Maximal square covered by the set of rectangles

Mikołaj Przybysz, Bartłomiej Wach and Łukasz Zdanowicz

MiNI, AO

Document metric					
Project:	Maximal square covered by the set of rectangles				
Name:	Description of the algorithms				
Topics:	Description of the optimal and approximation algorithms				
Authors:	Mikołaj Przybysz, Bartłomiej Wach and Łukasz Zdanowicz				
File:	AlgorithmsDescription.odt				
Version no:	01.50	Status:	Final	Opening date:	2010-10-19
Summary:	Description of the finding maximal square from the set of random rectangles				
Authorized by:				Last modification date:	2010-11-28

History of changes			
Version	**Date**	**Who**	**Description**
0.80	19.10.2010	Mikołaj Przybysz Bartłomiej Wach Łukasz Zdanowicz	Creation of version 0.80
1.00	21.10.2010	Mikołaj Przybysz Bartłomiej Wach Łukasz Zdanowicz	Version 1.00
1.50	23.11.2010	Mikołaj Przybysz Bartłomiej Wach Łukasz Zdanowicz	Correction of the algorithms Modification of the GUI Added deleting and editing Added view of every rectangle

Table of contents

Problem description

Abstract

Main purpose of the program is to find solution to how to cover a maximum possible square with a set of generated rectangles.

Summary

1. Purpose of the application is to find such a placement of a set of rectangles so they cover as big square area as possible, without overlapping each other. Rectangles will be generated with random width and height (random from interval 1 to 6 units). From the set of rectangles program should build the maximal square while taking **some restrictions** into account :

   - Rectangles cannot overlap.
   - Rectangles cannot expand beyond borders of the square.
   - There cannot be holes in created square.
   - Rectangles can be rotated.
   - Not all rectangles must be used.

2. **Input**

As an input user should specify number of rectangles wanted to use (for application it may be about 100-200, but for optimal – several rectangles)

3. **Output**

As an output user should get:

a) sketched square (in some scale if it is too big)

b) size of the square

Description of the algorithms

Problem to solve

The problem to solve is how to find the biggest possible rectangle one can cover with a subset of the input set of n rectangles of random sizes under certain constraints :

- rectangles cannot exceed the area of covered square
- not all rectangles have to be used
- there may be no solution

Description of the optimal algorithm

The algorithm will be split into two parts, one will find the set(s) of rectangles that will cover the biggest square, the second one will "pack" the rectangles into the square.

Dictionary:

Input : an array of pairs of integers being the width and height of each rectangle
w : denotes the width of a rectangle
h : denotes a height of a rectangle
n : denotes the **current** size of the input
r : denotes the root of the sum of areas of chosen rectangles

Part I : Finding the set of rectangles covering the biggest square

1. First we need to decide how big is the square that can be obtained from the set :
   1.1. Sum the areas of the rectangles and get the root (or **r)** of the sum
   1.2. If there are rectangles of **w** or **h** greater that the floor of the root, remove them and go to (1.1)
2. As the order of the rectangles resembles the order in which they are "packed" into the square, we need to generate all the **permutations** of the set of rectangles to make sure all the possibilities are considered.
   2.1. Create a new set of rectangles for each permutation of the input
3. Since we can rotate the rectangles, we will generate all the possible sets of rectangles of size **n**, treating rotated rectangles as new elements replacing previous rectangles.
   3.1. For each set of rectangles, for each **combination**(of **1** to **n** rectangles) of rectangles, create a new set of **n** rectangles and "rotate" the chosen ones(swap their **h** and **w**) in the combination.

After this part, we should have a set of sets of rectangles that will give us a rectangle or no solution.

Part II : "Packing" the rectangles into a square

Now we just have to put the rectangles into a square of size found in the first part.

1. For each generated set :
   1.1. Fill the square starting from upper left corner, left-right, taking rectangles from the set according to the order in the set.
   1.2. Check if there are any holes
   1.3. If there are none, save this set as a solution, **end execution**
   1.4. If there is a hole, continue the loop
2. If after trying all the sets, no solution is found :
   2.1. Lower **r** by one
   2.2. Go to Part I : (1.2)

Example execution

Input : { (1,1),(2,1),(3,1),(3,1),(4,4),(6,1) } // { (w,h) ...

Part I

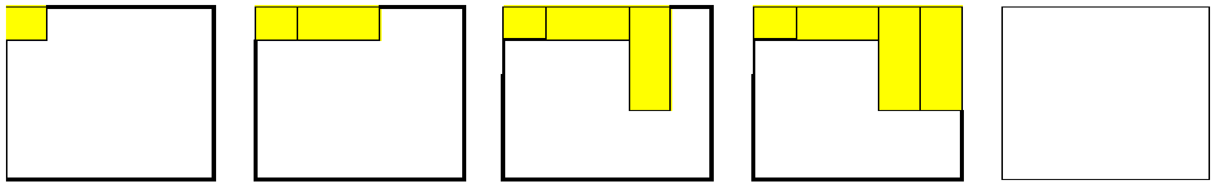
1.1 n = 6, r > 5, remove (6,1), n = 5,r = 5, go to 2.1

2.1 Generate x = n! sets from input

S1 = { (1,1),(2,1),(3,1),(3,1),(4,4) }
S2 = { (2,1),(1,1),(3,1),(3,1),(4,4) }
...
Sx = { (4,4),(3,1),(3,1),(2,1),(1,1) }
3.1 Generate y sets for each combination of each set and rotate chosen squares

S1.1 = { (1,1),(2,1),(3,1),(3,1),(4,4) }
S1.2 = { (1,1),(1,2),(3,1),(3,1),(4,4) }
S1.3 = { (1,1),(2,1),(1,3),(3,1),(4,4) }
...
S1.y = { (1,1),(1,2),(1,3),(1,3),(4,4) }

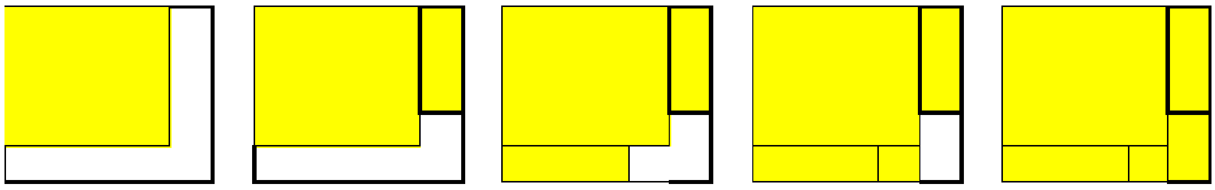
Part II



1.1 S1 = { (1,1),(2,1),(3,1),(3,1),(4,4) }

Filling fails at step 5, can't fit (4,4) into the rectangle.

1.1 Sk.z = { (4,4),(1,3),(3,1),(1,1),(1,2) }



Solution found, save solution, stop execution.

Description of the approximation algorithm – Mikołaj Przybysz

First we need to decide how big is square that can be obtained from the set.

1. Sum the areas of the rectangles and get the root of the sum
2. If there are rectangles of width or high greater that the floor of the root, remove them and go to step 1
3. Order set of rectangles from these with biggest area to lowest (decreasing)

After this step, we just have to put rectangles into a square of size found before.

4. While the set of rectangles is not empty
    - 4.1. Fill the square starting from upper left corner, left-right, taking rectangles from the set according to the order of the set
    - 4.2. If rectangle does not fit, rotate it and try again, if it still does not fit rotate it back and try to put it in next row
    - 4.3. If it does not fit skip this rectangle
    - 4.4. Check if area of used rectangles is equal area of possible square
        - 4.4.1. If it is true, save used rectangles as a solution and **end execution**, else continue
    - 4.5. Check if set of rectangles is empty
        - 4.5.1. If it is true, continue the loop, else go to step 5
5. Lower possible side of square by 1
    - 5.1. If possible square is less than 1, **end execution** with no solution, else go to step 2

Description of the approximation algorithm – Bartłomiej Wach

The algorithm will be split into two parts, one will find the set(s) of rectangles that will cover the biggest possible square, the second one will "pack" the rectangles into the square.

Dictionary:

Input : an array of pairs of integers being the width and height of each rectangle
w : denotes the width of a rectangle
h : denotes a height of a rectangle
n : denotes the **current** size of the input
r : denotes the root of the sum of areas of chosen rectangles

Part I : Finding the set of rectangles covering the biggest square

1.1. Sort the input in bigger-first manner

1.2. Sum the areas of the rectangles and get the root (or **r)** of the sum

1.3. If there are rectangles of **w** or **h** greater that the floor of the root, remove them and go to (1.1)

1.4. Try to insert each rectangle into first free space, try its rotated version if it doesn't fit

  1.4.1. after each inserted rectangles check if there are any holes in the square, if none, return solution, end computation

1.5. if no solution found, lower the assumed square size by 1 and goto 1.2

1.6. if no rectangles left, end computation

Description of the approximation algorithm – Łukasz Zdanowicz

Algorithm works as follows:

1. Sum the areas of all rectangles from instance of the problem. Get square root and then floor of calculated value to get side of possible maximal square that can be build.
2. Search the set of rectangles and remove rectangles with width or height greater than value gained in step 1.
    - 2.1. If any rectangle was removed, side of possible square must be calculated again - go to step 1, else continue

At this step we have a set of rectangles that can be put into possible square and the value of side of this square.

3. Search for rectangle with side nearest to initial value (at first initial value is equal to possible square side)
4. Put rectangle starting from upper-left corner of possible square (rotate if needed)
    - 4.1. If there is no possibility to put rectangle without overlaps or rectangle expand beyond borders skip this rectangle.
5. Check if area of used rectangles is equal to the area of possible square.
    - 5.1. If the answer is yes, **end computation** and save used rectangles as a solution, if no continue.
6. Check if there are rectangles in queue
    - 6.1. If the answer is no, go to step 8, else continue.
7. Check the side of possible square that left to be covered and go back to step 3 (put calculated value as initial value)
8. Decrease value of side of possible square by 1
    - 8.1. Check if value is less than 1
        - 8.1.1. If the answer is no go to step 2, else **end computation** with no solution.

Correctness proof

Thesis

For any input (input of any size),consisting of pairs of integers resembling width and height of rectangles, our algorithm will find an optimal solution(at a cost of computation time and memory space).

Proof

Our algorithm is separated into four parts.

1. The first part eliminates rectangles that won't fit for sure in the square since being too big, while defining the size of possible square to be formed.
2. The second part finds all the possible orders in which the rectangles can be inserted into the square (creates sets being all the permutations of the set of rectangles).
3. The third part multiplies number of sets created in part two by the size of the set of all possible combinations of rectangles of each set created in step two and swaps their sizes within each other, simulating the rotation of each rectangle. After step three, the algorithm has created set of all the possible approaches(in the way of putting the rectangles into square) to the input.
4. The fourth part is about filling the square, which is the process of elimination of the incorrect sets of rectangles from all those created in steps two and three until a solution is found or no solution is decided.

Conclusion

Therefore, by claiming that we created a proper “brute force” algorithm that considers all the possible approaches to the packing problem with the given input, we say it is a correct algorithm as no possibility is missed.

Complexity and space cost estimation for algorithms solving the problem of packing rectangles into square

Optimal algorithm

Space cost

Our algorithm first creates an array of **n!** arrays of size **n** being the set of all permutations of input. Then for each permutation, a new set of size **n** is created for each combination of the set, with chosen rectangles “rotated”. For each set of “computation sets”, being the set permutation + a set for each combination, algorithm tries to find a solution and reuses the space of “computation sets” for the next permutation. Therefore the space cost is **O(n*2^n)**.

Computational complexity

5. First, the algorithm sums the areas of rectangles to get optimal square edge length and removes extreme rectangles as long as there are no extreme rectangles:: **O(n)**
6. Then, the set of permutations is created : **O(n!)**
7. And the set of combinations, marking which rectangles to rotate is created : **O(2^n)**
8. Then the set of sets of rectangles is created for the algorithm to work which contains all the permutations + all the combinations of each permutation

: **O(2^n * n!)**

9. After that, the algorithm works while size of optimal square is > **O(max_length)**
   a. for each set created in step 8 : **O(2^n*n!*n)** for there are **n** rectangles in each set
      i. removes square that are too big from the set :**O(n)**
      ii. tries to make a square of the rectangles by putting them in the same order they are in the set: **O(n)**
      iii. lowers the square size by one if no solution found, goes to a : **O(1)**
10. Solution is either found or not.

Hence, the time complexity is as follows :

O(n) + O(n!) + O(2^n) + O(max_length) * (O(n) * O(2^n*n!*n) *O(n)*O(n)*O(1))

Approximation algorithm nr 1 (designed by Bartłomiej Wach)

Space cost

The algorithm uses only the input array : **O(n)**

Computational complexity

1. The algorithm finds the optimal rectangle edge length by getting the sum of areas of input rectangles and getting its sqrt : **o(n)**
2. The input is sorted in bigger-first manner. : **o(n)**
3. While the optimal rectangle edge length is > 0, the algorithm tries to fit the rectangles into the square, rotating each one if it doesnt fit as it is : **o(2n)**
4. Algorithm removes lowers the expected rectangle edge by one : **o(1)** and goes to 1.
5. Algorithm may run while **initial_square_size > 0** there fore **o(initial_square_size)** times

Hence, the time complexity is as follows :
o(n) +o(n)+ o(initial_square_size)*(o(2n)*o(1))

Approximation algorithm nr 2 (designed by Mikołaj Przybysz)

Space cost

The algorithm uses only the input array : **O(n)**

Computational complexity

1. The algorithm finds the optimal rectangle edge length by getting the sum of areas of input rectangles and getting its sqrt : **o(n)**
2. While the optimal rectangle edge length is > 0, the algorithm tries to fit the rectangles into the square, if one doesn’t fit anywhere it checks if its rotated version fits : **o(2n)**
3. Algorithm removes the biggest rectangle if no solution found : **o(1)** and goes to 1.
4. Algorithm may run while **initial_square_size > 0** there fore **o(initial_square_size)** times

Hence, the time complexity is as follows :
o(n) + o(initial_square_size)*(o(2n)*o(1))

Approximation algorithm nr 3
(designed by Łukasz Zdanowicz)

Space cost

The algorithm uses only the input array : **O(n)**

Computational complexity

1. The algorithm finds the optimal rectangle edge length by getting the sum of areas of input rectangles and getting its sqrt : **o(n)**
2. Algorithm removes rectangles with side greater than the optimal rectangle edge length: **o(n)**
3. Algorithm finds the rectangle with side nearest to initial value: **o(n)**
4. Put rectangle into square: **o(1)** (location and rectangle is known)
5. Check the side of possible rectangle that left to be covered: **o(sqrt(square_area))** and go back to step 3
6. Algorithm decreases side of possible square by 1: **o(1)** and go back to step 2.

Hence, the time complexity is as follows :
o(n) + o(n)*o(n)*o(1)*o(sqrt(square_area))*o(1)

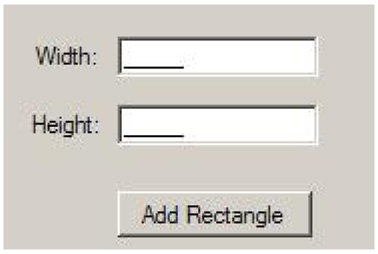
Application description

Application specification

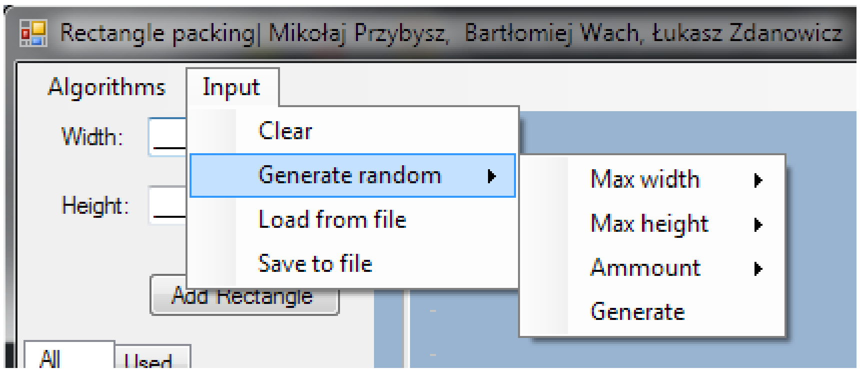
Programming language : C#
Environment : Microsoft Windows
Application dependencies : .NET platform
Application execution : The application is compiled into a single .exe file.

User's manual

The use of our application is very simple. First, the user has to provide input in the form of pairs of numbers being the width of height of each rectangle. This is done through two input fields.

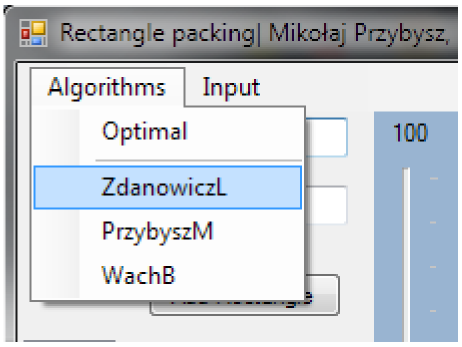


The input can also be generated randomly through menu "Input" where the user can set limits for generated dimensions of the rectangles as well as their amount and then use the "generate" button to fill the list of input rectangles with generated elements.



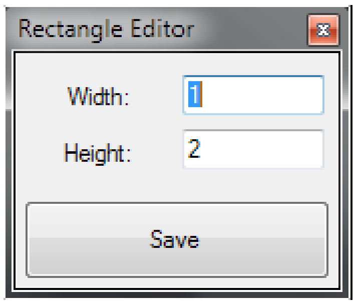
Input can also be imported from text file from local disk.
Input rectangles can be exported to text file to use with other programs.

The next step is performing calculations for which purpose "Algorithms" menu was designed, allowing the user to chose which algorithm to apply to the input. After computation is finished, the square (if a solution was found) is drawn in the GUI.



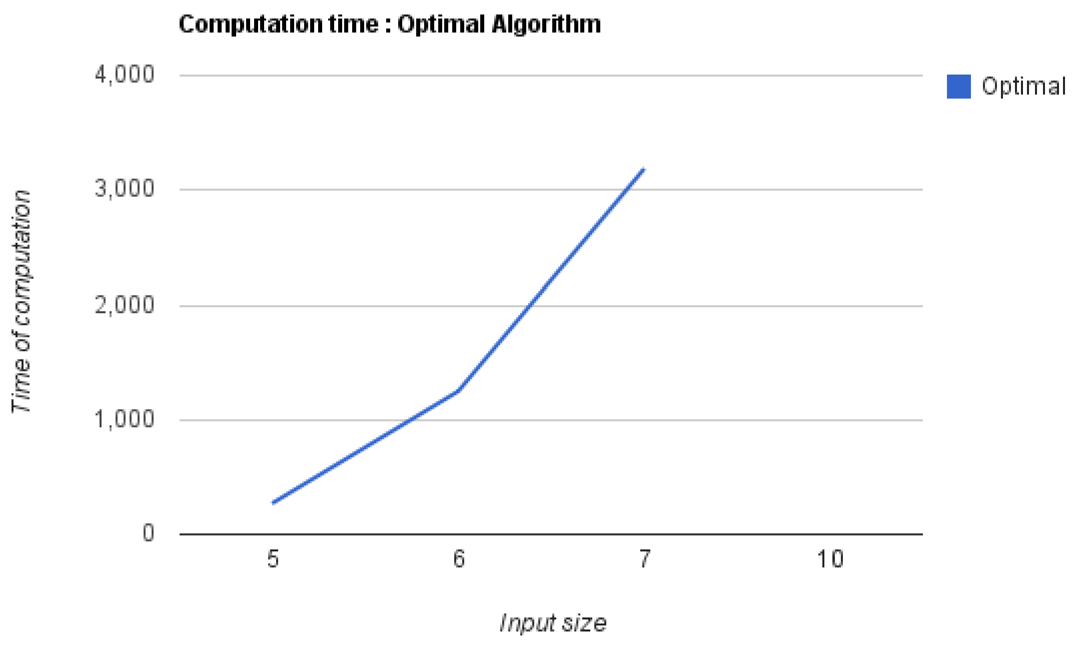
Any element from the input can be deleted by clicking on it and hitting "delete" button from keyboard.

Any element from the input can be edited by clicking it by mouse two times. In new window we specify width and height which will be changed after pushing "Save" button.

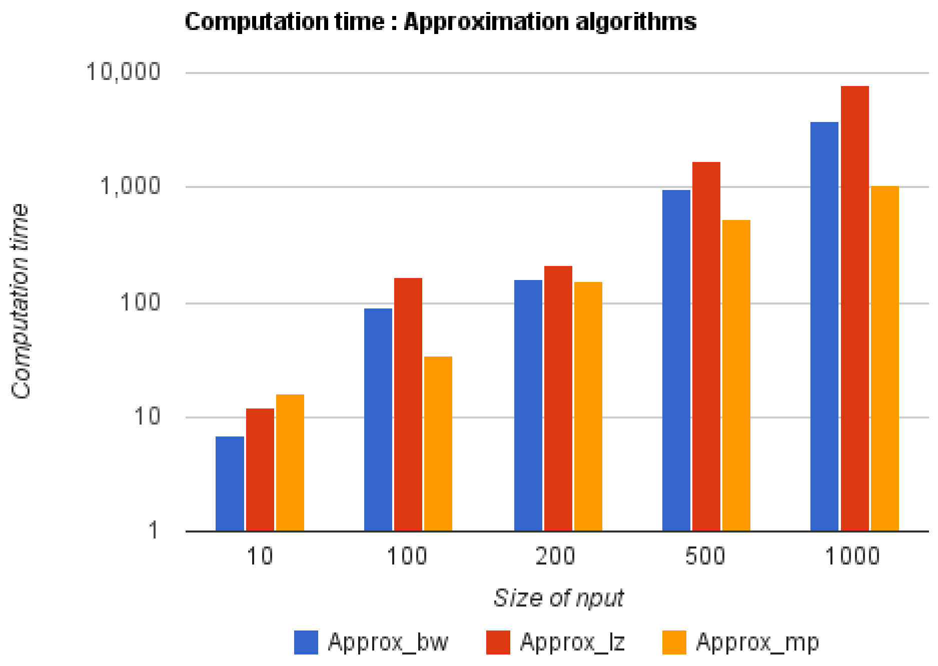


Performance statistics

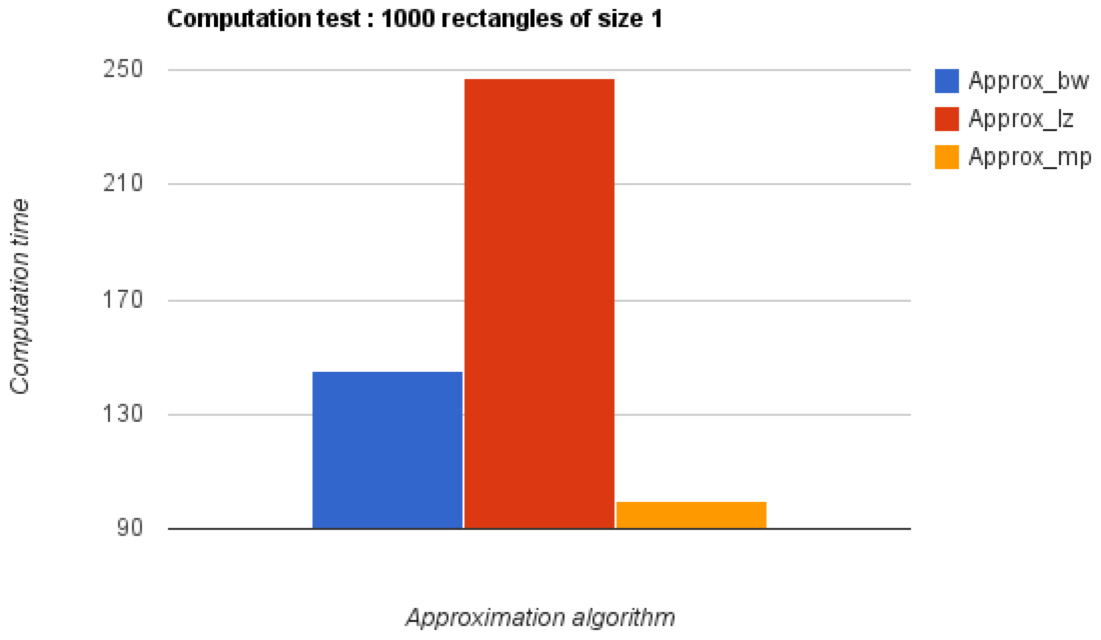
The optimal algorithm works for up to 8 rectangles, depending on the machine setting. For bigger sets the amount of memory needed to produce all the possible sets for computation usually exceeds the memory allowed for the application.
Computation time is given in milliseconds.



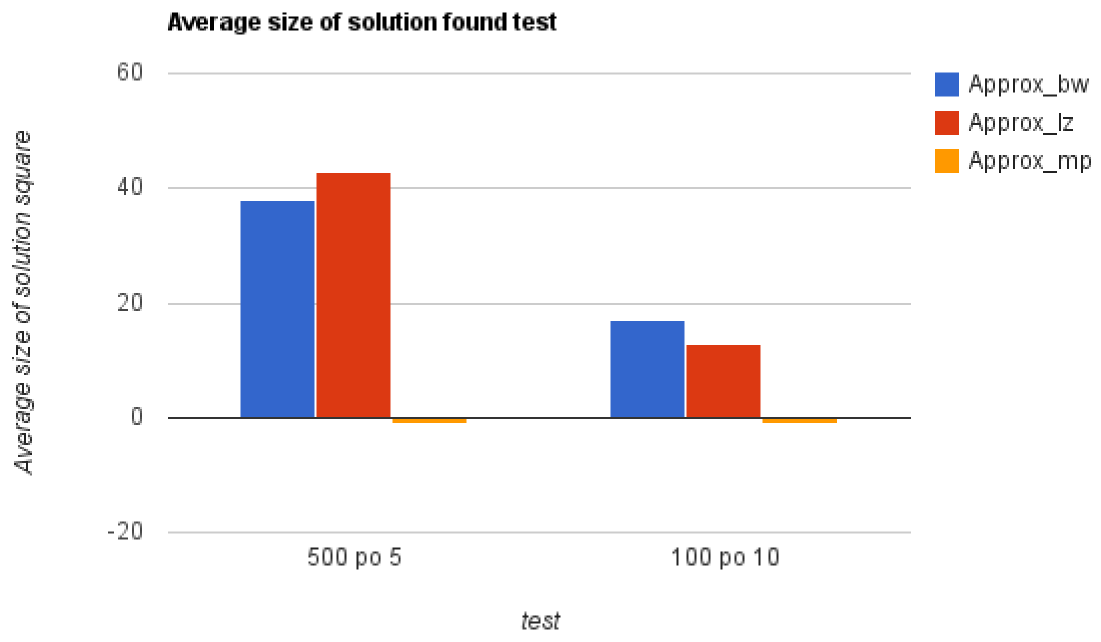
Comparison of performance of approximation algorithms for the same input of rectangles. Initial input was generated randomly. Computation time is given in milliseconds.



First test : 1000 rectangles of size 1 (1 by 1)



Second test : 500 rectangles of dimension less or equal 5 and 100 rectangles of dimension less or equal 10.



Report on rectangle packing algorithm designed by Mateusz Wypysiak and Artem Masalovych after a report by Michał Streja and Konrad Żaba

Optimal algorithm

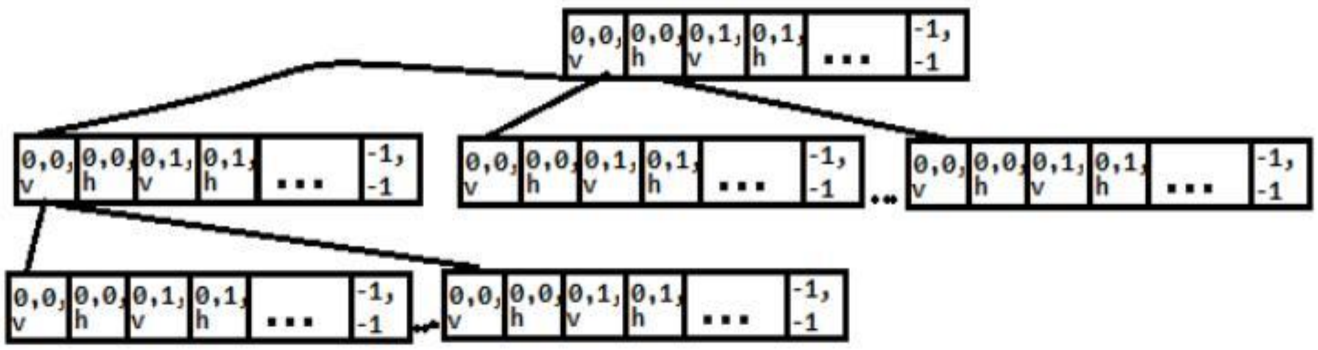
Section 5.1

Point 2 :

Language errors force reader to read it several times

"Each cell in "root" array creates a sub tree with all possible positions of the second element from the list."

This statement doesn't point to which list should we refer, therefore the sorted input is assumed which makes the visualisation of tree incorrect according to the description as each cell in each array (vector) should have only one child(it is unspecified if we consider each rectangle as a second element or not).



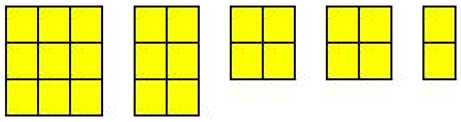
Remark :

"Number indicates particular cell filled with part of a rectangle. It will be useful in future, during coloring. So each rectangle will have its number according to which we differentiate them."

This statements are unclear, while first sentence suggests that each cell has a unique number, the second sentence suggests that all the cells covered by a certain rectangle have the same number or rectangles already put in the square are numbered in some other way.

Example execution as understood by reviewer

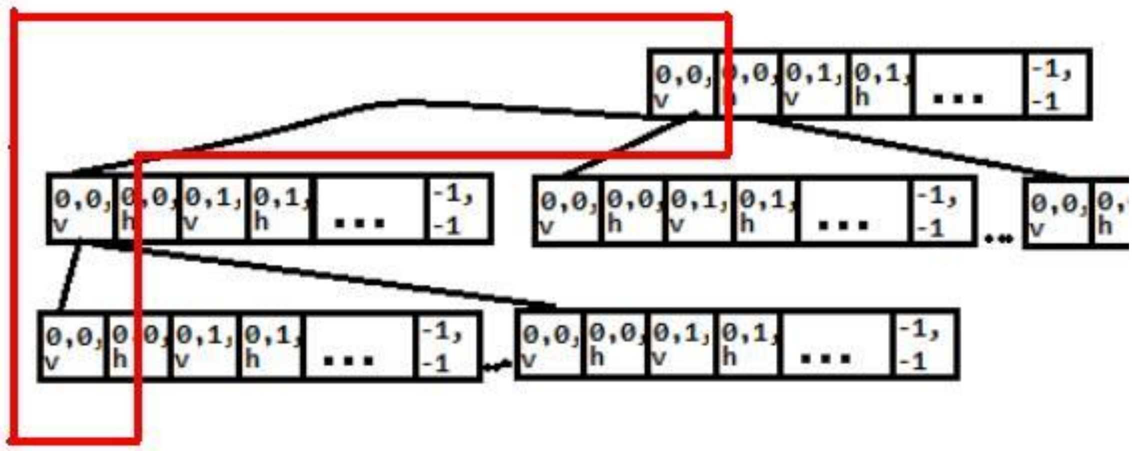
Sorted input :



S = 25

A = 5

First sequence considered in execution (going top to bottom) will always be a non-solution.



For one of the paths in the tree that gives a proper solution :

Rectangle :

Solution path (assuming (x_coordinate, y_coordinate, rotation)) :

0,0,v -> 0,3,h -> 3,0,v -> 3,2,v -> 3,4,h

1	1	1	3	3
1	1	1	3	3
1	1	1	4	4
2	2	2	4	4
2	2	2	5	5

No holes, solution found.

Conclusion : If all nuances will be solved in favor of the algorithm, its logic is correct.

Approximation Algorithm 1

Regarding errors mentioned in the algorithm report - all have been corrected, mainly:

Error 1:

“nieczytelność - czemu algorytmy aproksymacyjne NIE SĄ W PUNKTACH?”

Algorithms are pointed out and the form they are written in is pleasant for reading and easily understood.

Error 2 :

“Czemu algorytmy zakładają, że prostokąt może się nie zmieścić w 1 ruchu - to niemożliwe.”

In point 6, there is adnotation:

“6. If current rectangle has width (top side) = Awt and height (left side) = Aht, means rectangle fits all area, we can finish this step with success. Function call exits with success. Else go to step 6.a)“

Approximation Algorithm 2

New algorithm was created so we can omit errors mentioned in the algorithm report.
Found errors:

Error 1:

In point 12. undefined variable n (variable is described in point 13.)

Error 2:

Point 13. is not clear

Error 3:

In point 4.priority of choosing the proper position is not clear. In our opinion the description does not match *“Example of order of placing elements”* (5th element should be placed under 1st element as first, we consider y-axis and we find an empty spot along y-axis)

Final mark : We score this algorithm a score of 4 out of 5 as it contains only minor language errors while the overall logic is correct