Report for Advanced Data Structure and Algorithm Analysis

Texture Packing

Tinghao Xie[1]
Haoran Lin
Zihao Zhao

Zhejiang University

May 15th, 2020

[1]In this project, Tinghao Xie's work includes: proposing the idea of combining genetic algorithm with traditional ones; implementation of reverse-fit algorithm and part of genetic algorithm; implementation of random rectangle division program with a given height as input for test; part of the analysis job; documenting this paper.

1 Introduction

Cutting and packing problems are encountered in many industries, with different industries incorporating different constraints and objectives. A basic form of such problems is well known as *Bin Packing*, where a set of 1D items with different volumes are to be pack into a finite number of containers.

There are so many variations of *Bin Packing* problem, and our work is concerned with a 2D rectangular packing problem. The problem, also known as *Texture Packing*, originates from computer graphics, where packing different rectangle textures into a larger rectangle image is vital for rendering. In this problem, the task is to pack a collection of 2D items into a rectangular container while minimizing the used height of container. The packing process has to ensure that there is no overlap between the items. The specific problem we discuss has the following characteristics:

- All items are of rectangular shape
- All items can be rotated by 90°
- The container has a given width

A direct look of the *Texture Packing* problem is given below.

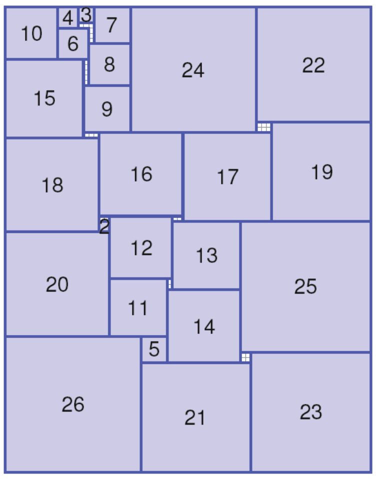


Figure 1: An example for *Texture Packing* problem

We are going to discuss several algorithms solving the problem in this report. In Chapter 2, these algorithms are specified. Besides traditional **Best-fit** algorithm, some variations and improvements of it are introduced. Particularly, **Genetic** algorithm is implemented to the problem as an innovative assist. After that, **Sleator's** algorithm and **Reverse-fit** algorithm are also briefly described. In Chapter 3, both correctness test and performance test are performed. Comparison of those algorithms and some possible factors contributing to approximation ratio are covered in Chapter 4, where further insight of genetic algorithm is also in scope.

2 Algorithm Specification

Since the rotation requirement will make the program quite complicated(it is hard to determine whether an item should be rotated), we discuss our algorithms for strip packing problem which is not allowed to rotate items in section 2.1, 2.2. Then we introduce genetic algorithm in section 2.3 to solve the rotation problem. Also, in section 2.4, we describe some known strip packing algorithms, which will be compared to our methods in the following chapters.

2.1 Best-fit Decreasing-Height

Best-fit Decreasing-Height is a level-oriented algorithm. The base idea of the algorithm is that we can divide the strip into different levels and place the items at the place which has the least remaining width. To be more specific, firstly, the algorithm sorts the items by order of nonincreasing height. Then, starting at the bottom left position, the algorithm places the items next to each other in the strip until the next item will overlap the right border of the strip. At this point, the algorithm scans the levels from bottom to top and places the item in the levels which has the least remaining width. If there is no existing levels satisfying the requirement, that is, the remaining widths of the existing levels are too small to place the current item, a new level is defined at the top of the tallest item in the current level and places the items next to each other in this new level.

For example, consider a strip with maximal width 18. And there are 7 items(the detailed information is in Table1). The solution generated by *Best-fit Decreasing-Height* is in Figure 2.

i	1	2	3	4	5	6	7
w(i)	6	5	4	7	3	4	6
h(i)	8	7	6	5	4	3	2

Table 1: Items Information

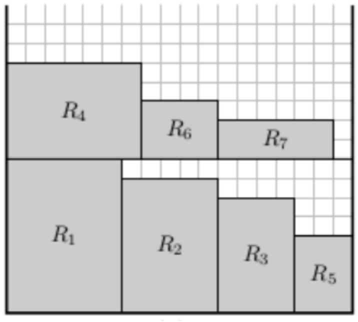


Figure 2: The solution of packing items in Table 1 into the strip(width is 18)

Algorithm 1 Best-fit Decreasing-Height

```
function BFDH($Items, MaxWidth$)
    $TotalHeight \leftarrow 0$
    $Levels \leftarrow empty$
    sort $Items$ by order of nonincreasing height
    for item in Items do
        level $\leftarrow$ the level in the $Levels$ with least remaining width enough for $item$
        if there exists such level then
            update the level: $level.remainWidth \leftarrow level.remainWidth - items.width$
        else
            new a level $newLevel$, and place the item in this level
            $newLevel.remainWidth \leftarrow MaxWidth - items.width$
            add $newLevel$ into $Levels$
            $TotalHeight \leftarrow TotalHeight + item.height$
        end if
    end for
    return $TotalHeight$
end function
```

- **Time Complexity**: The sort procedure cost $O(N \log N)$. And note that we can use BST(e.g. red-black tree) to find the least remaining width enough for items in the Line 6 in Algorithm 1. That is, the worst cost of Line 6 is $O(\log N)$. Therefore, the worst and average time complexity for Algorithm 1 is $O(N \log N)$.
- **Approximation Ratio**: In Algorithm 1, we sort the items by order of nonincreasing height. That is, the height will not place restrictions on the following step. Therefore, the algorithm works like best-fit for one-dimension bin packing. And we have learned that the approximation rate of best-fit for one-dimension bin packing is 1.7. Similarly, Coffman[1] proved that

  $$BFDH \leq 1.7OPT + h_{max}, \tag{1}$$

  where OPT is the optimal height and h_{max} is maximal height in these items. Since the proof is quite complex, please refer to the reference document we list if you are interested in the detailed proof. Also, note that $h_{max} \leq OPT$, the worst approximation ratio is 2.7.(Note that the approximation rate holds true only when the items are not allowed to rotate.)

2.2 Best-fit Decreasing-Height:Improved Version

Note that *Best-fit Decreasing-Height* only use the space next to rightmost items in each level. To utilize the space above the existing items in each level, we propose the improved version of *Best-fit Decreasing-Height*.

In general, this algorithm is also level-oriented algorithm. What makes it different from traditional *Best-fit Decreasing-Height* is the treatment of space in each level. Besides using the space next to rightmost items in each level, we also utilize the space between the top of existing items and the top of the level. To be more detail, we define a rectangular area, named *box*, associated with a item by the top line of the item, the top line of the level, the left line of the item and the right line of the item. Also, we consider the space next to rightmost items in each level as *box* too. For example, there are 7 boxes in Figure 3. And Figure4 illustrates the insertion of a new item (the red one) into a box(Box0), resulting in generation of Box1 and Box2.

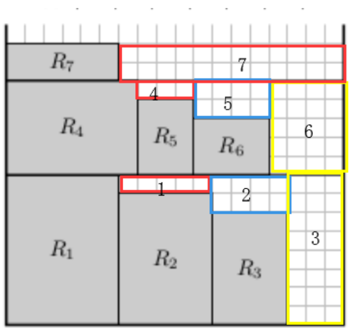
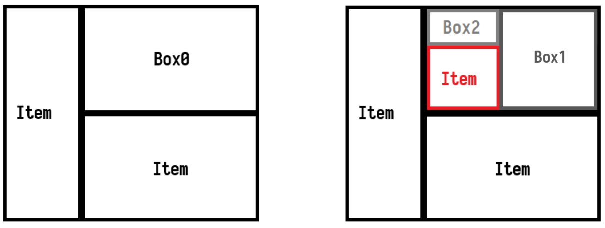


Figure 3: The boxes in the strip



(a) Before insertion (b) After insertion

Figure 4: Insertion of a new item (the red one)

For the detailed algorithm, this algorithm works like *Best-fit Decreasing-Height*. But when searching the "best" position for placing an item, this algorithm searches all the existing boxes and pick the fitted box whose width is least. This strategy usually results in better space utilization.

Algorithm 2 Best-fit Decreasing-Height : Improved Version

```
1: function BFDH-V2($Items, MaxWidth$)
2:     $TotalHeight \leftarrow 0$
3:     $Boxes \leftarrow empty$
4:     sort $Items$ by order of nonincreasing height
5:     for item in Items do
6:         box ← the box in the $Boxes$ whose space is enough for $item$ and has least width
7:         if there exists such box then
8:             place the item in this box
9:             new two boxes : $box1$ and $box2$
10:            $box1.width \leftarrow box.width - item.width$
11:            $box1.height \leftarrow box.height$
12:            $box2.width \leftarrow item.width$
13:            $box2.height \leftarrow box.height - item.height$
14:            erase $box$ from $Boxes$ and insert $box1$ and $box2$ into $Boxes$
15:        else
16:            new a box $newBox$, and place the item in this box
17:            $newBox.width \leftarrow MaxWidth - item.width$
18:            $newBox.height \leftarrow item.height$
19:            add $newBox$ into $Boxes$
20:            $TotalHeight \leftarrow TotalHeight + item.height$
21:        end if
22:    end for
23:    return $TotalHeight$
24: end function
```

- **Time Complexity**: The sort procedure cost $O(N \log N)$. Also, there are at most $N+1$ boxes in Algorithm 2 and we need to traverse all the existing boxes in Line 6 every time. Therefore, the average the worst time complexity for Algorithm 2 is $O(N^2)$.

- **Approximation Ratio**: Note that the worst condition for this improved version is the same as original BFDH. Therefore, they shared the same approximation ratio. That is,

$$BFDHv2 \leq 1.7OPT + h_{max} \tag{2}$$

  And the worst approximation ratio is 2.7.(Note that the approximation rate holds true only when the items are not allowed to rotate.)

2.3 Genetic Algorithm

In previous section, we assume that all the items cannot be rotated. But in this section, we use genetic algorithm to make the rotating decision and apply all the existing algorithms to these rotated items.

In general, genetic algorithm emulates evolution. The base idea is to generate high-quality chromosomes based on the previous generation and operators such as mutation, crossover and selection. And it generates chromosomes iteratively to get the approximate optimal solution.

2.3.1 Encoding

We use an array of bits to represent whether an item should be rotated. Formally, denote the items as $\mathcal{I} = \{item_1, item_2, \cdots, item_n\}$ and the bits array as $Chromosome = \{bit_1, bit_2, \cdots, bit_n\}$. We rotate

$item_i$ if and only if it can be rotated and $bit_i = 1$. Therefore, we can define a rotated function, named *Rotate*, to rotate items according to chromosome.

$$Rotate : \mathcal{I} \times Chromosome \to \mathcal{I}$$

And we denote f as the traditional strip packing algorithms(like BFDHv2), which takes a set of items as input and output the packing height. Then we can use the encoding to get the packing height of $Items$ as $f(Rotate(Items, Chromosome))$.

Therefore, we know that every encoding(chromosome) corresponds to a height. Since the encoding space is quite large(in face 2^N), it is impossible to brute all the possible encodings when N is large. Therefore, genetic algorithm, a heuristics method, is designed to search the encoding space based on several rules to find the approximate optimal height.

2.3.2 Initialization

The genetic algorithm should begin with several chromosomes(denote S as the size of initial number of chromosomes). And these chromosomes should be initialized randomly.

2.3.3 Evaluate Fitness

In previous subsection, we know that every chromosome corresponds to a height. That is, the chromosome with lower height is better. Then we can use these "good" chromosomes to generate new chromosomes, which is expected to be better. However, since the heights differ largely according to the input, they are not suitable for quantification. Therefore, we normalize the height to a fixed region. Denote the set of chromosomes as $\mathcal{C} = \{c_1, c_2, \cdots, c_S\}$, and the corresponding height as $\mathcal{H} = \{h_1, h_2, \cdots, h_S\}$. Then, the fitness of c_i is defined as

$$Fitness(c_i) = c \cdot \frac{\max \mathcal{H} - h_i}{\max \mathcal{H} - \min \mathcal{H}}. \tag{3}$$

Therefore, the fitness values of all the chromosomes are in $[0, c]$. And the higher fitness implies the better chromosome.

2.3.4 Secletion

Note that genetic algorithm follows the Darwinian evolution theory. That is, the higher fitness, the more possible a chromosome can survive and produce offspring. Therefore, we define a selection operator to select a chromosome from the population C. The selected probability is proportional to its fitness. That is

$$Pr(c_i \text{ is selected}) = \frac{fitness(c_i)}{\sum_{j=0}^{S} fitness(c_j)} \tag{4}$$

2.3.5 Reproduce

In general, we want that the chromosome with outstanding performance to survive regardless of selection. Therefore, we copy the top $\frac{1}{16}$ of these generation into next generation.

2.3.6 Crossover

The crossover operator, which server as the most important operator in genetic algorithm, simulates natural gene recombination. It happens with probability about $80\% - 95\%$.It takes two chromosomes as input, crossovers partial genes, and produce new chromosomes. Firstly, the crossover operator will choose a crossover position randomly. Then it exchanges the genes before or after the chosen position to generate new chromosomes.

For example, there are two chromosomes,c_1 = 0101 1010,c_2 = 1101 0001. And the crossover operator chooses to exchange the genes after 4-th position. Then two chromosomes c_3 = 0101 0001,c_4 = 1101 1010 are generated.

2.3.7 Mutate

The mutate operator emulates the mutation of genes. It helps the genetic algorithm to expand the searching space and avoid falling into local minimal. It choose several postions in the chromosome and filp it. And the mutate operator is applied rarely(normally, the probability is less than 1%).

For example, $c_1 = 0101\ 1010$ is chosen to be mutated. And the mutate operator chooses to filp the 2-th and 8-th bits. Then the new chromosome generated by mutate operator is $c_1' = 0001\ 1011$

2.3.8 Summary of Genetic Algorithm

Algorithm 3 Genetic algorithm

```
 1: function GA($strip - packing - algorithm$)
 2:     initialize the set of chromosomes(denoted as $Population$) randomly
 3:     $NewPopulation \leftarrow empty$
 4:     for k ← 1 to maxIter do
 5:         evaluate the fitness of chromosomes in $Population$ based on strip-packing-algorithm
 6:         copy the top $\frac{1}{16}$ of the chromosomes in $Population$ to $NewPopulation$
 7:         while The size of $NewPopulation$ is not enough do
 8:             $c_1, c_2 \leftarrow Select()$          ▷ select two chromosomes based on select operater
 9:             if rand() < CrossoverProbability then
10:                 $c_3, c_4 = Crossover(c_1, c_2)$          ▷ crossover based on certain probability
11:             else
12:                 $c_3 = Mutate(c_1), c_4 = Mutate(c_2)$          ▷ mutate based on certain probability
13:             end if
14:             add $c_3, c_4$ to $NewPopulation$
15:         end while
16:         $Population \leftarrow NewPopulation$
17:         $NewPopulation \leftarrow empty$
18:     end for
19:     evaluate the fitness of chromosomes in $Population$ based on strip-packing-algorithm
20:     return The best height recorded during evaluating
21: end function
```

- **Time Complexity**: The time complexity of genetic algorithm corresponds to two factor: the parameter of genetic algorithm(populatoin size S and iteration number $maxIter$) and the time complexity of strip packing algorithm(denoted $O(f(N))$) it based on. Then the time complexity of genetic algorithm is $O(S \cdot maxIter \cdot f(N))$.

- **Approximation Ratio**: The approximation ratio of genetic algorithm also depends on the strip packing algorithm it based on. In the worst case, the approximation ratio of genetic algorithm is the same as that of the based algorithm. However, normally, genetic algorithm will give a better approximation ratio than the algorithm it based.

2.4 Other known algorithms for strip packing

In this subsectoin, we introduce some known algorithms for strip packing which are also implemented in our source code. These algorithms will be compared to our algorithm in the following chapters. We will describe the basic idea of these algorithm, but not discuss implementation in detail(see our code or the reference we list if you are interested).

2.4.1 Sleator's Algorithm

Given a set of items and strip width W, the Sleator's Algorithm works as follows:

1. Place the items whose width larger than $W/2$ in the bottom of the strip(the placing order does not matter). Then the following steps will take place above these level.
2. Sort all the remaining items in nonincreasing order of height. The items will be placed in this order.
3. Place the items until no item is left or the next one does not fit.
4. Draw a vertical line at $W/2$, which cuts the strip into two equal halves. The following steps will place the remaining items in either the left half or the right half.
5. Let h_l be the highest point covered by any item in the left half and h_r the corresponding point on the right half. Choose the half which is lower and place the items on this half until no other item fits. Repeat this step until no item is left.

- **Time Complexity**: The sort procedure cost at most $O(N \log N)$. And the other steps cost at most $O(N)$. Therefore, the total time complexity of the algorithm is $O(N \log N)$.
- **Approximation Ratio**: Sleator[2] proved that the worst approximation ratio was 2.5 and this is a tight bound.(Note that the approximation rate holds true only when the items are not allowed to rotate.)

2.4.2 Reverse-fit Algorithm

Reverse-fit (short as *RF*) algorithm was first described by Schiermeyer[3] in 1994. Some additional notation would be necessary. For any item $i \in \mathcal{I}$, i's lower left corner is denoted by (a_i, c_i) and upper right corner by (b_i, d_i).

Given a set of items $\mathcal{I}$ and a strip of width W, RF algorithm works as follows:

1. Stack all rectangles with width more than $W/2$ on top of each other (in random order) at the bottom of the strip, with a total height as H_0. And other items would be packed above H_0.
2. Sort the remaining items in order of nonincreasing height. Denote the height of the tallest of the remaining items as h_{max}.
3. Construct the **first level**: pack items from left to right with their bottom along the line of height H_0 until there is no more room. Now let h_1 be the height of the tallest unpacked item.
4. Construct the **second reverse level**: pack items from right to left with their top touching $H_0 + h_{max} + h_1$, until no items left or the total width of the items in the second reverse level is at least $W/2$.
   Notice: Pack the items into the two levels due to First-Fit, i.e., placing the items in the first level where they fit and in the second one otherwise.
5. Shift down items in the second reverse level until an item touches some item in the first level. Denote H_1 as the new vertical position of the top of the second level. If no items are left, the algorithm is over with total height of H_1. Let f and s be the right most pair touching items, where f is at the first level and s at the second reverse level. Define $x_r = min(b_f, b_s)$. Actually, the 'touching-line' of f and s is given by $T(f, s) = \{\max(a_f, a_s), \min(b_f, b_s)\}$.
6. If $x_r \geq W/2$, define the third level (or the new 'first level' in the next round) at H_1. Skip step 7.
7. If $x_r < W/2$, then s must be the last rectangle in the second reverse level. Shift (again) down all items in the second reverse level except for s further, until an item touches another item at the first level. Define h_2 as the amount by which the second reverse level is shifted down. Denote the top of all items in the second reverse level except for s as H_2.

a) If $h_2 \leq h(s)$, then shift s to the left until it touches another item (from the first level) or the border of the strip. And the third level (or the new 'first level' in the next round) is defined at H_2.

b) If $h_2 > h(s)$, then define the third level (or the new 'first level' in the next round) at H_2. Then place s left-aligned to the third level, such that it touches an item from the first level or the border of the strip to its left.

8. Now after an execution of step 3 to 7, we'll say that a complete 'round' or 'iteration' is over. And that the third level is already defined, we'll let it be the new 'first level' in the next round, and we would execute step 3 to 7 all over again with the remaining items. Notice that each following level (starting at level three) is defined by a horizontal line through the top of the largest item on the previous level (e.g., the third level is defined by the top of the previous second reverse level H_2). And note that the first item placed in the next level might not touch the border of the strip with their left side, but an item from the first level or the item s.

- **Time Complexity**: The running time can reach $O(N^2)$, since there are at most N levels.
- **Approximation Ratio**: Schiermeyer[3] proved that RF algorithm is a 2-optimal algorithm and thus the worst approximation ratio was 2.(Note that the approximation rate holds true only when the items are not allowed to rotate.)

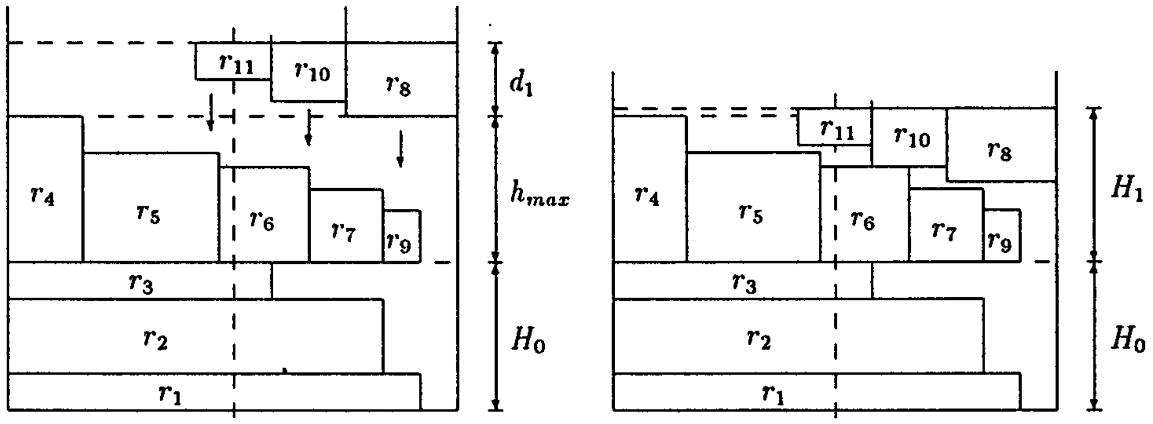


Figure 5: An illustration[3] for RF algorithm

3 Testing Results

In this chapter, we'll demonstrate our tests for both **correctness** and **performance**. *Best-fit Decreasing Height Improved Version* (short as *BFDHv2*) and *Genetic Algorithm - Best-fit Decreasing Height Improved Version* (short as *GA-BFDHv2*) would be focused on.

3.1 Correctness Test

Since no general baseline exists for large scale sets of input in 2D strip packing problem, we'll simply show that our program works correctly for some predictable inputs, while it's also working fine in some extreme cases without a known answer. Furthermore, since *GA-BFDHv2* is based on *BFDHv2* and is actually unpredictable, we'll test for original *BFDH* and *BFDHv2*'s correctness in case 1 and 2. And *GA-BFDHv2* would be tested later in case 4.

- **Case 1** A common case where the original *BFDH* and the improved *BFDHv2* work in the same way

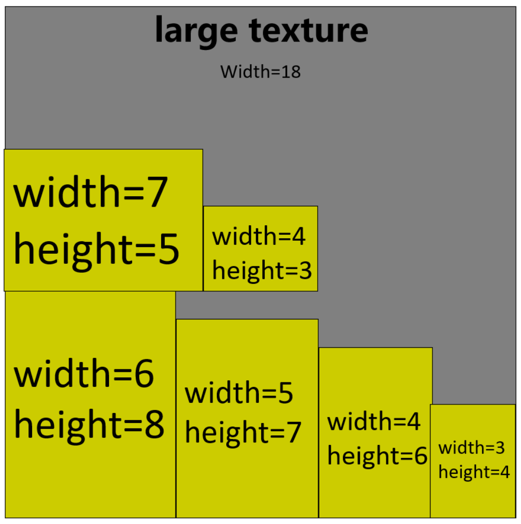
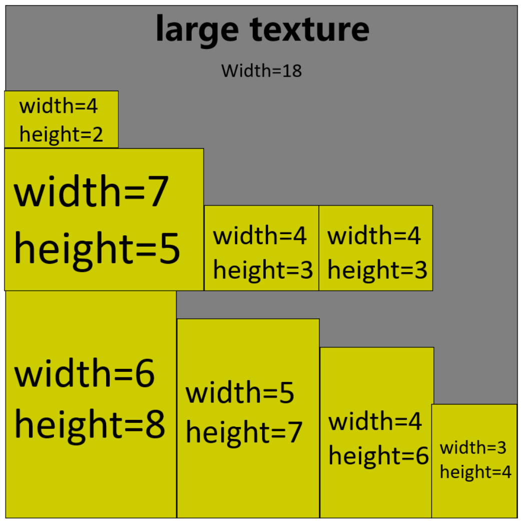


Figure 6: Case 1, total height = 13 for both *BFDH* and *BFDHv2*

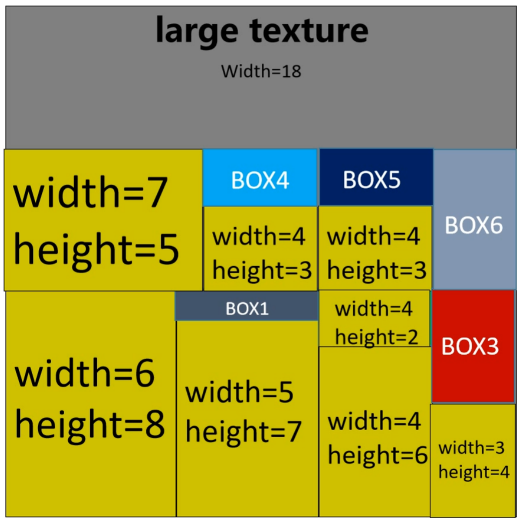
Testing result is as follows:

```
./project5
#Input:
            18  6
            6  8
            5  7
            4  6
            7  5
            3  4
            4  3
#Output:
            BFDH:               13
            BFDHv2:             13
```

- **Case 2** A common case that differentiates the original *BFDH* and the improved *BFDHv2*



(a) *BFDH*, total height = 15



(b) *BFDHv2*, total height = 13

Figure 7: Case 2

Testing result is as follows:

```
./project5
#Input:
        18 8
        6 8
        5 7
        4 6
        7 5
        3 4
        4 3
        4 3
        4 2
#Output:
        BFDH:           15
        BFDHv2:         13
```

- **Case 3** Cannot pack (even after a rotation) due to a large rectangle, say 10 × 10 for a strip with width = 8

  Testing result is as follows:

```
./project5
#Input:
        8 1
        10 10
#Output:
        Cannot Pack!
```

- **Case 4** Large-scale random input

  We use a test input generator **test_gen** to build such large-scale random input with:

$$number\ of\ textures = 1000$$
$$strip\ width = 3000$$
$$optimal\ height = 5000$$

  , which could be obtained at file **Input_case4.txt** together with test input generator's source code.Testing result is as follows:

```
./test_gen 1000 3000 5000 Input_case4.txt
./project5
#Input:
        3000 1000
        135 458
        214 227
        62 63
        174 65
        489 141
        908 1346
        ......
#Output:
        GA-BFDHv2:      5049
        BFDH:           6309
        BFDHv2:         5160
```

For case 1, 2 and 3, our program works correctly as expected. And for case 4, a large-scale random input, our program also gives reasonable output, and thus we'll directly affirm its general correctness.

3.2 Performance Test

To test run time performance of *BFDHv2* and *GA-BFDHv2*, we provide **test.cpp** (instead of **main.cpp**), which works as follow:

```
g++ -O3 test.cpp strip_packing.cpp -o test
./test max_width mini_num maxi_num step GAflag
```

, whose usage is explained in the **README.md** file provided together. Either run time of *BFDHv2* or *GA-BFDHv2* would be measured and eventually written in file **Runtime_performance.txt**.

The run times vs. input sizes(number of textures) tables are shown below:

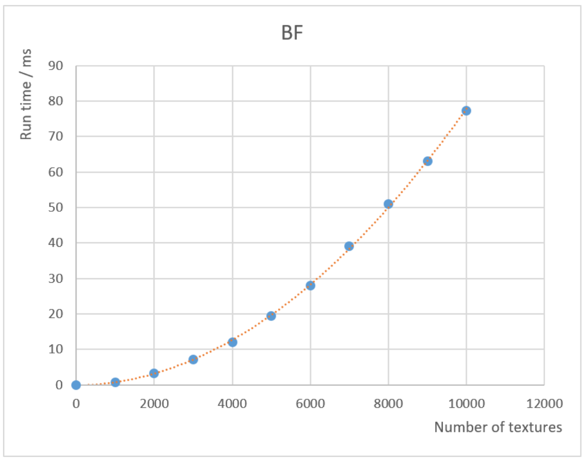
Number of textures	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Run time (ms)	0.80	3.27	7.10	12.00	19.33	28.00	39.00	51.00	63.00	77.33

Table 2: *BFDHv2* Run time

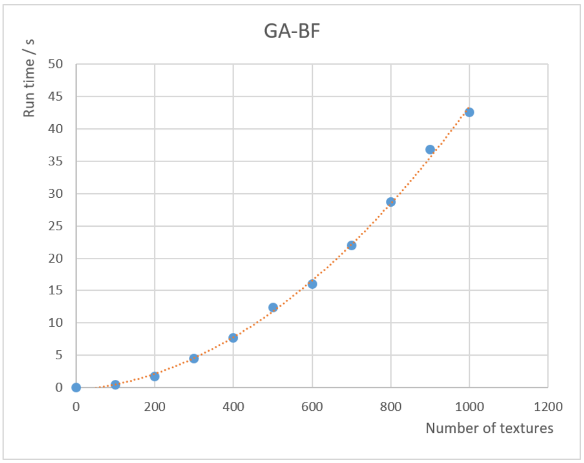
Number of textures	100	200	300	400	500
Run time (s)	0.418	1.697	4.444	7.725	12.398
Number of textures	600	700	800	900	1000
Run time (s)	16.022	21.932	28.700	36.769	42.571

Table 3: *GA-BFDHv2* Run time

And run times vs. input sizes(number of textures) plots are as follow:



(a) run times vs. input sizes for *BFDHv2*



(b) run times vs. input sizes for *GA-BFDHv2*

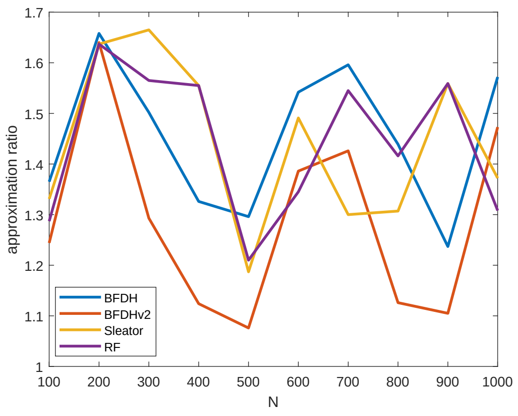
Figure 8: run times vs. input sizes

According to analysis earlier, the average and worst time complexity for *BFDHv2* is $O(N^2)$, and the actual run times fit to this bound perfectly. While *GA-BFDHv2*'s time complexity is $O(S \cdot maxIter \cdot f(N)) = O(N^2)$, where $f(N)$ stands for fucntion of *BFDHv2* here, the actual performance conforms to it as well.

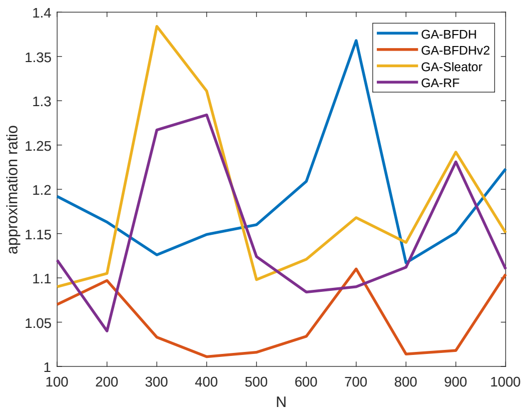
4 Analysis and Comments

4.1 Approximation Comparison

In this section, we will test the approximation ratio under varied numbers of textures for the algorithms mentioned above. In our tests, we fix the strip width(10^4) and the optimal height(10^4). The testing result is shown in the Figure 9. We can see that our *BFDHv2* performs best both among the traditional algorithms(do not allow rotation) and the genetic algorithms version.



(a) traditional algorithms



(b) genetic algorithms

Figure 9: approximation ratio for different algorithms

Also, the genetic algorithm helps those traditional algorithm reach a better approximation ratio. To illustrate the improvement introduced by genetic algorithm, we draw the Figure 10 to compare the *GA-BFDHv2* and *BFDHv2*. Evidently, Figure 10 implies that the genetic algorithm reduce the approximation ratio dramatically.

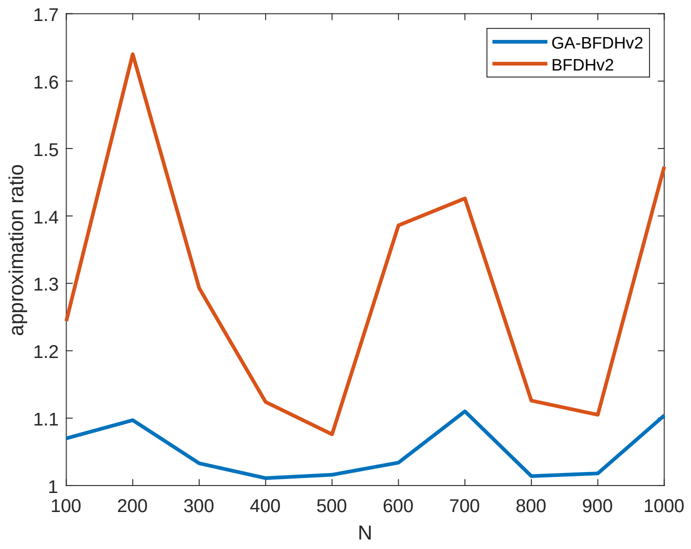


Figure 10: approximation ratio for different algorithms

4.2 Approximation Ratio and the Width of the Strip

In this section, we will focus on the *BFDHv2* and *GA-BFDHv2*, and discuss the relation between approximation ratio and the width of the strip.We use our test program to generator test cases with varied maximal width, while fixing the number of textures($N = 100$) and the optimal height($OPT = 10000$). The test results are shown in the Figure 11.

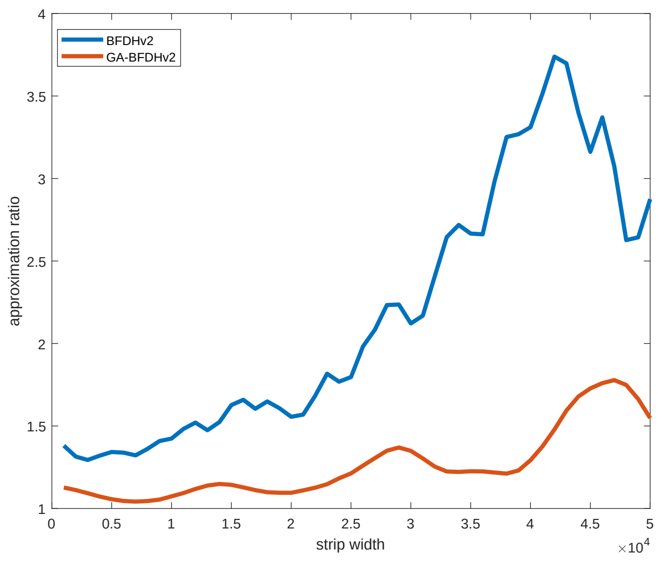


Figure 11: approximation ratio vs. strip width

As we can see, the approximation ratios for both *BFDHv2* and *GA-BFDHv2* increase when the maximal width increases. But the increasing speed of *GA-BFDHv2* is much smooth than the original *BFDHv2*.

We speculate that the phenomenon happens due to the following reasons. Firstly, When the optimal height is relatively low(i.e. the strip width is relative great), any absolute error made by algorithms would cause the approximation rate to increase dramatically. Moreover, when the strip width is relative large, the rotation decision becomes the dominant factor that affect approximation rate. For example, when the optimal height is 10^4, and the strip width is 10^5, there is a texture whose shape is $(10^5 - 1) \times 1$(height×width). If the algorithm do not rotate this item, the approximation rate is at least 10! And since our genetic algorithm would rotate these textures properly, the *GA-BFDHv2* performs much better than *BFDHv2* when strip width is relative high.

4.3 Approximation Ratio and Distribution of Widths and Heights

In this subsection, we will discuss the relations between approximation ratio and distribution of widths and heights.

4.3.1 Traditional Strip Packing Algorithms

We fix the strip width and optimal height so that they are equal. And we generate test cases so that all items shared the same aspect ratio($\min\{\frac{width}{height}, \frac{height}{width}\}$, since the problem allow rotation, we do not distinguish between width-height ratio and height-width ratio). Then we use the traidtonal strip packing algorithms described in Chapter 2 to solve these test cases. The result is shown in Table 4.

aspect ratio	1:1	1:2	1:3	1:4	1:5
BFDH	1.17	1.22	1.23	1.37	1.55
BFDHv2	1.00	1.03	1.05	1.15	1.12
Sleator	1.04	1.13	1.13	1.23	1.26
RF	1.31	1.29	1.32	1.35	1.36

Table 4: Approximation Ratio and Distribution of Widths and Height(traditional)

From Table 4, we can see that, in general, the approximation ratio of all the traditional algorihtms, expect *reverse-fit*, increase when aspect ratio decrease. The possible interpretation for this phenomenon is as follows.

When width and height are more and more unbalanced, the decision of the rotation is becoming more and more significant. However, the traditional algorithms cannot deal with rotation well, which causes the bad performance under low aspect ratio.

Also, note that our algorithm, BFDHv2, does best among all the listed algorithm, and under all aspect ratio.

4.3.2 Genetic Algorithm

When we apply the genetic algorithm to the traditoinal algorithm, the results change dramatically. Expect that the aspect ratio is 1(i.e. the rotation does not matter), the performance of all the algorithms is improved. And the approximation ratio seems to be independent to the aspect ratio when the aspect ratio is greater than 1.

aspect ratio	1:1	1:2	1:3	1:4	1:5
GA-BFDH	1.17	1.02	1.01	1.02	1.03
GA-BFDHv2	1.00	1.00	1.00	1.00	1.00
GA-Sleator	1.03	1.00	1.02	1.02	1.01
GA-RF	1.33	1.09	1.13	1.09	1.11

Table 5: Approximation Ratio and Distribution of Widths and Height(genetic algorithm)

4.4 Go Further into the Genetic Algorithm

4.4.1 The Comparison between Genetic Algorithm and Random Search

To verify that the genetic algorithm is an efficient algorithm instead of a random algorithm, we test the genetic algorithm(based on BFDHv2) and random search(based on BFDHv2) in different textures size. In this test, both the random search and the genetic algorithm search 5000 times.(since the test for this section is really time-consuming, we just test 10 times for every entry in Table 6.)

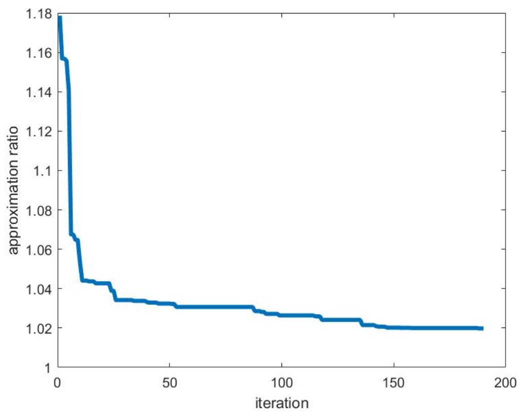
number of items	500	1000	1500	2000	2500	3000
genetic algorithm	1.04	1.03	1.13	1.09	1.02	1.02
random search	1.08	1.05	1.2	1.15	1.05	1.05

Table 6: The Comparison between Genetic Algorithm and Random Search

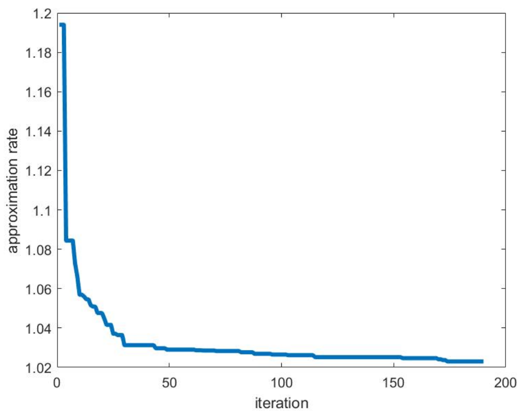
From the Table 6, we can see that the genetic algorithm introduce about 50% decrease in terms of the exceeding height above the optimal height.

4.4.2 The Convergence of Genetic Algorithm

To futher understand how genetic algorithm works, we draw the following figures(Figure 12) to illustrate how the best-height decreases during the iteration.



(a) test-case-1



(b) test-case-2

Figure 12: the convergence of genetic algorithm

From Figure 12, we can find that the most significant decreases happens during the top 50 iterations. Therefore, there is no need to run genetic algorithm for too many iterations. Usually, 50-100 iterations is enough for this task.

5 Conclusion

With this work, we propose the improved version of *Best Fit Decreasing Height* and introduce genetic algorithm to enhance the approximation. Also, the experiment we did shows that our algorithm outdoes many existing strip packing algorithms. Moreover, the further improvement can be done by considering other heuristic approaches. Besides, using heuristic algorithms to pack textures, instead of just deciding rotation, may lead to better performance.

6 Declaration

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

References

[1] Coffman Jr., Edward G.; Garey, M. R.; Johnson, David S.; Tarjan, Robert Endre (1980). ”Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms”. SIAM J. Comput. 9 (4): 808–826.

[2] Sleator, D. A 2.5 Times Optimal Algorithm for Packing in Two Dimensions. Information Processing Letters , 1:37–40, 1980.

[3] Schiermeyer, Ingo. (1994). Reverse-Fit: A 2-Optimal Algorithm for Packing Rectangles.. LNCS. 855. 290-299. 10.1007/BFb0049416.