Texture Packing

Author: Shen Yunfeng / Zhao Yilei / Wang Kedi

Date: 2022/5/15

Chapter 1: Introduction

In our class, we have learned the Bin Packing problem, which aims to fit a sequence of items with the fewest number of one-dimension bins, and three approximation algorithms were given to solve that NP-hard problem. Furthermore, after class, the two-dimension problem is introduced to us, which is the Texture Packing problem.

In the Bin Packing problem, we're provided with a length of bin, and lots of items to be input into the bins with that length. And in Texture Packing problem, there's a texture with infinite length whose length is given, which is just like the bin. What's more, when it comes to things to be fitted into, this time a more dimension is provided, which is the width beside the height. The question requires us to pack the items(shaped rectangle) into the texture, requiring that we can't make the length larger than the provided width. And what's more, just like we're searching for the minimum number of bins, this time we want to make the height to be minimized.

In this project, we try to approximate the problem's answer with the Next Fit algorithm.

Chapter 2: Data Structure / Algorithm Specification

The question is very easy to understand, and the main objects are easy to tell. We designed two classes, which are the bin and rec, corresponding to the texture and items (shaped rectangle) in the problem. What's more, here the bin is divided into several parts, each one contains a line of items.

Data Structure

1. bin(Texture in the problem)

```
class bin         //基于该层第一个长方形建立的桶
{
  public:
  int ban_x{-1};  //已占用宽度
  int height{-1}; //桶高

  bin(){};      //default constructor
  bool check(rec& r);//当前bin是否能装下长方形r，允许旋转
  void insert(rec& r);
};
```

Here the bin is not completely the same as that one with infinite length in the problem. Here the bin has its own height, which is the largest one among this line's items. In other words, each time we keep putting items into a bin until its width is completely occupied. Therefore, there must be one item whose height is the largest. Continually, we keep putting items into bins, with each bin having its height, until all the items have been packed. After that, we pile the bins up, and thus get a whole bin which we want.

2. rec(Items in the problem)

```
class rec//长方形
{
  public:
  int width;
  int height;

  rec() = delete;
  rec(int w, int h) : width(w), height(h) {};
};
```

There's nothing more to say about that. Each object is an item with its 'width' and 'height'.

Algorithm Specification:

Sketch

The algorithm is quite simple to understand. It is just like the Next Fit algorithm in the Bin Packing problem, except that this time it is two dimension. The main idea has been shown in the ***Data Structure*** Part, which is to fitting items into one bin randomly until the remaining width can't contain the next item. If so, we'll use another bin to contain the item, and so on. Finally, the sum of the heights is what we want.

Pseudo Code:

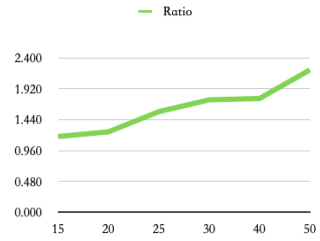
```
while(the items haven't all been put)
{
    if(this bin can occupt this item)
    {
        put the item into this bin;
        update the remaining width;
        update the bin's height if necessary;
    }
    else
    {
        create a new bin;
        if(still can't occupy the item)
           error: The items can't be packed;
        else
        {
            put the item into the bin;
            update the height and width;
        }
    }
}
```

Chapter 3: Testing Result

The relations between the approximation ratio and the width of the strip

As we know, the approximation ratio has the relationship with the width of the strip, because the width will terminate how many rectangles can be inserted in this level. Therefore, we create some samples to test the relation of them. There are samples containing 10, 100, 1000 records, which is generated randomly.

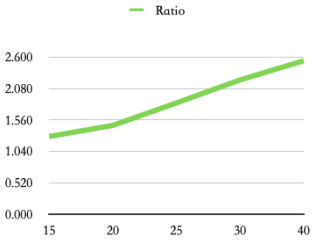
1. We use the sample of 10 records.



The relation between approximation ratio and width of 10 samples.

Width	Actual result/height	Best answer	Ratio
15	72	61.08	1.178781925
20	58	46.35	1.251348436
25	58	37.08	1.564185545
30	54	30.90	1.747572816
40	41	23.18	1.768766178
50	41	18.54	2.211434736

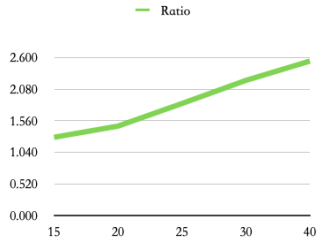
2. We use the sample of 100 records.



The relation between approximation ratio and width of 100 samples.

Width	Actual result/height	Best answer	Ratio
40	1,518	1,180.40	1.286004744
50	1,388	944.32	1.469840732
100	870	472.16	1.842595730
150	700	314.77	2.223845983
200	600	236.08	2.541511352

3. We use the sample of 1000 records.



The relation between approximation ratio and width of 1000 samples.

Width	Actual result/height	Best answer	Ratio
100	32,157	25,380.93	1.266974851
120	30,675	21,150.78	1.450301124
150	27,259	16,920.62	1.610992978
200	22,200	12,690.47	1.749344193
300	16,977	8,460.31	2.006664058

The relations between the approximation ratio and the different distributions of widths and heights

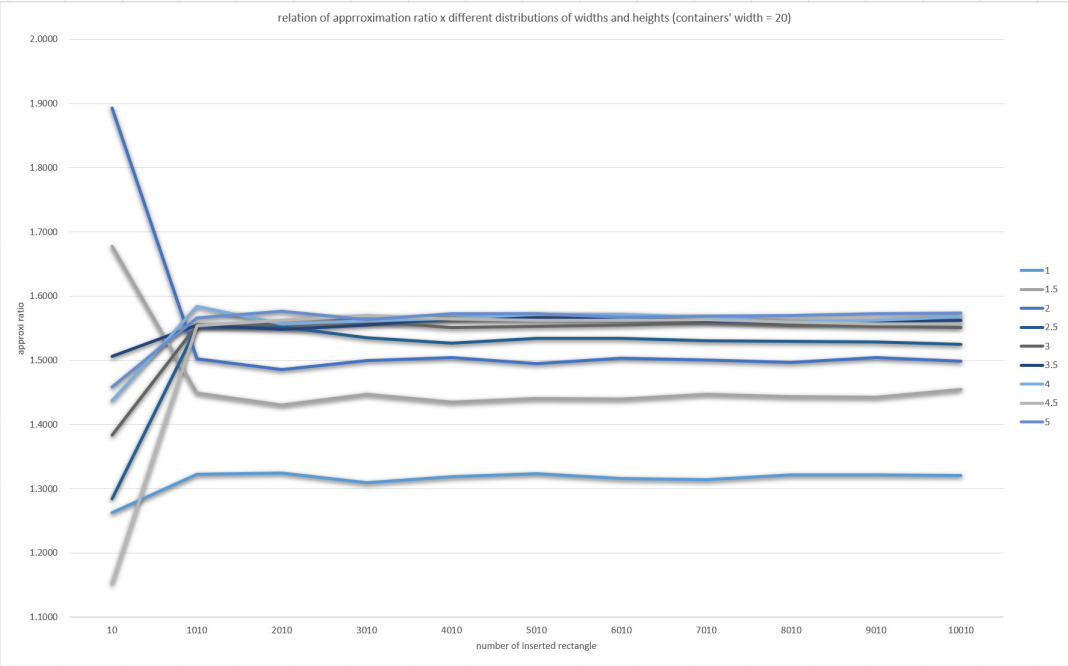
As the diff3rent distributions of widths and heights would affect the approximate ratio, for example, if the height of the rectangle is far bigger than its width, the approximate of Next-Fit Algorithm would be pretty large, as it always creates a new shelf when the current shelf could fit the new rectangle, and for the height of new rectangle is extremely large, the height of new shelf would be large alike.

In this test, we use random number generator to generate test cases, and renew the seed by function `srand(time(0))` to attain different cases each time. In order to observe how the different distributions of widths and heights would influence the ratio, we restrict the MAX_Height of rectangle = h/w * w_container each time.

Test on inserting 10, 1010, 2010, ..., 10010 rectangles when the width of container equals to 20, we can obtain the table as below:

n - the number of rectangles to insert; h/w - MAX_ Height of rectangle / width of container

		h/w								
		1	1.5	2	2.5	3	3.5	4	4.5	5
n	10	1.2630	1.6774	1.8929	1.2842	1.3839	1.5060	1.4369	1.1524	1.4583
	1010	1.3230	1.4491	1.5022	1.5523	1.5554	1.5546	1.5837	1.5543	1.5663
	2010	1.3250	1.4305	1.4857	1.5523	1.5558	1.5485	1.5572	1.5626	1.5770
	3010	1.3093	1.4476	1.4995	1.5355	1.5603	1.5548	1.5620	1.5701	1.5638
	4010	1.3192	1.4353	1.5042	1.5270	1.5513	1.5649	1.5657	1.5640	1.5724
	5010	1.3233	1.4405	1.4947	1.5340	1.5527	1.5660	1.5722	1.5623	1.5725
	6010	1.3158	1.4398	1.5033	1.5345	1.5552	1.5653	1.5720	1.5629	1.5669
	7010	1.3145	1.4473	1.5005	1.5308	1.5583	1.5617	1.5673	1.5632	1.5689
	8010	1.3220	1.4435	1.4965	1.5294	1.5555	1.5601	1.5596	1.5601	1.5697
	9010	1.3218	1.4423	1.5045	1.5292	1.5531	1.5610	1.5628	1.5567	1.5726
	10010	1.3207	1.4546	1.4991	1.5246	1.5514	1.5626	1.5676	1.5572	1.5734
AVERAGE		1.3143	1.4643	1.5348	1.5121	1.5394	1.5550	1.5552	1.5241	1.5602



Chapter 4: Analysis and Comments

The relations between the approximation ratio and the width of the strip

We can find in 3 pictures that all the slopes are positive, which means that the sample has larger width, the approximation ratio will be much bigger. That because the next-fit algorithm cannot deal with the situation that there are so many rectangles in the same level. Instead, it is easy to get more accurate answer when the width of the strip is small.

And also, the judgement of choosing which side of rectangles to be the width will affect the result of our tests. So we decide to let the longer side of the rectangle to be the width, the shorter side to be the height. There are many other factors that will affect our results, not only the width of the strip.

The relations between the approximation ratio and the different distributions of widths and heights

We can easily draw the conclusion from the table and plot that, the approximate ratio increases with the increment of both input size n and parament h/w.

For the same h/w, the ratio grows with the input size, but the worst case is around 1.5 and never exceed 1.6, which is close to the best case —— 1.5.

And for the same input size, we can see that the ratio grows when h/w is large. That means when the height of rectangle is much bigger than its width and both of them are larger than the width of the container, the Next-Fit Algorithm is tend to create many shelves with large height and can't fit many rectangles in single shelf, that would waste a lot of space, and thus leading to the increment of approximate ratio.

Appendix: Source Code

```
#include <iostream>
#include <vector>
#include <ctime>
#define MAX(x,y) (x>y)?x:y
#define MIN(x,y) (x<y)?x:y
using namespace std;
void result_next_fit(vector<int> a, vector<int> b);

int w;   //容器宽度
int n;   //待插入长方形个数

class rec//长方形
{
  public:
  int width;
  int height;

  rec() = delete;
  rec(int w, int h) : width(w), height(h) {};
};


class bin        //基于该层第一个长方形建立的桶
{
  public:
  int ban_x{-1};  //已占用宽度
  int height{-1}; //桶高

  bin(){};

  bool check(rec& r)//当前bin是否能装下长方形r，允许旋转
  {
    // cout << "@check" << endl;
    if     (this->ban_x+r.width<=w  && this->height>=r.height) return
true;
    else if(this->ban_x+r.height<=w && this->height>=r.width)//rotate
    {
      // cout << "  $ROTATE" << endl;
      int tmp = r.width;
      r.width = r.height;
      r.height = tmp;
      return true;
    }
```

```
    return false;
  }
  bool insert(rec& r)
  {
    // cout << "@insert" << endl;
    if(this->height==-1)//该桶首次插入长方形，初始化桶高=长方形的较短边
    {
      // cout << "  #ini" << endl;
      int minn = MIN(r.width, r.height);
      int maxx = MAX(r.height, r.width);
      if(maxx<w)
      {
        this->height = minn;
        this->ban_x =  maxx;
      }
      else
      {
        this->height = maxx;
        this->ban_x =  minn;
      }
      return true;
    }
    else if(this->check(r))//插的下，更新已占用桶宽
    {
      this->ban_x += r.width;
      return true;
    }
    return false;
  }

};


int main()
{

  int t;              //待插入长方形个数
  vector <int> a, b;
  cout << "Please input the width of the texture: ";
  cin >> w;
  cout << "Please input the number of rectangle: ";
  cin >> n;
  //生成随机数
  srand(time(0));
  for(int i=0; i<n; i++)
  {
    t = rand()%w + 1;
    // cout << "insert rec[" <<  t << ", ";
    a.push_back(t);
    t = rand()%(int)(5*w);
    // cout << t <<"]" << endl;
    b.push_back(t);
```

```
  }
  result_next_fit(a,b);

  system("pause");
  return 0;
}
void result_next_fit(vector<int> a, vector<int> b)
{
  double sum = 0;
  vector <bin*> shelf;//所有桶
  shelf.push_back(new bin());//首个桶
  for(int i = 0; i < n; i++)
  {
    sum += a[i]*b[i];
    rec cur_r(a[i],b[i]);
    if(shelf.back()->insert(cur_r)==false)//当前桶插不下，新建桶并重新插入
    {
      shelf.push_back(new bin());
      shelf.back()->insert(cur_r);
    }
  }
  //输出结果
  int tot = 0;
  int n_bin = shelf.size();
  for(int i = 0; i < n_bin; i++)
  {
    // cout << "height[" << i <<"] = " << shelf[i]->height << endl;
    tot += shelf[i]->height;
  }
  cout << "total height = " << tot << endl;
  cout << "ratio = " << tot/(sum/w) << endl;
}
```

References:

None.

Declaration:

We hereby declare that all the work done in this project titled "Huffman Codes" is of our independent efforts a group.