# Texture Packing Report

Cao Yijia
Feng Haibei
Ma Chenyue

**Date: 2020-05-31**

# Chapter 1: Introduction

*Problem description:*

Texture Packing is to pack multiple rectangle shaped textures into one large texture which have a given width and a minimum height.

Given a set of small rectangular pieces of different sizes and a rectangular container of fixed width and infinite length, the problem consists of orthogonally placing all the pieces within the container, without overlapping, such that the overall length of the packing is minimized.

*Purpose of this report:*

The problem belongs to NP-hard problem in theory. Therefore, we designed an approximation algorithm that runs in polynomial time in this project. And we use test cases of different sizes with different distributions of widths and heights to show the usability of our algorithm.

After the testing period, we finished a thorough analysis on all the factors that might affect the approximation ratio of our algorithm.

*Background of the algorithm:*

With the optimization goal of minimizing the unused area of the container, we propose a greedy algorithm for solving the texture packing problem.

# Chapter 2: Algorithm Specification

*Description of the approximation algorithm:*

We'd like to consider the solution as stacking some shelves one by one, each inside using a simple bin packing algorithm. In this way, we will be able to solve problems by continuously using greedy algorithms for several times.

Two structs, which are followed, are defined to declare the rectangle or to describe a horizontal slab of space where rectangles may be placed. And we use different bin packing algorithm in insert operations, which will be accomplished by several different functions in the followed context.

*The time bound and approximation ratio description:*

The main body of this algorithm is composed of sorting and insertion, and the sorting part adopts quick sort function, so the sorting time complexity is O(NlogN). In the insertion of ShelfBestHeightFit, we use nested loop therefore the time complexity is O(N^2). To conclude, the overall time complexity is O(N^2).

By theoretical analysis, the approximation ratio of this algorithm is 1.5. However, the reality is not completely consistent with the theory, you can read more about this in Chapter 4.

*Specifications of main data structures:*

We define two structs as “Rect” and “Shelf” in order to describe a horizontal slab of space where rectangles may be placed.

What’s more, ShelfBinPack is also defined to create a bin of given width.

An enum data type is given as follows to describe the approximation algorithm, the implementation of the algorithm is shown in the pseudo-code below.

```
struct Rect {
    int x;
    int y; //position of the left-bottom corner of the rectangle
    int width;
    int height;//size of the rectangle
};
```

```
//A horizontal slab of space where rectangles may be placed
struct Shelf
{
    //The x-coordinate that specifies where the used shelf space ends
    //[0, currentX]:already used. [currentX, binWidth]:still available.
    int currentX;

    //The y-coordinate of where this shelf starts
    int startY;

    //The height of this shelf
    //The topmost shelf is "open" and its height may grow.
    int height;

    //All the rectangles in this shelf
    vector<Rect> usedRectangles;
};
```

```
ShelfBinPack::ShelfBinPack(int width)
{
    binWidth = width;
    binHeight = 0;
    currentY = 0;
    shelves.clear();
    StartNewShelf(0);
}
```

```
//Different heuristic rules that can be used in the packing process
enum ShelfChoiceHeuristic
{
    ShelfNextFit, //NF: Always put the new rectangle to the last open shelf
    ShelfFirstFit, //FF: Always put the new rectangle to the first where it fits
    ShelfBestAreaFit, //BAF: Choose the shelf with smallest remaining shelf area
    ShelfWorstAreaFit, //WAF: Choose the shelf with the largest remaining shelf area
    ShelfBestHeightFit, //BHF: Choose the smallest shelf (height-wise) where the
rectangle fits
    ShelfBestWidthFit, //BWF: Choose the shelf that has the least remaining horizontal
shelf space available after packing
    ShelfWorstWidthFit, //WWF: Choose the shelf that will have most remainining
horizontal shelf space available after packing
};
```

*Description of several functions:*

In this part, we describe the functions of the algorithms other than insertion.

```
void StartNewShelf(int startingHeight);
```

To creates a new shelf of the given starting height, which will become the topmost 'open' shelf.

```
bool FitsOnShelf(const Shelf &shelf, int width, int height) const;
```

It returns true if the rectangle of size width * height fits on the given shelf, possibly rotated.

```
void RotateToShelf(const Shelf &shelf, int &width, int &height) const
{
    if ((width > height && width > binWidth - shelf.currentX) ||
        (width > height && width < shelf.height) ||
        (width < height && height > shelf.height && height <= binWidth - shelf.currentX))
        swap(width, height);
}
```

If the width > height and the long edge of the new rectangle fits vertically onto the current shelf, flip it. And if the short edge is larger than the current shelf height, store the short edge vertically.

```
void AddToShelf(Shelf &shelf, int width, int height, Rect &newNode)
{
    assert(FitsOnShelf(shelf, width, height));//if cannot fit on the shelf,error!

    //Swap width and height if the rectangle can fit better that way
    RotateToShelf(shelf, width, height);

    //Add the rectangle to the shelf
    newNode.x = shelf.currentX;
    newNode.y = shelf.startY; //position of the new rectangle
    newNode.width = width;
    newNode.height = height; //size of the new rectangle
    shelf.usedRectangles.push_back(newNode);

    //Advance the shelf end position horizontally
    shelf.currentX += width;
    assert(shelf.currentX <= binWidth);

    //Grow the shelf height
    binHeight -= shelf.height;
    shelf.height = max(shelf.height, height);
    binHeight += shelf.height;
    assert(shelf.height <= binHeight);
}
```

This function is used to add the rectangle of size width * height into the given shelf, possibly rotated.

*Pseudo-code of the insertion methods:*

The pseudo-code here is used to describe seven insertion methods in different situations.

```
ShelfNextFit:
    //Always put the new rectangle to the last open shelf
    if FitsOnShelf
    do AddToShelf
    end if
```

```
ShelfFirstFit:
    //Always put the new rectangle to the first where it fits
    for i=0:shelves.size
        if FitsOnShelf(shelves[i])
        do AddToShelf(shelves[i])
        end if
    end for
```

```
ShelfBestAreaFit://Choose the shelf with smallest remaining shelf area.
    bestShelf = 0;
    bestShelfSurfaceArea = -1;
    for i=0:shelves.size
        RotateToShelf(shelves[i]);//Pre-rotate the rect onto the shelf here already
        if FitsOnShelf(shelves[i])//find the shelf with smallest remaining shelf area
            surfaceArea;
            if (surfaceArea < bestShelfSurfaceArea)
            do  bestShelf = &shelves[i];
                bestShelfSurfaceArea = surfaceArea;
            end if
        end if
    end for
    if bestShelf
    do AddToShelf
    end if
```

```
ShelfWorstAreaFit://Choose the shelf with largest remaining shelf area.
    bestShelf = 0;
    bestShelfSurfaceArea = -1;
    for i=0:shelves.size
        RotateToShelf(shelves[i]);
        if (FitsOnShelf(shelves[i])
            surfaceArea;
            if (surfaceArea > bestShelfSurfaceArea)
            do  bestShelf = &shelves[i];
                bestShelfSurfaceArea = surfaceArea;
            end if
        end if
    end for
    if bestShelf
    do AddToShelf
    end if
```

```
ShelfBestHeightFit://Choose the shelf with best-matching height
    bestShelf = 0;
    bestShelfHeightDifference;
    for i=0:shelves.size
        RotateToShelf(shelves[i], width, height);
        if (FitsOnShelf(shelves[i])
            heightDifference;
            if (heightDifference < bestShelfHeightDifference)
            do  bestShelf = &shelves[i];
                bestShelfHeightDifference = heightDifference;
            end if
        end if//find the shelf with best-matching height
    end for
    if bestShelf
    do AddToShelf
    end if
```

```
ShelfBestWidthFit://Choose the shelf with smallest remaining shelf width.
    bestShelf = 0;
    bestShelfWidthDifference;
    for i=0:shelves.size
        RotateToShelf(shelves[i], width, height);
        if (FitsOnShelf(shelves[i])
            widthDifference;
            if (widthDifference < bestShelfWidthDifference)
            do  bestShelf = &shelves[i];
                bestShelfWidthDifference = widthDifference;
            end if
        end if//find the shelf with smallest remaining shelf widt
    end for
    if bestShelf
    do AddToShelf
```

```
ShelfWorstWidthFit://Choose the shelf with biggest remaining shelf width.
    bestShelf = 0;
    bestShelfWidthDifference;
    for i=0:shelves.size
        RotateToShelf(shelves[i], width, height);
        if (FitsOnShelf(shelves[i])
            widthDifference;
            if (widthDifference > bestShelfWidthDifference)
            do  bestShelf = &shelves[i];
                bestShelfWidthDifference = widthDifference;
            end if
        end if//find the shelf with biggest remaining shelf widt
    end for
    if bestShelf
    do AddToShelf
```

# Chapter 3: Testing Results

*Pseudo-code of the case generator program:*

```
input bin_width, rectangle_number;
for i=0:num
    x = rand() % width + 1;//restrict the length of one edge to the bin's width
    y = rand() % (2 * width) + 1);
    r[i].w = min(x,y);
    r[i].h = max(x,y);
end for
qsort(r, num, sizeof(r[0]), cmp);//sort in descending order by the shorter edge
for i = 0: num
do recarea += r[i].w * r[i].h;
end for
```

## *Pseudo-code of the test program:*

```
start clock;
for i = 0:num
    Rect newnode = Bin.Insert(r[i].w, r[i].h, choice);//choose different
    heuristic rule here
stop clock;
runtime = stop - start;
binarea = width * Bin.GetHeight();
ratio = binarea / recarea;
output runtime;
output Bin5.GetHeight();
output ratio;
```

## *Width = 40:*

Figure 1 table when width=40

| | | | | | | | width=40 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| input size | 10 | 50 | 100 | 500 | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 | 7000 | 8000 | 9000 | 10000 |
| rule0/ms | 0 | 0 | 0 | 2 | 5 | 15 | 13 | 15 | 23 | 28 | 30 | 41 | 41 | 47 |
| rule1/ms | 0 | 0 | 0 | 6 | 23 | 99 | 182 | 317 | 558 | 745 | 963 | 1359 | 1888 | 2112 |
| rule2/ms | 0 | 1 | 1 | 12 | 47 | 187 | 407 | 751 | 1182 | 1669 | 2253 | 2906 | 4060 | 4580 |
| rule3/ms | 0 | 0 | 1 | 13 | 49 | 183 | 474 | 707 | 1109 | 1620 | 2194 | 3130 | 4399 | 4889 |
| rule4/ms | 0 | 1 | 1 | 13 | 52 | 183 | 486 | 708 | 1160 | 1806 | 2160 | 3248 | 4037 | 4679 |
| rule5/ms | 0 | 0 | 1 | 12 | 57 | 191 | 405 | 847 | 1413 | 1700 | 2432 | 3004 | 4068 | 4643 |
| rule6/ms | 0 | 0 | 1 | 12 | 63 | 183 | 404 | 703 | 1199 | 1590 | 2493 | 2965 | 4150 | 5120 |
| ratio0 | 1.437 | 1.314 | 1.337 | 1.417 | 1.392 | 1.395 | 1.395 | 1.394 | 1.398 | 1.399 | 1.392 | 1.406 | 1.385 | 1.396 |
| ratio1 | 1.433 | 1.210 | 1.191 | 1.266 | 1.244 | 1.222 | 1.234 | 1.252 | 1.234 | 1.237 | 1.241 | 1.239 | 1.236 | 1.240 |
| ratio2 | 1.283 | 1.242 | 1.209 | 1.241 | 1.255 | 1.230 | 1.236 | 1.253 | 1.239 | 1.231 | 1.236 | 1.248 | 1.232 | 1.236 |
| ratio3 | 1.433 | 1.173 | 1.187 | 1.266 | 1.236 | 1.212 | 1.232 | 1.242 | 1.236 | 1.232 | 1.238 | 1.240 | 1.234 | 1.241 |
| ratio4 | 1.433 | 1.222 | 1.156 | 1.189 | 1.172 | 1.142 | 1.136 | 1.147 | 1.142 | 1.131 | 1.131 | 1.135 | 1.128 | 1.131 |
| ratio5 | 1.433 | 1.210 | 1.174 | 1.269 | 1.244 | 1.222 | 1.233 | 1.253 | 1.234 | 1.237 | 1.241 | 1.239 | 1.236 | 1.240 |
| ratio6 | 1.433 | 1.209 | 1.252 | 1.261 | 1.247 | 1.216 | 1.237 | 1.251 | 1.238 | 1.234 | 1.240 | 1.238 | 1.231 | 1.240 |

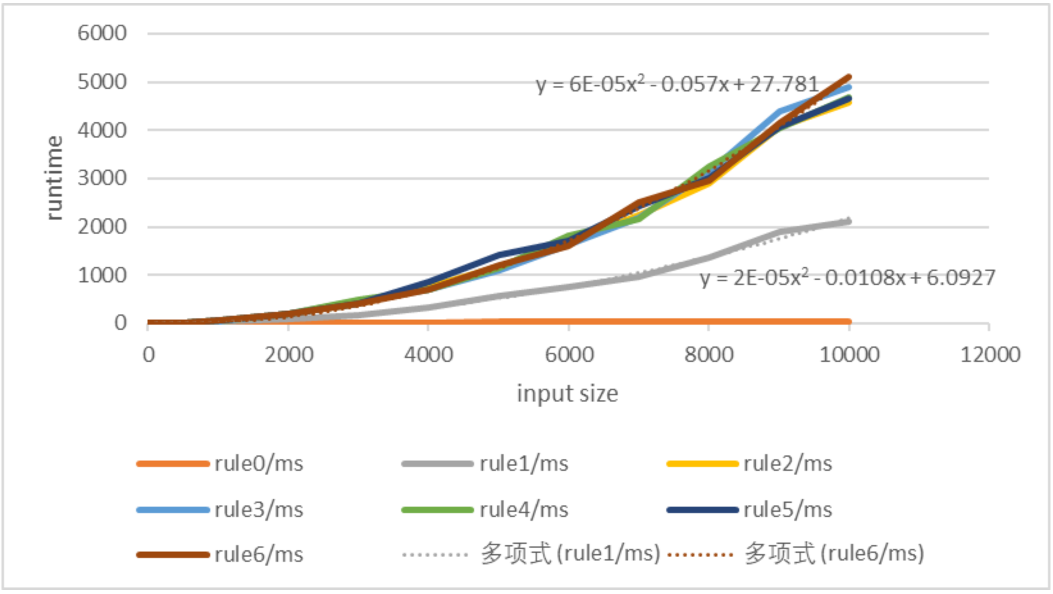

Figure 2 curve of run times vs input sizes when width=40

## *Width = 10:*

Figure 3 table when width=10

| | | | | | | | width=10 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| input size | 10 | 50 | 100 | 500 | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 | 7000 | 8000 | 9000 | 10000 |
| rule0/ms | 0 | 1 | 0 | 2 | 5 | 9 | 14 | 19 | 21 | 29 | 31 | 39 | 43 | 45 |
| rule1/ms | 0 | 0 | 1 | 7 | 26 | 94 | 201 | 360 | 658 | 834 | 1279 | 1431 | 1925 | 2153 |
| rule2/ms | 0 | 1 | 0 | 14 | 51 | 195 | 497 | 874 | 1343 | 1907 | 2881 | 3273 | 4050 | 4902 |
| rule3/ms | 0 | 0 | 1 | 14 | 51 | 195 | 449 | 777 | 1279 | 1831 | 3328 | 3237 | 4124 | 5358 |
| rule4/ms | 0 | 1 | 1 | 16 | 51 | 193 | 427 | 828 | 1290 | 1724 | 3223 | 3449 | 3870 | 5172 |
| rule5/ms | 0 | 0 | 1 | 14 | 51 | 192 | 444 | 791 | 1367 | 1876 | 2680 | 3226 | 3846 | 4720 |
| rule6/ms | 0 | 0 | 1 | 14 | 50 | 229 | 427 | 897 | 1273 | 1741 | 2835 | 3171 | 3953 | 4891 |
| ratio0 | 1.531 | 1.301 | 1.390 | 1.358 | 1.315 | 1.323 | 1.338 | 1.329 | 1.329 | 1.332 | 1.326 | 1.332 | 1.330 | 1.331 |
| ratio1 | 1.374 | 1.255 | 1.204 | 1.212 | 1.206 | 1.202 | 1.211 | 1.207 | 1.213 | 1.209 | 1.203 | 1.207 | 1.206 | 1.208 |
| ratio2 | 1.291 | 1.213 | 1.264 | 1.218 | 1.211 | 1.199 | 1.216 | 1.206 | 1.209 | 1.208 | 1.203 | 1.202 | 1.207 | 1.212 |
| ratio3 | 1.374 | 1.175 | 1.178 | 1.217 | 1.206 | 1.204 | 1.205 | 1.209 | 1.212 | 1.206 | 1.203 | 1.209 | 1.206 | 1.208 |
| ratio4 | 1.291 | 1.202 | 1.202 | 1.144 | 1.130 | 1.124 | 1.120 | 1.115 | 1.120 | 1.119 | 1.109 | 1.114 | 1.118 | 1.118 |
| ratio5 | 1.374 | 1.255 | 1.204 | 1.212 | 1.206 | 1.202 | 1.211 | 1.207 | 1.213 | 1.209 | 1.203 | 1.207 | 1.206 | 1.208 |
| ratio6 | 1.374 | 1.187 | 1.204 | 1.212 | 1.207 | 1.202 | 1.211 | 1.209 | 1.211 | 1.207 | 1.203 | 1.208 | 1.205 | 1.209 |

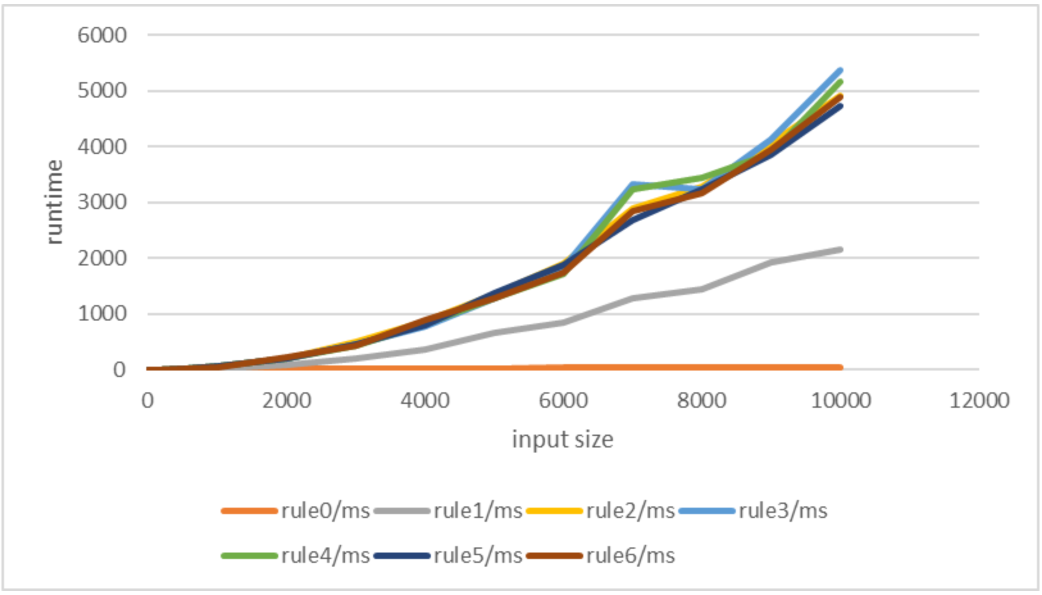

Figure 4 curve of run times vs input sizes when width=10

## *Width = 100:*

Figure 5 table when width=100

| | | | | | | | width=100 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| input size | 10 | 50 | 100 | 500 | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 | 7000 | 8000 | 9000 | 10000 |
| rule0/ms | 1 | 0 | 1 | 3 | 4 | 10 | 21 | 19 | 21 | 34 | 33 | 33 | 35 | 43 |
| rule1/ms | 0 | 0 | 1 | 8 | 25 | 81 | 220 | 311 | 479 | 761 | 1036 | 1302 | 1667 | 2207 |
| rule2/ms | 0 | 0 | 2 | 13 | 48 | 177 | 432 | 700 | 1096 | 1578 | 2166 | 3097 | 3692 | 4571 |
| rule3/ms | 0 | 0 | 0 | 14 | 48 | 183 | 428 | 699 | 1100 | 1600 | 2253 | 2866 | 3791 | 4396 |
| rule4/ms | 0 | 1 | 1 | 14 | 48 | 181 | 460 | 704 | 1104 | 1600 | 2660 | 2999 | 3663 | 4771 |
| rule5/ms | 0 | 0 | 1 | 15 | 47 | 179 | 405 | 704 | 1188 | 1695 | 2358 | 2995 | 3684 | 4427 |
| rule6/ms | 0 | 0 | 1 | 21 | 49 | 177 | 405 | 712 | 1176 | 1753 | 2150 | 3153 | 3665 | 4499 |
| ratio0 | 1.578 | 1.447 | 1.325 | 1.423 | 1.408 | 1.414 | 1.392 | 1.403 | 1.409 | 1.388 | 1.405 | 1.400 | 1.402 | 1.403 |
| ratio1 | 1.326 | 1.252 | 1.229 | 1.249 | 1.245 | 1.263 | 1.237 | 1.241 | 1.243 | 1.328 | 1.246 | 1.240 | 1.241 | 1.247 |
| ratio2 | 1.326 | 1.320 | 1.299 | 1.274 | 1.255 | 1.252 | 1.228 | 1.248 | 1.247 | 1.234 | 1.249 | 1.241 | 1.244 | 1.246 |
| ratio3 | 1.356 | 1.169 | 1.233 | 1.240 | 1.223 | 1.254 | 1.234 | 1.238 | 1.241 | 1.233 | 1.244 | 1.241 | 1.242 | 1.244 |
| ratio4 | 1.378 | 1.269 | 1.202 | 1.221 | 1.190 | 1.166 | 1.149 | 1.151 | 1.153 | 1.140 | 1.144 | 1.146 | 1.145 | 1.144 |
| ratio5 | 1.326 | 1.226 | 1.238 | 1.258 | 1.245 | 1.260 | 1.239 | 1.241 | 1.243 | 1.239 | 1.246 | 1.240 | 1.241 | 1.247 |
| ratio6 | 1.356 | 1.200 | 1.204 | 1.265 | 1.230 | 1.256 | 1.234 | 1.238 | 1.239 | 1.239 | 1.251 | 1.243 | 1.245 | 1.246 |

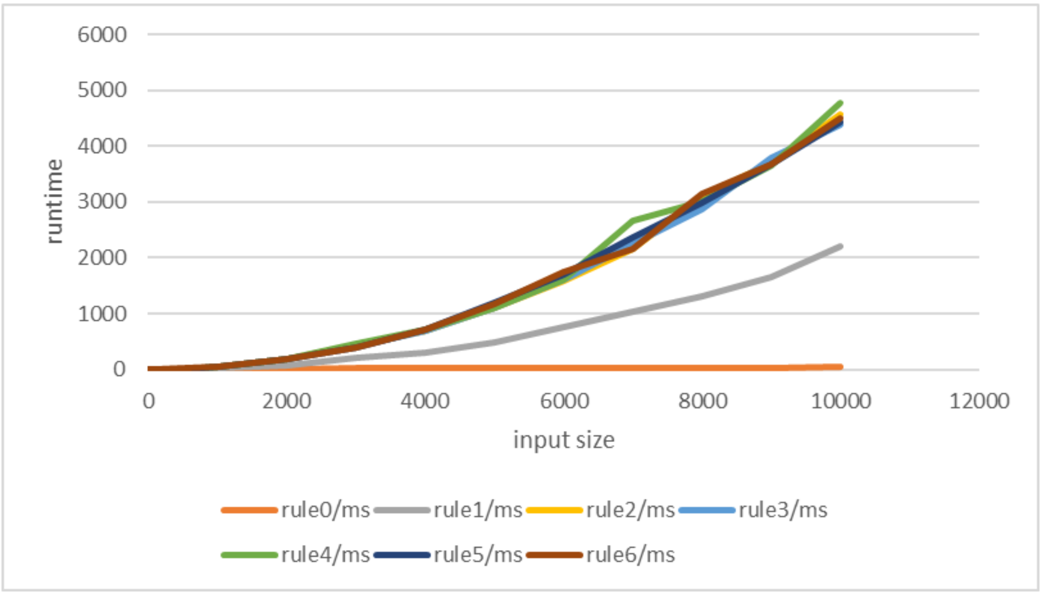

Figure 6 curve of run times vs input sizes when width=100

*Input Size = 1000:*

Figure 7 table when input size=1000

| input size=1000 | | | | | | |
|---|---|---|---|---|---|---|
| width | 50 | 100 | 150 | 200 | 250 | 300 |
| runtime/ms | 46 | 47 | 46 | 48 | 50 | 48 |
| rario | 1.175 | 1.171 | 1.201 | 1.189 | 1.187 | 1.208 |

We test 7 different heuristic rules to compare the run times: ShelfNextFit, ShelfFirstFit, ShelfBestAreaFit, ShelfWorstAreaFit, ShelfBestHeightFit, ShelfBestWidthFit, ShelfWorstWidthFit.

In first three table, we test cases of different input sizes. The last table we test cases of different widths using the same rule ShelfBestHeightFit. The expected result is that the algorithm runs in polynomial time, and by curve fitting, the actual result is consistent with the expected one. So the current status is “pass”.

# Chapter 4:  Analysis and Comments

*Complexities analysis:*

By theoretical analysis, the approximation ratio of this algorithm is 1.5. However, as we can't get the optimal solution when the data is random and the input size is large, we calculate the ratio of the area of the bin and the area of all rectangles. Because the optimal solution is no less than the sum of areas of the rectangles, the approximation ratio is no more than our ratio. By comparison, we conclude that different rules behave differently and in general ShelfBestHeightFit performs better.

We can conclude from the last table that width doesn't make a great influence. When the input size remains unchanged, the runtime and the ratio remain nearly the same.

As we obtain the widths and heights randomly, they already distribute differently. The problem with our algorithm is that we can't make use of the space in the top of each shelf. When some of the rectangles are very long in height and vert short in width, the algorithm may behave badly. But in general, the different distributions of widths and heights doesn't have much influence on the approximate ratio as the optimal solution may also behave badly.

The main body of this algorithm is composed of sorting and insertion, and the sorting part adopts quick sort function, so the sorting time complexity is O(NlogN). In the insertion of ShelfBestHeightFit, we use nested loop therefore the time complexity is O(N^2).To conclude, the overall time complexity is **O(N^2)**.

As the algorithm uses ordinary array and structural array to store the length and width of the rectangles and the built shelf information respectively. The space complexity is **O(N)**.

*Improvement:*

Our algorithm starts by sorting the input items according to descending width, and pack the items in shelves. The first shelf is the at the bottom of the bin. Rectangles are packed from left to right with their bottoms at the selected shelf. When a new shelf is needed, it is created along the horizontal line which coincides with the topmost of the item packed in the lower shelf.

As our algorithm packs rectangles from left to right on each shelf, there will be a waste of space above some of the rectangles. If we can make use of the space, the approximation ratio can be improved.

*Current well-known algorithm:*

Tabu Search algorithm is a meta-heuristic random Search algorithm. It starts from an initial feasible solution, selects a series of specific Search directions (moves) as heuristics, and selects the moves that change the value of a specific objective function the most. In order to avoid falling into the local optimal solution, a flexible "memory" technique is adopted in TS search to record and select the optimization process that has been carried out and guide the search direction in the next step. This is the establishment of Tabu table.

## Appendix: Source Code

*packing.h:*

```
#pragma once
#ifndef PACKING_H
#define PACKING_H
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <cassert>
#include <cstring>
#include <algorithm>
#include <time.h>
#include <iomanip>
using namespace std;

struct Rect {
    int x;
    int y; //position of the left-bottom corner of the rectangle
    int width;
    int height;//size of the rectangle
};

class ShelfBinPack
{
public:
    ShelfBinPack() :binWidth(0), binHeight(0), currentY(0) {}

    //Clears all previously packed rectangles and starts packing from scratch into a bin of the given width
    ShelfBinPack(int width);

    //Different heuristic rules that can be used in the packing process
    enum ShelfChoiceHeuristic
    {
        ShelfNextFit, //NF: Always put the new rectangle to the last open shelf
        ShelfFirstFit, //FF: Always put the new rectangle to the first where it fits
        ShelfBestAreaFit, //BAF: Choose the shelf with smallest remaining shelf area
        ShelfWorstAreaFit, //WAF: Choose the shelf with the largest remaining shelf area
        ShelfBestHeightFit, //BHF: Choose the smallest shelf (height-wise) where the rectangle fits
        ShelfBestWidthFit, //BWF: Choose the shelf that has the least remaining horizontal shelf space available after packing
        ShelfWorstWidthFit, //WWF: Choose the shelf that will have most remaining horizontal shelf space available after packing
    };

    //Inserts a single rectangle into the bin. The packer might rotate the rectangle.
    //method: the heuristic rule to use for choosing a shelf if multiple ones are possible.
    Rect Insert(int width, int height, ShelfChoiceHeuristic method);

    //Get the height of the bin
    int GetHeight();

private:
    int binWidth;
    int binHeight;

    int currentY;//The starting y-coordinate of the latest (topmost) shelf

                 //A horizontal slab of space where rectangles may be placed

    struct Shelf
    {
        //The x-coordinate that specifies where the used shelf space ends
        //[0, currentX]:already used. [currentX, binWidth]:still available.

        int currentX;

        //The y-coordinate of where this shelf starts
```

```cpp
        int startY;

        //The height of this shelf
        //The topmost shelf is "open" and its height may grow.
        int height;

        //All the rectangles in this shelf
        vector<Rect> usedRectangles;
    };

    vector<Shelf> shelves;

    //Returns true if the rectangle of size width * height fits on the given shelf, possibly rotated.
    bool FitsOnShelf(const Shelf &shelf, int width, int height) const;

    //If desirable, flips width and height so that the rectangle fits the given shelf the best.
    void RotateToShelf(const Shelf &shelf, int &width, int &height) const;

    //Adds the rectangle of size width * height into the given shelf, possibly rotated.
    void AddToShelf(Shelf &shelf, int width, int height, Rect &newNode);

    //Creates a new shelf of the given starting height, which will become the topmost 'open' shelf.
    void StartNewShelf(int startingHeight);
};

#endif // !PACKING_H
```

*packing.cpp:*

```cpp
#include "stdafx.h"
#include "packing.h"

using namespace std;

ShelfBinPack::ShelfBinPack(int width)
{
    //Create a bin of given width
    binWidth = width;
    binHeight = 0;
    currentY = 0;
    shelves.clear();
    StartNewShelf(0);
}

void ShelfBinPack::StartNewShelf(int startingHeight)
{
    //Create a new shelf of startingheight.
    Shelf shelf;
    shelf.currentX = 0;
    shelf.height = startingHeight;
    shelf.startY = currentY;

    shelves.push_back(shelf); //store the shelf
    binHeight += shelf.height; //update the binHeight
}

bool ShelfBinPack::FitsOnShelf(const Shelf &shelf, int width, int height) const
{
    if ((shelf.currentX + width <= binWidth) || (shelf.currentX + height <= binWidth))
        return true; //The rectangle can fit on the given shelf, possibly rotated.(consider the width)
    else
        return false; //cannot fit on the shelf
}

void ShelfBinPack::RotateToShelf(const Shelf &shelf, int &width, int &height) const
{
    //If the width > height and the long edge of the new rectangle fits vertically onto the current shelf,flip it.
    //If the short edge is larger than the current shelf height, store the short edge vertically.
    if ((width > height && width > binWidth - shelf.currentX) ||
        (width > height && width < shelf.height) ||
        (width < height && height > shelf.height && height <= binWidth - shelf.currentX))
        swap(width, height);
}

void ShelfBinPack::AddToShelf(Shelf &shelf, int width, int height, Rect &newNode)
{
    assert(FitsOnShelf(shelf, width, height));//if cannot fit on the shelf,error!

    //Swap width and height if the rectangle can fit better that way
    RotateToShelf(shelf, width, height);

    //Add the rectangle to the shelf
    newNode.x = shelf.currentX;
    newNode.y = shelf.startY; //position of the new rectangle
    newNode.width = width;
    newNode.height = height; //size of the new rectangle
    shelf.usedRectangles.push_back(newNode);

    //Advance the shelf end position horizontally
    shelf.currentX += width;
    assert(shelf.currentX <= binWidth);

    //Grow the shelf height
    binHeight -= shelf.height;
    shelf.height = max(shelf.height, height);
    binHeight += shelf.height;
    assert(shelf.height <= binHeight);
}

Rect ShelfBinPack::Insert(int width, int height, ShelfChoiceHeuristic method)
{
    Rect newNode;

    switch (method)
    {
    case ShelfNextFit:
        //Always put the new rectangle to the last open shelf
        if (FitsOnShelf(shelves.back(), width, height))
        {
            AddToShelf(shelves.back(), width, height, newNode);
            return newNode;
        }
        break;

    case ShelfFirstFit:
        //Always put the new rectangle to the first where it fits
        for (size_t i = 0; i < shelves.size(); ++i)
            if (FitsOnShelf(shelves[i], width, height))
            {
                AddToShelf(shelves[i], width, height, newNode);
                return newNode;
            }
        break;

    case ShelfBestAreaFit:
    {
        //Choose the shelf with smallest remaining shelf area.
        Shelf *bestShelf = 0;
        unsigned long bestShelfSurfaceArea = (unsigned long)-1;
        for (size_t i = 0; i < shelves.size(); ++i)
        {
            //Pre-rotate the rect onto the shelf here already
            RotateToShelf(shelves[i], width, height);
            if (FitsOnShelf(shelves[i], width, height))
            {
                unsigned long surfaceArea = (binWidth - shelves[i].currentX) * shelves[i].height;
                if (surfaceArea < bestShelfSurfaceArea)
                {
                    bestShelf = &shelves[i];
                    bestShelfSurfaceArea = surfaceArea;
                }
            }//find the shelf with smallest remaining shelf area
        }
        if (bestShelf)
        {
            AddToShelf(*bestShelf, width, height, newNode);
            return newNode;
        }
    }
    break;

    case ShelfWorstAreaFit:
    {
        //Choose the shelf with largest remaining shelf area.
        Shelf *bestShelf = 0;
        int bestShelfSurfaceArea = -1;
        for (size_t i = 0; i < shelves.size(); ++i)
        {
            //Pre-rotate the rect onto the shelf here already
            RotateToShelf(shelves[i], width, height);
            if (FitsOnShelf(shelves[i], width, height))
            {
                int surfaceArea = (binWidth - shelves[i].currentX) * shelves[i].height;
                if (surfaceArea > bestShelfSurfaceArea)
                {
                    bestShelf = &shelves[i];
                    bestShelfSurfaceArea = surfaceArea;
                }
            }//find the shelf with largest remaining shelf area
        }
        if (bestShelf)
        {
            AddToShelf(*bestShelf, width, height, newNode);
            return newNode;
        }
    }
    break;

    case ShelfBestHeightFit:
    {
        //Choose the shelf with best-matching height
        Shelf *bestShelf = 0;
        int bestShelfHeightDifference = 0x7FFFFFFF;
        for (size_t i = 0; i < shelves.size(); ++i)
        {
            //Pre-rotate the rect onto the shelf here already
            RotateToShelf(shelves[i], width, height);
            if (FitsOnShelf(shelves[i], width, height))
            {
                int heightDifference = max(shelves[i].height - height, 0);

                assert(heightDifference >= 0);

                if (heightDifference < bestShelfHeightDifference)
                {
                    bestShelf = &shelves[i];
                    bestShelfHeightDifference = heightDifference;
                }
            }//find the shelf with best-matching height
        }
        if (bestShelf)
        {
            AddToShelf(*bestShelf, width, height, newNode);
            return newNode;
        }
    }
    break;

    case ShelfBestWidthFit:
    {
        //Choose the shelf with smallest remaining shelf width.
        Shelf *bestShelf = 0;
        int bestShelfWidthDifference = 0x7FFFFFFF;
        for (size_t i = 0; i < shelves.size(); ++i)
        {
            // Pre-rotate the rect onto the shelf here already
            RotateToShelf(shelves[i], width, height);
            if (FitsOnShelf(shelves[i], width, height))
            {
                int widthDifference = binWidth - shelves[i].currentX - width;
                assert(widthDifference >= 0);
                if (widthDifference < bestShelfWidthDifference)
                {
                    bestShelf = &shelves[i];
                    bestShelfWidthDifference = widthDifference;
                }
            }//find the shelf with smallest remaining shelf width
        }

        if (bestShelf)
        {
            AddToShelf(*bestShelf, width, height, newNode);
            return newNode;
        }
    }
    break;

    case ShelfWorstWidthFit:
    {
        //Choose the shelf with smallest remaining shelf width.
        Shelf *bestShelf = 0;
        int bestShelfWidthDifference = -1;
        for (size_t i = 0; i < shelves.size(); ++i)
        {
            // Pre-rotate the rect onto the shelf here already
            RotateToShelf(shelves[i], width, height);
            if (FitsOnShelf(shelves[i], width, height))
```

```cpp
            {
                int widthDifference = binWidth - shelves[i].currentX - width;
                assert(widthDifference >= 0);
                if (widthDifference > bestShelfWidthDifference)
                {
                    bestShelf = &shelves[i];
                    bestShelfWidthDifference = widthDifference;
                }
            }//find the shelf with smallest remaining shelf width
        }
        if (bestShelf)
        {
            AddToShelf(*bestShelf, width, height, newNode);
            return newNode;
        }
    }
    break;

    }

    //The rectangle did not fit on any of the shelves. Open a new shelf.

    //Flip the rectangle so that the long side is horizontal.
    if (width < height && height <= binWidth)
        swap(width, height);

    StartNewShelf(height);
    assert(FitsOnShelf(shelves.back(), width, height));//if cannot fit on,error
    AddToShelf(shelves.back(), width, height, newNode);
    return newNode;

    //The rectangle didn't fit.
    memset(&newNode, 0, sizeof(Rect));
    return newNode;
}

int ShelfBinPack::GetHeight()
{
    return binHeight;
}
```

*test.cpp:*

```cpp
#include "stdafx.h"
#include "packing.h"

struct rec {
    int w;
    int h;
}r[10000];

int cmp(const void* a, const void* b) {
    return (*(struct rec*)b).w - (*(struct rec*)a).w;
}

int main()
{
    clock_t start, stop;//the start and end of the runtime
    time_t tim;//random variety
    double runtime;
    int x, y;
    int width;//the width given for the bin
    int num;//the number of rectangles given
    int minheight;//the final result--the min height of the bin
    int RectWidth, RectHeight;//size of the rectangle
    int op;//the heuristic rule
    int recarea = 0, binarea = 0;//compare the sum area of all rectangles and the area of the bin
    double ratio;

    cout << "Input the width of the bin and the number of rectangles:";
    cin >> width >> num;

    srand((unsigned)time(&tim));//random seed
    for (int i = 0; i < num; i++)
    {
        x = (int)(rand() % width + 1);//restrict the length of one edge to the bin's width
        y = (int)(rand() % (2 * width) + 1);
        if (x < y) { //assume w is shorter than h
            r[i].w = x;
            r[i].h = y;
        }
        else {
            r[i].w = y;
            r[i].h = x;
        }
    }
    qsort(r, num, sizeof(r[0]), cmp);//sort in descending order by the shorter edge
    for (int i = 0; i < num; i++) {
        recarea += r[i].w * r[i].h;
    }

    //the following program test 7 different heuristic rules
    ShelfBinPack Bin0(width);//create a bin of given width
    enum ShelfBinPack::ShelfChoiceHeuristic choice = (enum ShelfBinPack::ShelfChoiceHeuristic)0;
    start = clock();
    for (int i = 0; i < num; i++) {
        Rect newnode = Bin0.Insert(r[i].w, r[i].h, choice);
    }
    stop = clock();
    runtime = (double)(stop - start);//calculate the runtime of insertion
    cout << "The runtime of ShelfNextFit is " << runtime << " ms" << endl;
    cout << "The MinHeight is " << Bin0.GetHeight() << endl;
    binarea = width * Bin0.GetHeight();
    ratio = (double)binarea / recarea;
    cout << "The ratio is " << setprecision(4) << ratio << endl;
    cout << endl;

    ShelfBinPack Bin1(width);
    choice = (enum ShelfBinPack::ShelfChoiceHeuristic)1;
    start = clock();
    for (int i = 0; i < num; i++) {
        Rect newnode = Bin1.Insert(r[i].w, r[i].h, choice);
    }
    stop = clock();
    runtime = (double)(stop - start);
    cout << "The runtime of ShelfFirstFit is " << runtime << " ms" << endl;

    cout << "The MinHeight is " << Bin1.GetHeight() << endl;
    binarea = width * Bin1.GetHeight();
    ratio = (double)binarea / recarea;
    cout << "The ratio is " << ratio << endl;
    cout << endl;

    ShelfBinPack Bin2(width);
    choice = (enum ShelfBinPack::ShelfChoiceHeuristic)2;
    start = clock();
    for (int i = 0; i < num; i++) {
        Rect newnode = Bin2.Insert(r[i].w, r[i].h, choice);
    }
    stop = clock();
    runtime = (double)(stop - start);
    cout << "The runtime of ShelfBestAreaFit is " << runtime << " ms" << endl;
    cout << "The MinHeight is " << Bin2.GetHeight() << endl;
    binarea = width * Bin2.GetHeight();
    ratio = (double)binarea / recarea;
    cout << "The ratio is " << ratio << endl;
    cout << endl;

    ShelfBinPack Bin3(width);
    choice = (enum ShelfBinPack::ShelfChoiceHeuristic)3;
    start = clock();
    for (int i = 0; i < num; i++) {
        Rect newnode = Bin3.Insert(r[i].w, r[i].h, choice);
    }
    stop = clock();
    runtime = (double)(stop - start);
    cout << "The runtime of ShelfWorstAreaFit is " << runtime << " ms" << endl;
    cout << "The MinHeight is " << Bin3.GetHeight() << endl;
    binarea = width * Bin3.GetHeight();
    ratio = (double)binarea / recarea;
    cout << "The ratio is " << ratio << endl;
    cout << endl;

    ShelfBinPack Bin4(width);
    choice = (enum ShelfBinPack::ShelfChoiceHeuristic)4;
    start = clock();
    for (int i = 0; i < num; i++) {
        Rect newnode = Bin4.Insert(r[i].w, r[i].h, choice);
    }
    stop = clock();
    runtime = (double)(stop - start);
    cout << "The runtime of ShelfBestHeightFit is " << runtime << " ms" << endl;
    cout << "The MinHeight is " << Bin4.GetHeight() << endl;
    binarea = width * Bin4.GetHeight();
    ratio = (double)binarea / recarea;
    cout << "The ratio is " << ratio << endl;
```

```cpp
    cout << endl;

    ShelfBinPack Bin5(width);
    choice = (enum ShelfBinPack::ShelfChoiceHeuristic)5;
    start = clock();
    for (int i = 0; i < num; i++) {
        Rect newnode = Bin5.Insert(r[i].w, r[i].h, choice);
    }
    stop = clock();
    runtime = (double)(stop - start);
    cout << "The runtime of ShelfBestWidthFit is " << runtime << " ms" << endl;
    cout << "The MinHeight is " << Bin5.GetHeight() << endl;
    binarea = width * Bin5.GetHeight();
    ratio = (double)binarea / recarea;
    cout << "The ratio is " << ratio << endl;
    cout << endl;

    ShelfBinPack Bin6(width);
    choice = (enum ShelfBinPack::ShelfChoiceHeuristic)6;
    start = clock();
    for (int i = 0; i < num; i++) {
        Rect newnode = Bin6.Insert(r[i].w, r[i].h, choice);
    }
    stop = clock();
    runtime = (double)(stop - start);
    cout << "The runtime of ShelfWorstWidthFit is " << runtime << " ms" << endl;
    cout << "The MinHeight is " << Bin6.GetHeight() << endl;
    binarea = width * Bin6.GetHeight();
    ratio = (double)binarea / recarea;
    cout << "The ratio is " << ratio << endl;
    cout << endl;

    system("pause");
    return 0;
}
```

## References

[1] 陈端兵、黄文奇, “Greedy Algorithm for Rectangle-packing Problem”, *Computer Engineering*, page no.160-162, (February 2007)

## Author List

Programmer: Cao Yijia

Tester: Feng Haibei

Report Writer: Ma Chenyue

## Declaration

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

## Signatures

曹一佳　冯海贝　马辰越