Texture Packing

Group 18

Date:2020-05-20

CONTENT

Chapter 1 Introduction

1.1 Background

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 is a NP-hard problem, and we are going to solve it using approximate algorithm.

1.2 Problem Description

1.2.1 Input Specification

Each input file contains one test case. For each case, the first line gives two positive integers: N, the number of textures, and M, the width of the resulting texture.

```
N M
```

Then N lines follow. Each line gives two positive integers: w, the width of this texture, and h, the height of this texture.

```
w h
```

1.2.2 Output Specification

The first line gives the minimal height of the resulting texture. 'h' here stands for a specific integer.

```
minimum height=h
```

Then N bolcks follow. Each block contains 5 lines.

- The first line gives the number of this texture, num, starting at 0.
- The second line gives the width of this texture, w.
- The third line gives the height of this texture, h.
- The fourth line gives the w-position of the left-top cornor of this texture, w'.
- The fifth line gives the h-position of the left-top cornor of this texture, h'.

```
num
width=w
height=h
pos.w=w'
pos.h=h'
```

1.2.3 Problem Analysis

This project requires designing an approximation algorithm that runs in polynomial time. To test correctness, this problem needs to generat test cases of different sizes (from 10 to 10,000) with different distributions of widths and heights.

Chapter 2 Algorithm Specification

We used three kinds of algorithms to work out an approximating result.

2.1 First-Fit Decreasing Height Algorithm

First-Fit Decreasing Height Algorithm(FFDH) always adds the texture to the resulting texture linearly. Once the scanner gets one texture, it would take it directly.

The general idea is: from the first texture, try to add it. Then find the next texture, try to add it to **any possible level from 0 to the final**(all textures of the same level share the same 'w'). If fails, add it to a new level. Otherwise, add it to the first level that is able to contain the texture. FFDH continuely doing this utill no texture left.

The pseudocode of First-Fit Decreasing Height algorithm is:

```
FFDH(width, n)
    let l be a new set of pointers
    for i = 0 to n
    {
        flag = 0
        for j = 0 to l.size
        {
            if l[j]->rem_w >= texture[i].w
            {
                flag = 1
                update attributes of l[j]
                break
            }
        }
        if flag==0
        {
            let p be a new pointer
            update attributes of p
            add p to l
            update attributes of texture[i]
        }
    }
    return height of the resulting texture
```

2.2 Next-Fit Decreasing Height Algorithm

Next-Fit Decreasing Height Algorithm(NFDH) always adds the texture to the resulting texture linearly. Once the scanner gets one texture, it would take it directly.

The general idea is: from the first texture, try to add it. Then find the next texture, try to add it to **the former active level**(all textures of the same level share the same 'w'). If fails, add it to a new level. Otherwise, add it to the formaer level. NFDH continuely doing this utill no texture left.

The pseudocode of Next-Fit Decreasing Height Algorithm is:

```
NFDH(width, n)
    let l be a new set of pointers
    for i = 0 to n
    {
        flag = 0
        j = l.size - 1
        if l[j]->rem_w >= texture[i].w
        {
            flag = 1
            update attributes of l[j]
        }
        if flag==0
        {
            let p be a new pointer
            update attributes of p
            add p to l
            update attributes of texture[i]
```

```
        }
    }
    return height of the resulting texture
```

2.3 Best-Fit Decreasing Height Algorithm

Best-Fit Decreasing Height Algorithm(BFDH) always adds the texture to the resulting texture linearly. Once the scanner gets one texture, it would take it directly.

The general idea is: from the first texture, try to add it. Then find the next texture, try to add it to **the level with the minimal spare space**(all textures of the same level share the same 'w'). If fails, add it to a new level. Otherwise, add it to that level. BFDH continuely doing this utill no texture left.

The pseudocode of Best-Fit Decreasing Height Algorithm is:

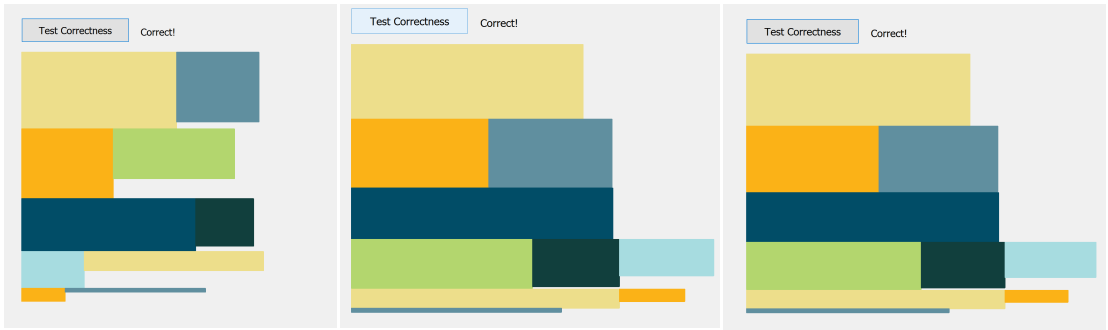
```
BFDH(width, n)
    let l be a new set of pointers
    for i = 0 to n
    {
        min_num = -1
        min_w = 0x3f3f3f3f
        for j = 0 to l.size
        {
           if l[j]->rem_w >= texture[i].w && min_w > l[j]->rem_w - texture[i].w
               update min_num and min_w
        }
        if min_num >= 0
        {
            let p be a new pointer
            update attributes of p
            add p to l
            update attributes of texture[i]
        }
    }
    return height of the resulting texture
```

Chapter 3: Testing Results

3.1 Results

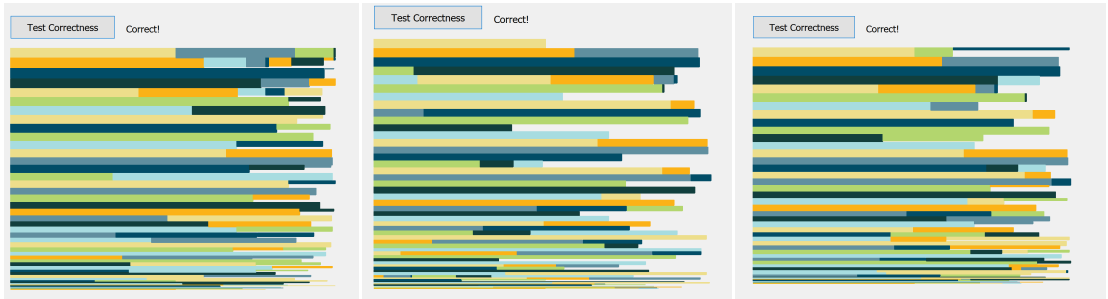
Sample1: test10timeswith50/data.in

> 10 rectangles
>
> Fixed width=50

	FFDH	NFDH	BFDH
height	114	125	125
image			

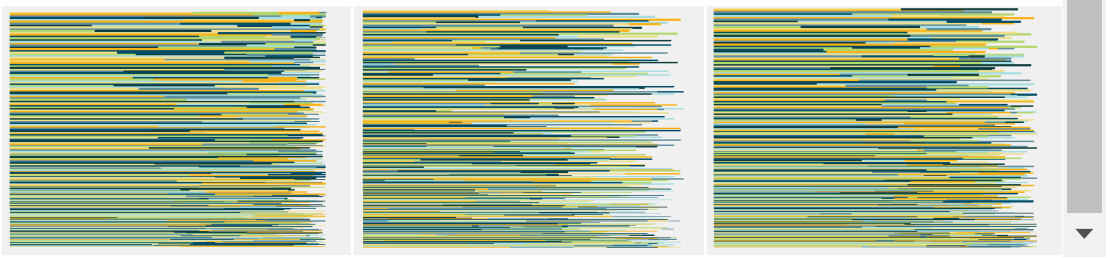
Sample2:test100timeswith200/data.in

> 100 rectangles
>
> Fixed width=200

	FFDH	NFDH	BFDH
height	924	1050	968
image			

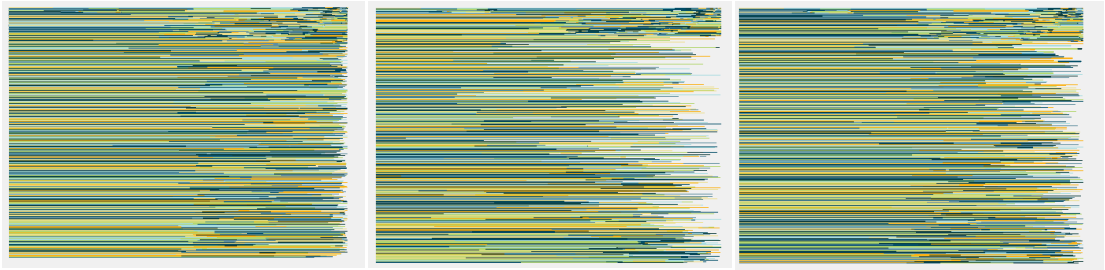
Sample3:test500timeswith100/data.in

> 500 rectangles
>
> Fixed width=100

	FFDH	NFDH	BFDH
height	3632	4493	3786
image			

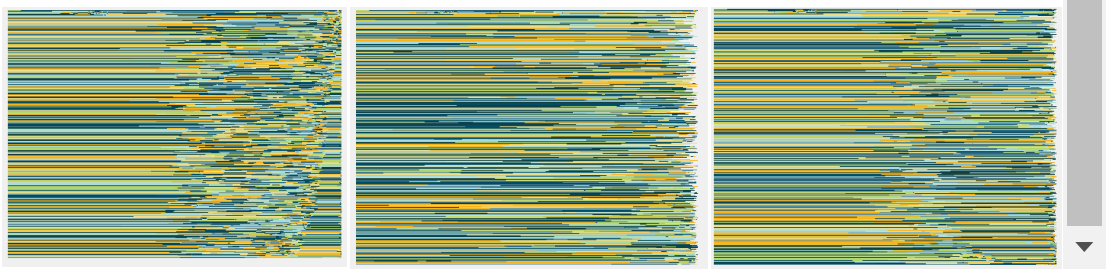
Sample4:test1000timeswith300/data.in

> 1000 rectangles
>
> Fixed width=300

	FFDH	NFDH	BFDH
height	189482	232809	197419
image			

Sample5:test10000timeswith1000/data.in

> 10000 rectangles
>
> Fixed width=1000

	FFDH	NFDH	BFDH
height	1299656	1679350	1399306
image			

3.2 Charts

n=100

times	1	10	50	100	500	1000	5000	8000	10000	50000
FFDH	0	0	0	0	4	7	36	57	71	280
NFDH	0	0	0	0	3	6	30	48	59	262
BFDH	0	0	0	1	5	8	41	58	81	354

n=500

times	1	10	50	100	500	1000	5000	8000	10000	50000
FFDH	0	0	4	6	37	70	264	537	641	2692
NFDH	0	0	2	2	17	34	150	229	345	1352
BFDH	0	1	4	9	47	88	356	544	749	3348

n=1000

times	1	10	50	100	500	1000	5000	8000	10000	50000
FFDH	0	2	12	25	111	196	1108	1459	2401	10813
NFDH	0	1	2	7	34	83	267	432	662	2662
BFDH	0	3	15	26	138	225	1243	2098	2770	16688

n=2000

times	1	10	50	100	500	1000	5000	8000	10000	50000
FFDH	1	10	40	89	412	866	3300	4727	8201	41235
NFDH	1	2	7	14	98	112	643	800	1152	3256
BFDH	1	11	58	107	463	935	4863	9030	11843	57818

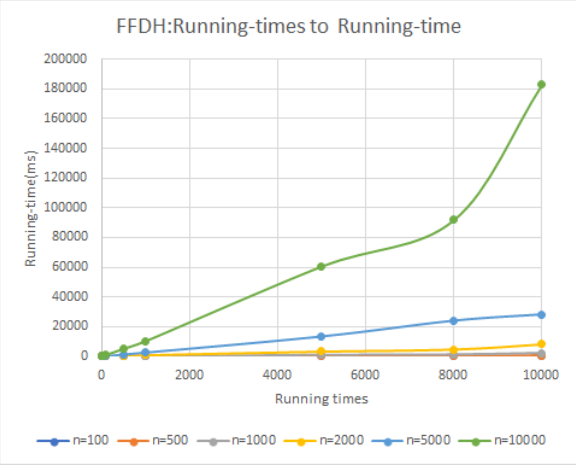
n=5000

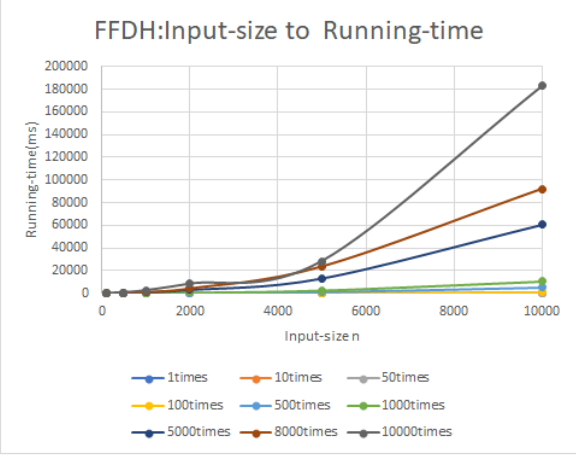
times	1	10	50	100	500	1000	5000	8000	10000
FFDH	3	27	133	265	1319	2652	13403	23918	28109
NFDH	0	2	9	16	82	164	867	1351	1743
BFDH	3	33	167	333	1688	3352	17520	28882	35522

n=10000

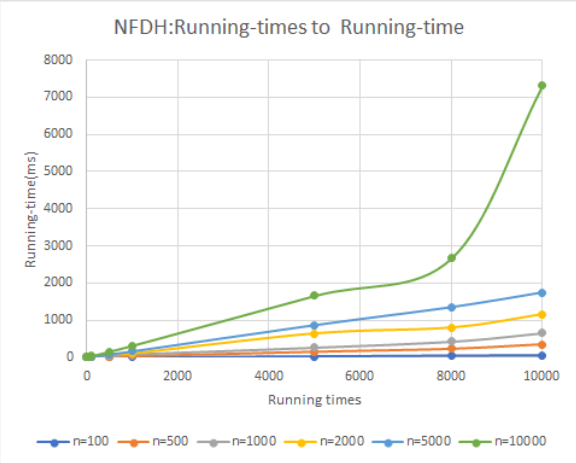
times	1	10	50	100	500	1000	5000	8000	10000
FFDH	13	101	501	1015	5148	10298	60588	91781	182876
NFDH	1	3	16	32	157	315	1658	2664	7318
BFDH	15	125	619	1262	6415	13638	71368	111665	340361

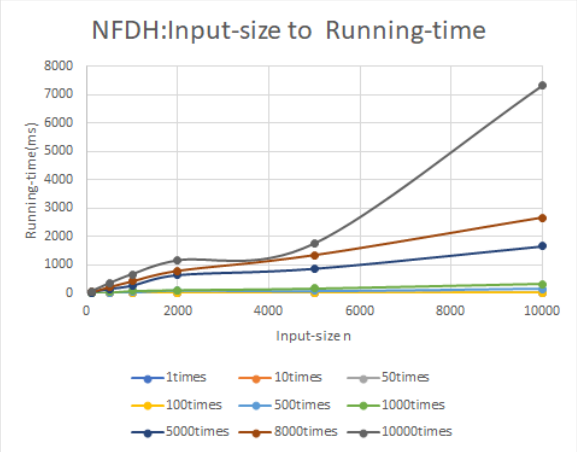
FFDH



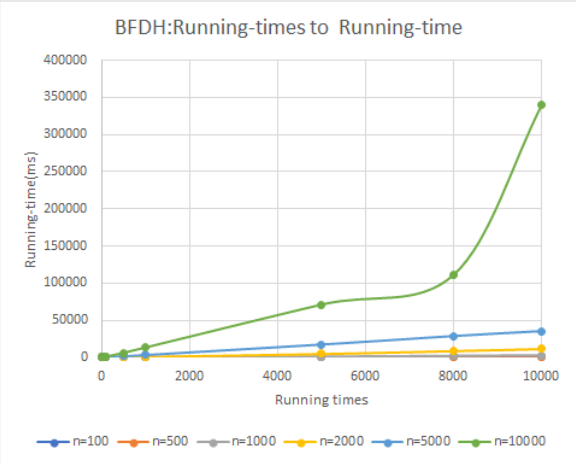


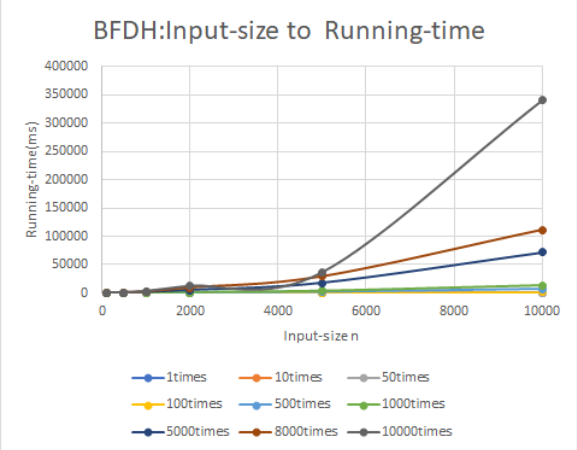
NFDH





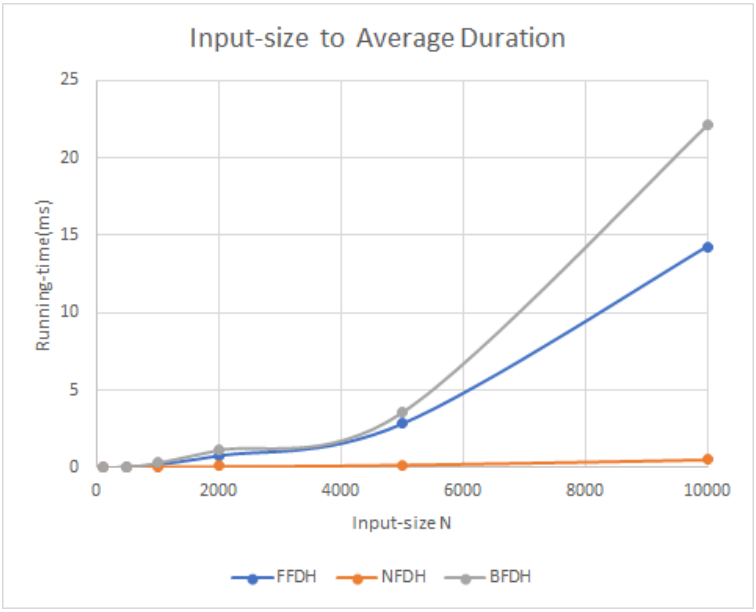
BFDH





Comparison of three algorithms

Average Duration	100	500	1000	2000	5000	10000
FFDH	0.006094	0.056937	0.216003	0.788645	2.831556	14.28657
NFDH	0.005465	0.028542	0.055585	0.081502	0.171688	0.493248
BFDH	0.00734	0.068925	0.310818	1.140207	3.548112	22.11865



Chapter 4: Analysis and Comments

	FFDH	**NFDH**	**BFDH**
Time Complexity	$O(N^2)$	$O(N \log N)$	$O(N^2)$
Space Complexity	$O(N)$	$O(N)$	$O(N)$

4.1 Analysis

FFDH packs the next rectangle R in non-increasing height on the first level where R fits. If no level can accommodate R, a new level will be created. So, the time complexity of FFDH is $O(n \log n)$(for sort)+$O(n * l)$, l is the max number of levels. In the worst case($l = n$), which mean each rectangle occupies a level with itself, the time complexity is $O(n^2)$.

NFDH packs the next rectangle R in non-increasing height on the current level if it can accommodate R, or the current level will be closed with a new level created. So, the time complexity of NFDH is $O(n \log n + n) = O(n \log n)$.

BFDH packs the next rectangle R in non-increasing height on the level, among those that can accommodate R, for which the residual horizontal space is the minimum. If there is no level can accommodate R, a new level will be created. So, the time complexity of BFDH is as same as FFDH.

The space complexity for 3 algorithms are simply $O(N)$ to store all the rectangles.

4.2 Comments

Comparing the actual test results, we can see that the time complexity is correct. At the same time, the program also passed the correctness test. So we can preliminarily judge that our program has no problem in describing the three algorithms.

For the three algorithms FFDH, NFDH, and BFDH, the time complexity of algorithm NFDH is significantly less than the others, this is because NFDH has one less cycle than the other two. It is obvious from the results that the NFDH algorithm is much more wasteful of space than FFDH and BFDH.

Although in the worst case, the time complexity of FFDH and BFDH is the same, FFDH is often quicker than BFDH in most cases because it does not need to traverse all levels. But the difference in results of FFDH and BFDH are not so significant. Who is better actually just depends entirely on the data characteristics of the input data.

Appendix

Source Code in C++

```cpp
//modify the code in line 57 to change the algorithm
#include <bits/stdc++.h>
using namespace std;
#define N 10005 //maximum number of textures
#define MIN 0x3f3f3f3f
//the structure to save a rectangle
struct Texture
{
    int w;    //width
    int h;    //height
    int posw; //the width axis
    int posh; //the height axis
};
//the structure to save a level
struct level
{
    int h;  //height of the level
    int rem_w;  //remaining width
    int tot_h;  //total height
};
//compare function
bool cmp(Texture a, Texture b){
    return a.h>b.h;
}
//First-Fit Decreasing Height algorithm
int FFDH(int width, int n);
//Next-Fit Decreasing Height algorithm
int NFDH(int width, int n);
//Best-Fit Decreasing Height algorithm
int BFDH(int width, int n);
vector<Texture> texture(N);
int main()
{
    int width, n;
    int i, j;

    //read in n and width
    freopen("data.in", "r", stdin);
    cin >> n >> width;
    START:
    for (i = 0; i < n; i++)
    {
        //input each texture
        cin >> texture[i].w >> texture[i].h;
        //if some rectangle's width is longer than the strip
        if (texture[i].w > width)
        {
            cout << "Error!";
            //reinput
            goto START;
        }
    }

    //sort the height in non-increasing order
    sort(texture.begin(), texture.end(), cmp);

    int min_height = BFDH(width, n);
    //output the minimum approximation height
    printf("minimum height=%d\n", min_height);

    //output the result of the packing
    for (i = 0; i < n; i++)
    {

        printf("%d\nwidth=%d\nheight=%d\npos.w=%d\npos.h=%d\n", i, texture[i].w, texture[i].h,
texture[i].posw, texture[i].posh);
    }

    return 0;
}

//First-Fit Decreasing Height algorithm
int FFDH(int width, int n)
{
    int i, j;
    vector<level *> l;  //pointers to levels

```

```
    for (i = 0; i < n; i++)
    {
        int flag = 0;   //flag=1 if the texture should be added
        //check each level
        for (j = 0; j < l.size(); j++)
        {
            //if the texture can be put into level j
            if (l[j]->rem_w >= texture[i].w)
            {
                flag = 1;
                //calculate the position of texture in the strip
                texture[i].posw = width - l[j]->rem_w;
                texture[i].posh = l[j]->tot_h;
                //update the remaining width
                l[j]->rem_w -= texture[i].w;
                goto NEXT;
            }
        }
    NEXT:
        //create a new level
        if (!flag)
        {
            level *p = new level;
            //the new level's height=current texture's height
            p->h = texture[i].h;
            //calculate the total height of the new level
            p->tot_h = (l.size() == 0) ? 0 : l[l.size() - 1]->tot_h + l[l.size() - 1]->h;
            //update the remaining width
            p->rem_w = width - texture[i].w;
            //add p to l
            l.push_back(p);
            //calculate the position of new texture in the strip
            texture[i].posw = 0;
            texture[i].posh = p->tot_h;
        }
    }
    return ((l.size()==0)?0:l[l.size()-1]->tot_h+l[l.size()-1]->h);
}
//Next-Fit Decreasing Height algorithm
int NFDH(int width, int n)
{
    int i, j;
    vector<level *> l;  //pointers to levels

    for (i = 0; i < n; i++)
    {
        int flag = 0;   //flag=1 if the texture should be added
        //check the last level
        for (j=l.size()-1; j < l.size(); j++)
        {
            //if the texture can be put into level j
            if (l[j]->rem_w >= texture[i].w)
            {
                flag = 1;
                //calculate the position of texture in the strip
                texture[i].posw = width - l[j]->rem_w;
                texture[i].posh = l[j]->tot_h;
                //update the remaining width
                l[j]->rem_w -= texture[i].w;
                goto NEXT;
            }
        }
    NEXT:
        //create a new level
        if (!flag)
        {
            level *p = new level;
            //the new level's height=current texture's height
            p->h = texture[i].h;
            //calculate the total height of the new level
            p->tot_h = (l.size() == 0) ? 0 : l[l.size() - 1]->tot_h + l[l.size() - 1]->h;
            //update the remaining width
            p->rem_w = width - texture[i].w;
            //add p to l
            l.push_back(p);                                                 //calculate the
position of new texture in the strip
            texture[i].posw = 0;
            texture[i].posh = p->tot_h;
        }
    }
    return ((l.size()==0)?0:l[l.size()-1]->tot_h+l[l.size()-1]->h);
}
//Best-Fit Decreasing Height algorithm
int BFDH(int width, int n)
{
    int i, j;
    vector<level *> l;  //pointers to levels
```

```
    //find the best fit level whose residual width is minimum
    for (i = 0; i < n; i++)
    {
        int min_num = -1;   //num of min level
        int min_w=MIN;  //min width
        //check each level
        for (j = 0; j < l.size(); j++)
        {
            //if the texture can be put into level j
            if (l[j]->rem_w >= texture[i].w && (min_w > l[j]->rem_w - texture[i].w))
            {
                //update the min num and width
                min_num = j;
                min_w=l[j]->rem_w - texture[i].w;
            }
        }
        //if there exists a best level
        if (min_num>=0)
        {
            //calculate the position of new texture in the strip
            texture[i].posw = width-l[min_num]->rem_w;
            texture[i].posh = l[min_num]->tot_h;
            //update the remaining width
            l[min_num]->rem_w-=texture[i].w;
        }
        //else create a new level
        else{
            level *p = new level;
            //the new level's height=current texture's height
            p->h = texture[i].h;
            //calculate the total height of the new level
            p->tot_h = (l.size() == 0) ? 0 : l[l.size() - 1]->tot_h + l[l.size() - 1]->h;
            //update the remaining width
            p->rem_w = width - texture[i].w;
            //add p to l
            l.push_back(p);                                                 //calculate the
position of new texture in the strip
            texture[i].posw = 0;
            texture[i].posh = p->tot_h;
        }
    }
    return ((l.size()==0)?0:l[l.size()-1]->tot_h+l[l.size()-1]->h);
}
```

Testing Set Generating Script in C++

```
#define _CRT_SECURE_NO_WARNINGS
#include<bits/stdc++.h>
#include<iostream>
using namespace std;
int main() {
    srand(time(0));
    cout << "how many rectangles?" << endl;
    int n;
    cin >> n;
    cout << "Fixed width?" << endl;
    int fixWidth;
    cin >> fixWidth;
    cout << "Biggest height?" << endl;
    int mHeight;
    cin >> mHeight;
    char s[100];

    sprintf(s, "mkdir test%dtimeswith%d", n,fixWidth);
    system(s);
    sprintf(s, "test%dtimeswith%d/data.in", n, fixWidth);
    freopen(s, "w", stdout);
    cout << n << ' ' << fixWidth << endl;
    int i;
    for (i = 0; i < n; i++) {
        int width, height;
        width = rand() % fixWidth;
        height = rand() % mHeight;
        cout << width << " " << height<<" ";
    }
    fclose(stdout);
    sprintf(s, "test%dtimeswith%d/Readme.txt", n, fixWidth);
    freopen(s, "w", stdout);
```

```
    cout << "This testing file is generated randomly.\n" << "n=" << n << "   fixed width=" << fixWidth << "\n";
    fclose(stdout);
}

```

Correctness Testing Source in C++

```
#include <bits/stdc++.h>
using namespace std;
#define N 10005
struct Texture
{
    int w, h, posw, posh, left, right, top, bottom;
} texture[N];
//check whether r1, r2 includes
bool isInclude(Texture &r1, Texture &r2)
{
    return (r1.posw < r2.posw && r1.posh < r2.posh && r2.right <= r1.right && r2.bottom <= r1.bottom) ||
           (r2.posw < r1.posw && r2.posh < r1.posh && r1.right <= r2.right && r1.bottom <= r2.bottom);
}
//check whether r1, r2  overlaps
bool isRectOverlap(Texture &r1, Texture &r2)
{
    return !(((r1.right <= r2.left) || (r1.bottom >= r2.top)) ||
             ((r2.right <= r1.left) || (r2.bottom >= r1.top)));
}

int main()
{
    int i,j;
    int width, height, n;
    freopen("in.txt", "r", stdin);
    //input
    cin >> n;
    cin >> width >> height;
    for (i = 0; i < n; i++)
    {
        //input the position and size of each texture
        cin >> texture[i].w >> texture[i].h >> texture[i].posw >> texture[i].posh;
        texture[i].left = texture[i].posw;
        texture[i].right = texture[i].posw + texture[i].w;
        texture[i].top = texture[i].posh;
        texture[i].bottom = texture[i].posh + texture[i].h;

        //check whether the texture cross the boarder
        if (texture[i].left < 0 || texture[i].top < 0 || texture[i].right > width || texture[i].bottom > height)
        {
            cout << "Error!";
            return 0;
        }
    }
    //check whether there exists a pair of texture that overlaps or include one another
    for (i = 0; i < n; i++)
        for (j = 0; j < i; j++)
            if (isRectOverlap(texture[i], texture[j]) || isInclude(texture[i], texture[j]))
            {
                cout << "Error!!";
                return 0;
            }

    //true answer if not return 0
    cout << "True!!!";

    return 0;
}
```

Declaration

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

Duty Assignments

Programmer:
Tester:
Report Writer: