Research Project 6

Texture Packing

GOUNP FIVE

Qian ZeCheng

Ji JunTao

Meng Hongli

Date 2018-6-1

Content

Chapter 1: Introduction 3

1.1 Problem Description 3

Chapter 2: Algorithm Specification 3

2.1 Structures 3

2.2 Description of the algorithms 6

Chapter 3: Testing Results 12

3.1 Result of Algorithm 12

3.2 The Statistic Result 16

Chapter 4: Analysis and Comments 20

4.1 Analysis of the time and space complexities 20

4.2 Comments on the testing results 23

4.3 Further possible improvements 25

Chapter 5: Code 26

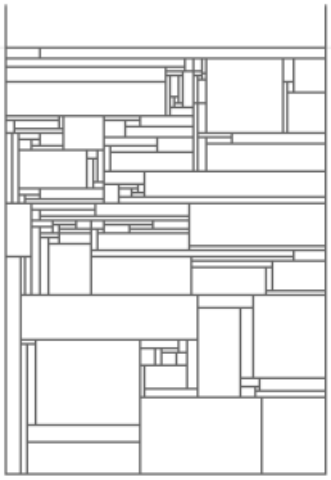
Declaration: 60

Chapter 1: Introduction

1.1 Problem Description

The 2D rectangular packing problem (2DRP) is a fundamental problem in cutting and packing literature. We are given a set of n rectangles with dimensions wi, hi, i = 1, . . ., n. The task is to pack the rectangles without overlap into a large rectangle of dimensions W H (which we call the sheet) such that the total area of the packed rectangles is maximized.

In this project, we used some approximation algorithm to reach the above target. The problem is similar to Bin Packing problem, the difference is that we can pack rectangles in two dimensions.

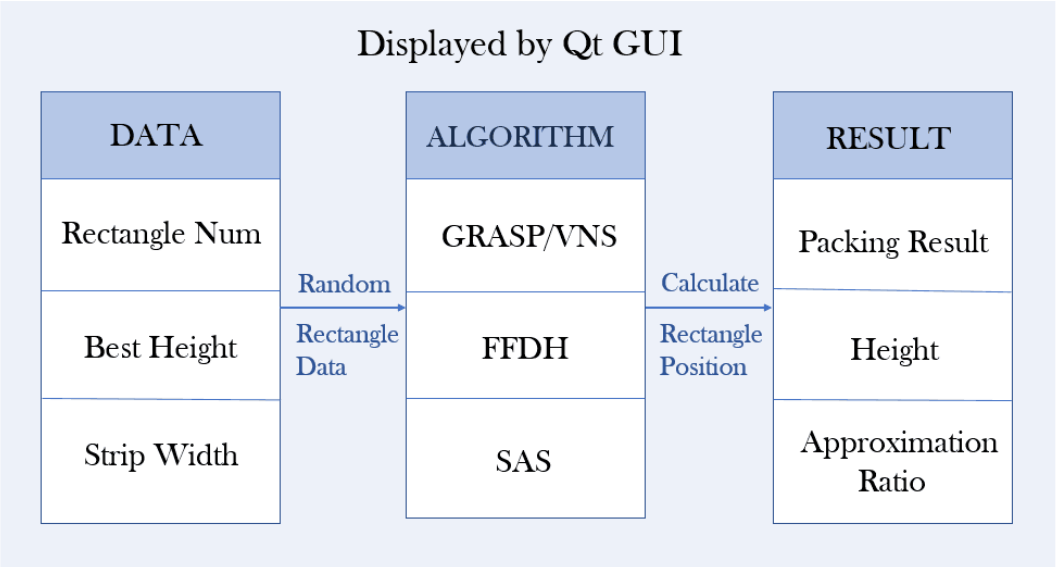


(a diagrammatic sketch of project 6)

Chapter 2: Algorithm Specification

2.1 Structures

2.1.1 The structure of the program



2.1.2 The data structure in the programme

A.Rectangle:

①Definition:

```
class Rectangle
{
public:
    Rectangle() : bl_x(-1), bl_y(-1) {};
    Rectangle(int w, int h) : width(w), height(h), bl_x(-1), bl_y(-1) {};             //overload ctor with two parameters
    Rectangle(int w, int h, int x, int y) : width(w), height(h), bl_x(x), bl_y(y) {}; //overload ctor with four parameters
    ~Rectangle() {};
    int get_width() const { return width; }    //get width of rectangle
    int get_height() const { return height; }  //get height of rectangle
    int get_x() const { return bl_x; }         //get cordinate x of rectangle
    int get_y() const { return bl_y; }         //get cordinate y of rectangle
    bool get_placed() const { return placed; } //get if the rectangle is placed

    bool operator==(const Rectangle &n);       //overload == to judge wheter two instances are the same
    void rotate();
    void set_bl_x(int x) { bl_x = x; }
    void set_bl_y(int y) { bl_y = y; }
```

```
    void place() { placed = true; }
    void reset() { placed = false; }

private:
    int bl_x, bl_y;
    int width, height;
    bool placed = false; //initialized not placed
};
```

②Explaination

The above structure is used to define rectangles in our programme. We can easily get the shape and the coordinate of rectangles by functions named as ‘get_’ , and we can change them by functions named as ‘set_’.

A.Segment:

```
class Segment
{ //horizontal segment class
public:
    Segment() {};
    Segment(int x1, int x2, int y_h) : xl(x1), xr(x2), y(y_h) {}; //overload ct
or with given parameters
    ~Segment() {};
    int get_xl() const { return xl; } //get left end point
    int get_xr() const { return xr; } //get right end point
    int get_y() const { return y; }   //get y
    void set_xl(int x) { xl = x; }    //set left end point
    void set_xr(int x) { xr = x; }    //set right end point
    void set_y(int y_h) { y = y_h; }  //set y
private:
    int xl, xr; //left end point and right end point
    int y;      //unique y because the segment is horizontal
};
```

②Explanation

This structure is used to define the segments in our GRASP algorithm. Its definition helps us get the length and coordinate of each segment.

2.2 Description of the algorithms

A. The First-Fit Decreasing Height(FFDH) algorithm:

①Explanation:

The first algorithm we picked is called FFDH, First-Fit Decreasing Height algorithm. In fact, there are many other conventional algorithms, like NFDH and BFDH. Since they are quite similar to the FFDH algorithm both in approximation performance and time complexity, we only implement FFDH without loss of generality. During the practice, it was observed that when the difference of heights of rectangles that fitted in the same level became extreme, the FFDH algorithm performed poorly.

②Pseudo-code:

```
1. //the main procedure of the FFDH algorithm
2. int FirstFit(vector<Rectangle> &list, int fixedWidth)
3. {
4.     preparation:
5.     use the vector reference to transfer the rectangle information
6.     sort the rectangles on the decreasing height order
7.     main procedure:
8.     while there rectangle exists
9.     begin:
10.          place the highest rectangle at a level's starting point
11.          add its width to this level's width
12.          if this level's width doesn't achieve the fixed width
13.          begin:
14.              then find next high rectangle to fit this level
15.              if it's width is less than the remaining width
16.                  then place it into this level
17.              else
18.                  check the next highest rectangle and decides whether i
    t can be placed
19.              endif
```

```
        else
            begin buliding a new level
            choose the current highest rectangle and place it at the n
    ew level's starting point
            update the height and the width of this level
            check the next points as the above procedure
        endif
    endwhile
    final:
    return the height
}
```

B. Size Alternating Stack algorithm

①Explanation

The Size Alternating Stack algorithm (Abbreviated as SAS below), its core idea is to sort the rectangles into narrow (height greater than width) and wide kinds and place them alternatively. For example, if we put a narrow rectangle at the beginning of a new level, then we put wide rectangles in its right parts, finally we use narrow rectangles again to fill the right parts of the wide, when a level can’t be filled then we go to the next level, the reverse with putting wide rectangles is also similar. And we determine this order by their height, always the higher first. While placing the wide rectangles alone or besides narrow ones, the wider first.

Therefore, after the procedure, we would get a result which has narrow and wide rectangles placed next to each other, it will be orderly but may have more wasted place in its right area.

Particularly, when most of the rectangles are the narrow or wide rectangles, the result would disappoint us.

②Pseudo-Code

```
1.  //the main procedure of the Size Alternating Stack algorithm
2.  int SAS(vector<rectangle>& rec_set){
3.      preparation:
4.      use the vector reference to transfer the rectangle information
5.      clear the narrow and wide rectangle vector
6.      sort the rectangles and push into narrow or wide vector
7.      sort the narrow and wide vector on the decreasing height and width
    order separately
8.      main procedure:
9.      while narrow or wide rectangles exist
10.     begin
11.         if narrow and wide both exist
12.         begin
13.             compare the first narrow and the first wide rectangle's he
    ight
14.             if the narrow is greater
15.             begin
16.                 put the first narrow rectangle at the starting point
17.                 update the height and the placing area(x1,y1,x2,y2)
18.                 erase this rectangle
19.                 let flag == 0 // flag determines whether to put narrow
    or wide next
20.             else
21.                 put the first wide rectangle at the starting point
22.                 update the height and the placing area(x1,y1,x2,y2)
23.                 erase this rectangle
24.                 let flag == 1 // flag determines whether to put narrow
    or wide next
25.             endif
26.         elseif only narrow exists
27.             place this narrow rectangle
28.             update the height and the placing area
29.             erase this rectangle
30.             let flag == 0 // the first must comes from narrow
31.         elseif only wide exists
32.             the same as above
33.             let flag == 1 // the first must comes from wide
34.         endif
```

```
        if flag == 0
            place the wide rectangles in its right part
        elseif flag == 1
            place the narrow rectangles in its right part
        endif
    endwhile
    final:
    return the height
}

//to place the wide rectangles
void packWide(vector<Rectangle> &list, int x1, int y1, int x2, int y2,
 int fixedWidth)
{
    find if there exists a wide rectangle can fit the narrow's right p
art
    if there exists
    begin:
        find the widest rectangle and place it
        update the height and the area coordinate values
        if the narrow rectangle exists
            call the placeNarrow function and place the narrow rectang
les in the wide's right
        else
            continuely call the placeWide function and place the wide
rectangles then erase it
    else if there not exist the appropriate wide rectangle
        call the placeWide function and place the wide rectangles next

    endif
}

//to place the narrow rectangles
void packNarrow(vector<Rectangle> &list, int x1, int y1, int x2, int y
2, int fixedWidth)
{
    find if there exists a wide rectangle can fit the wide's right par
t
    if there exists
    begin:
        find the highest rectangle and place it
        erase it from the narrow vector
        update the height and the area coordinate values
```

```
71.         recursively find the narrow rectangles to fill the right part
    then erase it
72.         update the area coordinate values
73.     else if there not exist the appropriate narrow rectangle
74.         call the placeWide function and place the wide rectangles next


75.         begin the next level of the narrow's right part or bulid a new
     level
76.     endif
77. }
```

C. An algorithm based on GRASP

①Explanation

The algorithm's full name is "Greedy Randomized Adaptive Search Procedure" (Abbreviated as GRASP below). It is a heuristic algorithm and is also offline, which aims to packing all rectangles into a nested shape. Briefly speaking, the algorithm tend to choose the "best fit" rectangle among unpacked rectangles. It maintains a segment contour which represents the current state of solution, selecting the lowest segment, and the "best fit" rectangle is chosen based on an evaluation function, allowing self rotation by ninety degree.

The evaluation value is calculated on every rectangle among unplaced one, choosing the rectangle with minimum evaluation value. If there isn't any rectangle can be placed on the lowest segment, segments updating will be implemented to dealing with contradictions, lifting the lowest segment to the level of reciprocal

second low segment(lengthening the lowest segment), and iterates until there isn't any unplaced rectangle.

Then we implement an improving procedure to the initial algorithm, bring about an even smaller approximation ratio.

②Pseudo-Code

```
1.  //the main procedure of the GRASP algorithm
2.  int IterativeFindSolution(vector<Rectangle> &rec_set, int strip_width,
     int k)
3.  {
4.      preparation:
5.      use the vector reference to transfer the rectangle information
6.      initialize the unplaced vector
7.      define and initialize the segment vector as Contour
8.      define the best contour vector for the best result in the iteratio
    ns
9.
10.     main procedure:
11.     for i = 0 : k-1
12.     begin
13.         find the lowest segment
14.         //call the function FillRectangle to place rectangles
15.         while the unplaced vector is not null
16.         begin:
17.             define the RCL vector to place the rectangles that can per
    fectly fit the segment
18.             call the FillRectangle function
19.             iteratively traverse the unplaced vector
20.             if the rectangle's width or height satisfies perfectly the
     segment's length
21.             begin
22.                 push it in the RCL vector
23.             endif
24.         endwhile
25.         if the RCL is not null
26.         begin:
27.             randomly select a rectangle from the RCL vector
```

```
28.             push it into the solution vector and erase it from the unp
    laced vector
29.             push its x y values into rec_set to record its coordinates

30.             erase it from the unplaced vector
31.             update the height
32.             resplit this segment based on the filled rectangle
33.             update the Contour vector
34.         else
35.         begin:
36.             find the less satisfying rectangle in the unplaced vector

37.             push the best satisfying rectangle into the RCL vector
38.             if the RCL is not null
39.             begin
40.                update the solution, rec_set, unplacedImplement optimiz
    ation    strategy vector and height
41.                resplit this segment based on the filled rectangle and
    update the Contour vector
42.             else
43.                 combine the lowest segment and second lowest segment

44.                 call the FillRectangle function to fill on the segment
     again
45.             endif
46.         endif
47.     endfor
48.     find the best result in the k-times iterations
49.     update the best_contour and the best rectangles
50.     carry out the optimization strategy to avoid the projecting situat
    ion
51.     final:
52.     return the height
53. }
```

Chapter 3: Testing Results

3.1 Result of Algorithm

(In the following pictures, our results will be shown in the left frame, while the best answer shown in the right frame.)

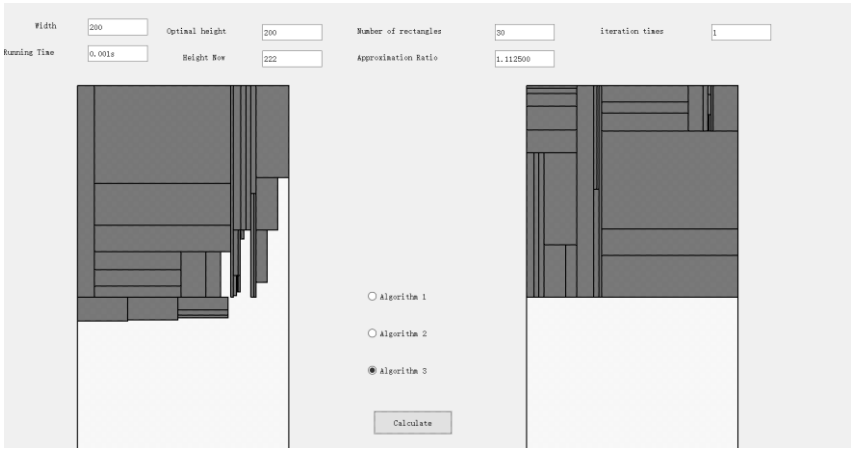
①FFDH

A. Small-scale data

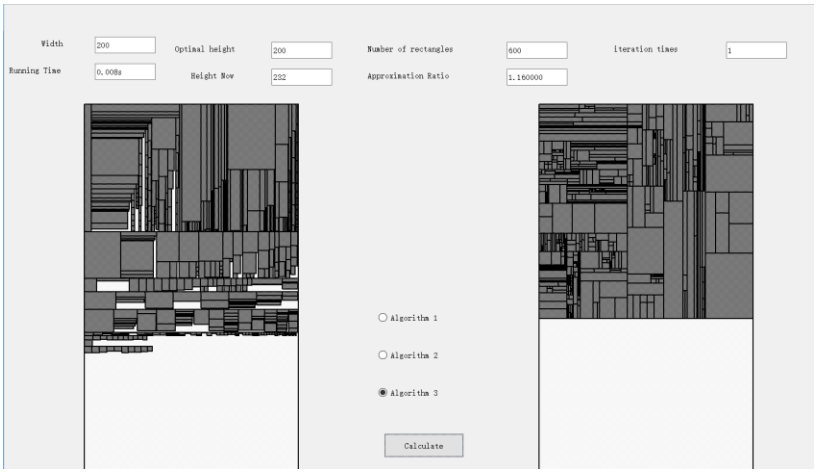
B. Big-scale data

②SAS

A. Small-scale data



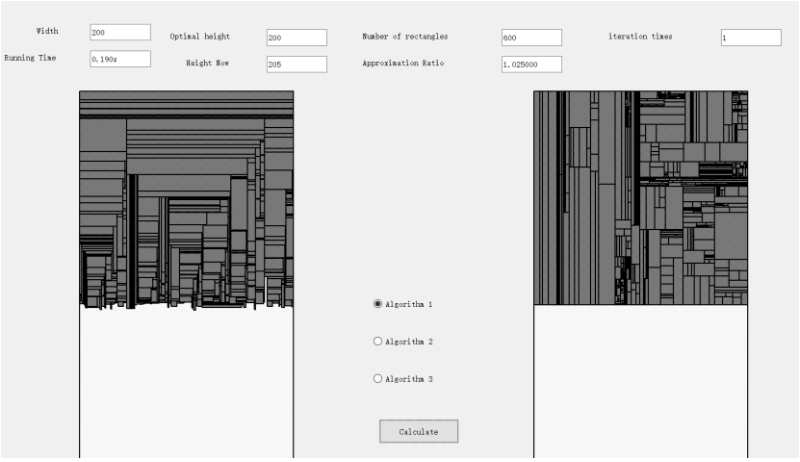
B. Big-scale data



③GRASP

A. Small-scale data

B. Big-scale data



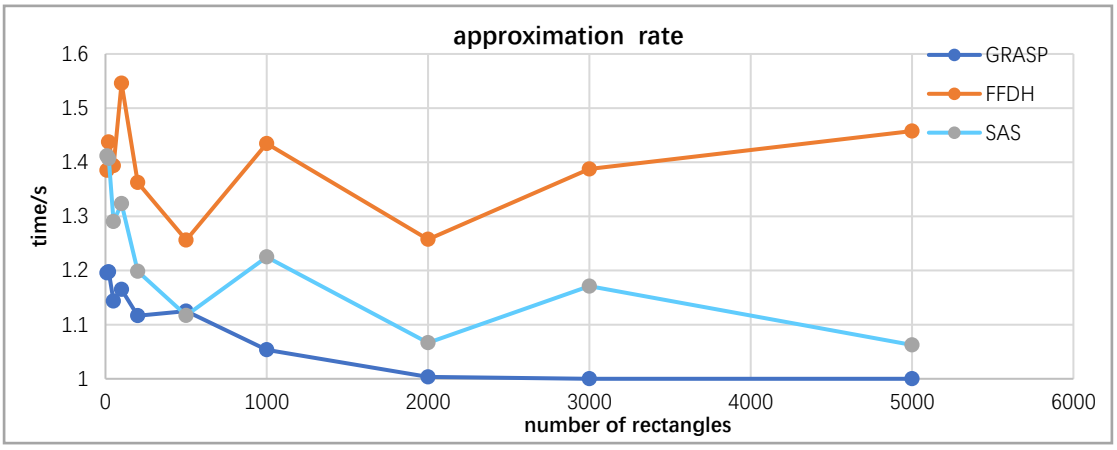
3.2 The statistical result

① All of testing result(each case was tested 5 times)

		500	1000	2000	3000	5000
GRASP	iteration times/s	1000	1000	100	100	10
	total time/s	0.2138	0.6782	0.2836	0.8396	0.2722
	Time/s	0.0002138	0.0006782	0.002836	0.008396	0.02722
	approximation rate	**1.1955**	**1.1975**	**1.1435**	**1.165**	**1.1165**
FFDH	iteration times	1000	1000	1000	1000	1000
	total time/s	0.0058	0.0126	0.0356	0.0824	0.2004
	Time/s	0.0000058	0.0000126	0.0000356	0.0000824	0.0002004
	approximation rate	**1.385**	**1.4375**	**1.3935**	**1.546**	**1.3625**
SAS	iteration times	1000	1000	1000	1000	1000
	total time/s	0.0206	0.0442	0.1514	0.3806	1.4264
	Time/s	0.0000206	0.0000442	0.0001514	0.0003806	0.0014264
	approximation rate	**1.4115**	**1.407**	**1.2905**	**1.3235**	**1.1985**

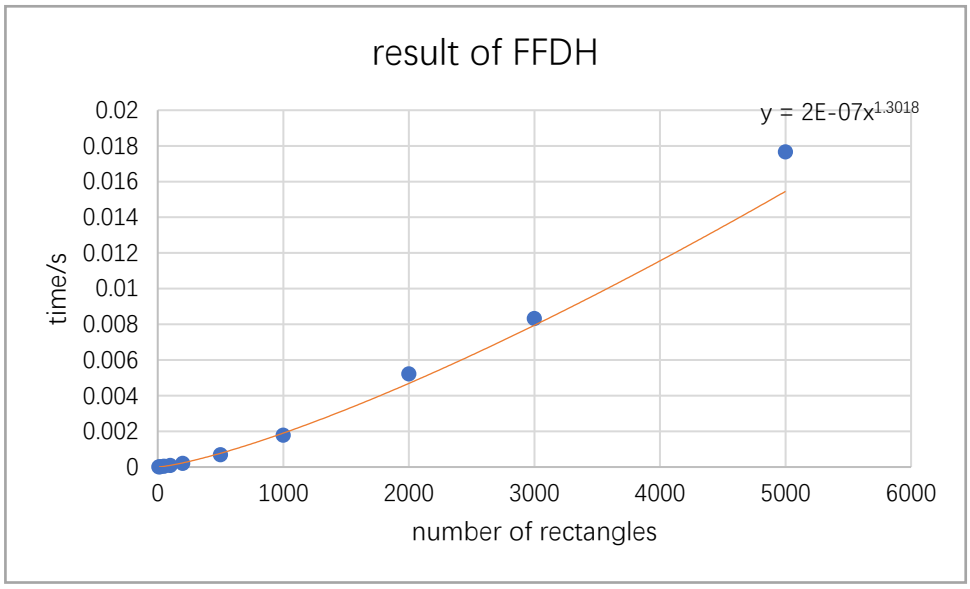
		500	1000	2000	3000	5000
GRASP	iteration times/s	10	1	1	1	1
	total time/s	1.3662	0.4878	1.7384	4.196	12.395
	Time/s	0.13662	0.4878	1.7384	4.196	12.395
	approximation rate	**1.125**	**1.0535**	**1.0035**	**1**	**1**
FFDH	iteration times	1000	100	100	100	100
	total time/s	0.6822	0.1776	0.5214	0.832	1.7656
	Time/s	0.0006822	0.001776	0.005214	0.00832	0.017656
	approximation rate	**1.2562**	**1.4345**	**1.2575**	**1.3875**	**1.4575**
SAS	iteration times	100	100	10	10	10
	total time/s	0.6174	1.7682	0.6622	1.5118	3.4394
	Time/s	0.006174	0.017682	0.06622	0.15118	0.34394
	approximation rate	**1.117**	**1.225**	**1.0665**	**1.171**	**1.0625**

②The statistical diagrams of approximation rate

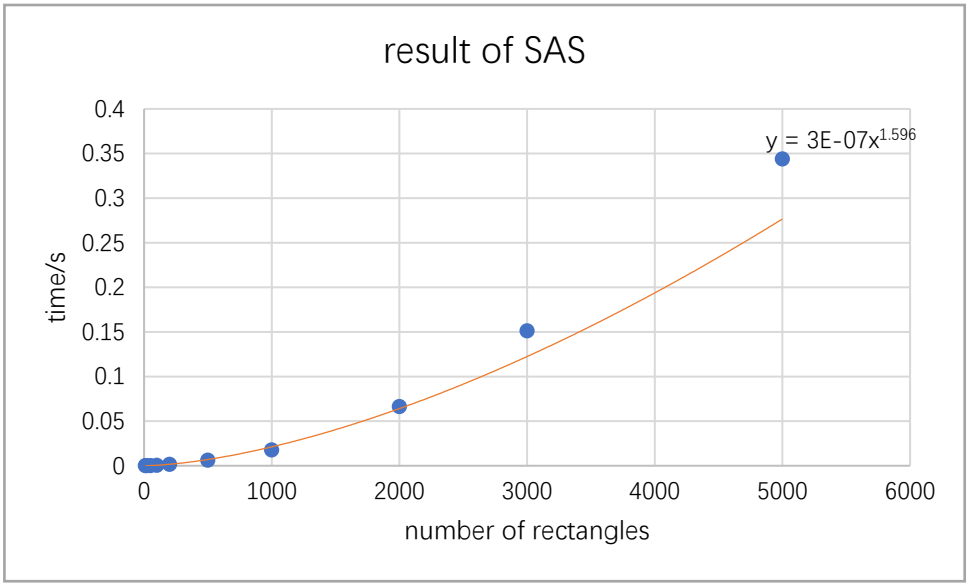


③The statistical diagrams of running time

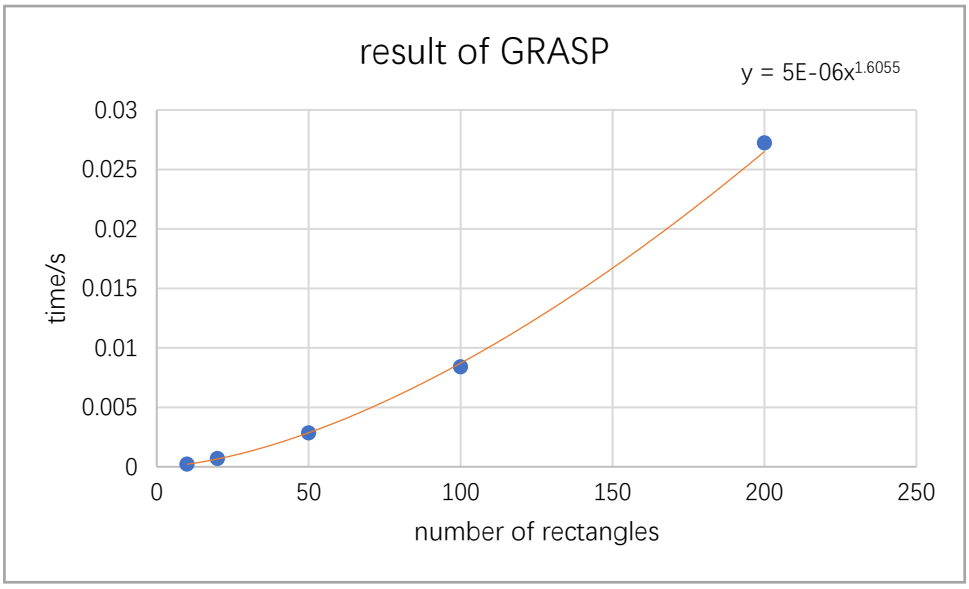
A. Power function fitting



(y=2E-7*X^1.3018)

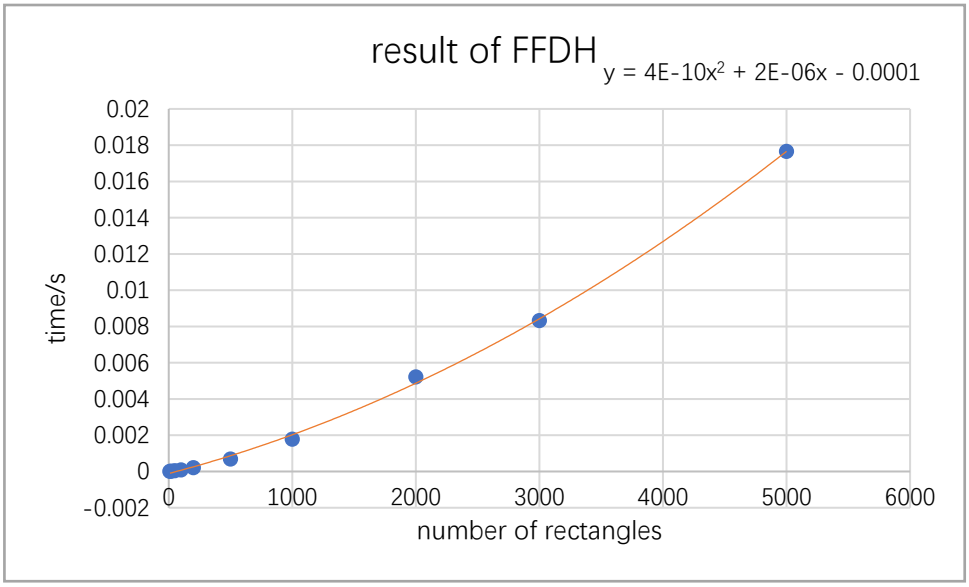


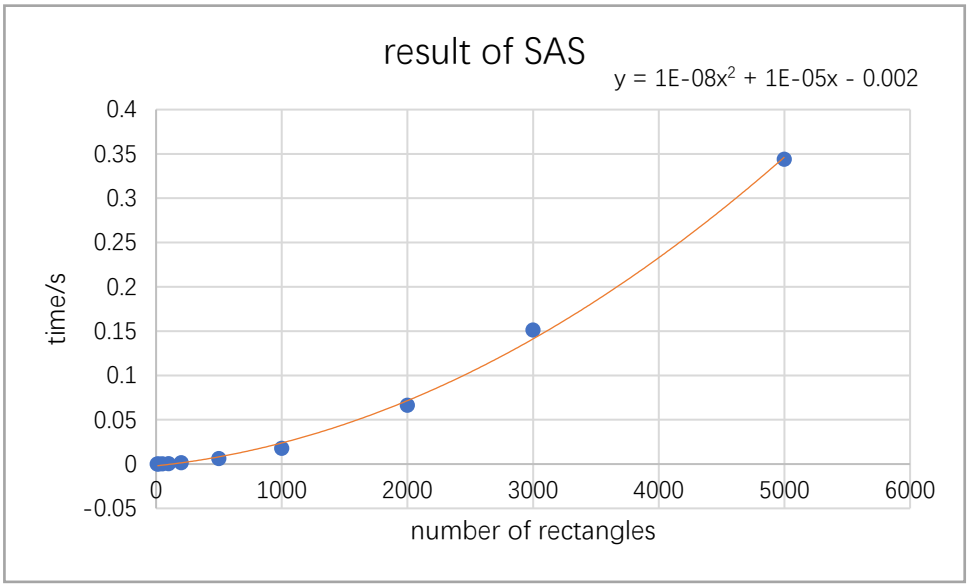
(y=3E-7*X^1.596)

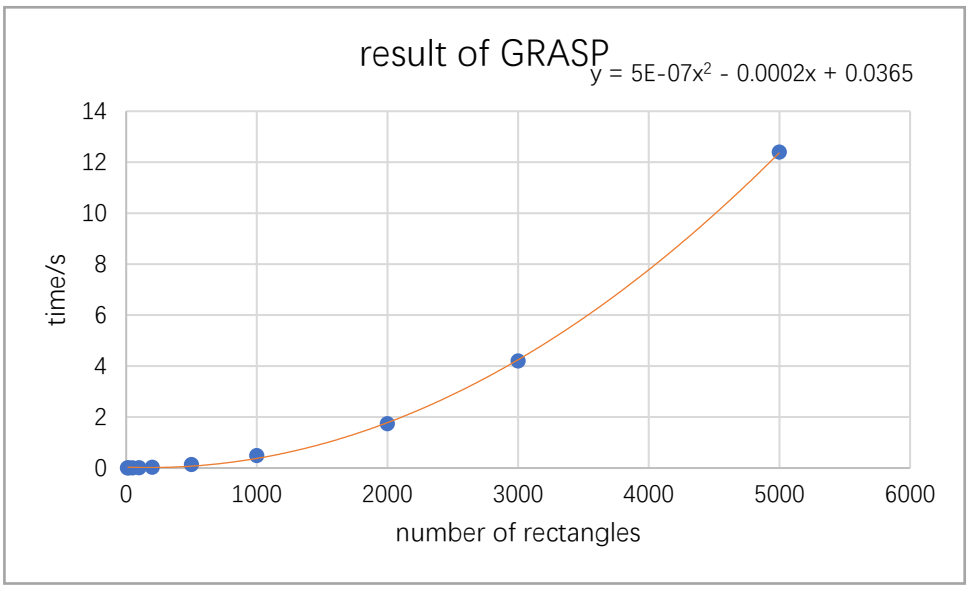


(y=5E-6*X^1.6055)

B. Polynomial fitting


result of FFDH
y = 4E-10x² + 2E-06x - 0.0001
time/s
number of rectangles
0.02
0.018
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
-0.002
0
1000
2000
3000
4000
5000
6000


result of SAS
y = 1E-08x² + 1E-05x - 0.002
time/s
number of rectangles
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
-0.05
0
1000
2000
3000
4000
5000
6000


result of GRASP
y = 5E-07x² - 0.0002x + 0.0365
time/s
number of rectangles
14
12
10
8
6
4
2
0
0
1000
2000
3000
4000
5000
6000

Chapter 4: Analysis and Comments

4.1 Analysis of the time and space complexities

①FFDH

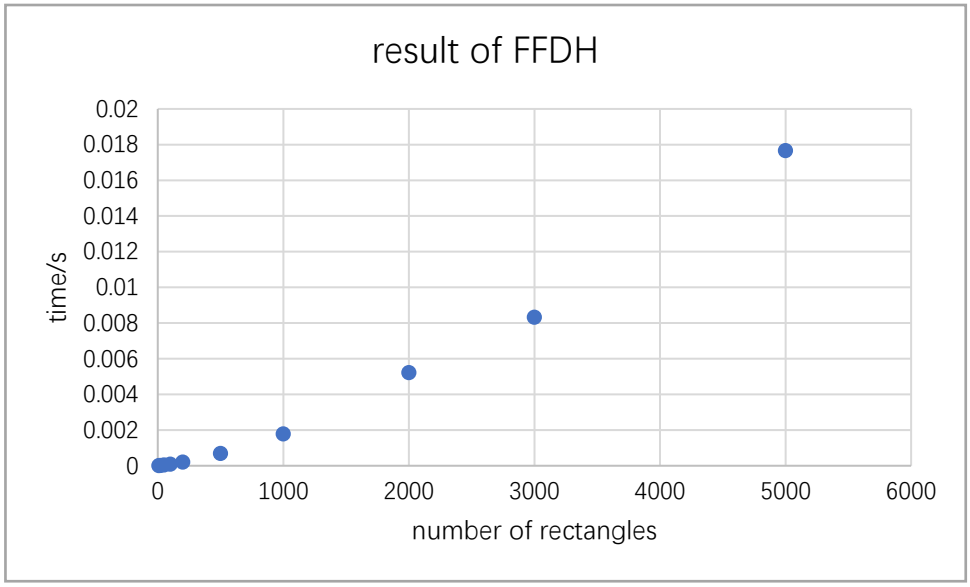
A. Time complexity

In PPT, we can get the time complexity of First-Fit algorithm. However, it is not equal to the time complexity of FFDH algorithm, since there is a sorting process in this algorithm. In this algorithm, we need to sort all rectangles by height. After that, insert them in accordance with the FF algorithm.

The analysis process of time complexity is very simple: The sorting algorithm contributes to the time complexity of O (NlogN), then we can implement the FFDH algorithm with the complexity of O (NlogN) .

So, the time complexity of this algorithm is O(NlogN).

It fits well with our experimental results:



B. Space complexity

In this algorithm, we need no extra auxiliary space, so it is O(1).

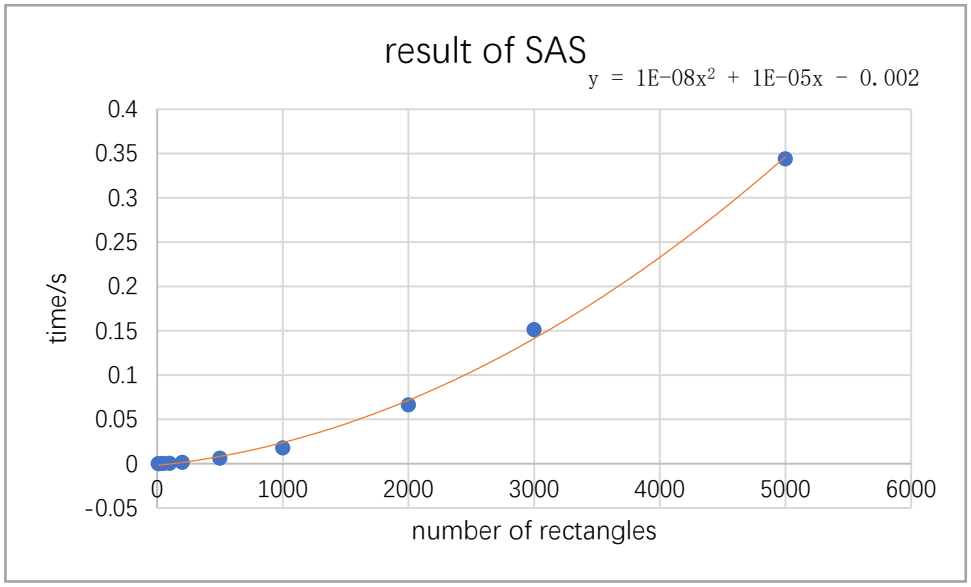
②SAS

A. Time complexity

It is hard to analyze the time complexity of SAS algorithm. It is hard to know, when the recursion process will end. So we can only know time complexity of fuzzy computation is between O (NlogN) and O (N^2).

The sorting process contributes the time complexity of O(NlogN), then for each selection, it may take 1~(N-1) more useless selections. So the worst case is that we get a time complexity of O(N^2).

It fits the following function formula very well:



B. Space complexity

For every possible case, it may call the N sub function recursively, so the extra space required is O (N).

③GRASP

A. Time complexity

Time complexities analysis of algorithm **GRASP**: it should be announced that the iteration time **"k"** will be ignore in time analysis, which means that time complexity is only related to each iteration.

As the algorithm principle has been introduced before(seen in Chapter 2.2 part C). The time taken by the program is

mainly produced in function “FillRectangle”(seen in source code function.cpp), which will find solution following greedy random search procedure. In this routine, this offline algorithm simply implement an ergodic among all unplaced rectangles to seak for the “best fit” rectangle to be placed on the lowest segment. And this procedure will iterate **N** times where **N** represents the number of rectangles. It should be note that even the “best fit” rectangle is finded in the first few steps, the other rectangles are also should be evaluated for the heuristic function. So time complexity of **GRASP** can always be denoted as:

$$O(N + (N-1) + \cdots + 2 + 1) = O(N^2)$$

B. Space complexity

For **space complexity** of this algorithm, it should be $O(N)$ because the stack segment of memory won’t be fill ceaselessly by recursive calling function and there isn’t any active allocation in loops.

4.2 Comments on the testing results

In this project, we have done a lot of tests. The main tests are focused on the GRASP algorithm. After obtaining the results, our

analysis is mainly based on time complexity and approximate proportions.

In terms of time complexity:

The ffdh algorithm is the most stable, and the speed is always the fastest, and stays at O (nlogn).

SAS algorithm‘s running speed is fast, but slower than ffdh.

The GRASP algorithm runs slowly, and the time complexity is always kept in O (N^2). We can see that it takes a lot of time in large-scale operations, for example, the 5000 query takes about 10 seconds.

In terms of approximation rate:

The ffdh algorithm is very unstable, and the approximation rate is always between 1.3 and 1.5. According to the dialogue, we can see that as long as there is a very long rectangle in the partition, the result will be very poor, because it wastes a lot of space at the bottom.

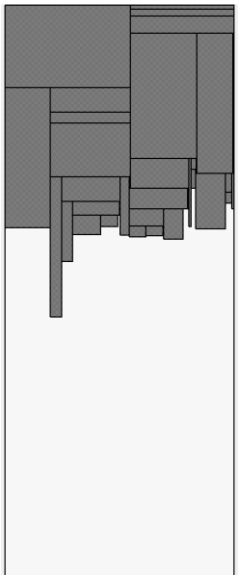
The performance of SAS algorithm is better than the FFDH algorithm, but is not obviously better when the number of rectangles is small. It performs very well in the rectangle and the approximation rate can even be around 1.1.

What is most worth mentioning is our GRASP algorithm, no matter how many rectangles, it always has excellent performance and can be stabilized at 1.0-1.1. After the number exceeds 1000, the approximate degree can basically reach 1.

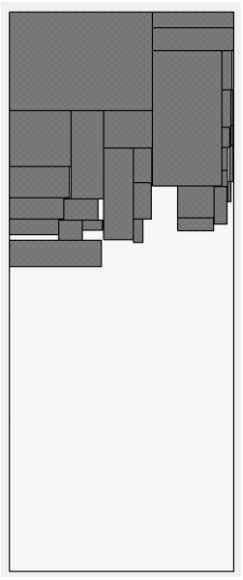
4.3 Further possible improvements

For algorithm **GRASP**, we implemented an improvement routine called **"Variable Neighbour Search"** to handle with the cases when placing the last few rectangles while they are "thin and high", forming peak shape and causing the height to be abnormally high(seen as following figures).

The optimization idea comes from these cases that if we force the last few "thin and high" rectangles to be replaced by their rotation shape "flat and low", then we may get a lower maximum height, like the right figure for instance.



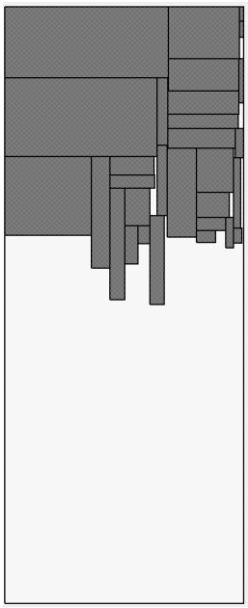
Before Improving



After Improving

The iteration time of **Variable Neighbour Search** is based on the problem size, the number of rectangles more explicitly. We set the iteration time to be **problem size/5**, which should be an hyper parameter of the heuristic function. Different choice of this parameter will lead to different performance of the algorithm.

While a drawback of both the algorithm and the **Variable Neighbour Search** routine should be point out, which has been shown in the following figure:



Among the last few rectangles, if there are two or more rectangles have nearly the same height. Then **Variable Neighbour Search** routine will do nothing to the initial solution, for the reason that **Variable Neighbour Search** is based on the difference of height between the highest segment and the reciprocal second high segment.

If the difference is too large to be accepted, then the last rectangle should be replaced. But if there are two or more rectangles have nearly the same height, the improvement algorithm will be invalid, which should be optimized later.

Chapter 5: Code

5.1 Texture Packing.h

```
#pragma once

#include <QtWidgets/QMainWindow>
#include "ui_TexturePacking.h"
#include <QWidget>
#include <QPainter>
#include <QPen>
#include <QColor>
#include <QBrush>
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <ctime>
#include <fstream>
#include <qwindowdefs.h>
#include <ctime>
#include <cstdlib>
#define LOWEST 1
#define HIGHEST 0
#define CANDIDATE_WIDTH 0
#define CANDIDATE_HEIGHT 1000
#define HEIGHT_WEIGHT 2
#define EPS 1e-8
#define maxsize 5000

using namespace std;
```

```
28.
29. class TexturePacking : public QMainWindow
30. {
31.     Q_OBJECT
32.
33. public:
34.     TexturePacking(QWidget *parent = Q_NULLPTR);
35.
36. public slots:
37.     void ClickButtom();
38.
39. private:
40.     Ui::TexturePackingClass ui;
41.
42. protected:
43.     void paintEvent(QPaintEvent *event);
44. };
45.
46. class Rectangle
47. {
48. public:
49.     Rectangle() : bl_x(-1), bl_y(-1) {};
50.     Rectangle(int w, int h) : width(w), height(h), bl_x(-1), bl_y(-
    1) {};            //overload ctor with two parameters
51.     Rectangle(int w, int h, int x, int y) : width(w), height(h), bl_x(x), bl
    _y(y) {}; //overload ctor with four parameters
52.     ~Rectangle() {};
53.     int get_width() const { return width; } //get width of rectangle
54.     int get_height() const { return height; }  //get height of rectangle
55.     int get_x() const { return bl_x; }         //get cordinate x of rectangl
    e
56.     int get_y() const { return bl_y; }         //get cordinate y of rectangl
    e
57.     bool get_placed() const { return placed; } //get if the rectangle is pla
    ced
58.     bool operator==(const Rectangle &n);       //overload == to judge wheter
     two instances are the same
59.     void rotate();
60.     void set_bl_x(int x) { bl_x = x; }
61.     void set_bl_y(int y) { bl_y = y; }
62.     void place() { placed = true; }
63.     void reset() { placed = false; }
64.
65. private:
```

```
66.     int bl_x, bl_y;
67.     int width, height;
68.     bool placed = false; //initialized not placed
69. };
70.
71. class Segment
72. { //horizontal segment class
73. public:
74.     Segment() {};
75.     Segment(int x1, int x2, int y_h) : xl(x1), xr(x2), y(y_h) {}; //overload
     ctor with given parameters
76.     ~Segment() {};
77.     int get_xl() const { return xl; } //get left end point
78.     int get_xr() const { return xr; } //get right end point
79.     int get_y() const { return y; }   //get y
80.     void set_xl(int x) { xl = x; }  //set left end point
81.     void set_xr(int x) { xr = x; }  //set right end point
82.     void set_y(int y_h) { y = y_h; }  //set y
83. private:
84.     int xl, xr; //left end point and right end point
85.     int y;      //unique y because the segment is horizontal
86. };
87.
88.
89. class Layer   // a class for FFDH algorithm to describe its layer
90. {
91. public:
92.     Layer() {};
93.     Layer(int width, int height, int floorh) :width(width), height(height),
    floorh(floorh) {};
94.     ~Layer() {};
95.     int get_width() const { return width; } //get the current width
96.     int get_height() const { return height; } //get the layer's height:the h
    ighest rectangle's y value
97.     int get_floorh() const { return floorh; } //get the start height of this
     layer
98.     void set_width(int w) { width = w; }      //set the width
99.     void set_height(int h) { height = h; }    //set the height
100.     void set_floorh(int fh) { floorh = fh; }  //set the floor_height
101. private:
102.     int width;
103.     int height;
104.     int floorh;
105. };
```

```
106.
107. class Area    //a class for the SAS algorithm to store the boundary values
     of an area to place the rectangles
108. {
109. public:
110.     Area() {};
111.     Area(int xl, int xr, int yb, int ya) : xl(xl), xr(xr), yb(yb), ya(ya){}
     ;
112.     ~Area() {};
113.     int get_xl() const { return xl; }   //get the left x value
114.     int get_xr() const { return xr; }   //get the right x value
115.     int get_yb() const { return yb; }   //get the below y value
116.     int get_ya() const { return ya; }   //get the above y value
117.     void set_xl(int x) { xl = x; }      //set the x_left
118.     void set_xr(int x) { xr = x; }      //set the x_right
119.     void set_yb(int y) { yb = y; }      //set the y_below
120.     void set_ya(int y) { ya = y; }      //set the y_above
121. private:
122.     int xl, xr;
123.     int yb, ya;
124. };
125.
126.
127.
128. vector<Rectangle> GetRectangle(int &strip_width); //read all required data
     from input
129. vector<Rectangle> GetOptimalSolution(int &strip_width);
130. bool LengthCheck(vector<Rectangle> &rec_set, int width);
                           //check if there exists a solution
131. void PrintRectangles(vector<Rectangle> rec_set);
                           //print all rectangle attributes in rectangle set
132. void PrintContour(vector<Segment> Contour);
                              //print all segment attributes in contour
133. void PrintSegment(Segment seg);
                            //print attributes of a single segment
134. bool SegmentLRSort(Segment a, Segment b);
                            //sort segments from left to right
135. vector<Segment>::iterator FindSegment(vector<Segment> &Contour, int flag);
                            //find segment with minimum y or maximum y
136. void ConstructRCL(vector<Rectangle> &rec_set, vector<Rectangle> &RCL, int l
     ength, int delta_y);     //construct a candidate list
137. vector<Rectangle>::iterator FindRectangle(vector<Rectangle> &rec_set, Recta
     ngle &rec);              //find if a rectangle is in rectangle set
```

```
138. void CombineSegment(vector<Segment> &Contour);
                                //combine neighbour segments which have the same
    height
139. void LevelUpdate(vector<Segment> &Contour);
                                //when no rectangle can be placed, then implement
     level update
140. bool BestFit(vector<Rectangle> &unplaced, vector<Segment> &Contour, Rectang
    le &best_fit_rectangle); //seak for the best fit unplaced rectangle
141. void FillRectangle(vector<Rectangle> &rec_set, vector<Segment> &Contour);
                               //fill rectangles in the strip
142. int IterativeFindSolution(vector<Rectangle> &rec_set, int strip_width, int
    k);
143. void VariableNeighbourSearch(vector<Rectangle> &rec_set, vector<Segment> &C
    ontour, int width);
144. void RandomRectangles(int width, int height, int recnum);
                               //randomly generate the rectangles
145. void placeFirstRec(vector<Rectangle>& rec, vector<Rectangle>& curRec, int&
    curX, int& curY, int& height);  //place a layer's first rectangle in the SAS
     algorithm
146. int FFDH(vector<Rectangle> &rec, int strip_width);                    //the
     core of the FFDH algorithm
147. int SAS(vector<Rectangle> &rec, int strip_width);                     //the
     core of the SAS algorithm
148. void placeNarrowRec(vector<Rectangle> &rec, vector<Rectangle> &narrowRec, v
    ector<Rectangle> &wideRec, Area& curArea, int& way);
149. void placeWideRec(vector<Rectangle> &rec, vector<Rectangle> &narrowRec, vec
    tor<Rectangle> &wideRec, Area& curArea, int& way); // the procedure of placi
    ng wideRec rectangles
150. int SumArea(vector<Rectangle> &rec_set);                              //to
    calculate the total area of all the rectangla
151. void setRecPosition(vector<Rectangle>& rec, vector<Rectangle>& curRec, Area
    & curArea, int index);   //set the position for the rectangle
152. void defineArea(Area& curArea, int xl, int xr, int yb, int ya);      //set
    the x&y values for the area
153. bool sortFFDH(Rectangle& rec1, Rectangle& rec2);             //define the s
    ort order of the FFDH algorithm
154. bool sortNarrow(Rectangle& rec1, Rectangle& rec2);           //define the s
    ort order of narrow rectangles in the SAS algorithm
155. bool sortWide(Rectangle& rec1, Rectangle& rec2);             //define the s
    ort order of wide rectangles in the SAS algorithm
```

5.2 random.cpp

```
#include "TexturePacking.h"
int rec[maxsize][4];

void RandomRectangles(int width, int height, int recnum)
{
    int size = 1;
    rec[0][0] = height;
    rec[0][1] = width;
    rec[0][2] = 0;
    rec[0][3] = 0;
    int type = 0;
    for (int i = 1; i < recnum; i++)
    {
        int chosen_to_devide = rand() % (size);
        type = type + 1;
        int count = 0;
        if (type % 2 == 0) //wide devide
        {
            int devide_height = rand() % rec[chosen_to_devide][0];
            while (devide_height < 0.3 * rec[chosen_to_devide][0] || devide_height > 0.7 * rec[chosen_to_devide][0])
            {
                devide_height = rand() % rec[chosen_to_devide][0];
                count++;
                if (count > 10)
                    break;
            }
            if (devide_height < 0.3 * rec[chosen_to_devide][0] || devide_height > 0.7 * rec[chosen_to_devide][0])
            {
                i--;
                continue;
            }
            rec[size][0] = rec[chosen_to_devide][0] - devide_height;
            rec[size][1] = rec[chosen_to_devide][1];
            rec[size][2] = rec[chosen_to_devide][2];
            rec[size][3] = rec[chosen_to_devide][3] + devide_height;
            rec[chosen_to_devide][0] = devide_height;
            size++;
        }
        else //height devide
        {
```

```
41.             int devide_width = rand() % rec[chosen_to_devide][1];
42.             while (devide_width < 0.3 * rec[chosen_to_devide][1] || devide_w
    idth > 0.7 * rec[chosen_to_devide][1])
43.             {
44.                 devide_width = rand() % rec[chosen_to_devide][1];
45.                 count++;
46.                 if (count > 10)
47.                     break;
48.             }
49.             if (devide_width < 0.3 * rec[chosen_to_devide][1] || devide_widt
    h > 0.7 * rec[chosen_to_devide][1])
50.             {
51.                 i--;
52.                 continue;
53.             }
54.             rec[size][1] = rec[chosen_to_devide][1] - devide_width;
55.             rec[size][0] = rec[chosen_to_devide][0];
56.             rec[size][2] = rec[chosen_to_devide][2] + devide_width;
57.             rec[size][3] = rec[chosen_to_devide][3];
58.
59.             rec[chosen_to_devide][1] = devide_width;
60.             size++;
61.         }
62.     }
63.     ofstream myfile("random.txt", ios::out); //example.txt 是你要输出的文件的名
    字
64.     myfile << width << " " << recnum << endl;
65.     for (int j = 0; j < recnum; j++)
66.     {
67.         myfile << rec[j][0] << " " << rec[j][1] << endl;
68.     }
69.
70.     ofstream anotherfile("optimal.txt", ios::out);
71.
72.     anotherfile << width << " " << recnum << endl;
73.     for (int j = 0; j < recnum; j++)
74.     {
75.         anotherfile << rec[j][2] << " " << rec[j][3] << endl;
76.     }
77.     anotherfile.close();
78.     myfile.close();
79. }
```

5.3 function.cpp

```
1.  #include "TexturePacking.h"
2.
3.  vector<Rectangle> GetRectangle(int &strip_width) //read all required data fr
    om input
4.  {
5.      vector<Rectangle> rec_set; //set of rectangles
6.      int rec_num;               //strip_width and rec_num are specified by us
    er
7.      int width, height;         //rectangle parameters
8.      int i;
9.      ifstream infile;
10.     infile.open("random.txt");
11.     infile >> strip_width >> rec_num;
12.     for (i = 0; i < rec_num; i++)
13.     {
14.         infile >> height >> width;
15.         Rectangle rec(width, height);
16.         rec_set.push_back(rec);
17.     }
18.     return rec_set;
19. }
20.
21. vector<Rectangle> GetOptimalSolution(int &strip_width) //read all required d
    ata from input
22. {
23.     vector<Rectangle> rec_set; //set of rectangles
24.     int rec_num;               //strip_width and rec_num are specified by us
    er
25.     int width, height;         //rectangle parameters
26.     int x, y;
27.     int i;
28.     ifstream infile_shape;
29.     ifstream infile_position;
30.     infile_shape.open("random.txt");
31.     infile_position.open("optimal.txt");
32.     infile_shape >> strip_width >> rec_num;
33.     infile_position >> x >> y; //discard
34.     for (i = 0; i < rec_num; i++)
35.     {
36.         infile_shape >> height >> width;
37.         infile_position >> x >> y;
38.         Rectangle rec(width, height, x, y);
```

```
39.         rec_set.push_back(rec);
40.     }
41.     return rec_set;
42. }
43.
44. bool LengthCheck(vector<Rectangle> &rec_set, int width)
45. {
46.     vector<Rectangle>::iterator iter;
47.     for (iter = rec_set.begin(); iter != rec_set.end(); iter++)
48.     {
49.         if ((*iter).get_height() < width + EPS || (*iter).get_width() < widt
    h + EPS)
50.             return true;
51.     }
52.     return false;
53. }
54.
55. void PrintRectangles(vector<Rectangle> rec_set) //print all rectangle attrib
    utes in rectangle set
56. {
57.     vector<Rectangle>::iterator rec_iter;
58.     int w, h;
59.     for (rec_iter = rec_set.begin(); rec_iter != rec_set.end(); rec_iter++)

60.     {
61.         w = (*rec_iter).get_width();
62.         h = (*rec_iter).get_height();
63.         cout << "width: " << w << " height: " << h << " x: " << (*rec_iter).
    get_x() << " y: " << (*rec_iter).get_y() << endl;
64.     }
65. }
66.
67. void PrintContour(vector<Segment> Contour) //print all segment attributes in
     contour
68. {
69.     vector<Segment>::iterator seg_iter;
70.     int xl, xr, y;
71.     for (seg_iter = Contour.begin(); seg_iter != Contour.end(); seg_iter++)

72.     {
73.         xl = (*seg_iter).get_xl();
74.         xr = (*seg_iter).get_xr();
75.         y = (*seg_iter).get_y();
76.         cout << "xl: " << xl << " xr: " << xr << " y: " << y << endl;
```

```
77.     }
78. }
79.
80. void PrintSegment(Segment seg) //print attributes of a single segment
81. {
82.     int xl, xr, y;
83.     xl = seg.get_xl();
84.     xr = seg.get_xr();
85.     y = seg.get_y();
86.     cout << "xl: " << xl << " xr: " << xr << " y: " << y << endl;
87. }
88.
89. bool SegmentLRSort(Segment a, Segment b) //function parameter in sort functi
    on
90. {
91.     return (a.get_xl() < b.get_xl());
92. }
93.
94. vector<Segment>::iterator FindSegment(vector<Segment> &Contour, int flag) //
    find segment with minimum y or maximum y
95. {
96.     int y_low = 10000;
97.     int y_high = -1;
98.     int y;
99.     vector<Segment>::iterator seg_iter, rseg;
100.
101.     if (flag == LOWEST)
102.     {
103.         for (seg_iter = Contour.begin(); seg_iter != Contour.end(); seg_ite
    r++)
104.         {
105.             y = (*seg_iter).get_y();
106.             if (y < y_low)
107.             {
108.                 rseg = seg_iter;
109.                 y_low = y;
110.             }
111.         }
112.     }
113.     else if (flag == HIGHEST)
114.     {
115.         for (seg_iter = Contour.begin(); seg_iter != Contour.end(); seg_ite
    r++)
116.         {
```

```
117.             y = (*seg_iter).get_y();
118.             if (y > y_high)
119.             {
120.                 rseg = seg_iter;
121.                 y_high = y;
122.             }
123.         }
124.     }
125.     return rseg;
126. }
127.
128. vector<Rectangle>::iterator FindRectangle(vector<Rectangle> &rec_set, Recta
    ngle &rec) //find if a rectangle is in rectangle set
129. {
130.     vector<Rectangle>::iterator rec_iter;
131.     for (rec_iter = rec_set.begin(); rec_iter != rec_set.end(); rec_iter++)

132.     {
133.         if ((*rec_iter).get_placed() == false && *rec_iter == rec)
134.             break;
135.     }
136.     return rec_iter;
137. }
138.
139. void ConstructRCL(vector<Rectangle> &rec_set, vector<Rectangle> &RCL, int l
    ength, int delta_y) //construct a candidate list
140. {
141.     vector<Rectangle>::iterator rec_iter;
142.     int width, height;
143.     for (rec_iter = rec_set.begin(); rec_iter != rec_set.end(); rec_iter++)

144.     {
145.         width = (*rec_iter).get_width();
146.         height = (*rec_iter).get_height();
147.         if (width <= length && width + CANDIDATE_WIDTH >= length && height
    < delta_y + CANDIDATE_HEIGHT) //three constrains should be satisfied
148.             RCL.push_back(*rec_iter);
149.         else if (height <= length && height + CANDIDATE_WIDTH >= length &&
    width < delta_y + CANDIDATE_HEIGHT) //rotate situation
150.         {
151.             (*rec_iter).rotate();
152.             RCL.push_back(*rec_iter);
153.         }
154.     }
```

```
155. }
156.
157. void CombineSegment(vector<Segment> &Contour)
158. {
159.     vector<Segment>::iterator iter, next_iter;
160.     int flag = 0; //iteration flag
161.     int length;
162.     while (flag != 1)
163.     {
164.         for (iter = Contour.begin(); iter != Contour.end(); iter++)
165.         {
166.             if (flag == 1)
167.                 break;
168.             next_iter = iter + 1;
169.             if (next_iter == Contour.end())
170.                 flag = 1;
171.             else
172.             {
173.                 if ((*iter).get_y() == (*next_iter).get_y())
174.                 {
175.                     length = (*next_iter).get_xr() - (*next_iter).get_xl();

176.                     (*iter).set_xr((*iter).get_xr() + length);
177.                     Contour.erase(next_iter);
178.                     break;
179.                 }
180.             }
181.         }
182.         if (iter == Contour.end())
183.             flag = 1;
184.     }
185. }
186.
187. void LevelUpdate(vector<Segment> &Contour)
188. {
189.     vector<Segment>::iterator iter, left_iter, right_iter;
190.     iter = FindSegment(Contour, LOWEST);
191.     if (iter != Contour.end())
192.         right_iter = iter + 1;
193.     if (iter != Contour.begin())
194.         left_iter = iter - 1;
195.     if (iter == Contour.begin()) //the first segment is the lowest
196.     {
197.         if (right_iter != Contour.end())
```

```
198.         {
199.             (*iter).set_y((*right_iter).get_y());
200.             (*iter).set_xr((*right_iter).get_xr());
201.             Contour.erase(right_iter);
202.         }
203.     }
204.     else if (iter == Contour.end() - 1) //the last segment is the lowest
205.     {
206.         (*iter).set_y((*left_iter).get_y());
207.         (*iter).set_xl((*left_iter).get_xl());
208.         Contour.erase(left_iter);
209.     }
210.     else //middle case
211.     {
212.         int left_y = (*left_iter).get_y();
213.         int right_y = (*right_iter).get_y();
214.         if (left_y < right_y) //left segment is lower than right segment
215.         {
216.             (*iter).set_y((*left_iter).get_y());
217.             (*iter).set_xl((*left_iter).get_xl());
218.             Contour.erase(left_iter);
219.         }
220.         else if (left_y > right_y) //left segment is higher than right segm
    ent
221.         {
222.             (*iter).set_y((*right_iter).get_y());
223.             (*iter).set_xr((*right_iter).get_xr());
224.             Contour.erase(right_iter);
225.         }
226.         else //left segment and right segment have the same height
227.         {
228.             (*iter).set_y((*right_iter).get_y());
229.             (*iter).set_xl((*left_iter).get_xl());
230.             (*iter).set_xr((*right_iter).get_xr());
231.             Contour.erase(right_iter);
232.             Contour.erase(left_iter);
233.         }
234.     }
235. }
236.
237. bool BestFit(vector<Rectangle> &unplaced, vector<Segment> &Contour, Rectang
    le &best_fit_rectangle) //return true for finded
238. {
239.     vector<Rectangle>::iterator rec_iter;
```

```
240.     vector<Segment>::iterator seg_iter_min_y, seg_iter_max_y;
241.     int length;
242.     int delta_y;
243.     int width, height;
244.     double min_evaluation = 10000; //minimum evaluation value should be sat
    isfied, initialize large
245.     double cur_evaluation;
246.
247.     seg_iter_min_y = FindSegment(Contour, LOWEST);  //find segment with min
    imum y in contour(iterator returned)
248.     seg_iter_max_y = FindSegment(Contour, HIGHEST); //find segment with max
    imum y in contour(iterator returned)
249.     length = (*seg_iter_min_y).get_xr() - (*seg_iter_min_y).get_xl();
250.     delta_y = (*seg_iter_max_y).get_y() - (*seg_iter_min_y).get_y();
251.     bool finded = false; //finded or not, 0 represents not finded, 1 for fi
    nded
252.     for (rec_iter = unplaced.begin(); rec_iter != unplaced.end(); rec_iter+
    +)
253.     {
254.         width = (*rec_iter).get_width();
255.         height = (*rec_iter).get_height();
256.         if (width > length && height > length)
257.             continue;
258.         else if ((width > length) || (height > width && height < length))
259.         {
260.             (*rec_iter).rotate();
261.         }
262.         cur_evaluation = 1.0 * (length - (*rec_iter).get_width()) / (*rec_i
    ter).get_height(); //evaluation function
263.         if (cur_evaluation < min_evaluation)
264.         {
265.             min_evaluation = cur_evaluation;
266.             best_fit_rectangle = *rec_iter;
267.             finded = true;
268.         }
269.     }
270.     return finded;
271. }
272.
273. void FillRectangle(vector<Rectangle> &rec_set, vector<Segment> &Contour)
274. {
275.     int length;  //length of the lowest segment
276.     int delta_y; //height difference between the highest segment and the lo
    west segment
```

```
277.     int i;
278.     vector<Rectangle> RCL;              //restricted candidate list for re
    ctangles
279.     vector<Rectangle> solution;         //partial solution list of rectang
    els
280.     vector<Rectangle> unplaced(rec_set); //partial solution list of rectang
    els, initialized all unplaced
281.     vector<Rectangle>::iterator rec_iter;
282.     vector<Segment>::iterator seg_iter_min_y, seg_iter_max_y;
283.     srand((unsigned)time(NULL)); //used to generate random number(for rando
    m choose)
284.     Rectangle best_fit_rectangle;
285.
286.     //PrintRectangles(RCL);
287.     while (!unplaced.empty())
288.     {
289.         RCL.clear();                                    //clear restricted
    candidate list for each iteration
290.         seg_iter_min_y = FindSegment(Contour, LOWEST);  //find segment with
     minimum y in contour(iterator returned)
291.         seg_iter_max_y = FindSegment(Contour, HIGHEST); //find segment with
     minimum y in contour(iterator returned)
292.         length = (*seg_iter_min_y).get_xr() - (*seg_iter_min_y).get_xl();
293.         delta_y = (*seg_iter_min_y).get_y() - (*seg_iter_min_y).get_y();
294.
295.         ConstructRCL(unplaced, RCL, length, delta_y); //use the lowest segm
    ent to construct restric candidate list RCL
296.         if (!RCL.empty())
297.         {
298.             i = rand() % RCL.size(); //random choose from RCL
299.             best_fit_rectangle = RCL[i];
300.         }
301.         else
302.         {
303.             if (BestFit(unplaced, Contour, best_fit_rectangle) == false) //
    RCL is empty so choose from unplcaed rectangles which best fit the segment
304.             {
305.                 LevelUpdate(Contour);
306.                 continue;
307.             }
308.         }
309.         rec_iter = FindRectangle(unplaced, best_fit_rectangle);
310.         solution.push_back(*rec_iter); //push into solution list
311.         unplaced.erase(rec_iter);      //remove the rectangle
```

```
312.         rec_iter = FindRectangle(rec_set, best_fit_rectangle);
313.         (*rec_iter).set_bl_x((*seg_iter_min_y).get_xl()); //place the recta
    ngle
314.         (*rec_iter).set_bl_y((*seg_iter_min_y).get_y());  //place the recta
    ngle
315.         (*rec_iter).place();
316.         if (length - best_fit_rectangle.get_width() == 0) //equivalent
317.         {
318.             (*seg_iter_min_y).set_y((*seg_iter_min_y).get_y() + best_fit_re
    ctangle.get_height()); //update the segment on which rectangle is placed
319.             CombineSegment(Contour);
                            //combine neighbour segments which have the same heig
    ht
320.         }
321.         else
322.         {
323.             Segment seg((*seg_iter_min_y).get_xl(), (*seg_iter_min_y).get_x
    l() + best_fit_rectangle.get_width(), (*seg_iter_min_y).get_y() + best_fit_r
    ectangle.get_height());
324.             (*seg_iter_min_y).set_xl((*seg_iter_min_y).get_xl() + best_fit_
    rectangle.get_width());
325.             Contour.push_back(seg);
326.             sort(Contour.begin(), Contour.end(), SegmentLRSort); //keep the
     left to right order of segments
327.             CombineSegment(Contour);                                //combine
    neightbour segments which have the same height
328.         }
329.     }
330. }
331.
332. int IterativeFindSolution(vector<Rectangle> &rec_set, int strip_width, int
    k)
333. {
334.     vector<Rectangle>::iterator rec_iter;
335.     vector<Segment> Contour;      //segment set represent the contour
336.     vector<Segment> best_Contour; //segment set represent the contour
337.     vector<Segment>::iterator seg_iter;
338.     vector<Rectangle> best_rec_set;
339.     int i;
340.     int min_height = 10000;
341.     for (i = 0; i < k; i++) //simply iterate k times to seak for the best s
    olution
342.     {
343.         Contour.clear();
```

```
344.         Segment seg(0, strip_width, 0);
        //the bottom segment
345.         Contour.push_back(seg);
        //Contour initialization
346.         for (rec_iter = rec_set.begin(); rec_iter != rec_set.end(); rec_ite
    r++) //reset all rectangles
347.             (*rec_iter).reset();
348.         FillRectangle(rec_set, Contour);            //seak for a simple sol
    ution
349.         VariableNeighbourSearch(rec_set, Contour, strip_width);     //impro
    vment
350.         seg_iter = FindSegment(Contour, HIGHEST); //find the highest segmen
    t which represents the height of strip
351.         if ((*seg_iter).get_y() < min_height)     //update or not
352.         {
353.             min_height = (*seg_iter).get_y(); //record information for the
    best solution
354.             best_Contour = Contour;
355.             best_rec_set = rec_set;
356.         }
357.     }
358.     return min_height;
359. }
360.
361. void VariableNeighbourSearch(vector<Rectangle> &rec_set, vector<Segment> &C
    ontour, int width)
362. {
363.     vector<Rectangle>::iterator rec_iter;
364.     vector<Segment>::iterator seg_iter, seg_iter_prior, seg_low;
365.     int delta_y;
366.     int length;
367.     int i, k;
368.     k = rec_set.size()/5;
369.     for (i = 0; i < k; i++)
370.     {
371.         seg_iter = FindSegment(Contour, HIGHEST);   //find highest segment
    in contour
372.         int highest = (*seg_iter).get_y();          //store the highest seg
    ment
373.         (*seg_iter).set_y(0);
374.         seg_iter_prior = FindSegment(Contour, HIGHEST);     //last but one
    highest segment
375.         (*seg_iter).set_y(highest);                 //recover the highest s
    egment
```

```
376.         delta_y = (*seg_iter).get_y() - (*seg_iter_prior).get_y();
377.         for (rec_iter = rec_set.begin(); rec_iter != rec_set.end(); rec_ite
    r++)
378.         {
379.             if ((*rec_iter).get_y() + (*rec_iter).get_height() == highest &
    & (*rec_iter).get_width() == (*seg_iter).get_xr() - (*seg_iter).get_xl() &&
    (*rec_iter).get_x() == (*seg_iter).get_xl()) //rec_iter points to the rectan
    gle which is highest
380.                 break;
381.         }
382.         if (rec_iter == rec_set.end())
383.             continue;
384.         if ((*rec_iter).get_width() < delta_y && (*rec_iter).get_height() <
    = width)
385.         {
386.             (*rec_iter).rotate();               //rotate this thin and tall
     rectangle, update segments
387.             (*seg_iter).set_y(highest - (*rec_iter).get_width());       //s
    egment descent
388.             CombineSegment(Contour);            //combine segments with the
     same height
389.             while (1)               //find the lowest segment or implement
    level update
390.             {
391.                 seg_low = FindSegment(Contour, LOWEST);
392.                 length = (*seg_low).get_xr() - (*seg_low).get_xl();
393.                 if (length >= (*rec_iter).get_width())   //place rectangle
    on it
394.                 {
395.                     (*rec_iter).set_bl_x((*seg_low).get_xl()); //place the
    rectangle
396.                     (*rec_iter).set_bl_y((*seg_low).get_y());  //place the
    rectangle
397.                     if (length - (*rec_iter).get_width() == 0) //equivalent

398.                     {
399.                         (*seg_low).set_y((*seg_low).get_y() + (*rec_iter).g
    et_height()); //update the segment on which rectangle is placed
400.                         CombineSegment(Contour);
                                    //combine neighbour segments which have t
    he same height
401.                     }
402.                     else
403.                     {
```

```
404.                         Segment seg((*seg_low).get_xl(), (*seg_low).get_xl(
    ) + (*rec_iter).get_width(), (*seg_low).get_y() + (*rec_iter).get_height());
405.                         (*seg_low).set_xl((*seg_low).get_xl() + (*rec_iter)
    .get_width());
406.                         Contour.push_back(seg);
407.                         sort(Contour.begin(), Contour.end(), SegmentLRSort)
    ; //keep the left to right order of segments
408.                         CombineSegment(Contour);
      //combine neightbour segments which have the same height
409.                     }
410.                     break;
411.                 }
412.                 else
413.                     LevelUpdate(Contour);
414.             }
415.         }
416.     }
417. }
418.
419. bool decreasing_height(const Rectangle & r1, const Rectangle & r2) //order
    the rectangles by the decreasing height
420. {
421.     return r1.get_height() > r2.get_height();
422. }
423.
424. bool decreasing_hw(const Rectangle & r1, const Rectangle & r2) //order the
    rectangles by the decreasing height then width
425. {
426.     if (r1.get_height() != r2.get_height())
427.         return r1.get_height() > r2.get_height();
428.     else
429.         return r1.get_width() > r2.get_width();
430. }
431.
432. bool decreasing_wh(const Rectangle & r1, const Rectangle & r2) //order the
    rectangles by the decreasing width then height
433. {
434.     if (r1.get_width() != r2.get_width())
435.         return r1.get_width() > r2.get_width();
436.     else
437.         return r1.get_height() > r2.get_height();
438. }
439.
```

```
440. int SumArea(vector<Rectangle> &rec_set) //calculate the total rectangle are
    a
441. {
442.     vector<Rectangle>::iterator iter;
443.     int sum_area = 0;
444.     for (iter = rec_set.begin(); iter != rec_set.end(); iter++)
445.         sum_area += (*iter).get_height()*(*iter).get_width();
446.     return sum_area;
447. }
```

5.4 sasffdh.cpp

```
1.  #include "TexturePacking.h"
2.
3.  int FFDH(vector<Rectangle>& rec, int strip_width)  //the core of  the FFDH a
    lgorithm
4.  {
5.      sort(rec.begin(), rec.end(), sortFFDH);        //sort the rectangles bas
    ed on their height
6.      rec[0].set_bl_x(0);
7.      rec[0].set_bl_y(0);
8.      int w = rec[0].get_width();
9.      int h = rec[0].get_height();
10.     vector<Layer> lay;
11.     lay.clear();
12.     Layer curlay(w, h, 0);                         //init the first layer
13.     lay.push_back(curlay);
14.
15.     for (int i = 1; i < rec.size(); i++)          //when the rectangle set i
    s not none
16.     {
17.         int j;
18.         for (j = 0; j < lay.size(); j++)
19.         {
20.             int emptyw = strip_width - lay[j].get_width();         //can use
    d to place the rectangle
21.             if (rec[i].get_width() <= emptyw)
22.             {
23.                 rec[i].set_bl_x(lay[j].get_width());
        //set the rectangle's position
24.                 rec[i].set_bl_y(lay[j].get_floorh());
```

```
25.                 lay[j].set_width(lay[j].get_width() + rec[i].get_width());
       //update this layer's width
26.                 break;
27.             }
28.         }
29.         if (j == lay.size())            //if all the existed layer can't pla
    ce the rectangle,then create a layer
30.         {
31.             curlay.set_width(rec[i].get_width());
32.             curlay.set_height(lay[j - 1].get_height() + rec[i].get_height())
    ;
33.             curlay.set_floorh(lay[j - 1].get_height());
34.             lay.push_back(curlay);
35.             rec[i].set_bl_x(0);
36.             rec[i].set_bl_y(lay[j].get_floorh());
37.         }
38.     }
39.     return (lay.back()).get_height();   //return the height
40. }
41.
42. bool sortFFDH(Rectangle& rec1, Rectangle& rec2)      //the sorting order of
    the FFDH algorithm
43. {
44.     return rec1.get_height() > rec2.get_height();
45. }
46.
47. int SAS(vector<Rectangle> &rec, int strip_width)     //the core of the SAS a
    lgorithm
48. {
49.     enum { LNARROW, LWIDE }choice1, choice2;         //LNARROW means that la
    st procedure,we put the narrow rectangles and we need to put the wide next
50.     int way = 0;
51.     int curX = 0;
52.     int curY = 0;
53.     int height = 0;
54.     Area curArea;    //curArea: the area can used to place the rectangle
55.     int choice;
56.
57.     vector<Rectangle> narrowRec;       //narrowRec:the set of the narrow rec
    tangles
58.     vector<Rectangle> wideRec;         //wideRec:the set of the wide rectang
    les
59.     vector<Rectangle>::iterator itRec;   //sort the rectangles
60.     for (itRec = rec.begin(); itRec < rec.end(); itRec++)
```

```
61.     {
62.         if ((*itRec).get_width() >= (*itRec).get_height())
63.             wideRec.push_back(*itRec);
64.         else
65.             narrowRec.push_back(*itRec);
66.     }
67.     rec.clear();     //clear the rec for placing the rectangles in our placi
    ng order
68.
69.     sort(narrowRec.begin(), narrowRec.end(), sortNarrow);
70.     sort(wideRec.begin(), wideRec.end(), sortWide);
71.
72.     while (narrowRec.size() || wideRec.size())    //when the set of the rect
    angles is not none
73.     {
74.         if (narrowRec.size() && wideRec.size())   //when the wide and narrow
     both exist
75.         {
76.             if (narrowRec[0].get_height() > wideRec[0].get_height())   //if
    the highest narrow is higher than the highest wide rectangle,we first put th
    e narrow
77.             {
78.                 placeFirstRec(rec, narrowRec, curX, curY, height);
79.                 choice = LNARROW;
80.             }
81.             else
82.             {
83.                 placeFirstRec(rec, wideRec, curX, curY, height);
84.                 choice = LWIDE;
85.             }
86.         }
87.         else if (narrowRec.size())    //if there is no wide rectangles
88.         {
89.             placeFirstRec(rec, narrowRec, curX, curY, height);
90.             choice = LNARROW;
91.         }
92.         else                          //if there is no narrow rectangles
93.         {
94.             placeFirstRec(rec, wideRec, curX, curY, height);
95.             choice = LWIDE;
96.         }
97.
98.         defineArea(curArea, curX, strip_width, curY, height);   //set the ar
    ea to place the rectangles
```

```
99.         if (choice == LNARROW)
100.             placeWideRec(rec, narrowRec, wideRec, curArea, way);
101.         else
102.             placeNarrowRec(rec, narrowRec, wideRec, curArea, way);
103.     }
104.     return height;
105. }
106.
107. void placeNarrowRec(vector<Rectangle> &rec,  vector<Rectangle> &narrowRec,
    vector<Rectangle> &wideRec,Area& curArea,int& way)
108. {
109.     enum { LNARROW, LWIDE }choice1, choice2;
110.     if (narrowRec.size())                           //if there exists the
    narrow rectangle
111.     {
112.         int i;
113.         for (i = 0; i < narrowRec.size(); i++)       //to find the narrow r
    ectangles to fit the area
114.         {
115.             int x_length = curArea.get_xr() - curArea.get_xl();
116.             int y_length = curArea.get_ya() - curArea.get_yb();
117.             if (x_length >= narrowRec[i].get_width() && y_length >= narrowR
    ec[i].get_height())
118.                 break;
119.         }
120.         if (i == narrowRec.size())
121.             return;
122.
123.         int tw = narrowRec[i].get_width();
124.         int th= narrowRec[i].get_height();
125.         setRecPosition(rec, narrowRec, curArea, i);   //set the x & y value
     for the rectangle in the rec set
126.         Area aboveArea(curArea.get_xl(), curArea.get_xl() + tw, curArea.get
    _yb() + th, curArea.get_ya()); //define this rectangle's above area
127.         placeNarrowRec(rec, narrowRec,wideRec,aboveArea,way);    //to put t
    he narrow rectangles in its above area
128.         Area rightArea(curArea.get_xl() + tw, curArea.get_xr(),curArea.get_
    yb(), curArea.get_ya()); //define this rectangle's right area
129.         placeNarrowRec(rec,narrowRec,wideRec,rightArea,way);     //to put t
    he narrow rectangles in its right area
130.     }
131.     else if (way == LWIDE)
132.     {
133.         way = LNARROW;
```

```
134.         return;
135.     }
136.     else         //if there is no narrow rectangle
137.         placeWideRec(rec,narrowRec,wideRec,curArea,way);
138. }
139.
140. void placeWideRec(vector<Rectangle> &rec, vector<Rectangle> &narrowRec, vec
    tor<Rectangle> &wideRec,Area& curArea,int& way) // the procedure of placing
    wideRec rectangles
141. {
142.     enum { LNARROW, LWIDE }choice1, choice2;
143.     int x_length = curArea.get_xr() - curArea.get_xl();
144.     int y_length = curArea.get_ya() - curArea.get_yb();
145.     int txl = curArea.get_xl();
146.
147.     if (wideRec.size() && x_length >= wideRec.back().get_width())  //to see
     if the narrowset wide rectangle can fit: if there exists a wide rectangle t
    o fit
148.     {
149.         int i;
150.         for (i = 0; i < wideRec.size(); i++)
151.             if (curArea.get_xr() - curArea.get_xl() >= wideRec[i].get_width
    ())
152.                 break;
153.         if (i == wideRec.size())
154.             return;
155.         int j;
156.         for (j = i; j < wideRec.size(); j++)       //select the widest rect
    angle to fit
157.         {
158.             if (curArea.get_ya() - curArea.get_yb() >= wideRec[j].get_heigh
    t())         //find the first rectangle to place
159.                 break;
160.         }
161.         if (j == wideRec.size())
162.             return;
163.
164.         int curX = wideRec[j].get_width() + curArea.get_xl();
165.         int curY = curArea.get_yb();
166.         while (j != wideRec.size() && curArea.get_ya() - curArea.get_yb() >
    = wideRec[j].get_height())
167.         {
```

```
168.             if (curArea.get_ya() - curArea.get_yb() >= wideRec[i].get_heigh
    t())   //if the rectangles'height is lower than this layer's highest wide re
    ctangle
169.             {
170.                 int tw = wideRec[j].get_width();
171.                 int th = wideRec[j].get_height();
172.                 int tx = wideRec[j].get_x();
173.                 setRecPosition(rec, wideRec, curArea, j);    //set the x&y
    value of this rectangle in the rec set
174.                 curArea.set_yb(curArea.get_yb() + th);
175.                 if (curArea.get_xr() - (curArea.get_xl() + tw) > 0)   //if
    this layer still has length but can't put a wide rectangle continusly,we put
     narrow
176.                 {
177.                     way = LWIDE;
178.                     if (narrowRec.size())
179.                     {
180.                         Area narrowArea(curArea.get_xl() + tw, curArea.get_
    xr(), curArea.get_yb()-th, curArea.get_ya());
181.                         placeNarrowRec(rec, narrowRec, wideRec, narrowArea,
     way);
182.                         curArea.set_xr(curArea.get_xl() + tw);
183.                     }
184.                 }
185.             }
186.             else
187.                 break;
188.             Area wideArea(curX, curArea.get_xr(), curY, curArea.get_ya());
      //put the wide rectangles in a new layer
189.             placeWideRec(rec, narrowRec, wideRec, wideArea, way);
190.         }
191.
192.     }
193.     else     //there doesn't have a wide rectangle to replace
194.     {
195.         way = LWIDE;
196.         placeNarrowRec(rec, narrowRec, wideRec,curArea,way);
197.     }
198. }
199.
200. void defineArea(Area& curArea, int xl, int xr, int yb, int ya)   //set the
    boundary value of the area
201. {
202.     curArea.set_xl(xl);
```

```
203.     curArea.set_xr(xr);
204.     curArea.set_yb(yb);
205.     curArea.set_ya(ya);
206. }
207.
208. void placeFirstRec(vector<Rectangle>& rec, vector<Rectangle>& curRec, int&
    curX, int& curY, int& height) //to place the first rectangle of a new layer

209. {
210.     curRec[0].set_bl_x(0);
211.     curRec[0].set_bl_y(height);
212.     rec.push_back(curRec[0]);
213.     curY = height;
214.     curX = curRec[0].get_width();
215.     height = height + curRec[0].get_height();
216.     curRec.erase(curRec.begin());
217. }
218.
219. void setRecPosition(vector<Rectangle>& rec, vector<Rectangle>& curRec, Area
    & curArea, int index)  //to put the rectangle in the rec set and set its x&y
     values
220. {
221.     curRec[index].set_bl_x(curArea.get_xl());
222.     curRec[index].set_bl_y(curArea.get_yb());
223.     rec.push_back(curRec[index]);
224.     curRec.erase(curRec.begin() + index);
225. }
226.
227. bool sortNarrow(Rectangle& rec1, Rectangle& rec2)     //to define the sort
    order of the narrow rectangles
228. {
229.     if (rec1.get_height() != rec2.get_height())
230.         return rec1.get_height() > rec2.get_height();
231.     else
232.         return rec1.get_width() > rec2.get_width();
233. }
234.
235. bool sortWide(Rectangle& rec1, Rectangle& rec2)      //to define the sort o
    rder of the wide rectangles
236. {
237.     if (rec1.get_width() != rec2.get_width())
238.         return rec1.get_width() > rec2.get_width();
239.     else
240.         return rec1.get_height() > rec2.get_height();
```

```
241. }
```

5.5 texture.cpp

```
1.  #include "TexturePacking.h"
2.
3.  bool Rectangle::operator==(const Rectangle &n) //overload operator "==" to j
    udge whether two rectangles have same width and height
4.  {
5.      if (this->width == n.get_width() && this->height == n.get_height())
6.          return true;
7.      else if (this->width == n.get_height() && this->height == n.get_width())

8.      {
9.          this->rotate();
10.         return true;
11.     }
12.     else
13.         return false;
14. }
15.
16. void Rectangle::rotate()
17. {
18.     int tmp = this->width;
19.     this->width = this->height;
20.     this->height = tmp;
21. }
```

5.6 Texture Packing.cpp

```
1.  #include "TexturePacking.h"
2.
3.  int paintflag = 0;
4.  TexturePacking::TexturePacking(QWidget *parent)
5.      : QMainWindow(parent)
```

```
{
    ui.setupUi(this);
    connect(ui.calculateButton, SIGNAL(clicked()), this, SLOT(ClickButtom())
);
}

void TexturePacking::ClickButtom()
{
    paintflag = 1; //calculation flag
    update();     //call paintEvent()
}

void TexturePacking::paintEvent(QPaintEvent *event)
{
    int zoom = 2;
    int lrec_x = 50, rrec_x = 700;
    int lrec_y = 200, rrec_y = 200;
    int maxheight = 750;
    QPainter background(this);
    QPixmap pix;
    pix.load("strawberry.ico");
    background.drawPixmap(530, 200, 100, 100, pix);

    if (paintflag == 1) //implement calculation
    {
        int strip_width;
        int min_height;
        int area;
        int i;
        vector<Rectangle>::iterator iter;
        QString str;
        str = ui.WidthEdit->text();
        int width = str.toInt() * zoom;
        str = ui.OptimalHeightEdit->text();
        int optimal_height = str.toInt() * zoom;
        str = ui.NumEdit->text();
        int recnum = str.toInt();
        str = ui.IterEdit->text();
        int k = str.toInt();
        RandomRectangles(width, optimal_height, recnum); //generate random c
ases
        vector<Rectangle> rec_set = GetRectangle(strip_width);
        vector<Rectangle> optimal_rec_set = GetOptimalSolution(strip_width);
```

```
47.         clock_t start, finish;
48.         double totaltime;
49.
50.         if (LengthCheck(rec_set, strip_width) == false)
51.         {
52.             return;
53.         }
54.
55.         if (ui.Algorithm_1->isChecked()) //call algorithm 1
56.         {
57.             start = clock(); //calculate running time
58.             min_height = IterativeFindSolution(rec_set, strip_width, k);
59.             finish = clock(); //calculate running time
60.         }
61.         else if (ui.Algorithm_2->isChecked()) //call algorithm 2
62.         {
63.             start = clock(); //calculate running time
64.             for (i = 0; i < k; i++)
65.                 min_height = FirstFit(rec_set, strip_width);
66.             finish = clock(); //calculate running time
67.         }
68.         else if (ui.Algorithm_3->isChecked()) //call algorithm 3
69.         {
70.             start = clock(); //calculate running time
71.             for (i = 0; i < k; i++)
72.                 min_height = SAS(rec_set, strip_width);
73.             finish = clock(); //calculate running time
74.         }
75.         else //do nothing
76.         {
77.             return;
78.         }
79.
80.         area = SumArea(rec_set);
81.         totaltime = (double)(finish - start) / CLOCKS_PER_SEC;
82.         str = QString::fromStdString(to_string(min_height / 2));
83.         ui.HeightEdit->setText(str);
84.         str = QString::fromStdString(to_string(1.0 * min_height / optimal_he
    ight));
85.         ui.RatioEdit->setText(str);
86.         string stringinit = to_string(totaltime);
87.         stringinit = stringinit.substr(0, stringinit.size() - 3) + "s";
88.         str = QString::fromStdString(stringinit);
89.         ui.TimeEdit->setText(str);
```

```
90.         str = QString::fromStdString(to_string(area * 1.0 / (strip_width * m
    in_height)));
91.         ui.UltilizationEdit->setText(str);
92.
93.         QPainter p;
94.         p.begin(this);
95.         //define a pen for drawing
96.         QPen pen;
97.         pen.setWidth(2);
98.         pen.setColor(QColor(0, 0, 0)); //set color
99.         pen.setStyle(Qt::SolidLine);
100.         //create a brush
101.         QBrush brush;
102.         brush.setColor(Qt::white);         //set color for filling
103.         brush.setStyle(Qt::Dense4Pattern); //set style
104.         p.setPen(pen);
105.         p.setBrush(brush);
106.         p.drawRect(lrec_x, lrec_y, strip_width, maxheight);
107.         p.drawRect(rrec_x, rrec_y, strip_width, maxheight);
108.         //draw all rectangles in rec_set
109.         brush.setColor(Qt::black); //reset color
110.         p.setBrush(brush);
111.         for (iter = rec_set.begin(); iter != rec_set.end(); iter++)
112.         {
113.             p.drawRect(lrec_x + (*iter).get_x(), lrec_y + (*iter).get_y(),
    (*iter).get_width(), (*iter).get_height());
114.         }
115.         for (iter = optimal_rec_set.begin(); iter != optimal_rec_set.end();
     iter++)
116.         {
117.             p.drawRect(rrec_x + (*iter).get_x(), rrec_y + (*iter).get_y(),
    (*iter).get_width(), (*iter).get_height());
118.         }
119.
120.         p.end();
121.     }
122.     paintflag = 0;
123. }
```

Declaration:

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