is based on the two-dimensional strip packing problem (see, for instance, Lodi, A. et al. "Two-dimensional packing problems: A survey." Eur. J. Oper. Res. 141 (2002)), which assumes that there is a single standardized unit of a given width, and the objective is to pack a finite number of items of a certain width and height in such a way that the height of the yielding unit will be minimal.

2. Problem description

If there is a unit of some width and number of items of predefined widths and heights, pack all of the given items so the height of the yielding unit will be minimal. Formalize the description of the problem, write a program that solves it, and provide information about the time complexity of the solution.

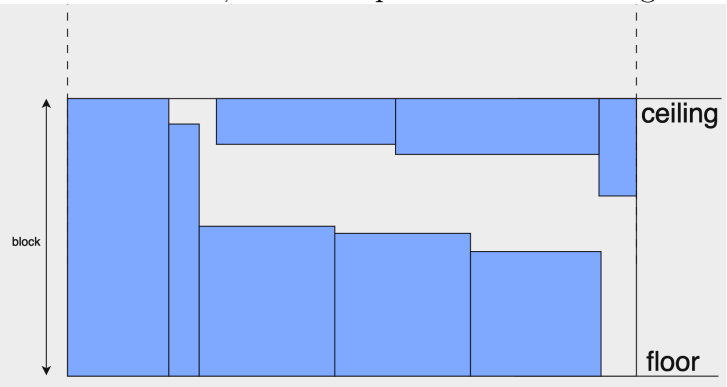
3. Implementation

First of all, i defined some primitives such as point(struct Point in file point.h), size(struct Size in file size.h) and frame(struct Frame in file frame.h):

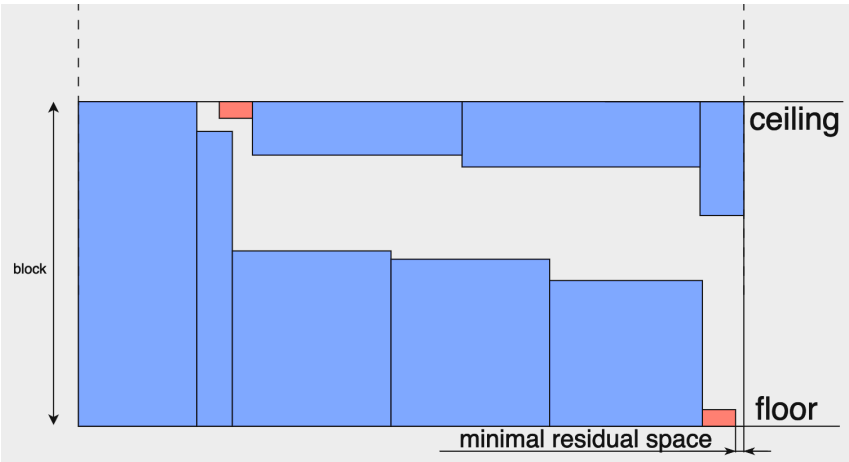
```
1  struct Point {
2      Point() : x(0), y(0) {};
3      Point(int x, int y) : x(x), y(y) {};
4
5      int x;
6      int y;
7  };
8
9  struct Size {
10     Size() : width(0), height(0) {};
11     Size(int width, int height) : width(width), height(height) {};
12
13     int width;
```

```
14      int height;
15
16      friend bool operator>(const Size& lhs, const Size& rhs) {
17          return lhs.width > rhs.width && lhs.height > rhs.height;
18      }
19
20      friend bool operator>(const Size& lhs, const int value) {
21          return lhs.width > value && lhs.height > value;
22      }
23
24      friend bool operator>=(const Size& lhs, const int value) {
25          return lhs.width >= value && lhs.height >= value;
26      }
27  };
28
29  struct Frame {
30      Frame() : size({0, 0}), origin({0, 0}) {};
31      Frame(Point origin, Size size) : origin(origin), size(size) {};
32
33      Point origin;
34      Size size;
35  };
```

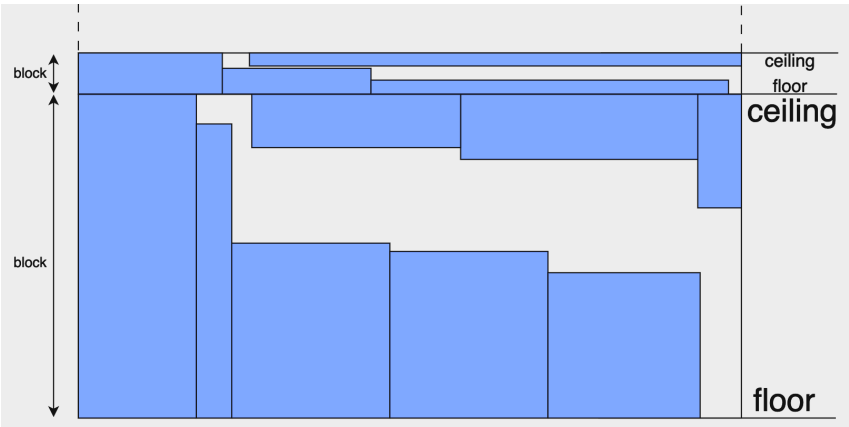
These primitives are the basis for our items.
Now let's describe the packaging algorithm verbally using code examples. The solution to this problem is based on the Floor-Ceiling algorithm with an additional recursive search for child blocks. At the first stage of this algorithm, the items(class Item in file item.h) are sorted by non-increasing height. After that, the items are laid out in turn, starting from the lower left corner of the unit(class Unit in file unit.h), forming blocks(class Block in file block.h) whose height is equal to the leftmost item. The upper line of this block is called "Ceiling and the lower line is called "Floor". If there is no space on the floor for the current item, then it is placed on the ceiling from right to left:



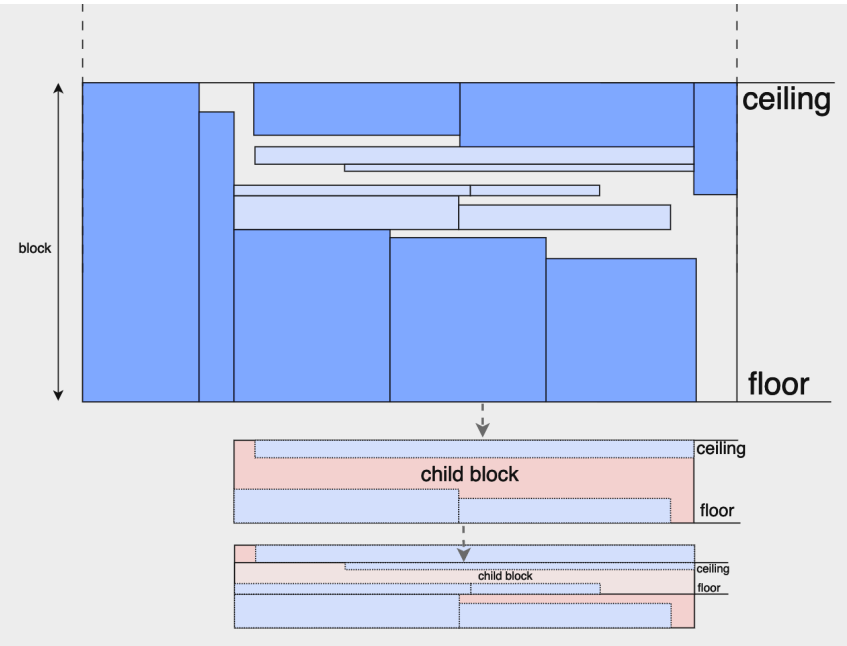
It is important to note that in the process of searching for a place for the current item, the algorithm chooses a **best place** which has a minimal residual space.



If the item cannot be placed in the current blocks, then the program creates a new block in the unit that is located above the current block:



In the process of implementing the algorithm, I found that if the difference between the heights of the aitems is large, then there is a lot of empty space between the ceiling and the floor that is not filled with anything. Therefore, to optimize the Floor Ceiling algorithms, I added the ability to add child blocks inside this block and do a recursive search for the best place inside it. Now if the current item doesn't fit on the floor or on the ceiling in the current block, then the search for the largest child block in terms of area inside the current block. This area is the area of a rectangle in which the upper right point is the lower right point of each item on the ceiling, and the lower left point of the rectangle is the upper left point of each item on the floor. The found child block must fit the current item in width and height.



The `find_block_with_largest_area` method searches for the largest child block inside the current block

```
1  std::shared_ptr<Block> Block::find_block_with_largest_area(const Item& item) {
2      Frame max_frame;
3      for(auto& ceiling_item : ceiling) {
4          for (auto& floor_item : floor) {
5              auto new_block_size = Size({ceiling_item.max_x() - floor_item.min_x(),
       ceiling_item.min_y() - floor_item.max_y()});
6              if(new_block_size >= 0 && new_block_size > item.size) {
7                  int area = new_block_size.width * new_block_size.height;
8                  int max_area = max_frame.size.width * max_frame.size.height;
9                  if(area > max_area) {
10                     max_frame = Frame({floor_item.min_x(), floor_item.max_y()},
       new_block_size);
11                 }
12             }
13         }
14     }
15
16     if(max_frame.size > 0) {
17         return std::make_shared<Block>(Block(max_frame.size, max_frame.origin,
       min_item_height));
18     }
19
20     return nullptr;
21 }
```

As we can see, the time complexity of search for child block is $O(n^2)$

The `find_best_place` method searches for the best place for the item inside the current block as well as recursively inside child blocks

```
ItemBestPlace Block::find_best_place(const Item& item) {
    ItemBestPlace best_place;

    bool can_floor_fit = can_floor_fit_item(item);
    bool can_ceiling_fit = can_ceiling_fit_item(item);

    if(can_floor_fit) {
        // if the floor can fit the item then update best_place for floor
    }

    if(can_ceiling_fit) {
        // if the floor and ceiling places can fit the item then update best_place for place which has a minimal residual space else just update best_place for ceiling
    }

    bool is_this_block_fit_item = can_floor_fit || can_ceiling_fit;
    if(child_block) {
        //Trying to find a place with minimal residual space for the item in a child block
        auto child_best_place = child_block->find_best_place(item);
        bool is_child_block_fit_item = child_best_place.residual_space >= 0;
        bool is_child_place_better = is_this_block_fit_item && child_best_place.residual_space < best_place.residual_space;
        if(is_child_block_fit_item && (!is_this_block_fit_item || is_child_place_better))
            best_place = child_best_place;
    } else if(!is_this_block_fit_item && !ceiling.empty()) {
        // Trying to find a child block if there is no space on the floor and ceiling
        child_block = find_block_with_largest_area(item);
        if(child_block)
            best_place = child_block->find_best_place(item);
    }

    return best_place;
}
```

The `can_floor_fit_item` and `can_floor_fit_item` methods check whether a given item fits in width and height on the floor and ceiling, respectively. If the item cannot be placed in the current blocks and their child blocks, then create a new block in the unit

4. Results

To build, it is enough to run a Makefile with the necessary target:
1. `make -C ./src unit_packaging` – Simple Floor Ceiling algorithm
2. `make -C ./src unit_packaging_recursive` – Floor Ceiling algorithm with child blocks
Then the execution files will appear in the `build` folder

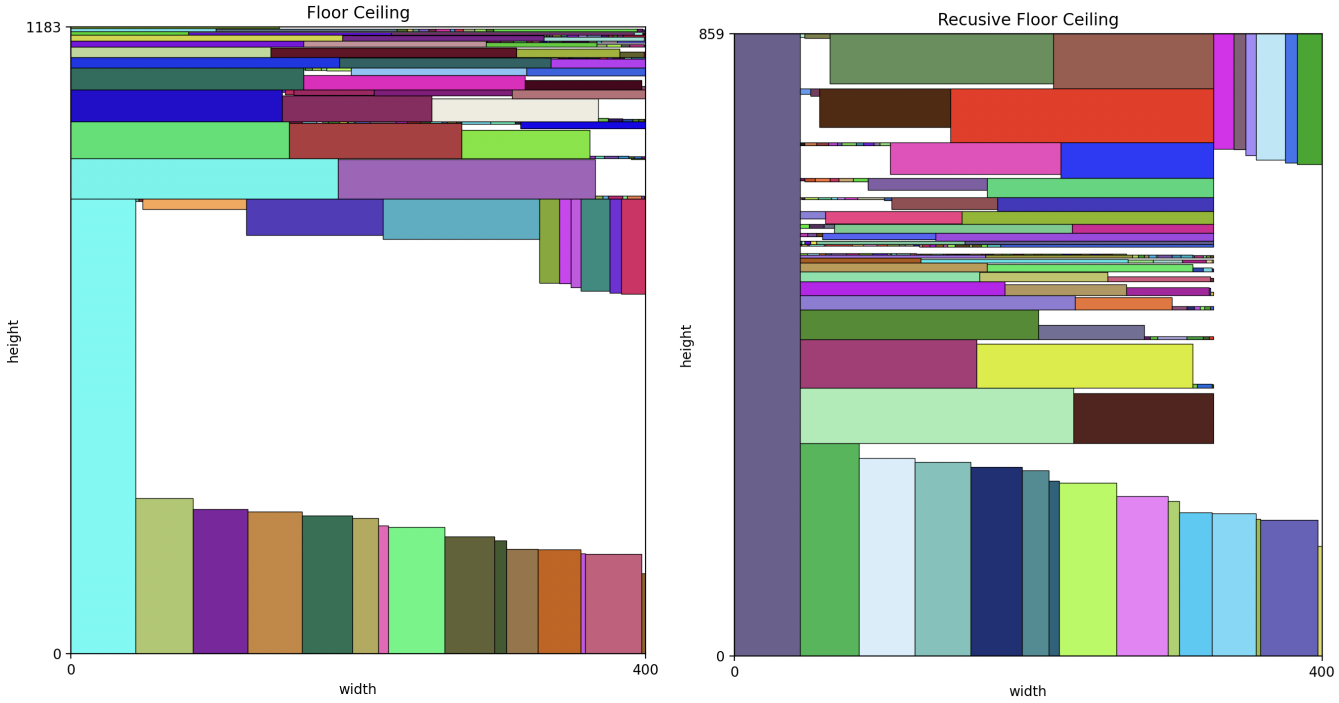
launch:
`./build/unit_packaging_recursive < ./test/input.txt > ./test/output_recursive.txt`

input example:
```
400
5
79 19
132 13
124 19
106 13
94 13
```
The first line shows the width of the unit, on the second line the number of items and on the rest their width and height respectively

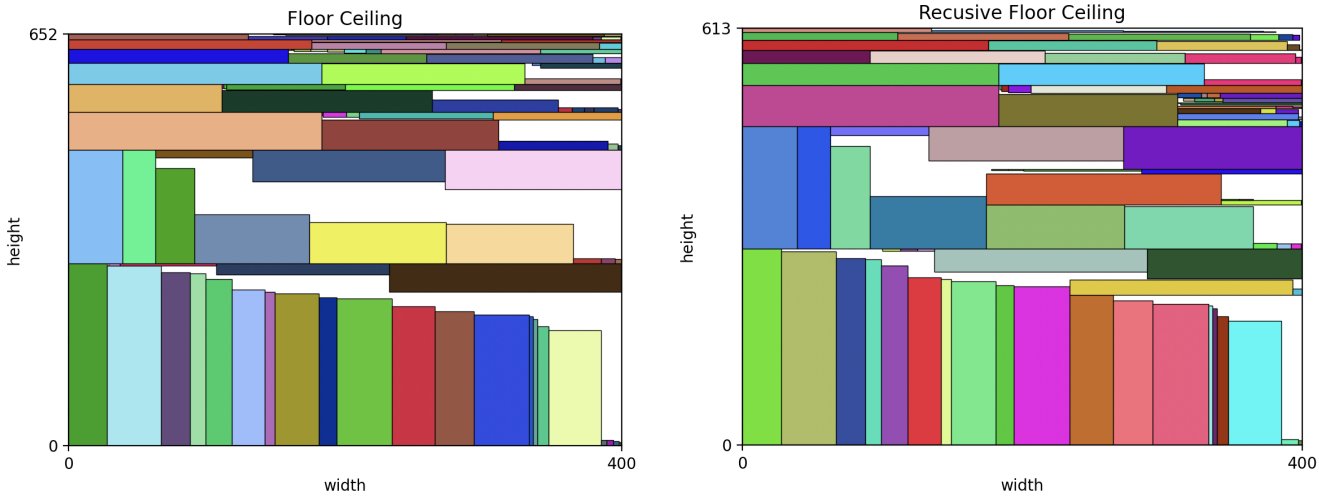
output example:
```
32 400
5
0 0 79 19
79 0 124 19
203 0 132 13
0 19 106 13
106 19 94 13
```
The first line shows the height and width of the unit, on the second line the number of items and on the rest their x, y, width and height respectively

To visualize the result of the algorithm, you need to install python and the matplotlib library. After that, run the command:
`python3 ./test/test_packaging.py < ./test/output_recursive.txt`

Now let's look at the difference between the usual Floor Ceiling(left image) and its optimization(right image) using recursive child blocks on two identical samples. To visualize the solution of this problem, I used Python language and matplotlib library:



As we can see, with a sufficiently large difference in the heights of the items, the algorithm effectively packed the items inside the block so that the total height became equal to the highest item. By reducing the height difference between the items, we get a correspondingly smaller difference in packaging:



The time complexity of this algorithm is calculated based on the iteration of all the items and the application of the `find_best_place` function to them, which in turn iterates through all the items of the block and additionally searches for child blocks using the `find_block_with_largest_area` function whose complexity is $O(n^2)$. Then the time

complexity of the `find_best_place` function for each block is equal to $O(n+n^2) \approx O(n^2)$ and then the total complexity of the algorithm for all items is equal to $O(n*n^2) \approx O(n^3)$. I think this algorithm can be optimized by caching the minimal residual spaces of block and a pointer to it in the binary search tree. In this case, when searching for a new place for an item, it is enough to iterate through the tree for $O(log(n))$, and the search will be successful and you will not have to create child blocks, then the time complexity will decrease on average to $O(n * log(n))$. But the upper limit of the time complexity in this case will remain the same, because if there is too much difference in the heights of the items, child blocks will still be created

5. Conclusion

As a result, we can see that this algorithm packs the items well inside the unit, minimizing the total height of the unit. In the process of applying the Floor Ceiling algorithm to this task, I found that although this algorithm packs quite effectively due to the presence of a floor and ceiling, it loses effectiveness due to the formation of huge voids between the floor and ceiling when the difference between the heights of the items is large. After the introduction of optimization with the creation of child blocks in empty spaces, I found that the total area of voids has noticeably decreased. My optimization loses out on performance relative to the standard Floor-Ceiling algorithm because the search for child blocks takes time, but it wins in terms of packaging quality
This task turned out to be not difficult for me, but very interesting. In the process of implementation, I discovered that this task has different practical applications, for example, by representing the width of the item for time, and the height for power, it is possible to simulate the minimum total power load per unit of time