Advanced Data Structures and Algorithm Analysis
Laboratory Projects

# Texture Packing

Nie Junzhe
Zhang Qi
Zhang Yichi


2020-05-17

# Contents

# 1 Introduction

We always encounter strip packing problems in our real life. This kind of problem is a 2-dimensional geometric minimize problem.

In this project, we ought to concern a 2D rectangle packing problem. We are given a large rectangle with a fixed width and several small rectangles, our work is to pack these small rectangles onto the larger one.

Pay attention to the following 2 points:

- The small rectangles must be packed without covering each other.
- All the small rectangles can be rotate by 90°.

Take the following picture as an example.

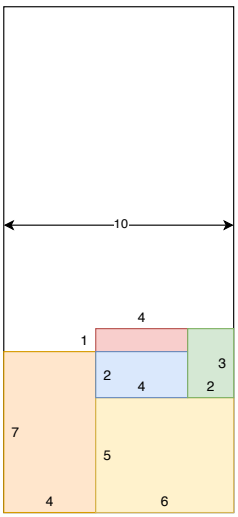


Figure 1.1: Example1

In this example, we are given a large rectangle with a fixed width as 10 and 5 small rectangles as the following table3.4:

Table 1.1: Example 1

| rectangles | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| height | 7 | 6 | 4 | 4 | 3 |
| width | 4 | 5 | 2 | 1 | 2 |

So, in this example the minimized height height is 8.

# 2 Algorithm Specification

## 2.1 First-fit(decreasing-height)

First-fit algorithm is always used in bin-packing problems. It was first described by Coffman. In this circumstance, it works as this way:

- Sorting the small rectangles in the order of their heights in non-decreasingly.
- Scanning from the bottom of the large rectangle to the top and packing the rectangle onto the first position which fits it well.

The pesudocode is listed as following:

**Algorithm 1** First-fit(decreasing height)

**Input:** $W$: the upper bound of the width of the large rectangle; $R$: the small rectangles need to be packed.
**Output:** an approximate result $H$

```
1: Initial H ← 0, blocks ← empty and flag ← 0
2: Sort R non-increasingly in height;
3: for rectangle in R do
4:     for block in blocks do
5:         if rectangle can be packed onto this block then
6:             pack rectangle onto this block;
7:             flag = 1
8:             break
9:         end if
10:    end for
11:    if flag == 0 then
12:        H = H + rectangle.height
13:        push rectangle into blocks
14:    end if
15: end for
16: Return H
```

Here, we take the former case in table3.4 as an example, with the First-fit algorithm, the result will be:

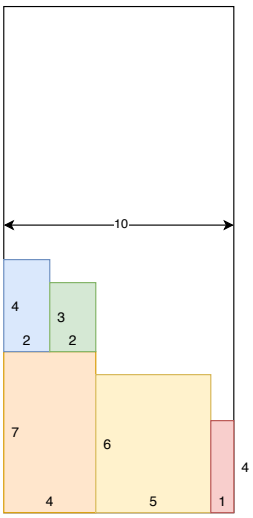


Figure 2.1: Example 2

So, using the method of First-fit algorithm, we get an approximate result, which is 11.

## 2.2 Bottom-up left-justified(not proper enough)

We tried a new algorithm though it's not a good one after considering its time complexity. Anyway, the pesudocode is listed as following:

- Description: When this algorithm iterates the rectangles, for each small rectangle, it searches from the lowest position to pack it and then move as far to the left as possible. Therefore, it packs the rectangle at the lowest and left-most required coordinate $(x, y)$ in the large texture.

**Algorithm 2** Bottom-up left-justified algorithm

**Input:** $W$: the upper bound of the width of the large rectangle; $R$: the small rectangles need to be packed.
**Output:** an approximate result $H$

```
1: Sort R descending order by width
2: for r_i in R do
3:     find the lowest place that can pack r_i
4:     pack r_i at the left-most position of this place
5: end for
6: Return max{r_i.y + r_i.height}
```

It sounds like an easy algorithm, when it comes to its time complexity, it needs 2 nested-loops. And in the inner loop, we need to erase the old blank and generate two new blanks, which takes $O(i)$. So, in total, the time complexity is $O(1^2 + 2^2 + ... + n^2)$, which is $O(N^3)$. It's much worse than First-fit algorithm(we will illustrate its complexity in detail in chapter 4). So we fail to choose it as our algorithm.

# 3 Testing Results

## 3.1 Testing cases

### 3.1.1 Case 1

In this case, we are given the fixed-width as 10 and 6 rectangles as Table 3.1. Using our algorithm, the result is shown in Figure 3.1, which is 32. Comparing with the result given by our program, correct. 32 is also the optimal result.

Table 3.1: Case 1

| rectangles | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| height | 13 | 11 | 11 | 10 | 8 | 6 |
| width | 8 | 4 | 4 | 2 | 7 | 3 |

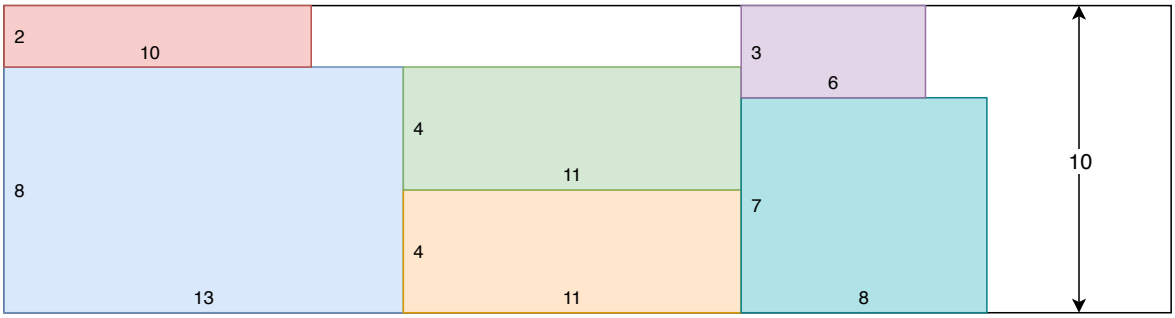


Figure 3.1: Case 1

```
10 6
11 4
13 8
6 3
8 7
11 4
10 2
Total height: 32
```

Figure 3.2: Result 1

### 3.1.2 Case 2

In this case, we are given the fixed-width as 15 and 7 rectangles as Table 3.2. Using our algorithm, the result is shown in Figure 3.4, which is 48. Comparing with the result given by our program, correct. 48 is also the optimal result.

Table 3.2: Case 2

| rectangles | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| height | 17 | 19 | 13 | 17 | 8 | 10 | 12 |
| width | 14 | 9 | 2 | 4 | 5 | 1 | 10 |

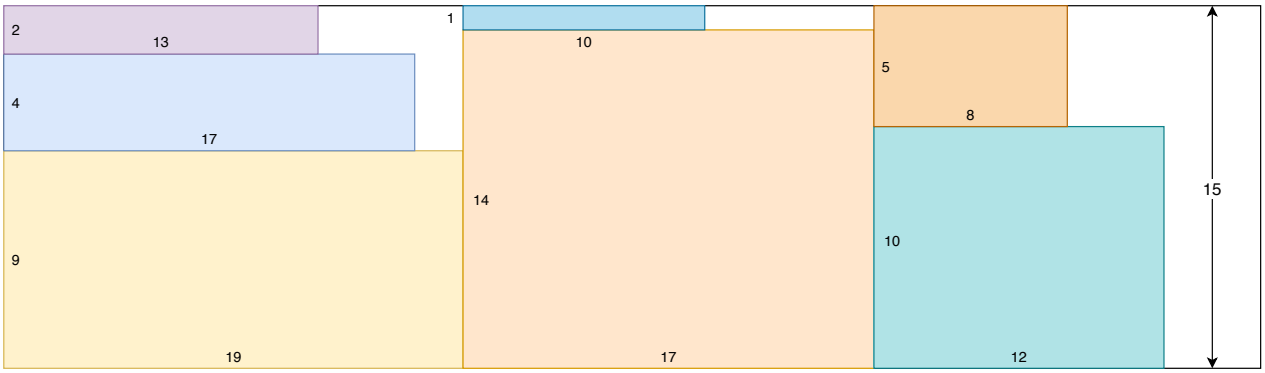


Figure 3.3: Case 2

```
15 7
17 14
19 9
13 2
17 4
8 5
10 1
12 10
Total height: 48
```

Figure 3.4: Result 2

### 3.1.3 Case 3

In this case, we are given the fixed-width as 20 and 8 rectangles as Table 3.3. Using our algorithm, the result is shown in Figure 3.6, which is 89. Comparing with the result given by our program, correct. However, 89 isn't the optimal result, in Figure 3.7, we give the optimal result, which is 77.

Table 3.3: Case 3

| rectangles | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| height | 24 | 9 | 12 | 26 | 27 | 18 | 19 | 22 |
| width | 17 | 6 | 5 | 7 | 16 | 6 | 4 | 4 |

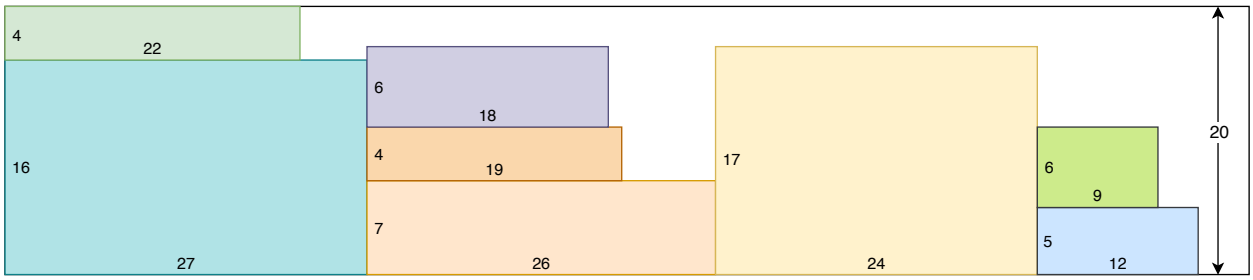


Figure 3.5: Case 3

```
20 8
24 17
9 6
12 5
26 7
27 16
18 6
19 4
22 4
Total height: 89
```

Figure 3.6: Result 3

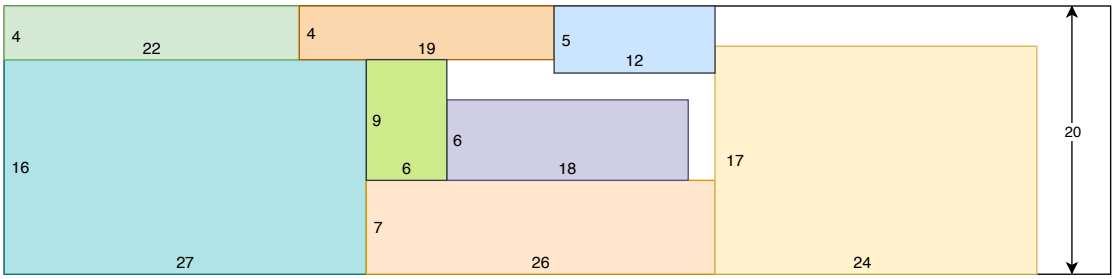


Figure 3.7: Case 3 optimal situation

Attention: All the testing cases are generated by test.cpp. Considering the limited space here, we put more testing cases in the file 'test cases', and the run time will be listed in the following sections.

## 3.2 Run time table

Table 3.4: Run time table

| number | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 | 7000 | 8000 | 9000 | 10000 |
|---|---|---|---|---|---|---|---|---|---|---|
| time(ms) | 2.962 | 6.938 | 14.91 | 25.137 | 33.389 | 45.915 | 61.633 | 77.845 | 100.888 | 120.125 |

## 3.3 Plots

Using matlab to fit the relation between number and time, we plot the regression curve as the following plot.

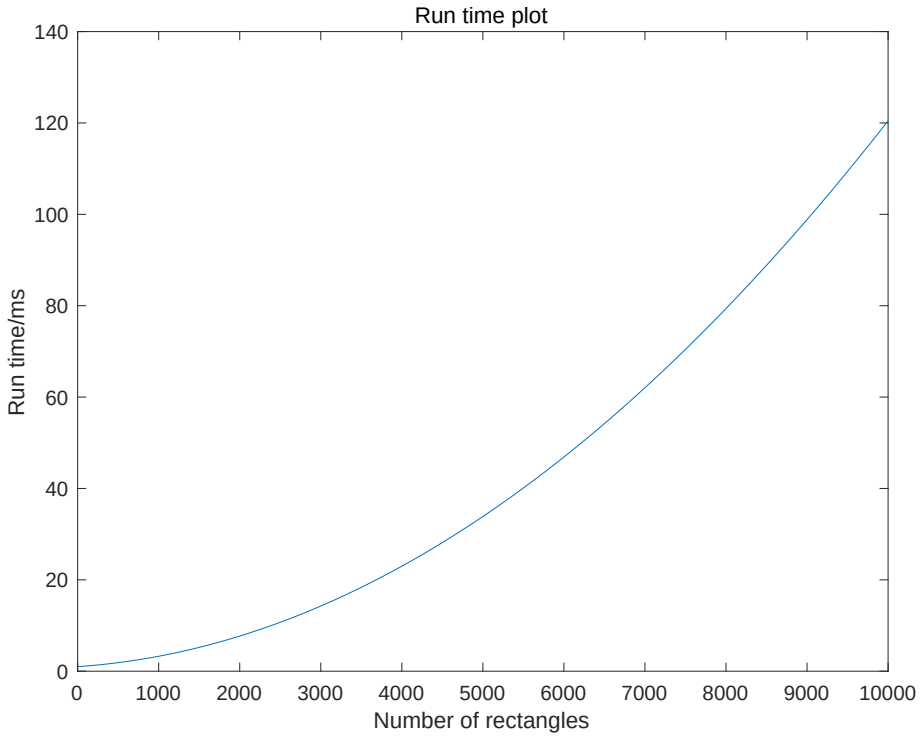


Figure 3.8: Plot

Also, we have calculated the regression function, which is:

$$T = 0.000001075352273N^2 + 0.001194191666667N + 1.005083333333248$$

And the coefficient of association is 0.9991, which is close to 1, so this plot fit our results well.

## 3.4 Analysis and comments

The results we've gotten above have showed that there is a quadratic relation between run times and the input sizes. In this processing, for every rectangle, we need to scan all the blocks to find the first fit block. So, for the $ith$ rectangle, we need $O(i)$ to find the suitable block. In total, $T(N) = O(1 + 2 + ... + N) = O(N^2)$. That why we finally get such a quadratic function plot.

# 4 Complexity Analysis and Approximate Ratio

## 4.1 Time complexity

Considering the worst case, every packing operation causes an insertion into the blocks.

- Sorting the rectangles: $O(N \log N)$
- Packing the rectangles: $O(N^2)$

So, the time complexity is $O(N^2)$.

## 4.2 Space complexity

- Saving the rectangles: $O(N)$
- Saving the blocks: $O(N)$

So, the space complexity is $O(N)$.

## 4.3 Approximate ratio

According to reference [2], the approximate ratio of this algorithm is approximately 2.7.

# 5 Appendix

## 5.1 Source code

### 5.1.1 First-fit

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
class rectangle {
public:
    int height, width;
    rectangle(int h, int w):height(h), width(w){}
};

bool cmp(const rectangle& a, const rectangle& b);

int main()
{
    int total_width, total_height, num, i, j;
    cin >> total_width >> num;
    total_height = 0;
    vector<rectangle> rectangles;
    for(i = 0; i < num; ++i) {
        int h, w;
        cin >> h >> w;
        rectangles.emplace_back(h, w);
    }
    sort(rectangles.begin(), rectangles.end(), cmp);
    vector<pair<int, int>> blocks;
    for(i = 0; i < num; i++) {
        for(j = 0; j < blocks.size(); j++) {
            if(rectangles[i].width + blocks[j].second < total_width) {
                blocks[j].second += rectangles[i].width;
                break;
            }
        }
        if(j == blocks.size()) {
            blocks.emplace_back(rectangles[i].height, rectangles[i].width);
            total_height += rectangles[i].height;
        }
    }
    cout << "Total height: " << total_height << endl;
    return 0;
}

bool cmp(const rectangle& a, const rectangle& b)
{
    if(a.height > b.height) return true;
    else if(a.height < b.height) return false;
    else return a.width > b.width;
}
```

### 5.1.2 First-fit for test

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>
#include <fstream>

using namespace std;

class rectangle { /* The rectangles need to be packed onto the texture. */
public:
    int height, width;
    rectangle():height(0),width(0){}
    rectangle(int h, int w):height(h), width(w){} /* Default constructor. */
};

clock_t start_time, end_time;

bool cmp(const rectangle& a, const rectangle& b); /* The way we compare two rectangles.

int main()
{
    int total_width, total_height, num, i, j;
    string filename;
    cout << "Please input the file you want to test:";
    cin >> filename;
    filename += ".txt";
    ifstream fin(filename.c_str());
    if(!fin) {
        cout << "Error" << endl;
        return 1;
    }
    fin >> total_width >> num;
    total_height = 0; /*Initialize the total height to zero. */
    vector<rectangle> rectangles; /* To store all the rectangles. */
    for(i = 0; i < num; ++i) {
        int h, w;
        fin >> h >> w;
        rectangles.emplace_back(h, w);
    }
    fin.close();
    start_time = clock();
    sort(rectangles.begin(), rectangles.end(), cmp); /* To sort all the rectangles as th
    vector<pair<int, int>> blocks;
    for(i = 0; i < num; i++) {
        for(j = 0; j < blocks.size(); j++) {
            if(rectangles[i].width + blocks[j].second <= total_width) { /* We find a su
                blocks[j].second += rectangles[i].width; /* Ignoring the height, just co
                break;
            }
        }
        if(j == blocks.size()) { /* We don't' find a suitable block to pack this rectang
            blocks.emplace_back(rectangles[i].height, rectangles[i].width); /* Append a
            total_height += rectangles[i].height; /* Add the height of the new block to
        }
```

```
    }
    cout << "Total height: " << total_height << "." << endl;
    end_time = clock();
    cout << "Total time: " << (double)(end_time - start_time) * 1000 / CLOCKS_PER_SEC << "ms
    return 0;
}

bool cmp(const rectangle& a, const rectangle& b) /* First, compare the height of a and b, th
{
    if(a.height > b.height) return true;
    else if(a.height < b.height) return false;
    else return a.width > b.width;
}
```

### 5.1.3 Test code

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
#include <ctime>

using namespace std;

int main()
{
    srand((int)time(nullptr));
    string filename;
    cout << "Please input a filename:"; /* The filename is defined by yourself. */
    cin >> filename;
    filename += ".txt";
    int MaxWidth, num, i;
    cout << "Please input the max width of the texture:";
    cin >> MaxWidth; /* Choose the max width you want to test. */
    cout << "Please input the number of the rectangles:";
    cin >> num;
    ofstream fout(filename);
    fout << MaxWidth << " " << num << endl;
    for(i = 0; i < num; i++) {
        int h, w;
        h = rand() % (3 * MaxWidth / 2) + 1; /* Constrain the rectangles that the width mus
                                                MaxWidth and the height must be smaller than
        w = rand() % MaxWidth + 1;
        if(h < w) swap(h, w);
        fout << h << " " << w << endl;
    }
    cout << "Succeeded.";
    return 0;
}
```

## 5.2 Reference


[1] Thomas H.Cormen, Charles E.Leiserson. Introduction to Algorithms. China Machine Press. 2006.

[2] E.G.Coffman.JR, M.R.Garey, D.S.Johnson, R.E.Tarjan. Performance Bounds For Level-oriented Two-dimensional Packing Algorithms. In Society for Industrial and Applied Mathematics. 1980.

## 5.3 Author list

Nie Junzhe 3180103501
Zhang Qi 3180103162
Zhang Yichi 3180103772

## 5.4 Declarations

**We hereby declare that all the work done in this project titled "Texture Packing" is of our independent effort as a group.**

## 5.5 Signatures