Lehigh University
Department of Computer Science and Engineering
CSE441 Advanced Algorithms
Spring 2013

EXPERIMENTS ON TWO SPECIAL CASES OF RECTANGLE PACKING PROBLEM

Yi Luo
yil712@lehigh.edu

ABSTRACT

In this experiment report, we have a detailed discussion on two special cases of rectangle packing problem. The first case has restrictions on the general packing rule so that the new rectangle can only be packed against the enclosing rectangle of all previous rectangles. Our experiment shows that this case is not easy to solve and our proposed search algorithm cannot properly handle too many rectangles. In the second case, we pack only four rectangles. This is a problem coming from IOI1995 and we discuss a naïve method and an improved search algorithm to tackle it. Our experiment shows that this case is relatively easy to handle and our proposed algorithm work efficiently.

1. INTRODUCTION

Given a number of rectangles with different sizes, how to pack them in an optimal way so that the final enclosing rectangle has the minimum area? This is an interesting problem and we are hoping to explore it to a certain degree in this experimental report. Figure.1 shows two example structures of packing three rectangles. It is easy to recognize that structure (a) has a smaller enclosing rectangle than structure (b).

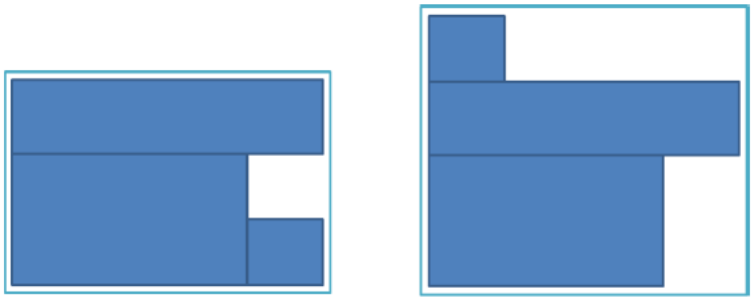


Figure.1 Two example structures of packing three rectangles

When the number of rectangles is small, we may enumerate all possible packing structures and figure out the optimal one. However, when the number of rectangles increases to a large scale, it turns to be unrealistic to enumerate all results. People are allowed to pack these rectangles in any ways and there is no limitation on the rule, the order and the orientation of packing process. Facing such a difficult problem, we don't directly solve it in the most general perspective; instead, we firstly add some restrictions to make it more specific. By dealing with these special cases, we are able learn some meaningful illumination so as to have a better understanding of the original problem.

In this report, we are going to discuss two special cases of the initial rectangle packing problem. The first case has constraints on the packing rule, which requires that whenever the next rectangle is packed, it can only be put against the enclosing rectangle of all previous rectangles. The second case has no constraint on the packing rule but restricts the number of rectangles to be a very small scale which is exactly four. Both cases are more specific than the original problem. Our experiments reveal that, the first case is pretty difficult to scale, while the second case is relatively easy to handle due to the small size of input. Let's move on to the detailed discussions.

2. CASE ONE: RESTRICTIONS ON THE PACKING RULE

2.1 Problem description

In this case, we restrict that "*the n+1 rectangle can only be packed against to the enclosing rectangle of all previous 1to n rectangles*$(n \geq 1)$". For example, in figure.2, we need to pack 4 rectangles in ascending order. As it shows, the 3rd rectangle can only be packed against the enclosing rectangle (in blue) of the 1st and 2nd ones. After putting the 3rd one, the previous

three rectangles generate a new enclosing rectangle (in red). The 4^{th} rectangle then can only be appended against the red lines, which produces the final enclosing rectangle (in green).

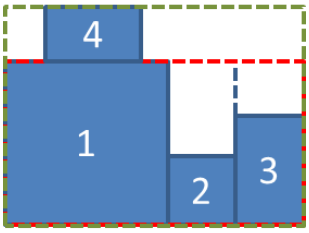


Figure.2 An example packing structure of 4 rectangles under case 1

It is apparent that different final packing structures have different area values, like the initial problem, we want to get the solution with the smallest enclosing rectangle. Figure.3 displays two ways of packing three rectangles; we definitely prefer structure (b).

Figure.3 Two packing structures of 3 rectangles under case 1

Before we start to explore this problem, we firstly define it in a formal description as follows.

Case 1: Rectangle Packing Problem with Restrictions on the Packing Rule

Given K rectangles with different sizes, try to figure out the smallest enclosing (new) rectangle into which these rectangles may be fitted without overlapping. Meanwhile, the following conditions should be satisfied:

a. The heights and widths of all rectangles are integers;
b. The $n+1$ rectangle can only be packed against the enclosing rectangle of all previous 1 to n rectangles ($n \geq 1$);
c. The orientation of any rectangle is fixed.

Input:

The first line is an integer K ($K \geq 1$) which indicates the number of rectangles to be packed. In the following K lines, each line contains two integers which specify the height h ($h \geq 1$) and width w ($w \geq 1$) of a rectangle.

Output:

The only one line is an integer indicating the area of the smallest enclosing rectangle.

Sample Input:

```
7
10 2
8 3
```

```
2 8
3 5
5 2
2 3
1 1
```

Sample Output:

```
100
```

2.2 Preliminary ideas

Seen from this description, we notice that the possible packing structures have been restricted since we don't need to care about float sizes or any rotations of rectangles. Also, some layouts of packing can be unified to the same one if we apply the reflection. For example, in figure.4, both directions of packing the 2nd one to the previous rectangle just create the same enclosing rectangle. We treat them as the same and call it attaching a new rectangle "on the width".

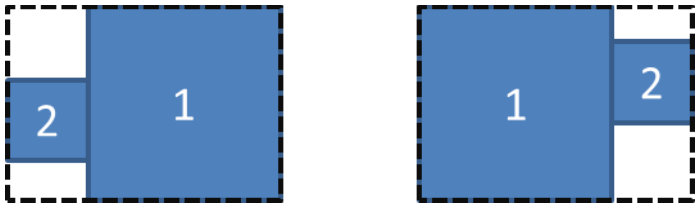


Figure.4 Two ways of attaching a new rectangle "on the width"

Similarly, in figure.5, two directions of attaching a new rectangle "on the height" can also be regarded as the same.

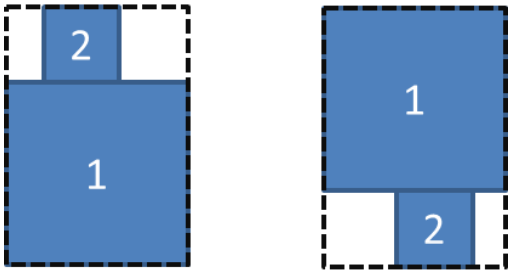
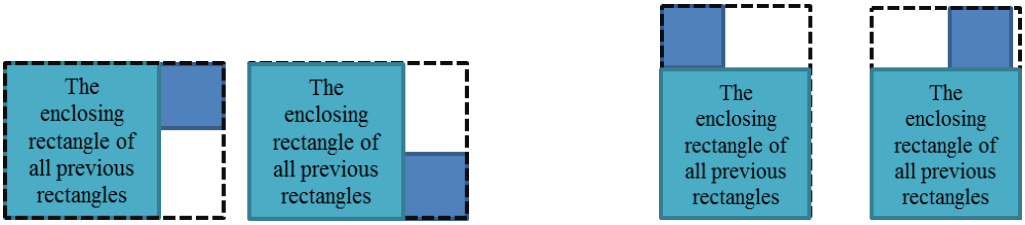


Figure.5 Two ways of attaching a new rectangle "on the height"

Additionally, as shown in figure.6, all possible ways of attaching a new rectangle on width can be further unified to the same one, we define it as attaching rectangles "on the right"; all possible ways of attaching a new rectangle on height can be unified to the same one, we define it as attaching rectangles "on the top".



(a) (b)

Figure.6 Different ways of packing a new rectangle can be unified into only two categories

Therefore, each time when adding a new rectangle, we only have **two options** – attaching it *on the right* or *on the top*. Then we are able to think of solutions based on such formalized packing rules. We now try various algorithms learnt from the CSE441 course to tackle it.

2.3 Does the greedy algorithm work?

At first glance, it seems there should be a greedy strategy: to get an optimal structure at each step then we may get a global optimal solution at last.

2.3.1 Greedy algorithm attempt 1

The first greedy idea is to get the immediate smallest enclosing rectangle every time after we attach a new rectangle. The intuition is *to get the smallest one at each step will result in the smallest one at last*. The algorithm can be described as follows:

```
procedure greedyPackingAttempt1(int K, Rectangle[] rectList)
Input: the number of rectangles K and the list of rectangles (each element contains the height and width)
Output: the area of the minimum enclosing rectangle

while rectList.size > 1:
      int minArea = a very large integer;
      Rectangle newRect = new Rectangle();
      Rectangle r1, r2;
      int H, W, A;
      for each binary-combination (a, b) from K rectangles:
            //put b on the top of a
            H  = a.height + b.height;
            W = max(a.width, b.width);
            A = H * W;
            if (A < minArea):
                  minArea = A;
                  newRect.height = H;
                  newRect.width = W;
                  r1 = a;
                  r2 = b;
            //put b on the right of a
            H  = max(a.height, b.height);
            W = a.width + b.width;
            A = H * W;
            if (A < minArea):
                  minArea = A;
                  newRect.height = H;
                  newRect.width = W;
                  r1 = a;
                  r2 = b;
```

```
        remove r1 and r2 from rectList;
        add newRect to rectList;
```

This procedure detects the smallest enclosing rectangle each time when attaching a new rectangle to the existing ones. However, this greedy idea does not work correctly because it cannot guarantee the global optimization at last. As illustrated in figure.7, the above greedy algorithm produces the packing structure (a) which has area = 33; however, the optimal packing structure should be (b) which has area = 22.

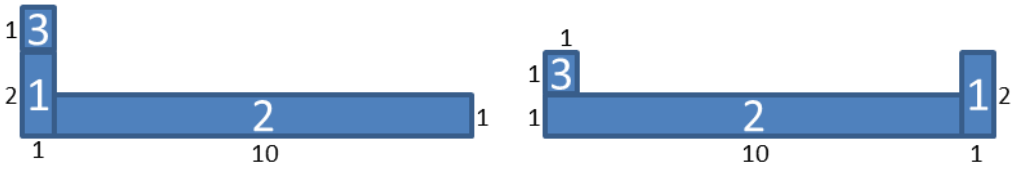


Figure.7 The packing structure (a) found by greedyPackingAttempt1 and (b) the optimal one

This counterexample beats our first attempt to solve the problem via greedy algorithms. So we start to think of another greedy strategy.

2.3.2 Greedy algorithm attempt 2

This greedy strategy attempts to get the smallest wasted area introduced by a new rectangle. It is because every time when we pack a new rectangle, the newly enclosing rectangle may waste some space. The intuition is that *to guarantee the minimum wasted area at each step will result in the smallest rectangle at last*. The algorithm can be described as follows:

```
procedure greedyPackingAttempt2(int K, Rectangle[] rectList)
Input: the number of rectangles K and the list of rectangles (each element contains the height and width)
Output: the area of the minimum enclosing rectangle

while rectList.size > 1:
      int minWastedArea = a very large integer;
      Rectangle newRect = new Rectangle();
      Rectangle r1, r2;
      int H, W, wastedArea;
      for each binary-combination (a, b) from K rectangles:
            //put b on the top of a
            H  = a.height + b.height;
            W = max(a.width, b.width);
            figure out wastedArea;
            if (wastedArea < minWastedArea):
                  minWastedArea = wastedArea;
                  newRect.height = H;
                  newRect.width = W;
                  r1 = a;
```

```
            r2 = b;
        //put b on the right of a
        H  = max(a.height, b.height);
        W = a.width + b.width;
        figure out wastedArea;
        if (wastedArea < minWastedArea):
            minWastedArea = wastedArea;
            newRect.height = H;
            newRect.width = W;
            r1 = a;
            r2 = b;
remove r1 and r2 from rectList;
add newRect to rectList;
```

Unfortunately, this attempt of greedy algorithm fails as well. Figure.7 just serves as the counterexample again: to save the wasted time at each step, we may firstly pack the 1st and 3rd rectangles and finally get the packing structure (a); however, the optimal one is still (b).

Both greedy algorithm attempts just fail. Why?

It is because the area of the current enclosing rectangle is not only decided by 1) the area of the previous enclosing rectangle, but also decided by 2) the wasted area introduced by the new rectangle. However, neither 1) nor 2) are fixed. They cannot be fulfilled at the same time and they are always dependent on each other. Therefore, we cannot find the global optimal result based any single optimization of them.

2.4 Let's try the search algorithm!

Consequently, we start to consider several search ideas. In this specific case, the packing rule has been restricted, making it easier to list all possible structures at each step when attaching a new rectangle. We firstly try the brute force search.

2.4.1 The brute force search

In this problem, different orders of packing these rectangles will result in different final enclosing rectangle. For n rectangles, there are $n!$ possible permutations; each time when attaching a new rectangle, there are 2 different options (*on the right* or *on the top*). As a result, there are in total $2^n n!$ possibilities to search. Figure.8 shows the partial searching tree and corresponding packing structures for the result lists.

We use a "x: y" notation for each packing step where x refers to the number of the rectangle and y indicates the option of the packing rule (1 stands for packing on the top, 0 stands for packing on the right).

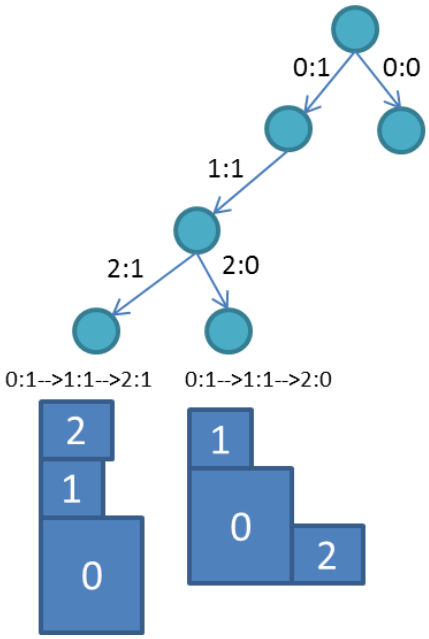


Figure.8 The partial searching tree and two complete results

We've implemented this brute force search idea in Java (the *PackSearch1.java* in the attachment) and get following outputs for the example in figure.7.

```
0:1-->1:1-->2:1-->Area = 40
0:1-->1:1-->2:0-->Area = 33
0:1-->1:0-->2:1-->Area = 33
0:1-->1:0-->2:0-->Area = 24
0:1-->2:1-->1:1-->Area = 40
0:1-->2:1-->1:0-->Area = 33
0:1-->2:0-->1:1-->Area = 30
0:1-->2:0-->1:0-->Area = 24
0:0-->1:1-->2:1-->Area = 40
0:0-->1:1-->2:0-->Area = 33
0:0-->1:0-->2:1-->Area = 33
0:0-->1:0-->2:0-->Area = 24
0:0-->2:1-->1:1-->Area = 40
0:0-->2:1-->1:0-->Area = 33
0:0-->2:0-->1:1-->Area = 30
0:0-->2:0-->1:0-->Area = 24
1:1-->0:1-->2:1-->Area = 40
1:1-->0:1-->2:0-->Area = 33
1:1-->0:0-->2:1-->Area = 33
1:1-->0:0-->2:0-->Area = 24
1:1-->2:1-->0:1-->Area = 40
1:1-->2:1-->0:0-->Area = 22
1:1-->2:0-->0:1-->Area = 33
1:1-->2:0-->0:0-->Area = 24
1:0-->0:1-->2:1-->Area = 40
1:0-->0:1-->2:0-->Area = 33
1:0-->0:0-->2:1-->Area = 33
```

```
1:0-->0:0-->2:0-->Area = 24
1:0-->2:1-->0:1-->Area = 40
1:0-->2:1-->0:0-->Area = 22
1:0-->2:0-->0:1-->Area = 33
1:0-->2:0-->0:0-->Area = 24
2:1-->0:1-->1:1-->Area = 40
2:1-->0:1-->1:0-->Area = 33
2:1-->0:0-->1:1-->Area = 30
2:1-->0:0-->1:0-->Area = 24
2:1-->1:1-->0:1-->Area = 40
2:1-->1:1-->0:0-->Area = 22
2:1-->1:0-->0:1-->Area = 33
2:1-->1:0-->0:0-->Area = 24
2:0-->0:1-->1:1-->Area = 40
2:0-->0:1-->1:0-->Area = 33
2:0-->0:0-->1:1-->Area = 30
2:0-->0:0-->1:0-->Area = 24
2:0-->1:1-->0:1-->Area = 40
2:0-->1:1-->0:0-->Area = 22
2:0-->1:0-->0:1-->Area = 33
2:0-->1:0-->0:0-->Area = 24
The total searching time is: 48
The smallest area is: 22
```

This brute force method searches $2^3 * 3! = 48$ times. This is acceptable when n is small. However, when n increases a little bit, say $n = 7$, the total searching time turns to be $2^7 * 7! = 645120$ which is dramatically augmented from 48. In fact, the running time of the brute force search algorithm is $O(2^n n!)$ which is definitely intractable.

2.4.2 Intelligent exhaustive search - backtracking

However, it is possible to apply some smart searching techniques to prune the searching tree so as to reduce the total searching time. One good choice is the backtracking as in figure.9.

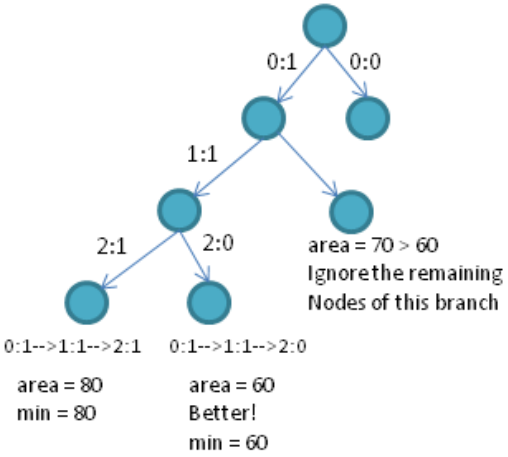


Figure.9 The search algorithm optimized by backtracking

In fact, when we firstly encounter a complete solution, we record this result area as the minimum solution. Then, when we go through other branches, we just ignore that branch if we notice that the current area has been larger than the minimum one and backtrack to search other branches. We keep updating the minimum result whenever getting a better complete result. This idea has also been implemented in the *PackSearch1.java* file. Here is the result acquiring by this backtracking approach upon the example in figure.7:

```
0:1-->1:1-->2:1-->Area = 40, which is better.
0:1-->1:1-->2:0-->Area = 33, which is better.
0:1-->1:0-->2:0-->Area = 24, which is better.
1:1-->2:1-->0:0-->Area = 22, which is better.
The total searching time is: 4
The smallest area is: 22
```

The searching time reduces from 48 to 4. We also carry on an experiment on 7 rectangles with the following input file.

Input:

```
7
10 2
8 3
2 8
3 5
5 2
2 3
1 1
```

The backtracking search has results as follows.

```
0:1-->1:1-->2:1-->3:1-->4:1-->5:1-->6:1-->Area = 248, which is better.
0:1-->1:1-->3:1-->4:1-->5:1-->6:0-->2:1-->Area = 240, which is better.
0:1-->1:1-->3:1-->4:1-->5:0-->2:1-->6:1-->Area = 232, which is better.
0:1-->1:1-->3:1-->4:0-->2:1-->5:1-->6:1-->Area = 208, which is better.
0:1-->1:1-->3:1-->4:0-->5:1-->6:0-->2:1-->Area = 200, which is better.
0:1-->1:1-->4:0-->3:1-->5:0-->2:1-->6:1-->Area = 192, which is better.
0:1-->1:0-->2:1-->3:1-->4:1-->5:1-->6:1-->Area = 184, which is better.
0:1-->1:0-->2:1-->3:1-->4:0-->5:1-->6:1-->Area = 180, which is better.
0:1-->1:0-->3:1-->4:1-->5:1-->6:0-->2:1-->Area = 176, which is better.
0:1-->1:0-->3:1-->4:1-->5:0-->2:1-->6:1-->Area = 168, which is better.
0:1-->1:0-->3:1-->4:0-->2:1-->5:1-->6:1-->Area = 144, which is better.
0:1-->1:0-->3:1-->4:0-->5:1-->6:0-->2:1-->Area = 136, which is better.
1:1-->4:0-->3:1-->0:0-->2:1-->5:1-->6:1-->Area = 128, which is better.
1:1-->4:0-->3:1-->0:0-->5:1-->6:0-->2:1-->Area = 120, which is better.
4:1-->5:1-->1:0-->3:1-->0:0-->2:1-->6:1-->Area = 112, which is better.
4:1-->5:1-->6:1-->1:0-->3:1-->0:0-->2:1-->Area = 104, which is better.
5:1-->6:1-->4:0-->3:1-->1:0-->2:1-->0:0-->Area = 100, which is better.
The total searching time is: 17
The smallest area is: 100
```

Fortunately, the searching time reduces from 645120 to merely 17. Figure.10 exhibits the optimal solution and it looks good☺.

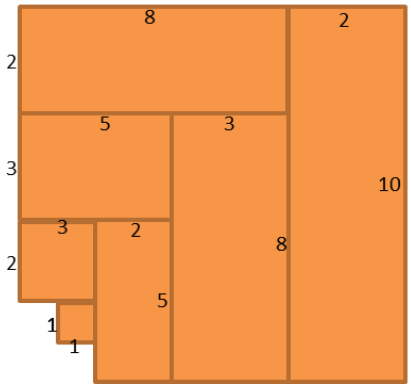


Figure.10 The optimal solution of an experiment on 7 rectangles

2.5 Conclusions

In this special case, we restrict the packing rule of the original problem so that the new rectangle can only be packed against the enclosing rectangle of all previous rectangles. As a result, various packing options are unified to only two options – on the right or on the top of the previous enclosing rectangle. We start from two greedy strategies to acquire global optimal result based on local optimization. However, our greedy algorithm attempts do not work. Then we try the search algorithm and implement the brute force search for a preliminary exploration. To improve the search performance, we further bring in an intelligent search algorithm – backtracking - to prune the searching tree and reduce the search time. Fortunately, our experiments reveal several good improvements.

However, the search algorithm even optimized with backtracking strategy is still not scalable. The running time is highly dependent on the testing data. In the worst case, the running time is the same as the brute force search algorithm.

3. CASE TWO: RESTRICTIONS ON THE NUMBER - ONLY 4 RECTANGLES

3.1 Problem description

In this case, we restrict that "*there are only 4 rectangles to be packed*" and there is no limitation on the order, the rule and the orientation. Consequently, this specific case is almost the same as the original problem except that the number of rectangles is only 4.

Before we start to explore this problem, we firstly define it in a formal description as follows. In fact, this is a revision of a programming competition problem coming from IOI 1995.

Case 2: Rectangle Packing Problem with 4 rectangles [IOI1995]

Four rectangles are given. Find the smallest enclosing (new) rectangle into which these four may be fitted without overlapping. By smallest rectangle we mean the one with the smallest area. All four rectangles should have their sides parallel to the corresponding sides of the enclosing rectangle.

Input:

The input consists of four lines. Each line describes one given rectangle by two positive integers: the lengths of the sides of the rectangle. Each side of a rectangle is at least 1 and at most 50.

Output:

The single line contains an integer: the minimum area of the enclosing rectangles.

Sample Input:

1 2
2 3
3 4
4 5

Sample Output:

40

3.1 Preliminary ideas and a naive solution

Since we only need to pack 4 rectangles, it appears pretty easy to enumerate all possible packing structures and figure out the smallest candidate. Here comes the most straightforward idea.

Because we know the maximum length of a rectangle is less than 50, then the height and width should respectively be less than 4*50 = 200. So the eventual enclosing rectangle has a size at most $N \times N$, where $N \leq 200$.

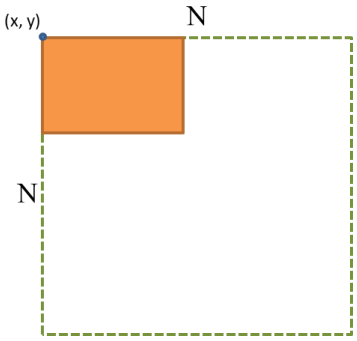


Figure.11 For each rectangle, we put them in the enclosing rectangle in every possible position

As figure.11 shows, we can simply put each rectangle into the dashed enclosing square and try them at all possible positions (x, y). The algorithm is as follows.

```
for x = 1 to N:
      for y = 1 to N:
            put this rectangle at position (x,y)
```

We just deal with all four rectangles in the same method and each of them can be rotated at every position. After packing all four rectangles, we just verify whether there are intersections. If there is no intersection, we call the function to calculate the area of the enclosing rectangle and update the optimal solution. For the attempts of all possible positions we have to run $N^{2\times4}$ times

and for the rotations of each rectangle we have to try 2^4 times. In total, we have to run $16 \times N^8$ times where $N \leq 200$. This is rather time-consuming.

We may as well apply some smart search strategies, such as backtracking, to improvement this naive algorithm. Some good ideas are:

a. Ignore the remaining work if the area of the current enclosing rectangle has exceeded the minimum area.
b. Ignore the remaining work if the first two or three rectangles have already caused intersections.

However, the adapted algorithm still costs too much time especially when N increases to a large scale. [P. Kluit, IOI1995]

3.2 If we know all possible layout, we may have a better solution

In fact, the original problem description in IOI 1995 has one more condition which provides all possible layouts of packing four rectangles. Figure.12 shows the whole six possible layouts and people are able to get other packing structures merely based on these six layouts via reflection, rotation or different packing permutation of rectangles.

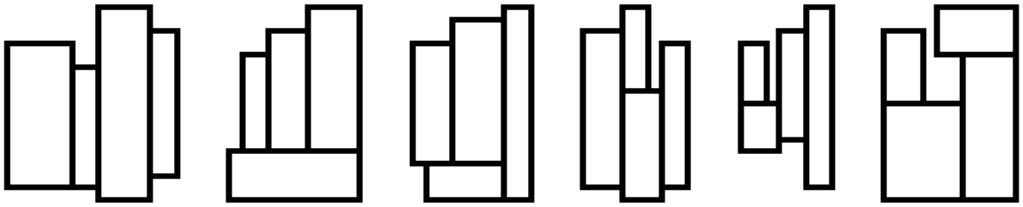
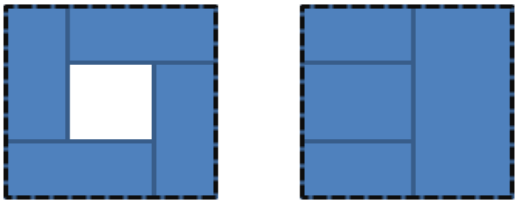


Figure.12 Six possible layouts of packing four rectangles [IOI1995]

However, the competition problem description fails to give the reason why there are only six possible layouts. Before we directly adopt this condition, we firstly have an informal discussion about how to get these six representative layouts. One idea is to list all layouts and unify equivalent ones to a representative layout. But we would like to inspect it in a more general perspective as follows. Actually, we notice that there are two categories of partitions for the final enclosing rectangle into four rectangles:

a. To pack them according to layout (a) in figure.13; each rectangle occupies a corner of the enclosing rectangle.
b. To pack them according to layout (b) in figure.13; the final enclosing rectangle is partitioned into two parts and one of them is further partitioned for the other three rectangles.



(a) (b)

Figure.13 Two ways of partitioning the final enclosing rectangle

Layout (a) is a representative layout in figure.12. As for layout (b), it has several variations. The left part can only be further partitioned into two parts, in which one is a single rectangle while the other one contains exactly two rectangles. There is only one way to partition a rectangle into two parts as shown in figure.14.

Figure.14 The only way to partition a single rectangle into two ones

To get the left side of layout (b) in figure.13, there are two methods, as shown in figure.15. Then, we are able to get layout (b) according to figure.16 in five different ways.

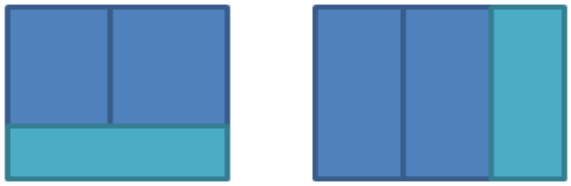


Figure.15 Two ways of getting three rectangles in one enclosing rectangle

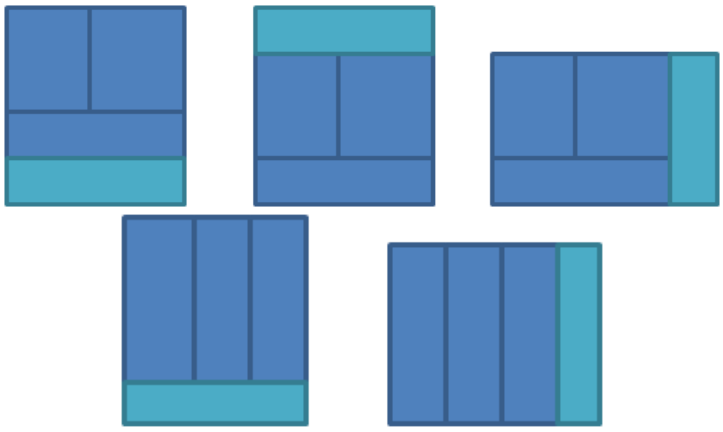


Figure.16 Five variations of layout (b) in figure.13

Overall, we get the 6 representative layouts for packing four rectangles. All the other packing structures can be obtained merely based on these 6 basic layouts.

Then we provide a search algorithm to enumerate all candidate packing structures and figure out the optimal result. For the order of packing rectangles, there are in total $4! = 24$ permutations. Each rectangle has two orientations, in all there are $2^4 = 16$ possibilities. In addition, there are 6 basic layouts, so the total search time is $24 \times 16 \times 6 = 2304$.

To compute the area of the enclosing rectangle of each layout, we get their heights and widths as follows.

Height = max(h1, h2, h3, h4)
Width = w1+w2+w3+w4

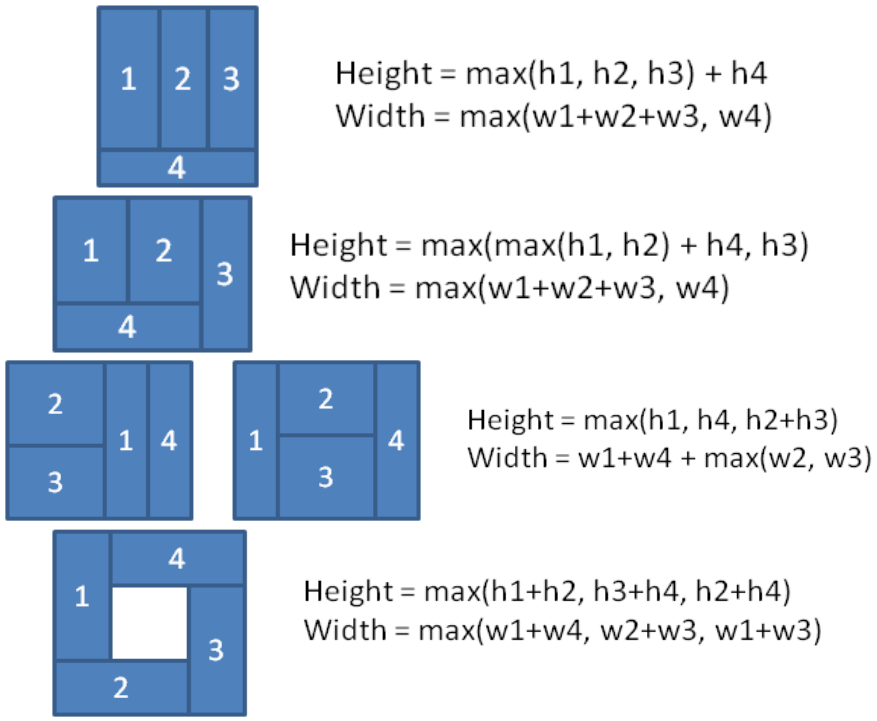


Figure.17 Eventually, we only have 5 types of layouts

At last, we only have 5 types of layouts because two of them can be regarded as the same when calculating the area. Consequently, the search time reduce from 2304 to $24 \times 16 \times 5 = 1920$. We implement this algorithm as *PackSearch2.java* in the attachment. Here is an experiment result.

Input:

```
2 10
5 8
2 2
1 1
```

Output:

```
search time:      1920
final layout:     3
min area:   70
3 : 0
4 : 0
1 : 1
2 : 1
```

In the output, the notation "*x:y*" means the *xth* rectangle has been put with orientation *y* (*y=0*, no rotation; *y=1*, with rotation). Translating this optimal result into a picture, we can get figure.18.

3.1 Conclusions

In this special case, we only pack 4 rectangles and we don't have limitations on the packing rule, the order or the orientation of rectangles. At beginning, we come up with a naïve method but it requires too much search time. Then we start from the hints of a competition problem from IOI1995 to get 6 basic layouts for packing 4 rectangles and show that other

packing structures can be mapped to these 6 representative ones by reflection or rotation. As a result, we are able to search all possibilities in 1920 times. Our algorithm in *PackSearch2.java* successfully solves this special case.

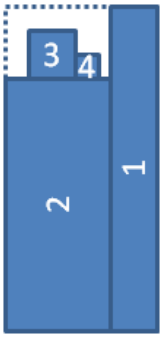


Figure.18 The optimal result getting by our search algorithm

However, the number of rectangles is very small. If we enlarge the scale of testing data, the number of basic layouts will increase dramatically, making it unrealistic to enumerate all possibilities.

4. SUMMARY

In this experiment report, we discuss two special cases of the interesting rectangle packing problem. In fact, the rectangle packing problem is a famous difficult problem in the world. It is a typical variation of the "bin packing problem" [3], which is de facto a NP-hard problem. However, in this report, we do not discuss about whether the rectangle packing problem is NP-hard or how to prove it is NP-hard. We focus on two more specific cases of the initial problem.

The first special case restricts the general packing rule so that there are only two options every time when attaching a new rectangle. However, even though the every-step packing options have been restricted, the comprehensive problem is still very difficult. We are not able to design a greedy algorithm or figure out a suitable sub-problem for dynamic programming. Finally, we provide a search algorithm optimized with backtracking strategy. Our method obtains good results when the input data is in a small scale, however, it does not perform well when the number of rectangles increases to be very large.

The second special case deals with only 4 rectangles and does not have limitations on the packing rule. We discuss a naïve idea at beginning. Then we make use of the hints from IOI1995 to figure out 6 basic layouts of packing four rectangles and propose an efficient search algorithm.

For both cases, we are not able to come up with any greedy algorithm or dynamic programming strategy. We are only able to search them and optimize the algorithm with some smart search techniques. However, the search algorithm cannot always scale well especially when the size of input data grows pretty large. In the future, we will try to explore some approximate algorithms for them. Then we will also explore the approximate algorithm for the original rectangle packing problem.

5. REFERENCES

[1] Sanjoy Dasgupta et al, Algorithms, 1st edition, 2006

[3] Bin packing problem, http://en.wikipedia.org/wiki/Bin_packing_problem

[4] Richard E. Korf, Optimal Rectangle Packing: New Results, 2004

[5] 7th International Olympiad in Informatics (IOI1995), http://olympiads.win.tue.nl/ioi95/