Research Project 6:

Texture Packing

G23

闵思韬 万园 陈思雅

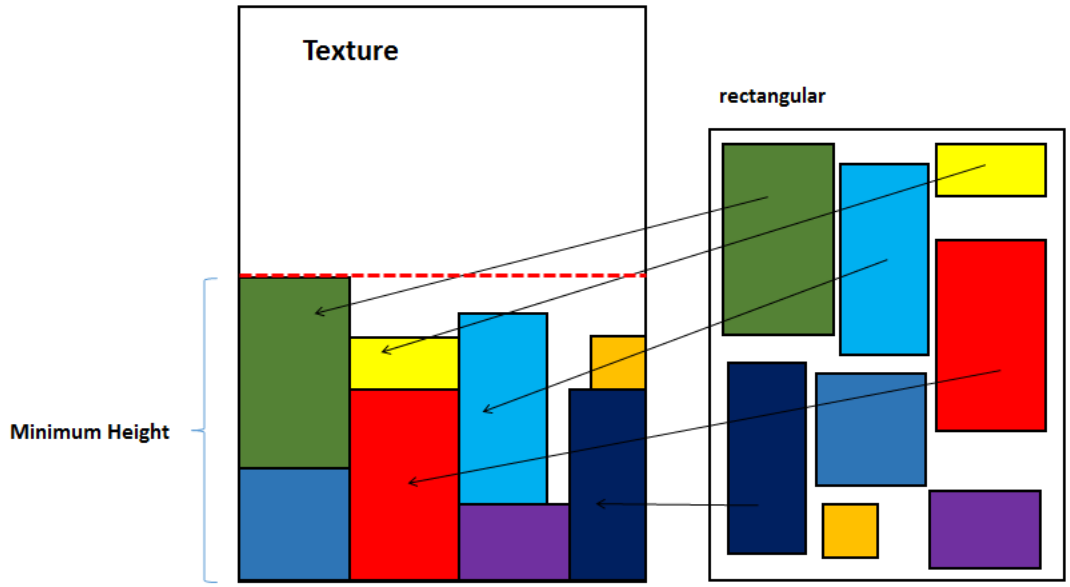
Date: 2016-05-15

Chapter 1: Introduction

Problem description,

Texture Packing is to pack multiple rectangle shaped textures into one large texture. The resulting texture must have a given width and a minimum height. This project requires you to design an approximation algorithm that runs in polynomial time. You must generate test cases of different sizes (from 10 to 10,000) with different distributions of widths and heights. A thorough analysis on all the factors that might affect the approximation ratio of your proposed algorithm is expected.

We draw a picture to show what this problem means.



Related knowledge

Texture and packing

This problem is one of the main objectives in the computer graphics area is highly realistic image synthesis. Texture, which is an attribute of an object's surface, is critical to generating realistic images. It's is one of the surface attributes of an object, consisting of many components, such as color, gloss, and bumps. By simply mapping a texture to a surface, we can obtain a much richer image than a flat color surface.

Realistic image synthesis of an arbitrarily shaped pavement is one of the applications of textures. It would be tedious and time consuming, and thus not practical, for a graphics designer to manually illustrate such texture of paving stone. On the other hand, although it is possible to generate a reasonably satisfactory image by performing texture mapping using a real photograph of a pavement, this approach has the following problems:

(1) When the surface of a pavement is bumpy, the shades and shadows in a real

photograph do not match a computer-generated image.

(2) When the aspect ratio of a photograph differs from that of a pavement, there will be unnatural seams on the texture-mapped surface.

(3) The boundary of a photograph usually differs from that of a pavement, and this causes

undesirable cut-offs of the paving stones at the boundaries or gaps between paving stones and the boundaries.

The proposed pavement simulation method offers a solution to these problems. The packing –pattern that dictates the placement of each paving stone is obtained by means of the square packing technique. The packing pattern for a given road area is generated by performing dynamic simulation of scattered square particles with proximity-based inter=particle forces. A pavement texture is then obtained usually by: generating the geometry of each stone by a subdivision surface method ,and rendering a texture image with fractal noise for each stone.

Square Cell Packing

This section describe our basic approach to the cell generation problem by which we generate a patter for a stone pavement.

Given :

- 2D geometric domain
- Desired size distribution of cell, as scalar field , and
- Desired directionally, given as a vector field.

Generate :

- A set of well packed and well aligned square cells that is compatible with the given cell sizes and directionality.

Approaching way is using a paticle model to obtain the optimal locations of cell. A proximity based force field is defined between two cells, so that they touch each other along their edges. It is important is important that the directionality is specified over the entire domain so that the packed square cells are well-aligned and natural-looking. In this paper, we assume that square

cells can form a pattern resembling a stone pavement by having a directionality well-aligned

along the domain boundary. Our implementation automatically generates such a directionality by using these equaction,

$$\mathbf{v} = \frac{1}{n}\sum_{i=1}^{n}\frac{\mathbf{s}_i}{d_i^2}$$

Here, we assume that the domain boundary consists of n segments, and s_i denotes the vector of the ith segment of the domain boundary. V is the direction vector at an arbitrary point P inside the domain and di denotes the distance between p and s_i.

Bin Packing Problem and NP

Bin packing problem, objects of different volumes must be packed into a finite number of bins or containers each of volume V in a way that minimizes the number of bins used. In computational complexity theory, it is a combinatorial NP-hard problem. The decision problem (deciding if a certain number of bins is optimal) is NP-complete.

So much variety; texture packing, linear packing and others cases. These can be seen as a special case of the cutting stock problem. When the number go to 1 and each has characterized it known as knapsack problem.

Bin Packing problem has an NP-hard computational complexity, it's a solution to very large scale of problem can be solved with algorithms.

The first algorithm, fast but not give a maximal solution, requires NlogN time. So the algorithm can be much more useful after sorted and decreasing the algorithm.

Analysis of approximate algorithm

The ***best fit decreasing*** and ***first fit decreasing*** strategies are among the simplest heuristic algorithms for solving the bin packing problem. They have been shown to use no more than 11/9 OPT+1 bins (where OPT is the number of bins given by the optimal solution). The simpler of these, the ***First Fit Decreasing*** (FFD) strategy, operates by first sorting the items to be inserted in decreasing order by their sizes, and then inserting each item into the first bin in the list with sufficient remaining space. Sometimes, however, one does not have the option to sort the input, for example, when faced with an online bin packing problem. In 2007, it was proven that the bound 11/9 OPT+6/9 for FFD is tight. MFFD (a variant of FFD) uses no more than 71/60 OPT+1 bins (i.e. bounded by about 1.18 OPT, compared to about 1.22 OPT for FFD). In 2013, Sgall and Dósa gave a tight upper bound for the first-fit (FF) strategy, showing that it never needs more than 17/10 OPT bins for any input.

It is NP-hard to distinguish whether OPT is 2 or 3, thus for all $\varepsilon>0$, bin packing is hard to approximate within $3/2-\varepsilon$. (If such an approximation exists, one could determine whether n non-negative integers can be partitioned into two sets with the same sum in polynomial time. However, this problem is known to be NP-hard.) Consequently, the bin packing problem does not have a polynomial-time approximation scheme (PTAS) unless P = NP. On the other hand, for any $0 < \varepsilon \leq 1$, it is possible to find a solution using at most $(1+\varepsilon)$ OPT+1 bins in polynomial time. This approximation type is known as asymptotic PTAS.

Purpose of Report

In our report, we try to f**ind a solutions (algorithm)** to the texture packing problem and **compare their time complexity and space complexity** so that we can have **a deeper understanding of algorithm** we learned and we can **understand the advantages and shortcomings of these algorithms** which can make us to find a more appropriate algorithm when we meet some problems in our future works and researches.

Chapter 2: Data Structure / Algorithm Specification

Description (pseudo-code preferred) of all the algorithms involved for solving the problem, including specifications of main data structures. Or, if you are to introduce a new data structure and its related operations, do it in this chapter.

1. Our Approximation Algorithm

This problem of the texture packing is actually called the **2-Dimensional bin packing** problem. And the 2-dimensional bin packing problem is a **N-P hard problem,** so it can't be perfect solved in the polynomial time. But there are many approximation algorithms that can solve this problem in polynomial time, but they not find the approximation answer of the problem but not the most optimal answer of this problem. So in our research project we find a approximation algorithm to solve this problem. Our algorithm is a kind of **greedy algorithm**. There are many steps in our algorithm. **In each step we define some rules to find the best answer of the local problem**. **And step by step we finally can find the approximate optimal answer.** This answer is the local optimal answer but not the optimal solution of the global scale. However it can be not far away from the truly optimal solution of this problem.

Now, let me describe our approximation algorithm. When we load rectangular to the given texture, there will be a contour of the rectangular in the given strip. So if we want to load the rectangular and make them have the smallest value of the height, we must insert each rectangular to the lowest **concave** of the contour of rectangular which have been loaded as figure2.1 shows.

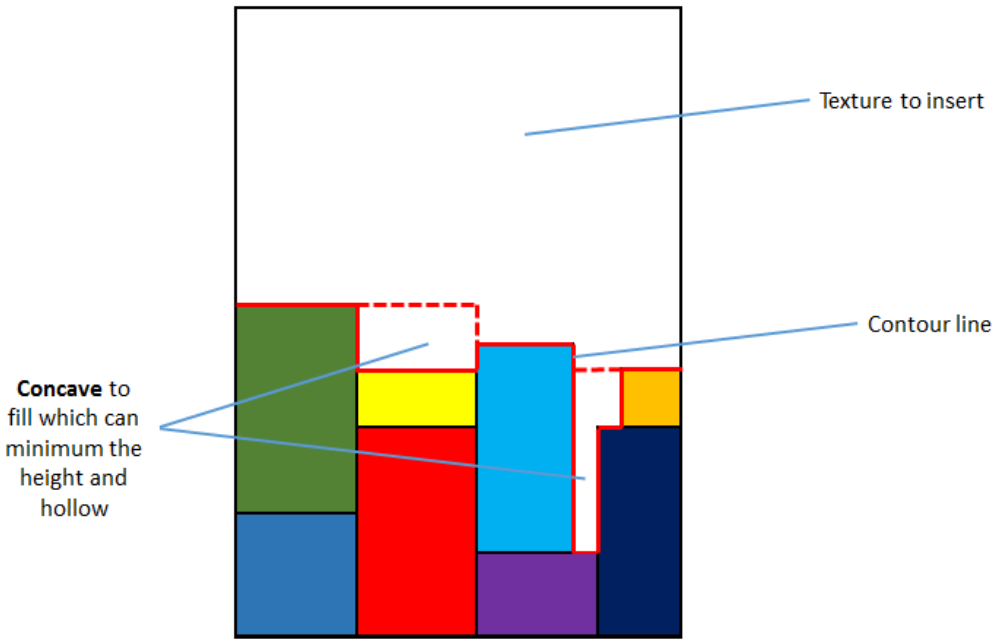
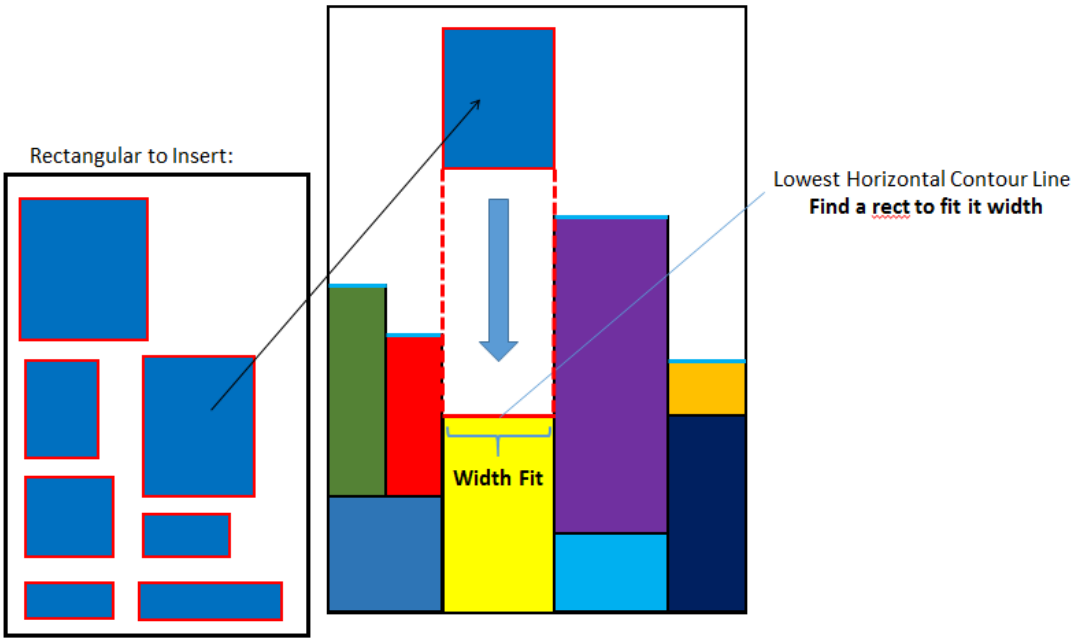


Figure 2.1

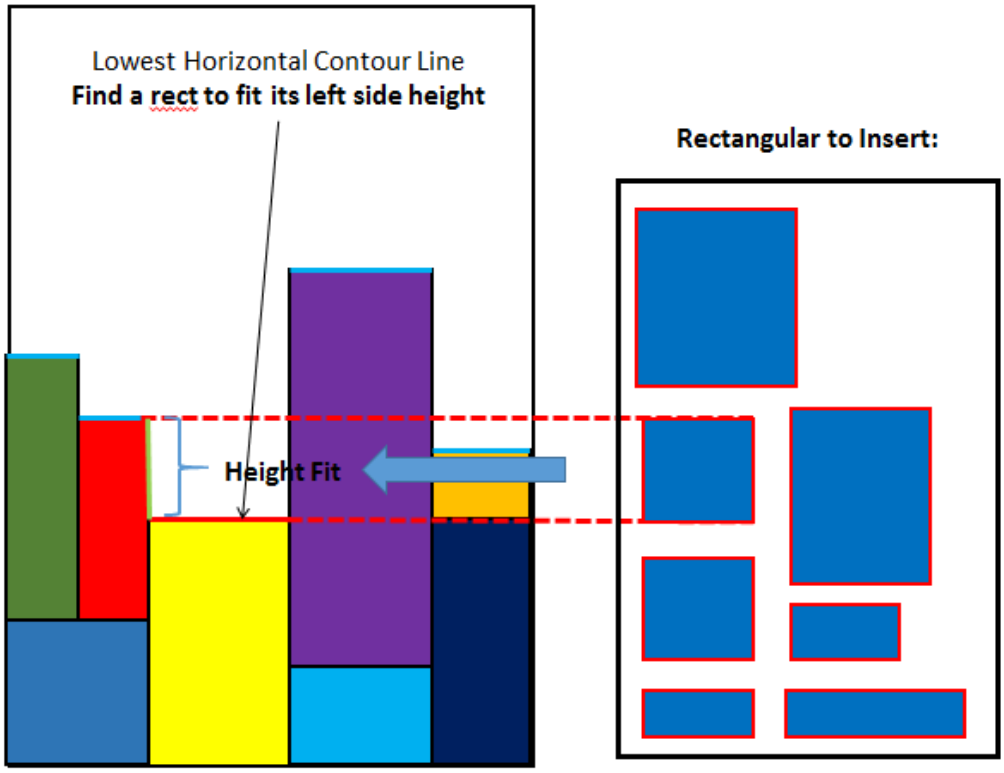
Using this idea, we make a list to store the horizontal lines of contour. **And in each step we find the lowest horizontal line of the contour and insert a most fit rectangular to the**

space upper the lowest contour line. To find the most fit rectangular that can insert into the space that surrounded by the lowest horizontal line of the contour and vertical line of contour in the left and right side of the lowest horizontal line of the contour, we draw 4 rules(the 4 rule is list in the order of priority):

Rule1: Width fit first: among all rectangular we load the rectangular whose length or width is the same as the width of the lowest horizontal contour line first. If there are many rectangular which can fit this rule, we select one which has the largest area. As figure shows below.



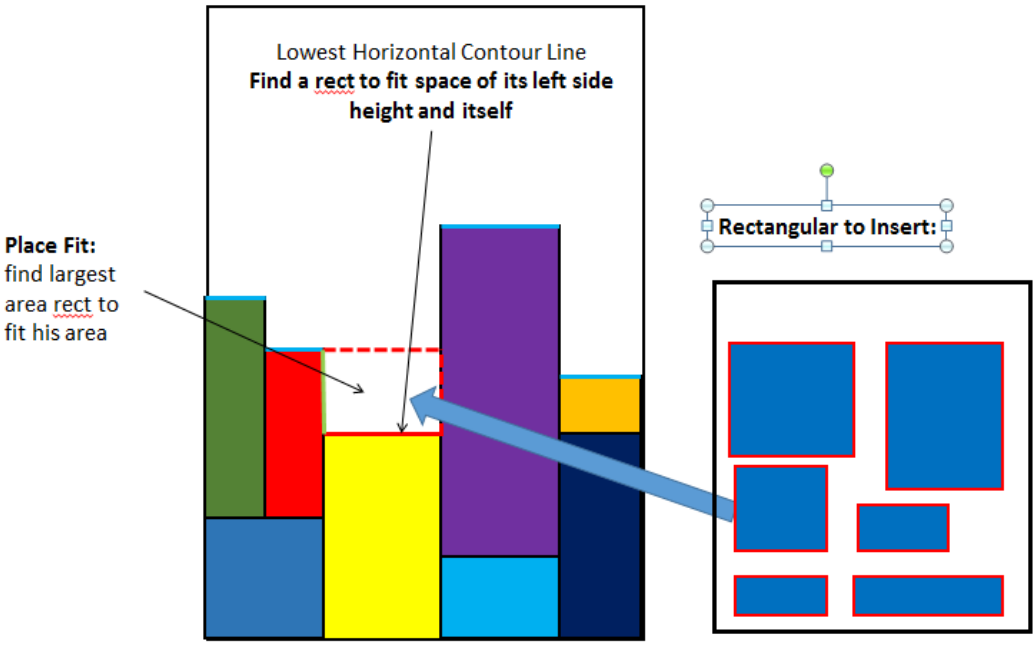
The purpose of this rule is to make rectangular fill the concave of the contour and make the hole of the strip less.



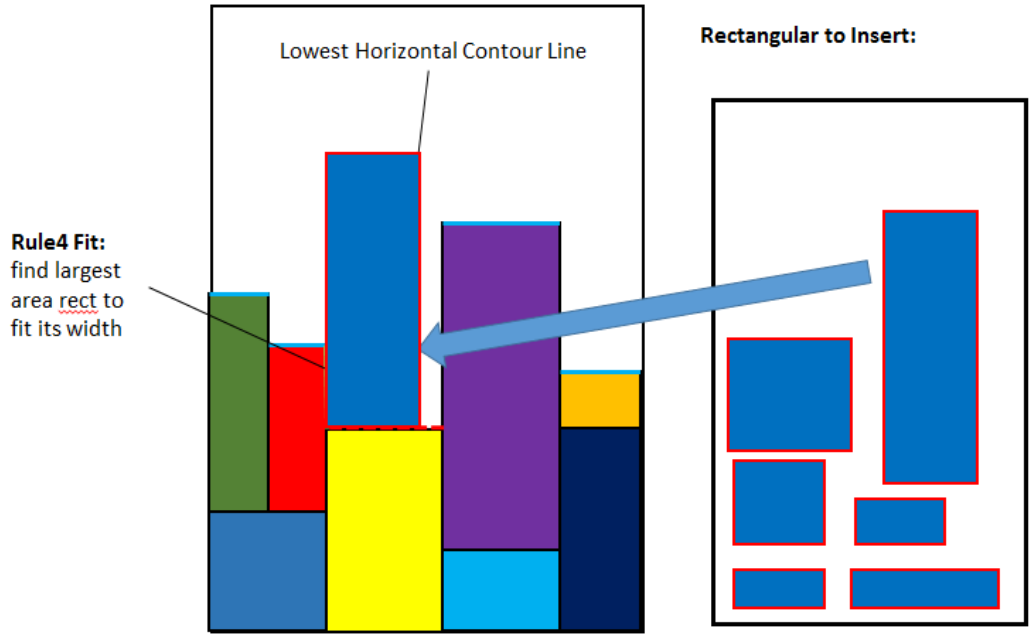
Rule2: Height fit first: among all rectangular we load the rectangular whose length or width is the same as the height of vertical line contour of the left side of the lowest horizontal contour line first. If there are many rectangular which can fit this rule, we select one which has the largest area. As figure shows below.

The purpose of this rule is to make rectangular fill the concave of the contour and make the hole of the strip less.

Rule3: Can be placed first: among all rectangular we load the rectangular whose length or width is less than the height of vertical line contour of the left side of the lowest horizontal contour line and the width of the lowest horizontal contour line first. If there are many rectangular which can fit this rule, we select one which has the largest area. As figure shows below.



Rule4: Larger area first: if all 3 rule we draw above is not play a role, we use the last rule that we find the rectangular whose length or width is less than the width of the lowest horizontal contour line and has the largest area. As figure shows below.



If all rules above can't find a fit rectangular that means the space of lowest contour line is too small for all rectangular, we level up this contour line and we choose another contour line to find a fit rectangular until we can find a rectangular to insert to the texture strip.

It's a kind of greedy algorithm. In each step, we find a most fit rectangular and in N steps we can insert all rectangular into the strip and find a approximate minimum height of all rectangular.

pseudo-code:

```
1./*Initialization*/
    InitContours ( C) ;          //initialization of the contour list
    Height_packing = 0 ;         // initialization of the packing height
for ( i = 0 ; i < N ; i + + )
    if ( rect[i]. x > - 1)  continue ;    // if the rectangular has been insert ignore
    2. /*Find the lowest contour line*/
    Clowest=FindLowestContuor( contuorlist C );   //find the lowest contour line
    3./*Find most fit rectangular*/
    .FitRectID=Rule1 (rect,ClowestContour) ;        //rule 1
    if (FitRectID > =0) goto 3) ;                   // if find goto step4
    FitRectID =Rule2 (rect,ClowestContour);         //rule 2
    if (FitRectID > =0) goto 3) ;                   // if find goto step4
    FitRectID =Rule3 (rect,ClowestContour);         //rule 3
    if (FitRectID > 0) goto 3) ;                    // if find goto step4
    FitRectID =Rule4 (rect,ClowestContour);         //rule 4
    if (FitRectID > =0) goto 3) ;                   // if find goto step4

    4./*Insert rectangular and Update the contour */
    Rect[FitRectID].x=CLowst.x;
    Rect[FitRectID].y=CLowst.y;
    UpdateContour(Contour Clowest,ContourList C);
    5./*Update Height*/
    if(Rect[FitRectID].rotation=0)
        Heigth_packing=max{Height_packing , Clowest.y+rect[FitRectID].width};
    if(Rect[FitRectID].rotation=90)
        Heigth_packing=max{Height_packing , Clowest.y+rect[FitRectID].length};
```

2. Date Structure

Data structure we used is **the linked list** which is to store the list of the horizontal contour line and some **array** to store the information of the rectangular. Because these data structure is too simple and common, we do not describe them in detail.

Chapter 3: Testing Results

Input: the width of box, number of rectangle, the height and width of rectangles (**When the number of rectangles is larger than 9, we get the data by reading files, the files are available in the folder, and the data can be seen at the end of this document.**) For the number of rectangular above 300, we use random number to generate random length and width of different rectangular.

Testing result

	Iuput	**purpose**	**output**	**result**
1	1 1 1 1	simplest sample	1	*Optimal*
2	5 2 2 3 3 2	Precise sample	3	*Optimal*
3	40(width) T10.txt(10 rectangles)	a small sample	40	*optimal*
4	50 T100.txt	A larger sample	166	*16 more than the optimal height*
5	80 T70.txt	a worse example	158	*58 more than the optimal height*
6	70 T300.txt	The largest known example we new	151	*1 more than the optimal height*
7	100(width) 10 random rectangles	Testing the result for random case	135|0.079s	*Run successfully*
8	100 10000 random rectangles	Large amount of data	253110|25.32s	*Run successfully*

Testing result of approximation ratio

N	$H_{optimal}$	$H_{our\ output}$	Time(s)	Given Width	Ratio
10	40	40	0	40	1
20	50	52	0.18	30	1.04
30	50	61	0.32	30	1.22
40	80	134	0.41	80	1.675
50	100	113	0.52	100	1.13
60	100	104	0.61	50	1.04
70	100	158	0.742	80	1.58
80	80	85	0.828	100	1.06
100	150	166	0.842	50	1.106
200	150	172	1.141	70	1.15
300	150	151	1.331	70	1.01

Approximation Ratio for N<300 is less than 2.

Comments

According to the texting results, our algorithm get a minimum height which is close to the optimal height in most circumstances, and it didn't take unacceptable time to solve a big problem concerning 10000 random rectangles. So this algorithm is practicable and useful when solving texture packing program.

Bonus: comparison with a known algorithm

GA+BLF is a meta-heuristic algorithm raised by E. Hopper and B.C.H. Turton in 2001.
GA: Genetic algorithms, BLF: the BL-Fill heuristic came from the BL algorithm.
GA+BLF is one of the methods that yielded the best results according to the author's test. So we choose it to compare our algorithm with.

Compare table

problem	number	optimal height	GA+BLF	our result	
1	10	40	40	40	**0**
2	20	50	51	52	**-1**
2	30	50	52	61	-9
4	40	80	83	134	-51
5	50	100	106	113	**-7**
6	60	100	103	104	**-1**
7	70	100	106	158	-52
8	80	80	85	85	**0**

9	100	150	155	166	**-11**
10	200	150	154	172	-18
11	300	150	155	151	**4**
12	500	300	313	303	**10**

Analysis

According to the test table, facing with the same problem, our algorithm is not as good as GA+BLF, however, our algorithm did not lose a lot (which means that the discount is no more than 10% of the optimal height) in 67% of problems. We think this comparing result is acceptable. And it tells us that there is large space for us to improve our algorithm to get better results.

It is a pity that we get data about GA+BLF from Hopper's paper, so it is unfair to compare the execution time used since the computing capacity of our computers is higher than theirs. But the execution time is still worth mentioning here, because the running of GA+BLF is so slow that it can't run if the rectangles are more than 500, so we can't make a further comparison in the heights we get.

Chapter 4: Analysis and Comments

Theoretical time and space complexity:

1. Time complexity

In each step we insert one rectangular and in each step we scan the all N rectangular to find the most fit rectangular to insert into the large texture. And that we insert N rectangular in the large texture means we use N steps to insert all rectangular. So, the worst time our algorithm we used is N*N, that is N^2.

So we can draw conclusion that the **worst time complexity is O(N^2).**

2. Space complexity

We use an Array to store the information of the rectangular and a linked list to store the information of the contour. Because the number of the rectangular and the contour horizontal line is not more than N, we can draw the conclusion that th**e space complexity of our algorithm is O(N).**

Actual testing of the time complexity:

Testing run time table

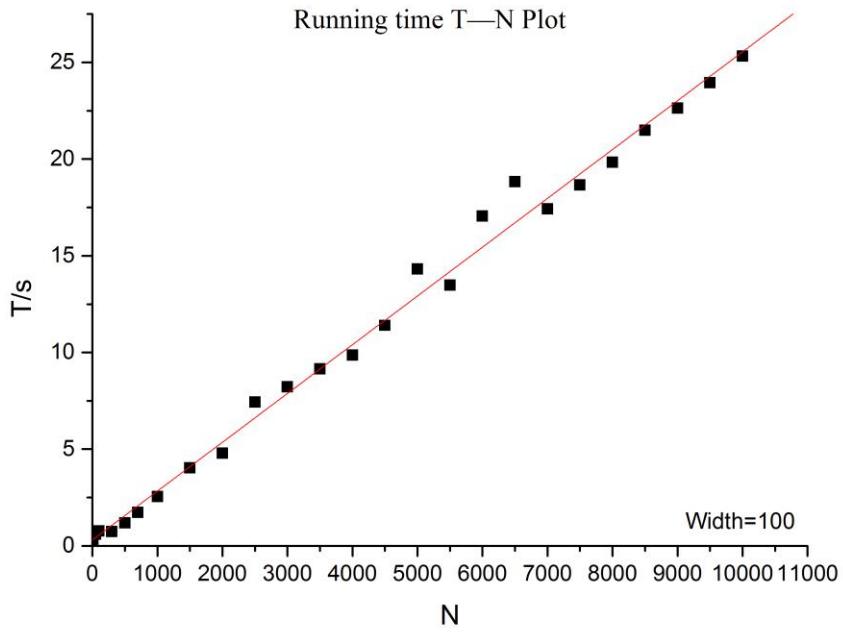
Set Texture Width=100

Input Size—N	Running Time(s)
10	0.088
50	0.603
100	0.767
300	0.74

500	1.181
700	1.729
1000	2.544
1500	4.027
2000	4.785
2500	7.427
3000	8.222
3500	9.141
4000	9.862
4500	11.404
5000	14.318
5500	13.476
6000	17.047
6500	18.833
7000	17.424
7500	18.664
8000	19.841
8500	21.483
9000	22.642
9500	23.948
10000	25.32

Plot the run times vs. input sizes for illustration:

Here is our testing result: you can see that all time complexity is same with our analysis.



Approximation Algorithm T-N

Analysis

From the charts of actual time used, we can see that the time which the algorithm we used is obviously O(N). It is not the $O(N^2)$ and it is faster than the theoretical time. The reason why it is O(N) ,not $O(N^2)$, we don't know. Maybe the average time complexity of our algorithm is faster than $O(N^2)$ and the actual time used is represent the average time complexity.

References

[1] E. Hopper, B.C.H. Turton, "An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem", European Journal of Operational Research 128 (2001) 34-57,2001

[2] J iang Xingbo , L üXiaoqing, Liu Chengcheng, Li Monan, "A Dynamic2Fit Heuristic Algorithm for the Rectangular Strip Packing Problem", Journal of Computer Research and Development, 2009

Author List

Specify ***who did what*** to show that particular contributors deserve to have their names printed in the cover page of your report.

Programming: Write the program **Wan Yuan & Min Sitao**

Testing: Provide a set of test cases to fill in a test report. **Chen Siya**

Documentation: a complete report. **Wan Yuan & Min Sitao**

Declaration

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

Signatures

闵思韬

万园

陈思雅

2016.5.15