Project

Texture Packing

Group 17

Date:2022-05-20

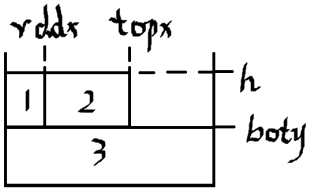
Chapter 1: Introduction

```
In class we have learnt approximation algorithms to solve binpacking problem in polynomial time. Here we are
to solve a 2-dimensional problem similar to binpacking - texture packing problem.
Given a maximum width and lots of rectangle textures, we shall fill these textures into a large texture with
that width and a fixed height. We should write approximation algorithms to get the minimum height while
fulfilling the limit, and compare their performance.
```

Chapter 2: Algorithm Specification

Data Generator

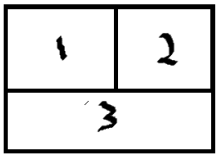
For the code, We'd prefer that you read the original code. Here we only tell that how we generate the data.



We first divide the optimal result into 3 parts, as above. Part 1 is only one rectangle, to guarantee the optimal height (Part 3 + Part 1). The remaining rectangles should be arranged in Part 2 and Part 3.

Then we divide Part 2 and Part 3 by another mothod.

- If that part can contain 3 or more rectangles:

  We divide it into 3 parts, and arrange each part at least one rectangle. Do recursion.
- If that part can contain 2 parts:
  We randomly divide it by width or by height.
- If that part can contain only 1 part:
  We store the width and height as the rectangle, then return.

Using above method, we can guarantee the fixed width and minimum height.

Our Algorithm: Next Fit Decreasing Height(NFDH) Algorithm

The idea of NFDH algorithm is like the Next Fit method of binpacking problem.

We consider the rectangles one by one in height-decreasing order. If one cannot fit the last block we have created, we create a new block with its height and the desired width, the block may add new blocks in some following steps. If the rectangle fit in the last block we have created, then we put it in that block.

Using height-decreasing order, we can guarantee each block could be checked easily by only considering the width(height will always be lower than any block we have considered).

```
Procedure NextFit(N:integer)
    vector< vector<int> > packing;
    vector<int> level;

    current_width:=0,total_height:=blocks[0].height;
    for i=0 To N-1 Step 1
        if current_width+blocks[i].width<=max_width then
        {
            level.push_back(blocks[i].width);
            current_width:=current_width+blocks[i].width;
        }
        else
```

```
        {
            packing.push_back(level);
            level.clear();
            level.push_back(blocks[i].width);
            total_height:=total_height+blocks[i].height;
            current_width:=blocks[i].width;
        }

    packing.push_back(level);
    Output total_height;

    for(auto i:packing)
        for(auto j:i)
            Output j;
```

- Approximation Ratio and Proof
  We can prove the Approxiamation Ratio of NFDH being 2.
  Let the demanded width being unit 1. A block means the rectangles we used to fill one width under a limited height.
  Consider the NFDH packing with blocks B_1, ,B_2 ..., B_t. For each i, let x_i be the width of the first rectangle in B_i and y_i be the total width of B_i. For each $i < t$, the first rectangle in B_{i+1} does not fit in B_i. Therefore $y_i + x_{i+1} > 1$. Since each rectangle in B_i has height at least H_{i+1}, and the first rectangle in B_{i+1} has height H_{i+1}, $A_i + A_{i+1} \geq H_{i+1}(y_i + x_{i+1}) > H_{i+1}$. Therefore, if A denotes the total area of all the rectangles,

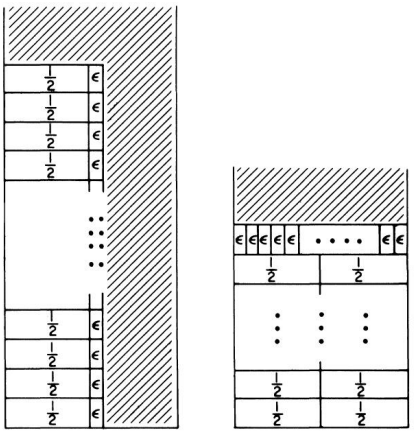
$$\begin{aligned} NFDH(L) = \sum_{i=1}^{t} H_i &\leqq H_1 + \sum_{i=1}^{t-1} A_i + \sum_{i=2}^{t} A_i \\ &\leqq H_1 + 2A \\ &\leqq 1 + 2OPT(L) \end{aligned}$$

  which is the desired bound.

Factors Affecting the Approximation Ratio

We can easily discover the Approximation Ratio is affected by the blocks. However, we cannot easily give out a simple mathmatic proof, so we just talk about the worst case of approximation. Also, we will give out a Block/Height-Approximation Ratio table in the Testing Results part.

Let the list L has n rectangles, where n is a multiple of 4. All the rectangles have height 1, the odd numbered ones have width $\frac{1}{2}$, and the even numbered ones have width ϵ, for a suitably small $\epsilon > 0$. The optimum and NFDH packings of L are shown in the figure. In this case we have $NFDH(L) = n/2$ and $OPT(L) = n/4 + 1$, so the ratio $NFDH(L)/OPT(L)$ can be made arbitrarily close to 2 by choosing n suitably large and ϵ suitably small.



First Fit Decreasing Height(FFDH) Algorithm

The idea of FFDH algorithm is like the First Fit method of binpacking problem.

We consider the rectangles one by one in height-decreasing order. If one cannot fit any block we have created, we create a new block with its height and the desired width, the block may add new blocks in some following steps. If the rectangle fit in any one of the blocks we have created(we choose the first), then we put it in that first block.

Using height-decreasing order, we can guarantee each block could be checked easily by only considering the width(height will always be lower than any block we have considered).

```
Procedure FirstFit(N:integer){
    vector< vector<int> > packing;
    vector<int> init;
    init.push_back(0);
    packing.push_back(init);
    total_height:=blocks[0].height;
    for i=0 To N-1 Step 1
            flag:=0;
        for(vector<int>& j:packing)
            if j[0]+blocks[i].width<=max_width then
      {
                j.push_back(blocks[i].width);
                j[0]:=j[0]+blocks[i].width;
                flag:=1;
                break;
            }
        if flag=0 then
        {
            vector<int> new_level;
            new_level.push_back(blocks[i].width);
            new_level.push_back(blocks[i].width);
            packing.push_back(new_level);
            total_height:=total_height+blocks[i].height;
        }
    Output total_height;
    for(auto i:packing)
        for(auto j:i)
            Output j;
```

Main Program

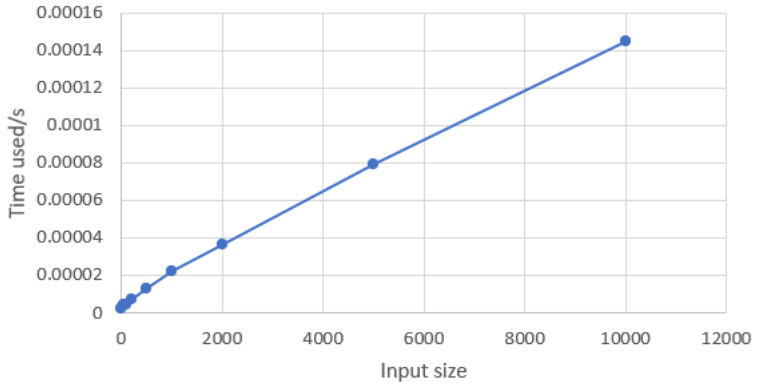
The main program is used to get the input, sort the rectangles by height-decreasing order, and call the two functions which indicates the different approximation algorithms.

```
Procedure main()
    INPUT max_width,N;
    T* blocks:=(T*)malloc(N*sizeof(struct texture));
    for i=0 To N-1 Step 1
        Output blocks[i].width,blocks[i].height;
    sort(blocks,blocks+N);

    NextFit(blocks,N);
    FirstFit(blocks,N);

    free(blocks);
    return 0;
```

Chapter 3: Testing Results

We use the fore-mentioned data generator to generate the test data randomly. We generate data by setting width and height 1e9. The approximation ratio corrects to 4 decimal places. Note that the ratios won't fit all conditions.

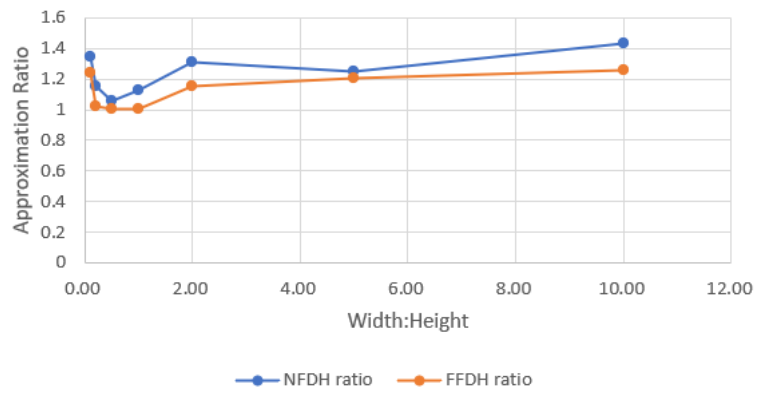
Input size	NFDH ratio	NFDH time(s)	FFDH ratio
10	1.0366	0.0000018	1.0366
20	1.0548	0.0000028	1.0474
50	1.0862	0.0000046	1.0089
100	1.4538	0.0000045	1.1385
200	1.1697	0.0000068	1.1359
500	1.1032	0.0000127	1.0487
1000	1.3111	0.0000220	1.1962
2000	1.5132	0.0000362	1.4776
5000	1.0391	0.0000789	1.0070
10000	1.2617	0.0001446	1.1504



The NFDH time didn't include the time used for the sort. We can see it's O(n). If we add up the sort, it's surely like O(nlogn).

For the analysis of factors affecting the approximation ratio performed, see the following table. We generate it by setting the width 120000, and number of blocks 100.

Width:Height	NFDH ratio	FFDH ratio
10:1	1.4299	1.2589
5:1	1.2500	1.2022
2:1	1.3150	1.1545
1:1	1.1242	1.0088
1:2	1.0595	1.0029
1:5	1.1494	1.0180
1:10	1.3485	1.2402



Notably, the 2 algorithms both perform quite well when Width:Height is around 0.5,the farther from that rate, the worse the result is. Also we can find that, FFDH always performs better than NFDH.

Chapter 4: Analysis and Comments

Space Complexity

- To store the basic information, we need $O(1)$ for constants and $O(n)$ for the rectangles.
- To solve the problem, we need $O(n)$ push_backs on std::vector, which occupies $O(n)$ space.
- In all, the space complexity is $O(n)$.

Time Complexity

- To use the FFDH or NFDH Algorithm, we need to arrange the rectangles in height-decreasing order. It uses $O(nlogn)$ time to deal a sort process.
- The FFDH or NFDH Algorithm both use $O(n)$ time to fill each rectangle into a large texture.
- In all, the time complexity is $O(nlogn)$.

Appendix(README.txt)

```
We suppose read the source file to get more comments. The files:
Data Generator - datagen.cpp.
Main Program - main.cpp.

The data generator may run a very long time if hard to generate the random data easily, wait, or choose a
larger width and height.
We recommend using TDM-GCC9.2.0 to compile,
the C++ standard shall be C++11 or above.
```