Texture Packing

Authors: XXX

Date: 2022/05/18

Chapter 1: Introduction

Texture Packing is to pack multiple rectangle shaped textures into one large texture. When it comes to the bin packing problem, the only restriction is the capacity of the bin, and the volume of the loaded item is the only property.

But for this problem, that is given a width restriction of the resulting texture, you need to pack all the rectangle and return the minimum height of the resulting texture, which is known as **strip packing problem**. We need to consider the width and height of the outermost and largest texture, and the loaded rectangle also has two properties, width and height, similar to the two-dimensional expansion problem of the previous question.

And we consider to use First Fit 、Next Fit、Best Fit as well as other approximation algorithms to solve this problem.

Chapter 2: Data Structure / Algorithm Specification

This problem is actually a strip packing problem, so we'd like to denote texture as rect when specifying algorithms. What we have to do is to put a lot of rects into a strip with given width and try to get as small height as we can in a short time. We have implemented several algorithms. Among them, First Fit、Next Fit、Bottom Left algorithms has been proposed by someone. And Best Fit is extended from one dimension bin pack, Reverse Fit is a simplified version of a well-known algorithm Reverse Fit.

Data Structure

We have only used STL vector in our program, which is implemented in a way of dynamic array.

Algorithm

Level Oriented Algorithm

First-Fit Decreasing Height (FFDH) algorithm

FFDH packs the next item R (in non-increasing height) on the first level where R fits. If no level can accommodate R, a new level is created. As for the pseudocode, we check whether the rect_list (which involve the width and height of all the input rectangle) is empty. If not, we traverse all the rectangle and insert them one by one following the rules started earlier, that is, if can insert in this level, insert and renew the used width, otherwise, check the next level. If no exist level can insert this rectangle, create a new level and insert it as well as renewing the current_height.

```
long long FFDH(int width, Rect[] rect_list){ /* First-Fit Decreasing Height */
    if (IsEmpty(rect_list)) return 0;
    SortByHeight(rect_list);
    int[] width_taken_list; /* the used width of every exist level */
    long long int current_height := 0;
    for (rect in rect_list) {
        if (rect.width > width) return -1;
        for (level in levels) {
            if (can store in this level) {
                Update Width_taken of this level; /* renew the uesd width */
                break;
            }
        }
        if (cannot store in any exist levels) {
            current_height := current_height + rect.height;
            Create new level and Update width_taken of the new level;
        }
    }
    return current_height;
}
```

Approximation Ratio: $FFDH(L) \leq 1.7OPT(L) + h_{max} \leq 2.7 * OPT(L)$

Proof: The proof process is quite long, so will not be given in this report. For detailed proof, see our reference list [3].

Next-Fit Decreasing Height (NFDH) algorithm

NFDH packs the next item R (in non-increasing height) on the current level if R fits. Otherwise, the current level is "closed" and a new level is created. As for the pseudocode, we check whether the rect_list (which involve the width and height of all the input rectangle) is empty. If not, we traverse all the rectangle and insert them one by one following the rules started earlier, that is, if can insert in the now top level, insert and renew the used width, otherwise, create a new top level and insert it as well as renewing the current_height.

```
long long NFDH(int width, Rect[] rect_list)
{
    if (IsEmpty(rect_list)) return 0;
    SortByHeight(rect_list);
    int remaining_width := width;
    long long current_height := rect_list[0].height;
    for (rect in rect_list){
        if (rect.width > width) return -1;
        if (remaining_width >= rect.width)
        {
            remaining_width := remaining_width - rect.width;
        }
        else
        {
            remaining_width := width - rect.with;
            current_height := current_height + rect.height;
        }
    }
    return current_height;
}
```

Approximation Ratio: $NFDH(L) \leq 2 * OPT(L) + h_{max} \leq 3 * OPT(L)$

Proof: The proof process is quite long, so will not be given in this report. For detailed proof, see our reference list [3].

Best-Fit Decreasing Height (BFDH) algorithm

BFDH packs the next item R (in non-increasing height) on the level, among those that can accommodate R, for which the residual horizontal space is the minimum. If no level can accommodate R, a new level is created. As for the pseudocode, we check whether the rect_list (which involve the width and height of all the input rectangle) is empty. If not, we traverse all the rectangle and insert them one by one following the rules started earlier, that is, for each insertion, we traverse all the exist levels and find the biggest used width level which is also suitable to insert this rectangle, then insert it and renew the used width, otherwise, create a new level and insert it as well as renewing the current_height.

```
long long BFDH(int width, Rect[] rect_list)
{
    if (IsEmpty(rect_list)) return 0;
    SortByHeight(rect_list);
    long long current_height := 0;
    for (rect in rect_list){
        if (rect.width > width) return -1;
        Find Min remaining_width no less than rect.width;
        if (found) {
            min_remaining_width := remaining_width - rect.width;
        }
        else{
            height := height + rect.height;
            remaining_width := width - rect.width;
            add remaining_width to remaining_width_list;
        }
    }
    return current_height;
}
```

Approximation Ratio: A bound 3 can be given, that is, $BFDH(L) \leq 3 * OPT(L)$. But we are not sure whether it is the most tight upbound.

Proof:

It is obvious that $BFDH(L) \leq NFDH \leq 3 * OPT(L)$.

Non Level Oriented Algorithm

Bottom-Left (BL) Algorithm

BL first order items by non-increasing width. BL packs the next item as near to the bottom as it will fit and then as close to the left as it can go without overlapping with any packed item. Note that BL is not a level-oriented packing algorithm.As for the pseudocode, we check whether the rect_list (which involve the width and height of all the input rectangle) is empty. If not, we traverse all the rectangle and insert them one by one following the rules started earlier, that is, we always put the rectangle that need to insert on the top right corner, then we check if it can move down, if is, keep moving down, if not, keep moving left, then check as the earlier rules, until the rectangle nether can move down nor can move left. Then place this rectangle and keep insert next rectangle. And the result is the max_y, that is the biggest height place in all the insert rectangles.

```
long long BL(int width, Rect[] rect_list)
{
    if (IsEmpty(rect_list)) return 0;
    SortByWidth(rect_list);
    for (rect in rect_list){
        if (rect.width > width) return -1;
        Place rect at rightmost and top most;
        Move rect down until rects already in pack block it;
        Move rect left until rects already in pack block it;
        Move rect down until rects already in pack block it;
        Update max_y;
    }
    return max_y;
}
```

Details are omitted in the pseudo-code above, to see more details, pseudo-code below are preferred.

```
/* More detailed implementation */
long long BL(int width, Rect[] rect_list)
{
    if (IsEmpty(rect_list)) return 0;
    SortByWidth(rect_list);
    for (rect in rect_list){
        if (rect.width > width) return -1;
        Place rect at right most;
        y0 := 0;
        for (texture in y_desc_sorted_pack){
            if (Block(rect, texture)) {
                y0 := texture.y + texture.height;
                break;
            }
        }
        Move rect to (right, y0);
        x := 0;
        for (texture in x_desc_sorted_pack){
            if (Block(rect, texture)){
                x := texture.x + texture.width;
            }
        }
        Move rect to (x, y0);
        y1 := 0;
        for (texture in y_desc_sorted_pack){
            if (Block(rect, texture)) {
                y1 := texture.y + texture.height;
                break;
            }
        }
        Move rect to (x, y1);
    }
    return y_desc_sorted_pack[0];
}
```

Approximation Ratio: $BL(L) \leq 3 * OPT(L)$

Proof:

Let h_1 denote the height of the bottom edge of a highest texture, denote the height ordinate of its higher edge as h_{BL}. Let y denotes th height of the texture. So $h_{BL} = y + h_1$. If the total texture is half occupied, then $h_{opt} \geq max\{y, h_1/2\}$.

If y > $h_1/2$, $\frac{h_{BL}}{h_{OPT}} \leq \frac{y+h_1}{y} < \frac{y+2y}{y} = 3$.

if $y \leq h_1/2$, $\frac{h_{BL}}{h_{OPT}} \leq \frac{h_1/2+h_1}{h_1/2} = 3$.

For more detailed proof, please see paper in our reference list.

Reverse-fit (RF) algorithm

RF is an algorithm ignited by the algorithms above. We can find that when we pack rects level oriented, the rightmost and topmost of one level is often empty, which causes a lot of space wasting. RF first stacks all items of width greater than 1/2. Remaining items are sorted in non-increasing height and will be packed above the height H0 reached by those greater than 1/2. Then RF repeats the following process. Roughly speaking, RF packs items from left to right with their bottom along the line of height H0 until there is no more room. Then packs items from right to left, piling them on first level directly(called second-level) until the total width is at least 1/2. Repeat this by establishing a new first level on the highest rect in second level. Note that there is a famous RF implemented by someone, but our version is a little different from it.

```
long long RF(int width, Rect[] rect_list)
{
    Rect[] rest_rect_list;
    Point[] first_level_list;
    long long current_height := 0;
    if (IsEmpty(rect_list)) return 0;
    for (rect in rect_list){
        if (rect.width > width) return -1;
        if (rect.width > width / 2) /* Stack rect that is wider than half at the
bottom */
        {
            current_height := current_height + rect.height;
        }
        else /* Copy rect that has not been stacked */
        {
            Add rect to rest_rect_list;
        }
    }
    int i := 0;
    while (exist rect in rest_rect_list has not been packed)
    {
        Pack rects from left to right on current_height; /* First Level */
        Drop rects from top to bottom until it hit rects in first level or
ground on current_height
            from right to left until rects has taken up at least half of the
width. /* Second Level */
    }
}
```

Approximation Ratio: The approximation ratio of the well-known RF algorithm is 2. But for our simplified version, surely we have not reached such tight conclusion.

STL Algorithm

We use STL sort function, which is implemented in form of quick sort when data quantity is large and insertion sort when data quantity is small.

Chapter 3: Testing Results

Correctness Testing

- Case 1
    - **Purpose**: A random small case.
    - **Input**:

    > 8
    > 10
    > 7 8
    > 3 3
    > 5 7
    > 3 5
    > 6 6
    > 4 4
    > 5 2
    > 1 1
    > 2 5
    > 4 1

    - **Output**:

Algorithm	FFDH	NFDH	BFDH	BL	RF
Expected result	28	33	28	34	35
Actual result	28	33	28	34	35
Status	Pass	Pass	Pass	Pass	Pass

- Case 2
    - **Purpose**: Same case with Case 1 except for larger width. To test whether algorithm can work properly for different width.
    - **Input**:

> 12
> 10
> 3 3
> 5 2
> 6 6
> 3 5
> 4 1
> 2 5
> 5 7
> 7 8
> 4 4
> 1 1

- **Output**:

Algorithm	FFDH	NFDH	BFDH	BL	RF
Expected result	19	19	19	19	22
Actual result	19	19	19	19	22
Status	Pass	Pass	Pass	Pass	Pass

- Case 3
  - **Purpose**: A random case with larger size. To test whether algorithms can work properly when texture numbers grow larger.
  - **Input**:

> 18
> 25
> 1 7
> 3 5
> 4 4
> 2 3
> 11 2
> 2 4
> 13 4
> 15 6
> 17 8
> 16 3
> 14 2
> 7 8
> 6 9
> 7 7
> 1 5
> 3 3
> 5 6
> 6 10
> 11 3
> 3 7
> 14 4
> 4 5
> 6 9
> 1 3
> 5 7

  - Output:

Algorithm	FFDH	NFDH	BFDH	BL	RF
Expected result	57	64	57	63	63
Actual result	57	64	57	63	63
Status	Pass	Pass	Pass	Pass	Pass

- Case 4
  - Purpose: Case with texture width larger than given bin width. To test whether the algorithm can work out error cases.

- Input:

> 18
> 25
> 1 7
> 3 5
> 4 4
> 2 3
> 11 2
> 2 4
> 13 4
> 19 6
> 17 8
> 16 3
> 14 2
> 7 8
> 6 9
> 7 7
> 1 5
> 3 3
> 5 6
> 6 10
> 11 3
> 3 7
> 14 4
> 4 5
> 6 9
> 1 3
> 5 7

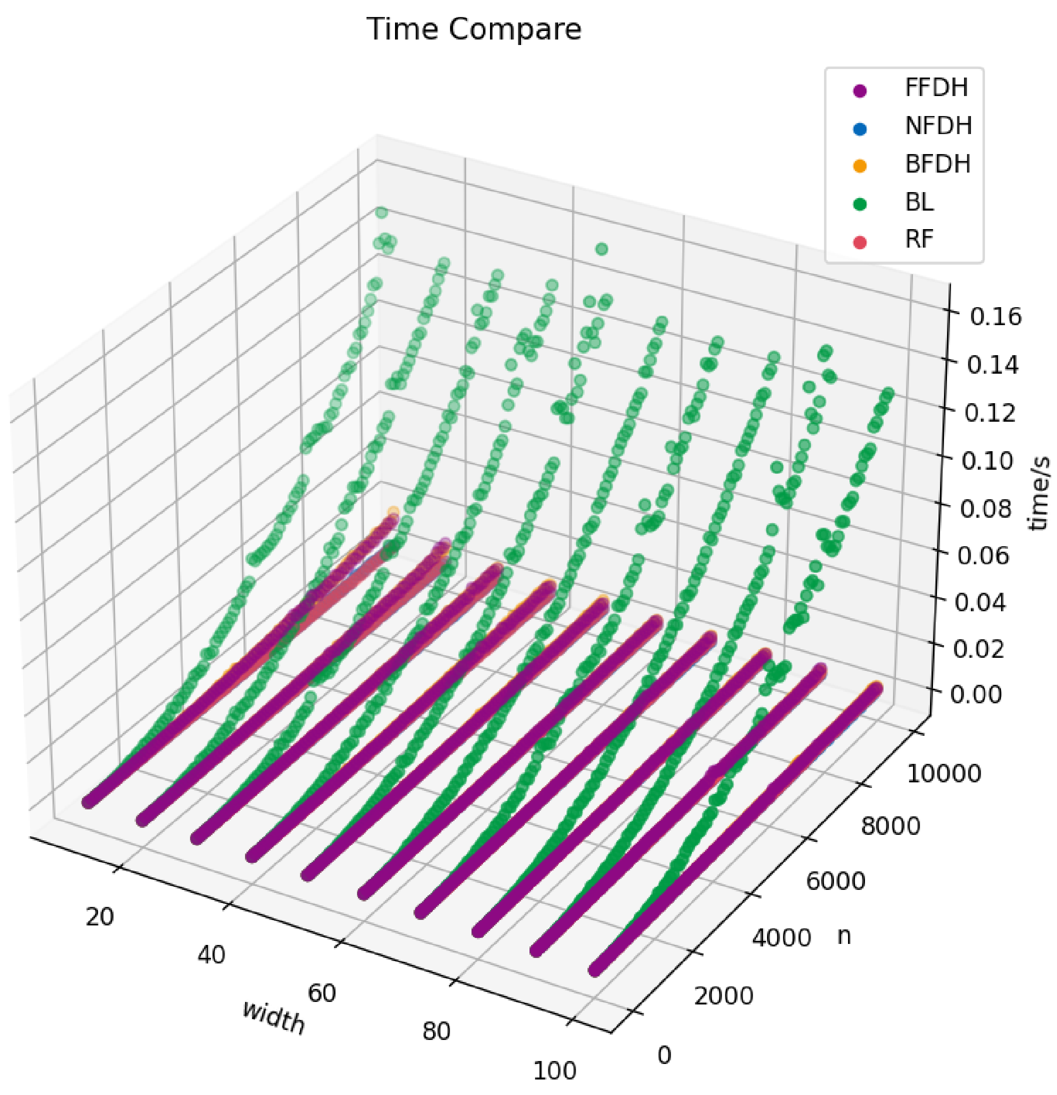
- Output:

Algorithm	FFDH	NFDH	BFDH	BL	RF
Expected result	-1	-1	-1	-1	-1
Actual result	-1	-1	-1	-1	-1
Status	Pass	Pass	Pass	Pass	Pass

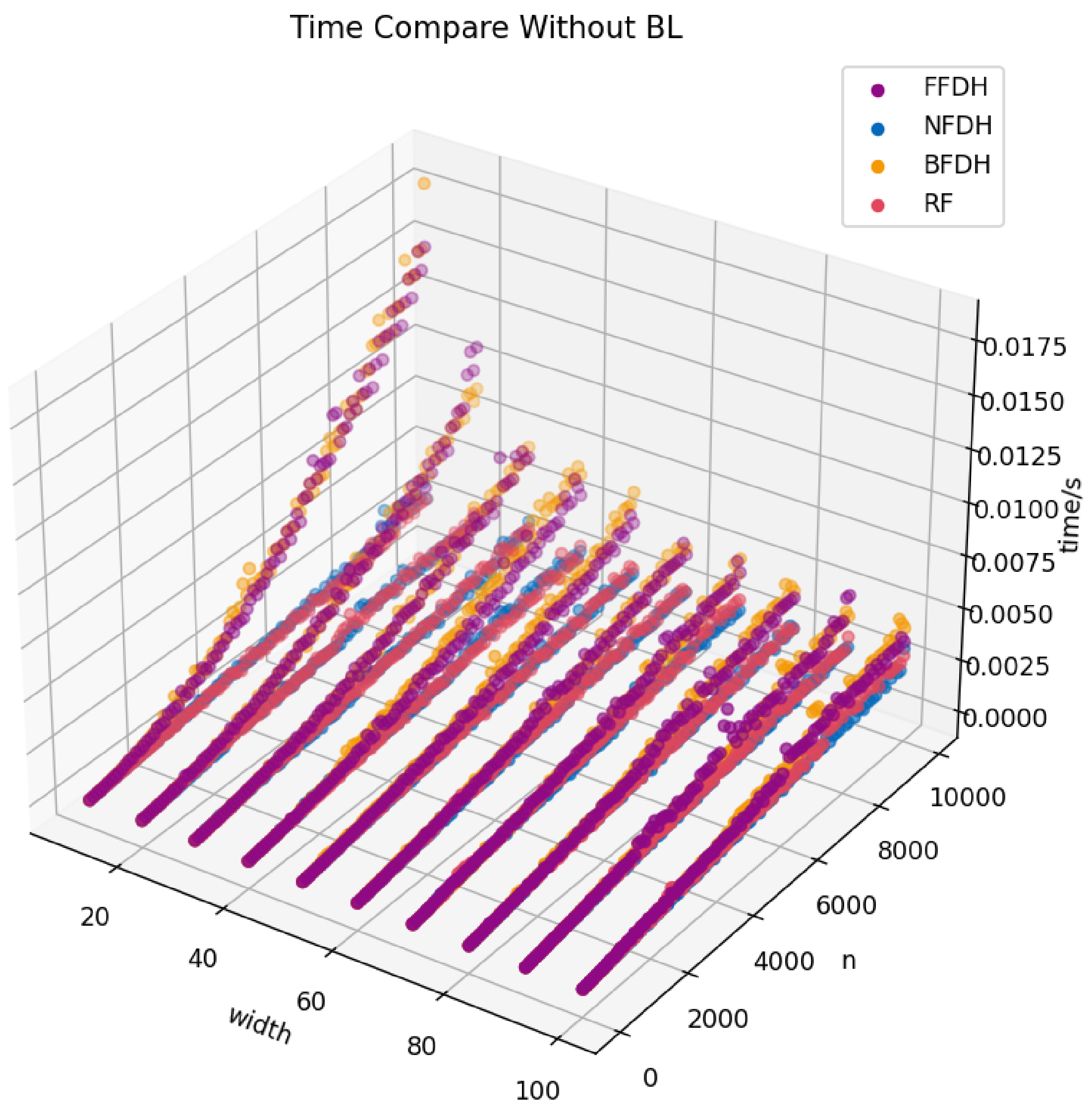
Performance Testing

To test the performance of our different algorithms, we use Gen.cpp to generate different size data, for width ranging from 10 to 100 and for n ranging from 10 to 10000. We assure that for small cases the time error will still not be very large by repeating running the algorithm and getting average time. As width will matter little if the texture width also differ linearly according to width, so the maximum texture is limited to a constant value that is no larger than the least width.

Time Performance

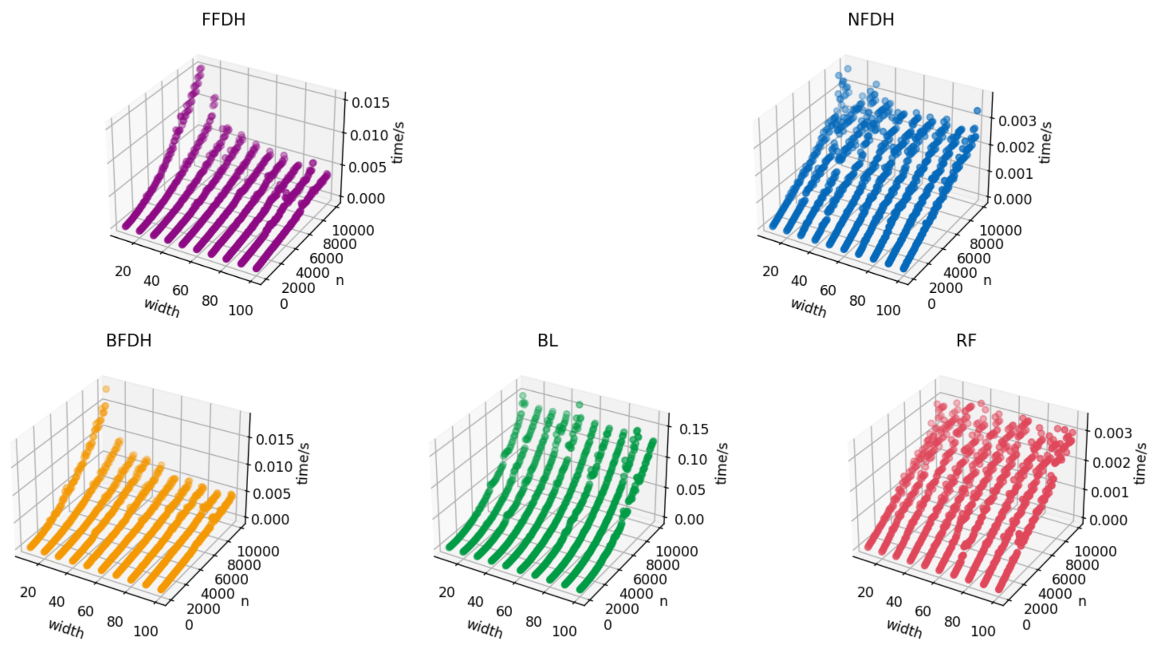


Though the time complexity of BL is just the same as FFDH and BFDH, its actual performance when dealing with the random data generated by our program is quite poor. And because it is far slower than other algorithms, the comparison among the other algorithms is not revealed by this graph. To compare them, we remove BL from the graph. And the graph is as follow.



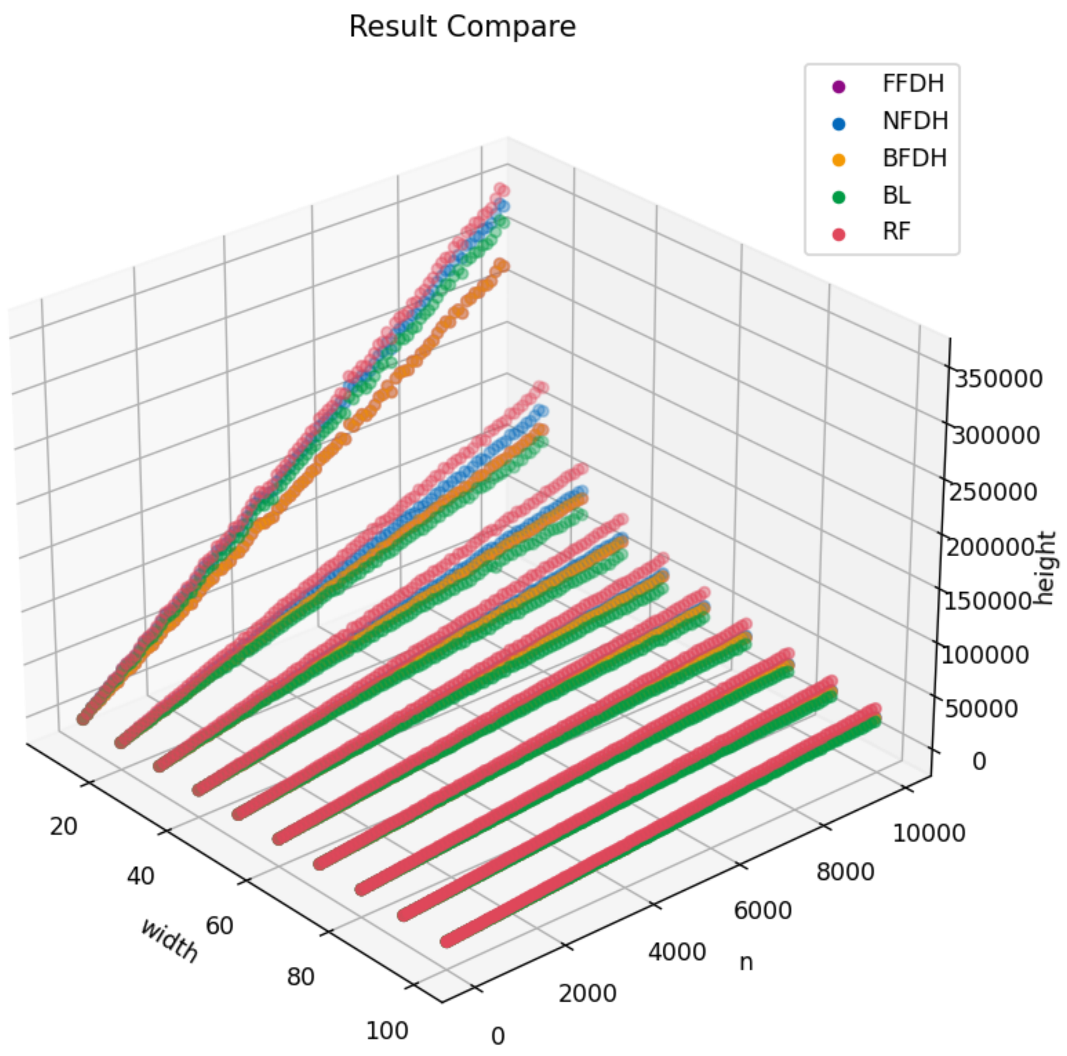
Through the graph, we can find that the performance of FFDH and BFDH is very close and when the width of the strip is larger, both of them perform better while width seemed to matter little to other algorithms. Also, it can be further confirmed by the graph that our algorithm FFDH and BFDH($O(N^2)$ proved in Analysis) is much slower than NFDH($O(NlogN)$ proved in Analysis).

To make their performance more clear, we also draw split graph for each algorithm.



Result Performance

The result comparison of five algorithms is drawn below. Note that you can not find FFDH(Purple) in the graph because it is almost the same as BFDH(Yellow) and thus been covered.



According to the graph, we can find BFDH and FFDH work good when width is small but when width grows bigger, BL works better.

Conclusion

According to the graph above, we come to a conclusion that among all the algorithms we have implemented, BFDH and FFDH give consideration to both speed and result when width is very small. And BL, though quite slow, work well when width is not very small. NFDH, though quite fast, will often give a result that is not so tight.

Chapter 4: Analysis and Comments

Analysis

Time Complexity

FFDH

The worse is every rectangle need to create a new level, that is $1+2+3+...+N = O(N^2)$

NFDH

Though this algorithm only has to traverse all the rect in order, which is $O(N)$, it takes time to sort them before traverse, which is $O(NlogN)$. So the time complexity is $O(NlogN)$.

BFDH

We need to check each level whenever it can store this rectangle or not, that is $1+2+3+...+N = O(N^2)$

BL

The worse is never be blocked and can move down forever , that is （$1+2+3+...+N$）$/2 = O(N^2)$

RF

This algorithm traverse rects and for each rect in second level, it checks collision with first level. The number of rect in the first level is relevant to width and rects, and for the worst case, it is $O(N)$. So the time complexity is $O(N*N) = O(N^2)$.

Space Complexity:

Store all the rectangles need $2N = O(N)$, and for each algorithm, it will create new vector , each size won't exceed N, and the number of the vector is constant. So all the space complexity of each algorithm is $O(N)$

Comments

The BFDH is similar to the FFDH, just in BFDH, we need to traverse all the known levels under any cricumstances and find the smallest suitable level and insert while FFDH only find the first suitable level and insert. But in fact, when we do insertion, we can pile up on the last rectangle if the whloe height still not exceed the height of this level, that's means we can save some used width in this level. But the difficulty is that we need to record the unused height of this level and the last used width, which is hard to know. and for the pile up rectangle, we may see this as a new small level and keep piling up, which means the record of unused height and last used width is change, that's a huge project.

As for the NFDH, the advantage is obvious, because it can reduce the time complexity greatly, but also the disadvantage is that the result may be bad, that's because for a new rectangle, it only check whether this level is suitable but not check the other levels, and if not suitable, then it create new level, which makes the final height bigger.

Then follows BL， it provide a new way to do insertion. for each rectangle coming, it will be placed in the top right corner. and it will check whether the rectangle can move down, if can, then it will keep moving down until cannot, otherwise, it will keep moving left until cannot. Then it keeps moving down again if it can, and do the same options. At last, the rectangle can't move down or move left, then it's position is fixed. Then we keep doing those options on the next rectangle until finished. So although the insertion become a little complex, it's also a great way and may get better result for some data, and it's time complexity is similar to the FFDH.

When talks to RF, it also opens up a new idea. It doesn't do insertion like the previous algorithm, but takes two parts simultaneously —— pile up those width more than half of the resulting texture width as width and the others as height, then it merge those two triangular like structure. As you can see, the time complexity of BL is not small, and whether the result is good or not depends greatly on the given data of the retangles.

Bonus:

There is already a well-known Reverse-Fit algorithm(also short for RF) proposed by someone and it's result performance is quite good. It's approximation ratio is 2. The RF we proposed is somehow a poorly designed version compared to it and thus don't give a satisfactory result though it is quite fast. Both algorithms stack the textures wider than half at the bottom and pack textures in two level scheme. What makes the awesome algorithm go far than ours is its operations on second level. While we simply base another first level on the highest first level and second level, which may waste a lot space, it base another first level on last second level, even if last first level is higher.

Appendix: Source Code

As the whole code will be too long to display in appendix, We only paste the main algorithm in appendix.

```
#include "StripPack.h"

long long FFDH(int width, std::vector<Rect> rect_list){ // First-Fit Decreasing
Height
    long long current_height = 0; // the whole height of the sum of the height
each level
    int j;

    if (rect_list.empty()) return 0;

    std::sort(rect_list.begin(), rect_list.end(), HeightDescSort); // sort
according to height in descending order
    std::vector <int> width_taken_list; // the already used width of each level
    for (int i = 0; i < rect_list.size(); i++) {
        if (rect_list[i].width > width){
            return -1;
        }

        for (j = 0; j < width_taken_list.size(); j++) {
            if (width_taken_list[j] + rect_list[i].width <= width) { // can
store the rectangular in this level which be pointed to by the pointer "point".
                width_taken_list[j] += rect_list[i].width; // renew the already
used width of this level
                break;
            }
        }
        if (j == width_taken_list.size()) { // can't store the rectangular in
all the level that has been existed
            current_height += rect_list[i].height; // renew the whole new Height
and create a new level
            width_taken_list.push_back(rect_list[i].width);
        }
    }
    return current_height;
}

long long NFDH(int width, std::vector<Rect> rect_list) // Next-Fit Decreasing
Height
{
    int current_width = 0;
    long long current_height = 0;
    int i = 0;
    if (rect_list.empty()) return 0;

    std::sort(rect_list.begin(), rect_list.end(), HeightDescSort); // sort
according to height in descending order
    current_height += rect_list[0].height;

    while (i < rect_list.size())
    {
        if (rect_list[i].width > width) // can not be packed
        {
            return -1;
        }
        else if (current_width + rect_list[i].width <= width) // can still be
packed in this level
        {
            current_width += rect_list[i].width;
        }
        else // can not be packed in this level, new a level
        {
            current_width = rect_list[i].width; // the width of this texture
will be next level's current width
            current_height += rect_list[i].height; // add height to new a level
        }
        i++; // pack next texture
    }

    return current_height;
}

long long BFDH(int width, std::vector<Rect> rect_list) // Best-Fit Decreasing
Height
{
    int min, max, max_index;
    long long current_height = 0;
    if (rect_list.empty()) return 0;

    std::sort(rect_list.begin(), rect_list.end(), HeightDescSort); // sort
according to height in descending order

    std::vector<int> width_taken_list;
    for (int i = 0; i < rect_list.size(); ++i)
    {
        if (rect_list[i].width > width) // can not be packed
        {
            return -1;
        }
        else
        {
            // find the min and the max width left
            min = INF;
            max = -1;
            for (int j = 0; j < width_taken_list.size(); ++j)
            {
                int width_taken = width_taken_list[j] + rect_list[i].width;
                if (width_taken <= width && width_taken > max)
                {
                    max = width_taken;
                    max_index = j;
                }
                if (width_taken < min) min = width_taken;
            }
            if (min > width) // if even the min can not be placed in all the
levels existing, new a level
            {
                width_taken_list.push_back(rect_list[i].width);
                current_height += rect_list[i].height; // update height as a new
level is created
            }
            else // insert this texture into the level that will leave least
width
            {
                width_taken_list[max_index] = max;
            }
        }
    }

    return current_height;
}

long long BL(int width, std::vector<Rect> rect_list) // Bottom-Left
{
    int bottom_y, top_y, left_x, right_x;
    std::vector<Point>::iterator j, k;
    if (rect_list.empty()) return 0;

    std::sort(rect_list.begin(), rect_list.end(), WidthDescSort);

    std::vector<Point> y_sorted_list; // in descending order of y ordinate
    std::vector<Point> x_sorted_list; // in descending order of x ordinate

    for (int i = 0; i < rect_list.size(); ++i)
    {
        if (rect_list[i].width > width)
        {
            return -1;
        }
        // Move the texture towards bottom from rightmost and upmost
        // Find the texture that will prevent this texture from moving lower
        for (j = y_sorted_list.begin(); j != y_sorted_list.end(); j += 2)
        {
            if (j[1].x > width - rect_list[i].width) // be blocked and cannot
move down any more
            {
                break;
            }
        }
        if (j == y_sorted_list.end()) // can move to the bottom
        {
            bottom_y = 0;
            top_y = rect_list[i].height;
        }
        else // be blocked by some texture
        {
            bottom_y = (*j).y;
            top_y = (*j).y + rect_list[i].height;
        }

        // Move the texture to left
        for (k = x_sorted_list.begin(); k != x_sorted_list.end(); k += 2)
        {
            if (k[0].y < top_y && k[0].y > bottom_y || k[1].y < top_y && k[1].y > bottom_y || k[0].y <= bottom_y && k[1].y >= top_y) // be blocked and cannot move left any more
            {
                break;
            }
        }
        if (k == x_sorted_list.end()) // can move to the leftmost
        {
            left_x = 0;
            right_x = rect_list[i].width;
        }
        else // be blocked by some texture and cannot move to left any more
        {
            left_x = (*k).x;
            right_x = left_x + rect_list[i].width;
        }

        // Try to move down the texture again
        for (; j != y_sorted_list.end(); j += 2)
        {
            if (j[0].x > left_x && j[0].x < right_x || j[1].x > left_x && j[1].x < right_x || j[0].x <= left_x && j[1].x >= right_x) // be blocked by some texture
            {
                break;
            }
        }

        if (j == y_sorted_list.end()) // can be moved to the bottom
        {
            bottom_y = 0;
            top_y = rect_list[i].height;
        }
        else // be blocked by some texture
        {
            bottom_y = (*j).y;
            top_y = bottom_y + rect_list[i].height;
        }

        // Update y_sorted_list
        for (j = y_sorted_list.begin(); j != y_sorted_list.end(); j += 2)
        {
            if (top_y >= (*j).y) break;
        }
        y_sorted_list.insert(y_sorted_list.insert(j, Point{ right_x, top_y }), Point{ left_x, top_y });

        // Update x_sorted_list
        for (k = x_sorted_list.begin(); k != x_sorted_list.end(); k += 2)
        {
            if (right_x >= (*k).x) break;
        }
        x_sorted_list.insert(x_sorted_list.insert(k, Point{ right_x, top_y }),
Point{ right_x, bottom_y });
    }

    return y_sorted_list[0].y;
}

long long RF(int width, std::vector<Rect> rect_list)
{
    // Stack texture wider than half at the bottom.
    // Repeat building first level and second level until all textures are
packed
    // First level from left to right, second level from right to left. But for
second level, each texture are directly dropped,
    // until width taken is no less than half. While dropping, record the
highest.
    // First level is then established on this highest.
    int current_height = 0;
    std::vector<Rect>::iterator i;
    std::vector<Rect> rest_rect_list;
    std::vector<Point> first_level_list;
    int current_width, j, current_x;
    // 1. Stack textures with width more than half of the strip width
    for (i = rect_list.begin(); i != rect_list.end(); ++i)
    {
        if (i->width > width)
            return -1;
        else if (i->width > (double)width / 2)
        {
            current_height += i->height;
        }
        else
        {
            rest_rect_list.push_back(*i); // extract textures with width no
greater than half
        }
    }
    // 2. Sort the rest texture in descending order of width
    std::sort(rest_rect_list.begin(), rest_rect_list.end(), HeightDescSort);

    i = rest_rect_list.begin();
    while (i != rest_rect_list.end())
    {
        current_width = 0;
        first_level_list.clear();
        // 3. First level from left to right
        // only the ordinates of rightmost and topmost points are necessary to
store
        while (i != rest_rect_list.end() && current_width + i->width <= width)
// texture can still be put in the first level
        {
            current_width += i->width;
            if (first_level_list.empty())
            {
                first_level_list.push_back(Point{i->width, current_height + i-
>height});
            }
            else
            {

 first_level_list.push_back(Point{first_level_list[first_level_list.size() -
1].x + i->width, current_height + i->height});
            }
            i++;
        }
        if (!first_level_list.empty())
        {
            current_height = first_level_list[0].y;
        }
        // 4. Second level from right to left
        // max_y should be recorded
        current_width = 0;
        while (i != rest_rect_list.end() && current_width < (double)width / 2)
        {
            current_x = width - current_width - i->width;
            // From right to left
            for (j = first_level_list.size() - 1; j >= 0; j--)
            {
                if (first_level_list[j].x <= current_x) // No possibility to be
blocked by more left
                {
                    break;
                }
                if (j > 0)
                {
                    if ((first_level_list[j].x > current_x && first_level_list[j
- 1].x <= current_x + i->width) ||
                        (first_level_list[j].x >= current_x + i->width &&
first_level_list[j - 1].x <= current_x)) // be blocked
                    {
                        current_height = current_height >= first_level_list[j].y
+ i->height ? current_height : first_level_list[j].y + i->height;
                    }
                }
                else
                {
                    if (first_level_list[j].x > current_x) // be blocked by the
left most texture in the first level
                    {
                        current_height = current_height >= first_level_list[j].y
+ i->height ? current_height : first_level_list[j].y + i->height;
                    }
                }
            }
            current_width += i->width; // update width taken
            i++;
        }
    }
    return current_height;
}


bool HeightDescSort(Rect a, Rect b)
{
    return (a.height > b.height) || (a.height == b.height && a.width > b.width);
}
```

```
bool WidthDescSort(Rect a, Rect b)
{
    return (a.width > b.width) || (a.width == b.width && a.height > b.height);
}
```

References

[1] Schiermeyer, "Reverse-Fit: A 2-optimal algorithm for packing rectangles", Springer.

[2] Brenda S. Baker, E. G. Coffman, Jr., Ronald L. Rivest, "Orthogonal Packings in Two Dimensions", *SIAM J. Comput*. 9.

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

[4] "Strip packing problem Wikipedia", https://en.wikipedia.org/wiki/Strip_packing_problem#cite_note-Coffman1980-9

Declaration

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