Rectangle Packing Algorithms

T.P.J. Beurskens M.C. Crone M.L. Hofsté A.I.O. Nijsten
E.C. Takke D.A. Tuin G.C. van Wordragen

Group 14

Abstract

In this paper, the problem of finding a bounding box of minimum size and a placement for a set of rectangles in the plane, without overlapping, is considered. This is a problem that has numerous applications in production processes and in daily life. Four variations of the problem will be looked at; a combination of allowing or disallowing rotation of rectangles, and having a fixed or flexible height of the bounding box. After a short overview of previous work on these problems, five algorithms are presented to help solve them. Of these five, two are independently derived and three are modifications of existing algorithms. One of these algorithms provides optimal solutions through brute forcing, others give approximations, e.g. by using a greedy strategy or trying to fill several bounding boxes with bottom left packing. With a comparison using different benchmarks, weaker and stronger points of these algorithms will be stated.

1 Introduction

Background and motivation. Suppose you need to cut rectangular objects from a sheet of metal with a laser while creating as little waste as possible. This problem is closely related to determining the optimal packing of (rectangular) shapes in a 2D space. Somewhat less straightforward, but also related, is the problem of scheduling similar jobs that can be executed by one or more workers, with a minimal amount of time and workers. This can be solved by representing the jobs as non-rotatable rectangles [9]. Furthermore, the problem can be extended to 3 dimensions, when loading a van of a delivery service with packages for example.
In all these problems, an optimal packing can reduce costs, and solutions provided in this report may be of interest to anyone searching for improvement in such processes.

Problem statement and previous work. We are looking for an algorithm to pack rectangles into the smallest possible rectangular area. The problem is defined in a two-dimensional space and all rectangles should be aligned with the vertical and horizontal sides of the box. In order to make this problem more concrete, some notation is introduced.

First, let S denote the list of rectangles in the plane that need to be packed. So $S = \{R_i \mid i = 0, 1, ..., n-1\}$ with n the number of rectangles and R_i a rectangle. Also let r be a boolean indicator of whether the rectangles can rotate or not. Furthermore, each rectangle R_i has the following properties: a width w_i, a height h_i, a variable position of its bottom left corner in the first quadrant (x_i, y_i) and possibly a variable orientation (rotated by 90 degrees or not).

Next, let B denote the set of all possible bounding boxes in which the rectangles of S fit without overlapping. These bounding boxes are possibly of a fixed height H. A solution is formed by a fixed combination of locations and orientations of all rectangles R_i such that there exists a cover in B for these R_i in these positions, and an optimal solution is one where the cover is one of minimal area.

The rectangle packing problem is NP-complete [9], which means that, in general, finding the optimal bounding box is considered to be impossible to do in polynomial time. A large number of papers about this subject has been published. This amount of activity can partly be explained by variations of this problem, such as allowing rotations and fixing one dimension of the bounding box. Some papers aim to find an optimal solution, others an approximation. Finding better and faster solutions is an active field of research, because they can be crucial in some production processes. By decreasing resource usage, costs can be decreased and sustainability can be improved.

The problem without rotation, with a flexible bounding box and bounding box with fixed height is considered in the paper by Korf [9]. In this paper, algorithms are described to solve these problems, starting with an approximating solution and continuing with finding better solutions, until an optimal solution is found. It also proposes a set of squares of increasing size as a benchmark to compare algorithms. Further, extensions are introduced that make the algorithm faster. In later papers of Korf, more benchmarks, searching and pruning techniques are introduced for the algorithm in his 2003 paper. [7, 10]

Another way of looking at the bounding box problem without any rotation allowed, is as if it is a Tetris game. This is exactly what is considered in the paper by Coffman et al. [5]. The article introduces an algorithm that is an improvement of the readily known Next Fit Level algorithm (NFL). In another article by Belov et al. regarding the same problem, the Bottom Left Right (BLR) algorithm is described [3].

When rectangles are allowed to be rotated by 90 degrees, the complexity of the problem naturally increases. The most logical, naive solution is to run an algorithm designed to work without rotations with all configurations of rotated and non-rotated rectangles, however this would produce a running time that is higher by an exponential factor. Especially when given large inputs this is undesirable. Hence there exists no general way to adapt any algorithm to also accept rotations efficiently. As a result, this problem needs its own specialized algorithms. One way to generalize existing algorithms to accept rotations is described by Simonis and O'Sullivan [12].

Given a fixed height, one can consider so called staircase packing. In the paper by Kenmochi et al. an algorithm is described that packs rectangles like a staircase [8]. This algorithm works very fast and optimal for perfect packing problems, i.e. problems where the rectangles can fill a bounding rectangle completely. They extend the problem to non-perfect packing problems as well. In their experiments they get optimal and fast results for problems up to 25 rectangles, in cases where rotation is not allowed as well as cases where it is allowed.

A lot of these algorithms do well in their own regard. However, combining them can lead to significant performance gains. This is done in the paper by Baker [2]. In this article a new algorithm is proposed which was derived by analyzing the behavior of Next-Fit Decreasing Height (NFDH), First-Fit Decreasing Height (FFDH), and Bottom-Leftmost (BL) [4] algorithms. This algorithm provides a solution whose area is at most 5/4 times the optimal area.

Our contribution Some of the algorithms implemented were inspired by aforementioned papers, however others are independently derived and most of include improvements or extensions. Firstly, the OptRotMARP algorithm implemented is a brute-force algorithm sped up by using pruning. Another innovative creation is the use of Korf's 2003 algorithm [9] in conjunction with Bottom-Left packing. This combination has shown to be very successful. The Wall Stacker algorithms have not been based on a paper at all. Empirical evidence has shown these algorithms to be a good heuristic. Additionally, the G-Staircase algorithm by Kenmochi et al. [8] has been extended to solve problems with flexible height and time limits were introduced to allow for cases with larger number of rectangles to be solved as well. Lastly, the Recursive algorithm [13] has been implemented to work multiple times for distinct orderings, heights and orientations.

2 The algorithms

2.1 OptRotMARP

OptRotMARP (Optimal solver for Minimal Area Rectangle Packing with Rotations) is an independently derived branch and bound style algorithm that uses backtracking on the position of each rectangle to find a solution that is guaranteed to be optimal. The basic structure of the algorithm is described as pseudocode in Algorithm 1. The algorithm uses a fast heuristic to provide a solution and initializes *bestArea* as the area of that solution (A) to improve pruning.

Algorithm 1 OptRotMARP

Stores the optimal configuration of a given set of rectangles $R_0, ..., R_{n-1}$.
Initially $k = n - 1$, $bestArea = A$, and the bounding box $w \times h = 0 \times 0$.

```
if w · h ≥ bestArea then
    return                                  ▷ This branch does not improve the solution.
if rectangle R_k overlaps any others then
    return                                  ▷ This branch violates the constraints.
if k = 0 then
    Store the current configuration of rectangles
    bestArea ← w · h
    return
for all relevant configurations of rectangle R_{k-1} do
    Make a recursive call with k ← k − 1
     and w × h as the size of the smallest rectangle covering R_{k-1}, ..., R_{n-1}
```

2.1.1 Runtime analysis

The use of backtracking, when not considering pruning, implies that all possible configurations of rectangles are considered. This makes for a tree of depth equal to the amount of rectangles n and as branching factor p the possible configurations for each rectangle, thus having p^n leaf nodes.

p is determined by whether rotating is possible and the upper bound on the area (A), which is initially received from some fast heuristic such as NFDH. Each configuration takes

$\mathcal{O}(n)$ time to evaluate, as it has to check overlap with all other rectangles. This gives upper bounds $\mathcal{O}(A^n n)$ and $\mathcal{O}(2^n A^n n)$ on the running time for problems with and without rotation, respectively. As can be seen, the quality of the pre-calculated solution has a huge impact on the running time.

2.1.2 Optimality

Backtracking makes sure all possible configurations are explored, which guarantees optimality. Left to prove is that none of the pruning techniques impact this optimality.

First of all, branches stop being considered if they no longer improve on the best solution found thus far, which can be quickly checked by comparing the area of the needed bounding boxes. This can be done because no rectangles are moved after being placed.

Once this style of pruning happens, there is also no point in placing the last placed rectangle (R_k) at an even higher position. If, after moving the rectangle to some $(x, 0)$ further right, still $w \cdot h \geq bestArea$, that means all good placements of R_k have been considered for the given configuration of the rectangles $R_{k+1}, ..., R_{n-1}$ before it and new configurations of R_{k+1} can be considered.

Secondly, the solution is not allowed to have overlapping rectangles. To ensure it obeys that constraint, the algorithm checks whether the most recently placed rectangle R_k overlaps any of the previously placed rectangles $R_{k+1}, ..., R_{n-1}$ and prunes if it does. Since overlapping rectangles are not allowed, pruning this branch will not get rid of optimality and ensures a valid final solution. Similarly, if a rectangle crosses the maximum height, the configuration is no longer valid and its branch can be pruned with the same reasoning as before.

To still work relatively well with large rectangles, only coordinates that can be made by summing up the sides of unique rectangles are considered. An optimal solution can always be adjusted such that each rectangle has either another rectangle or one of the axes directly to the left and bottom of it, so a rectangle's coordinates can always be a sum of the sides of the rectangles to the left or bottom of it.

2.2 Korf 2003 and Bottom-Left rectangle packing

In this section an algorithm is described that solves all variations of the problem and is based on a combination made of the algorithm described by Korf in his 2003 paper [9] and Bottom-Left rectangle packing.

2.2.1 Korf2003

Korf2003 is an algorithm that tries to fit the rectangles in several bounding boxes, always returning a solution. Algorithm 2 describes the algorithm of Korf, where the conditions on H in line 5 extend the algorithm for fixed height. It first calculates a trivial solution in lines 1 to 4 with a height equal to the maximum height of the rectangles in S. Then it adjusts one of the dimensions of the bounding box in lines 6 to 9 depending on its earlier success with finding a solution and checks whether this gives a box that can fit the rectangles and could give a better solution than already found in lines 10 to 13. If it finds a way of placing the rectangles in this new box with dimensions w and h in line 14 this solution is stored as the new best solution and the process is repeated until a bounding box having the width of the widest rectangle is found or the h reaches the height limit H.

Algorithm 2 Korf2003(S)

▷ computes a bounding box of minimal size of the rectangles in S

1: $h \leftarrow \max\{h_i : R_i \in S\}$; $w \leftarrow \infty$
2: Place all rectangles in order of decreasing height in the leftmost available position in the $h \times w$ rectangle, as far down as possible, like in figure 1
3: $w \leftarrow \max\{x_i + w_i : R_i \in S\}$; $s \leftarrow true$
4: Store placement of rectangles as first solution sol with area $w \times h$
5: **while** $w >= \max\{w_i : R_i \in S\}$ and ($h \leq H$ or $H = \infty$) **do**
6:   **if** $s = false$ **then**
7:     Increment h by one
8:   **else**
9:     Decrement w by one
10:   **if** $w \cdot h <$ total area of rectangles in S **then**
11:     $s \leftarrow false$;   continue
12:   **else if** $w \cdot h >$ area(sol) **then**
13:     $s \leftarrow true$;   continue
14:   $s \leftarrow$ place(w, h, S)
15:   **if** $s = true$ **then**
16:     store placement in sol; $w \leftarrow \max\{x_i + w_i : R_i \in S\}$
17: **return** sol

To extend the algorithm for rotations for small amounts of rectangles (up to ten), it can be called on all possible permutations of rotations of rectangles. The method place(w, h) indicates whether the rectangles can be successfully placed in a bounding box of width w and height h or not, and places the rectangles if possible. When it does this correctly, Korf claims that the algorithm gives an optimal solution. Furthermore, when place(w, h) cannot find a way in which the rectangles fit, while there exists a way, the algorithm always returns a correct solution, which is possibly the trivial solution computed in the first few lines.

2.2.2 Place

Huang et al. prove in their paper that if it is possible to place n rectangles into a given bounding box without overlapping, then a packing without overlapping for this bounding box can be found through placing the rectangles in a sequence of bottom left placements [6]. There are $n!$ permutations of n rectangles, and O_n is defined as the highest possible number of bottom left placements for each rectangle. This means that there are at most $n! \cdot (O_n)^n$ possible ways to place rectangles using a bottom left placement algorithm, and at most $n! \cdot 2^n \cdot O_n^n$ ways if rotations are allowed. This already gives a large number of possibilities for a small number of rectangles. Hence, a version that uses a carefully chosen part of these possibilities is implemented, as explained later.

For placing rectangles, a version of Bottom-Left packing that places rectangles in a given sequence is used, as introduced by Baker et al. [1] One by one the rectangles in the sequence are placed as far left in the bounding box as possible and then as close to the bottom as they can, without overlapping another rectangle (and thus our implementation could be called Left-Bottom packing). Since rectangles are only placed when they do not overlap others, the

given placement by this algorithm is correct. To check whether a rectangle fits somewhere, a data structure based on a 2D array indicating occupied areas can be used, where widths and heights of the columns and rows in the grid are aligned with the already placed rectangles, as pictured in figure 1.

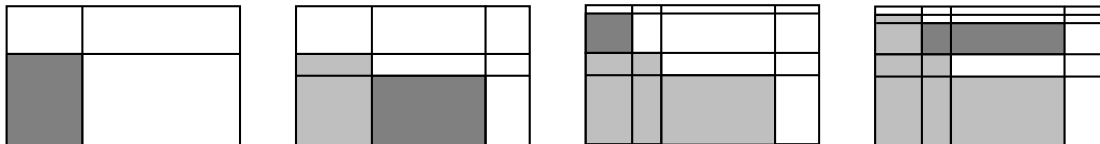

Figure 1: Placing algorithm on 4 rectangles: when a rectangle is placed, columns and rows in the grid are split according to the edges of the rectangle.

When the sequence of rectangles is sorted on decreasing height, the area of the resulting placement in the bounding box of given height is proven to be at most three times the optimal solution size for that height [1]. As the algorithm by Korf loops over all possible heights for an optimal solution, with Baker's algorithm as the placing algorithm, the resulting solution area is at most three times the optimal solution area.

With small amounts of rectangles (≤ 5), the placing algorithm can be improved by trying all permutations of the sequence of rectangles. For up to ten rectangles, rotation can be implemented by finding a solution using the Korf2003 algorithm on all 2^n combinations of orientations of rectangles. For more than five rectangles, without rotation, the algorithm attempts to place the rectangles into the bounding box based on decreasing height. When that fails, placing based on decreasing width can be tried, in case of success, this will improve the result.

The place(w, h) algorithm can also give a solution by itself when a bounding box is chosen, e.g. one with $w = \infty$ and $h = \max\{h_i : R_i \in S\}$. In the case where rotations are allowed, all rectangles that initially have $h_i > H$ if $H > 0$ should be rotated to find correct solutions using these algorithms.

2.2.3 Runtime analysis

For placing a rectangle, at most n^2 positions in the grid need to be checked. Splitting a column or row in the grid takes at most $\mathcal{O}(n)$ time, because there are at most n cells that must be split for the column or row in the 2D array. Thus placing according to Bottom Left packing can be done in $\mathcal{O}(n^2)$ time. Korf2003 examines $\mathcal{O}(\sum_{i=0}^{n-1} w_i + \sum_{i=0}^{n-1} h_i) = \mathcal{O}(n \cdot (W_S + H_S))$ rectangles in the worst case, where W_S and H_S are the maximum width and height of a rectangle in S. [9]

For the variation on the algorithm for small amounts of rectangles, flexible and fixed height, but no rotation, there are at most $n!$ possible permutations of the rectangles that are checked. This results in a running time of $\mathcal{O}(n \cdot (W_S + H_S) \cdot n! \cdot n^2) = \mathcal{O}(n^3 \cdot n! \cdot (W_S + H_S))$. For the same cases with 6 to 25 rectangles without rotation, the algorithm takes $\mathcal{O}(n^3 \cdot (W_S + H_S))$. With rotation, the cases with $n \leq 10$ take a factor 2^n more time.

2.3 Wall Stack Family

The Wall Stack family is a set of independently derived algorithms to stack large numbers of rectangles in a short amount of time. The main idea behind these algorithms is that they provide a solution that can be used for cases where e.g. back-tracking based algorithms would take an unreasonable amount of time. However, empirical evidence shows that it is able to keep up with these other algorithms quite well. All three algorithms that are part of the Wall Stack family heavily rely on use of a recursive placing function. The difference between the algorithms is their placing function, which means that each version can be used in different cases. A necessity for these algorithms is to have a height limit under which the rectangles have to be placed. If no such limit exists, a small algorithm is used to determine a promising height limit.

2.3.1 Simple Wall Stack

The first of three members of the family is Simple Wall Stack. The algorithm was designed for use in non-rotation cases and is, as mentioned before, based on recursion. First, it determines the width of the box it wants to fill. The height of this initial box corresponds to the given height limit. The width is determined by dividing the total area of all rectangles that still have to be placed by the height limit. It then starts to fill this box. It takes the tallest rectangles it can find that fit in the desired box and places them there, in a way similar to Bottom-Left algorithms. After that, it divides the remaining space in the box according to figure 2. Then, the placing function is recursively called on those new boxes. This process is repeated until either the maximum recursion depth has been reached, no more rectangles can be placed in the boxes, or the initial box is full. When no more rectangles can be placed in this box, the total area of the remaining rectangle is again divided by the height limit to determine a new box in which the remaining rectangles are to be placed. This process continues until all rectangles are placed. An upper time bound for this algorithm is $\mathcal{O}(b \cdot k^2 + n^2 \cdot \log(n))$, where b is the maximum number of variations between dimension sizes, i.e. sum of number of different widths rectangles have, and n is the number of elements. k is the maximum number of elements that share a dimension, i.e. their widths or heights are the same.

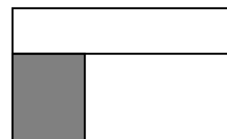

Figure 2: How the algorithm divides the space.

2.3.2 Dual Wall Stack

Dual Wall Stack is an algorithm based on the principles of Simple Wall Stack, but can only be used when rotation is allowed. The main difference between both algorithms is the fill method they use. Dual Wall Stack first rotates all rectangles to stand up straight, i.e. let the height be greater than the width. Then, it starts placing rectangles in a staircase-like fashion: rectangles under the staircase laying flat, and above the staircase standing straight up, see figure 3. In the end, the algorithm "converges" to what seems to be a Bottom-Left stacking algorithm when placing small rectangles, which are placed last. A time bound on this algorithm is $\mathcal{O}(b^3 + n^2)$. This means that the algorithm works quite well, when rectangles have similar shapes and sizes.

2.3.3 Triple Wall Stack

Triple Wall Stack performs the best of the three on cases with a very large amount of rectangles, even though it does not rotate any rectangles actively. If rotation is available, it rotates all rectangles that are too long to fit under the height limit. It then applies a placing algorithm similar to Simple Wall Stack. The only difference is that the placing method handles several edge-cases better. This includes cases that have a large variety in rectangle sizes and cases that have primarily standing rectangles. An upper time bound on the algorithm is $\mathcal{O}((b+k)^2 \cdot \log(b+k)+n+b\cdot k^2)$. To bring this in perspective, if all elements shared a dimension, i.e. $n=k$, the time bound would be $\mathcal{O}(n^2 \cdot \log(n))$. This holds when all elements have different sizes, i.e. $b=n$.

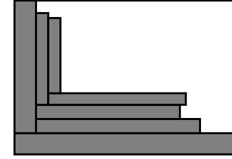

Figure 3: How the algorithm divides the space.

2.4 G-Staircase

This section concerns an algorithm that is based on the G-Staircase algorithm from the paper by Kenmochi et al. [8]. The algorithm uses nodes, which are placements of rectangles such that the rectangles make up a staircase shape (see figure 4). Here, an extension of Kenmochi's algorithm is discussed, as it can also handle problems with no fixed height. For this, it iterates through bounding boxes created according to the papers by Korf [9], [7].

In order to allow the algorithm to also be used for larger cases, the amount of nodes checked per given bounding box is limited. This gets rid of optimality and generally gives solutions that are not as compact as they could be. To combat this, the algorithm implements an extra method which searches for gaps. It then examines if moving rectangles, that have been placed elsewhere, into the gap could decrease the total area. An upper

Lastly, the algorithm shifts all rectangles to the left as far as possible, i.e. for all rectangles R_i finding the lowest possible x_i without overlapping other rectangles.

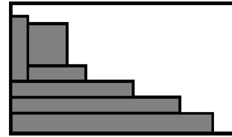

Figure 4: How the algorithm divides the space.

With a constant bound on the amount of nodes checked per bounding box, G-Staircase's running time is bounded as $\mathcal{O}(n \cdot \frac{\sum_i w_i h_i - A}{H})$ for problems where the height is fixed. If the height is flexible, then the running time will be $\mathcal{O}(n^2 \cdot (W_S + H_S))$. Worth noting is that the guaranteed quality of solutions will inevitably go down as n goes up, as it will check a smaller and smaller part of all nodes.

2.5 Recursive

Recursive is a Heuristic Recursive (HR) algorithm based on the paper by Zhang, Kang and Deng[13]. The algorithm on which it is based has an average running time of $\theta(n^3)$. Recursive extends the HR algorithm by providing extra methods to be able to deal with the flexible height test cases and by evaluating test cases at different orderings, manners of rotation and height limits. These extensions add at most a factor of 2048 to the time complexity. Therefore the average running time of Recursive is also equal to $\theta(n^3)$. This algorithm is applied to medium sized inputs, where its heuristic strategy and recursive structure result in good solutions. Its running time complexity is too large for it to be run effectively on large

inputs. The heuristic is also not precise and optimal enough to apply it to smaller test cases that can instead be brute forced efficiently.

3 Experimental evaluation

In this section, the algorithms will be compared using some benchmarks from papers and test cases with certain characteristics. The tests are run using Windows 10 on a 2.6 GHz Intel(R) Core(TM) i7-6700HQ CPU with 8GB of RAM.

3.1 Benchmarks

Consecutive rectangles. In his 2003 paper, Korf introduces the use of squares increasing in size, starting with a 1×1 square and ending with an $n \times n$ square, as the consecutive square benchmark for comparing algorithms. In this and later papers optimal solutions were found for up to 25 squares. These will be used in this report to compare results. [9, 10]
Larger amounts of rectangles are used here to verify the running time analysis of several algorithms described above.

As rotation is not relevant on squares, we look at the problem concerning rotation with the use of $N \times N+1$ rectangles, where $1 \leq N \leq n$, the consecutive rectangle benchmark. As Huang and Korf argue, these problems are rather easy to solve because of the shared dimensions between rectangles. Therefore we also look at the set of n rectangles with dimensions $N \times 2n - N$ with rotation, introduced as the double-perimeter benchmark by Huang and Korf in 2013. [7]

Perfect packing. When a rectangle packing problem has a solution with no empty space, this is called a perfect packing. Problems that have a perfect packing as the optimal solution are used here to compare results of algorithms that work on larger problems.
An example algorithm to generate problems with a perfect packing solution is:

i) Start with a large initial rectangle in a list.

ii) Select and remove one of the rectangles from the list.

iii) Cut it into two smaller rectangles.

iv) Add those rectangles to the list.

v) Repeat steps ii) to iv) until the list is of the desired size.

3.2 Specific cases

Large variance All algorithms in the Wall Stacker family are very fast and efficient. However they do make some assumptions that can result in a very bad solution. In figure 5 such a solution can be found. In this figure you can clearly see the assumptions that are made concerning the bounding box width, and how those affect the result.

The reason for this anomaly is a combination of things. It first tries to calculate the width of the bounding box in which it will try to fit in all the rectangles, since the height is given. Next, it fills this bounding box in a recursive way as illustrated in figure 3. Lastly, the

first rectangle standing upright (gray) fits perfectly on top of the two horizontally oriented rectangles.

Now, what happens is that the algorithm chooses the gray rectangle to stand upright, but then the pink rectangle, now placed in the bottom right corner of the figure, does not fit in the bounding box. As a result, the algorithm needs to place the pink rectangle in a location where it will most definitely hurt the outcome of this test.

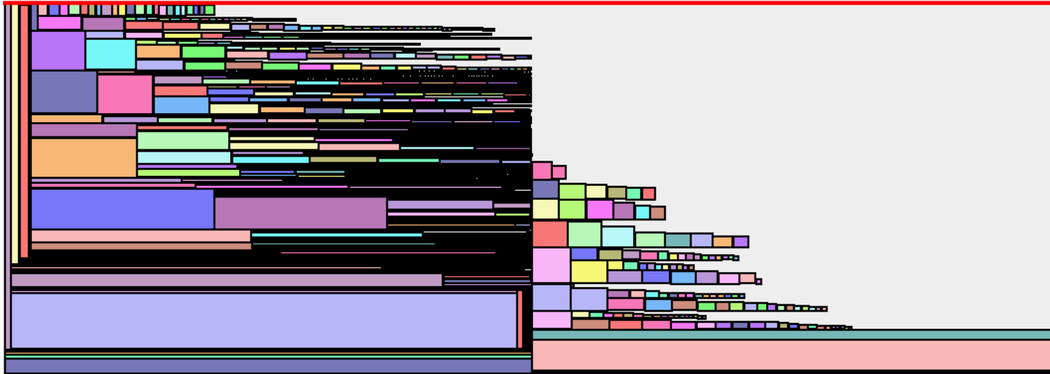

Figure 5: An anomaly in Wall Stacker (Triple Wall Stack).

In figure 6 the same case is solved by Korf2003, which is significantly more computationally expensive, but makes no such restricting assumptions.

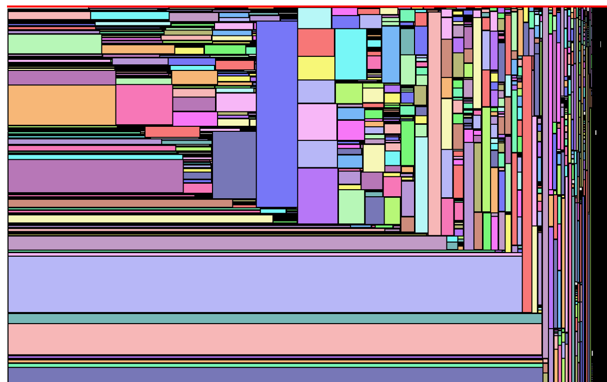

Figure 6: How Korf2003 handles the same case.

Small Variance When the rectangles are very similar - their heights and widths do not vary a lot - Korf2003 performs relatively bad. The algorithm chooses to work with a height equal to the height of the tallest rectangle. Since the smallest rectangle has almost the same height as the tallest rectangle, the rectangles will not pile up. Therefore, the algorithm computes a queue of all rectangles. This is not a very good solution, as the heights of the rectangles slightly differ, so a lot of space is wasted.

For these specific cases, Wall Stacker performs significantly better, as can be seen in the example in Figure 7.

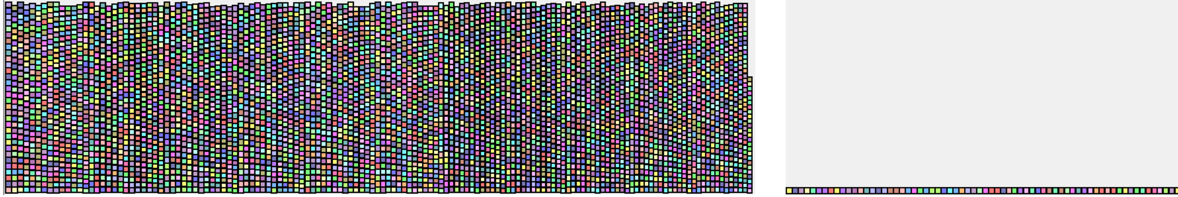

Figure 7: Difference in how Wall Stacker (left) and Korf2003 (right) handle the same case

Large rectangles OptRotMARP takes significantly longer when rectangles are very large, as also follows from the runtime analysis. For example, the case in figure 8 takes about 0.3 ms for OptRotMARP and 2.0 ms for Korf2003. When the lengths of all sides are multiplied by 1000, OptRotMARP takes about 220 s and Korf2003 about 2.1 s. For OptRotMARP, the increase in time can be explained by the grown number of possible locations of the rectangles that are tried by the brute force algorithm. For Korf2003, the growth can be explained by the number of bounding boxes the algorithm tries to fill, which linearly depends on the largest height and width of the rectangles in the set.

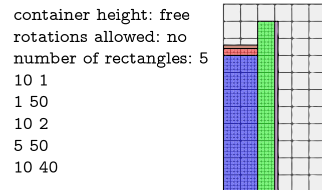

Figure 8: How the algorithm divides the space.

3.3 Timing results

The Korf2003 algorithm was run on testcases based on the consecutive square benchmark, with the amount of rectangles ranging from 26 to 1000. When the time it takes to find a solution is divided by the amount of rectangles squared for each case, as in figure 9, it can be observed that this becomes constant. This confirms the n^2 bound found for Bottom-Left packing in section 2.2.3. A large deviation can be seen in the first run (blue) on tests with 25 to about 80 recangles. This could possibly be due to optimizations within Java (like just-in-time compilation) that improve the timing results of the runs.

A similar analysis of the OptRotMARP algorithm, with the amount of rectangles varying between 1 and 14 shows that there cannot be found a very tight timebound. This is due to the pruning used by OptRotMARP, which makes some cases faster, but hardly speeds up other cases. It is not run on larger cases than those with 14 rectangles, as these runs take hours to finish.

All five algorithms described in section 2 were run on the consecutive squares, consecutive rectangles and double perimeter benchmark, with the number of rectangles varying from 1 to 25. OptRotMARP was run on these benchmarks with 1 to 11 rectangles. The timing results can be viewed in figure 10. The algorithms perform similarly relative to each other for each benchmark. Absolutely, they take more time for each of the cases from left to right, top to bottom, which confirms the relative difficulty of the benchmarks as argued by Huang and Korf [7]. Wall Stacker and Korf2003 are both very fast. For Korf2003, this is the result of differing in the amount of permutations tried when using Bottom-Left packing on cases of different sizes. For Wall Stacker, this is because once placed, the rectangles will not change position anymore. The Recursive and G-Staircase algorithms have running times in between, but quickly take significantly longer than Korf2003 and Wall Stacker.

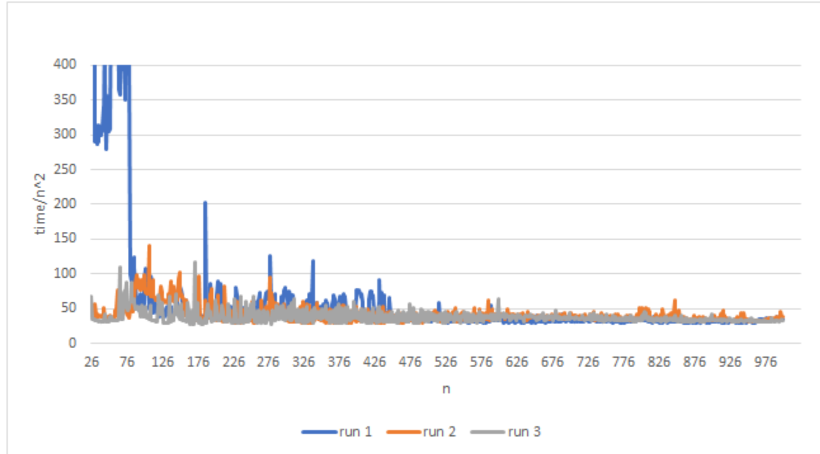

Figure 9: Korf2003 time/n^2

Similar relative results were found when the five algorithms were run on the consecutive rectangle and double perimeter benchmarks with rotations allowed. Absolutely, the algorithms took longer to run, because making the choices on rotating rectangles takes some time.

3.4 Optimality results

Consecutive squares One way the optimality of the five algorithms was tested, was by running all of them on the consecutive square benchmark, as this benchmark was used to measure performance by many papers [7]. Subsequently, the results were compared with the results given in the paper by Huang and Korf [7], which are optimal. This comparison was done based on the percentage of wasted space calculated by

$$\%\text{ wasted space } = 1 - \frac{\text{total area rectangles}}{\text{area found solution}} = 1 - \frac{\sum_{i=0}^{n-1} w_i \cdot h_i}{\text{area found solution}}. \tag{1}$$

The results are listed in table 1. It can be seen that OptRotMARP indeed gives optimal results for these cases. Furthermore, it follows that solutions by Korf2003, Wall Stacker, Recursive and G-Staircase have on average 3.4, 4.0, 3.8 and 1.4 percent more wasted space respectively than the optimal solution, respectively.

Perfect packings Another way to evaluate the results of the algorithms is by using sets of rectangles of which it is known that there exists a solution that is a perfect packing. Five times one thousand of such unique sets of rectangles with a certain number of rectangles (3, 5, 10, 25 or 5000) were generated using the method described earlier from a rectangle with a height of 543 units and a width of 632 units.

In table 2 the amount of times each of the five algorithms finds the optimal packing for each set of one thousand test cases is listed. The - indicates that the algorithm was not used on the set of test cases, because it would take too long to finish. In table 3 the average percentage of wasted space for each set of one thousand test cases is listed.

Over all, while for cases with a larger amount of rectangles, there are less and less cases that are solved optimally, solutions that are on average very close to optimal are given.

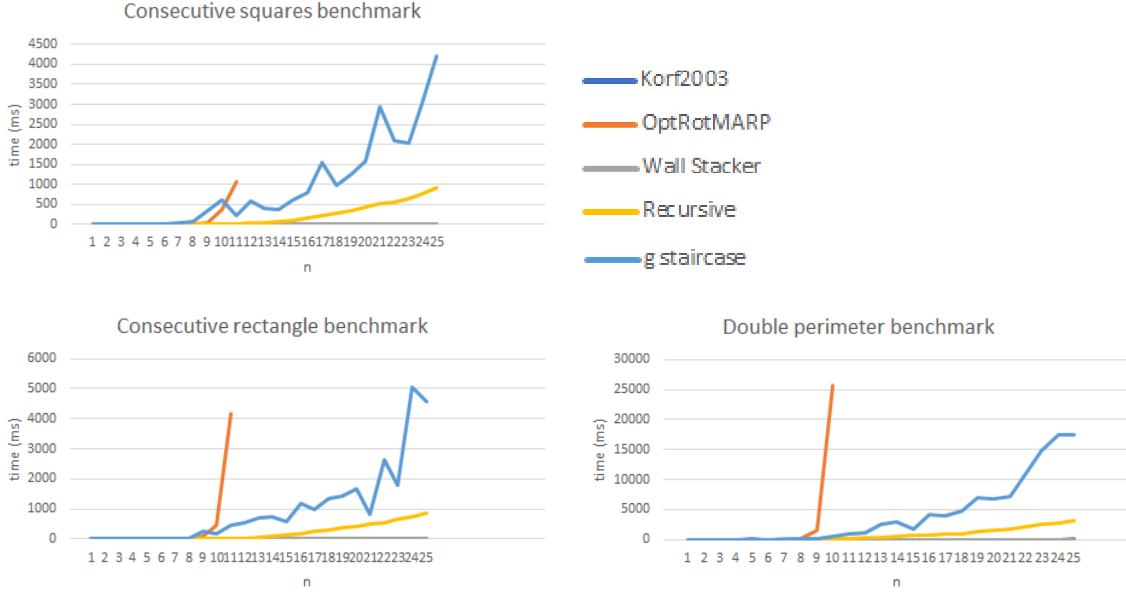

Figure 10: Three benchmarks, cases with 1 to 25 rectangles

OptRotMARP always gives optimal results, as it was designed to do. Remarkably, the worst result by Korf2003 in all these cases is one with 5000 rectangles that has a wasted space of 23.1%. This means that the solution by Korf2003 is 1.3 times the optimal solution area, which is much less than the bound of 3 times the optimal solution area found in section 2.2.2. This may be due to what is described as the empty-corner model [7], which says that every perfect packing has a solution using bottom left placements, contrary to more general problems [1].

n	optimal	Korf2003	OptRotMARP	Wall Stacker	Recursive	G-Staircase
3	1000	1000	1000	1000	1000	1000
5	1000	1000	1000	936	886	1000
10	1000	907	1000	483	732	946
25	1000	179	-	161	68	-
5000	1000	84	-	82	-	-

Table 2: Amount of perfectly packed sets with n rectangles

n	optimal	Korf2003	OptRotMARP	Wall Stacker	Recursive	G-Staircase
3	0.00	0.00	0.00	0.00	0.00	0.00
5	0.00	0.00	0.00	0.61	1.37	0.00
10	0.00	0.45	0.00	2.33	4.83	0.36
25	0.00	2.94	-	6.92	7.65	-
5000	0.00	0.11	-	0.51	-	-

Table 3: Average percentage of wasted space in 1000 runs

n	optimal	Korf2003	OptRotMARP	Wall Stacker	Recursive	G-Staircase
1	0.00	0.00	0.00	0.00	0.00	0.00
2	16.7	16.7	16.7	16.7	16.7	16.7
3	6.67	6.67	6.67	6.67	6.67	6.67
4	14.3	14.3	14.3	14.3	14.3	14.3
5	8.33	8.33	8.33	8.33	8.33	8.33
6	8.08	8.08	8.08	8.08	8.08	8.08
7	9.09	9.09	9.09	9.09	9.09	9.09
8	2.86	8.93	2.86	11.69	11.69	2.86
9	5.00	8.65	5.00	8.65	8.65	5.00
10	4.94	4.94	4.94	4.94	4.94	4.94
11	1.36	6.3	1.36	9.64	8.17	1.36
12	2.55	8.96	2.55	8.96	8.96	3.27
13	2.03	6.29	2.03	6.29	6.29	2.50
14	1.93	6.02	-	6.11	6.11	1.93
15	1.98	6.06	-	9.16	9.16	1.98
16	1.06	6.44	-	7.08	7.08	2.79
17	0.50	4.80	-	4.80	4.80	5.31
18	1.40	5.85	-	8.10	5.85	3.03
19	0.84	7.42	-	7.14	7.59	3.21
20	0.69	5.65	-	5.75	5.75	3.56
21	0.99	6.78	-	5.35	5.35	6.23
22	0.71	5.41	-	6.43	5.71	4.05
23	0.64	6.02	-	6.89	6.89	3.48
24	0.58	4.76	-	6.08	6.08	4.72
25	0.40	5.59	-	6.89	5.59	4.97

Table 1: Consecutive square benchmark wasted space

4 Concluding Remarks

The algorithms discussed in this report have been shown to give solutions to the variations of the rectangle packing problem as described in the introduction. These algorithms have been evaluated against different benchmarks, from which it is derived how they perform in terms of time and optimality.

Even though the presented results are very promising, there remain possibilities for improvement. Therefore, next to the stronger points of the algorithms, an overview of some proposed improvements for weaker points will be given here.

To start with the Korf2003 and Bottom-Left packing algorithms, it would be an improvement to have an algorithm that helps finding bounding boxes when packing a large number of rectangles. Currently, the method of finding bounding boxes for smaller input sizes does not scale to larger inputs and the selected bounding boxes for larger inputs may give bad results when there is a low variance in rectangle size. Additionally, it may be interesting to compare the current algorithm with one using a placing algorithm other than Bottom-Left packing. Further, more pruning methods may be implemented to make the Korf2003 algorithm faster. As for now, the combination algorithm performs well on a large variety of cases when looking

at solution optimality and timing.

The weak point of OptRotMARP is the computational time complexity, although this is inherent to the algorithm design. To improve the practical running time of OptRotMARP, more pruning methods could be implemented. Care must be taken that these pruning methods do not affect the optimality of the solutions returned by the algorithm. This guarantee of optimality for the solutions returned by OptRotMARP is a strong point.

Another improvement to the OptRotMARP algorithm would be to use a better heuristic algorithm than NFDH, which currently delivers relatively large initial bounds. If such a heuristic finds a tighter upper bound for the solution, OptRotMARP can prune more possibilities and thus making it faster for larger cases. A proposal would be to use the Wall Stacker algorithms as such a heuristic. Do note, however, that this only lowers the average case, and not the worst case scenario.

The Recursive algorithm was originally derived from the literature. In its current state, it does not prune any solutions and therefore recursively iterates over many possible states. For future works, an effective pruning policy could be implemented in order to speed up the execution in practice. A strong point is its relative simplicity in approach. Recursively packing rectangles is a concept that is easily implemented while delivering relatively good results.

Finally, the Wall Stacker algorithms could still profit of improvements with regard to rectangles with a large height. The algorithms are currently unable to deal sufficiently with important edge cases. A point in favor of Wall Stacker is the speed it runs at. In all cases, but especially for the larger inputs, it gives solutions relatively fast compared to the other algorithms discussed in this report.

Despite the possible future works as posed in this conclusion, a combination of all algorithms presented in this report is already capable of producing better solutions for the rectangle packing problem with different types of inputs than the algorithms on their own.

References

[1] B.S. Baker, E.G. Coffman, Jr. and R.L. Rivest. Orthogonal Packings in Two Dimensions. *SIAM J. Comput.* 9(4): 846–855 (1980).

[2] B.S. Baker, D.J. Brown and H.P. Katseff. A 5/4 algorithm for two-dimensional packing. *Journal of Algorithms* 2(4): 348–368 (1981).

[3] G. Belov, G. Scheithauer and E.A. Mukhacheva. One-dimensional heuristics adapted for two-dimensional rectangular strip packing. *Journal of the Operational Research Society* 59: 823-832 (2008).

[4] B. Chazelle. The Bottom-Left Bin-Packing Heuristic: An Efficient implementation. *IEEE Transaction on Computers* C-32(8): 697–707 (1983).

[5] E.G. Coffman, Jr. , P.J. Downey and P. Winkler. Packing rectangles in a strip. *Acta Informatica* 38: 673–693 (2002).

[6] W. Huang, T. Ye and D. Cheng. Bottom-Left Placement Theorem for Rectangle Packing. *arXiv:1107.4463* (2011).

[7] E. Huang and R.E. Korf. Optimal Rectangle Packing: An Absolute Placement Approach. *J. Artif. Intell. Res.* 46: 47–87 (2013).

[8] M. Kenmochi, T. Imamichi, K. Nonobe, M. Yagiura and H. Nagamochi. Exact Algorithms for the 2-Dimensional Strip Packing Problem with and without Rotations. *European Journal of Operational Research* 198(1): 73–83 (2009).

[9] R.E. Korf. Optimal Rectangle Packing: Initial Results. In *Proc. 13th Internat. Conf. on Automated Planning and Scheduling (ICAPS)*, pages 287–295, 2003.

[10] R.E. Korf. Optimal Rectangle Packing: New Results. In *Proc. 14th Internat. Conf. on Automated Planning and Scheduling (ICAPS)*, pages 142–149, 2004.

[11] N. Lesh and M. Mitzenmacher. BubbleSearch: A simple heuristic for improving priority-based greedy algorithms. *Information Processing Letters* 97: 161-169 (2006).

[12] H. Simonis and B. O'Sullivan. Almost Square Packing. In *CPAIOR'11 Proceedings of the 8th international conference on Integration of AI and OR techniques in constraint programming for combinatorial optimization problems*, pages 196–209, 2011.

[13] Zhang, D., Kang, Y. and Deng, A. A new heuristic recursive algorithm for the strip rectangular packing problem. *Computers & Operations Research* 33(8): 2209–2217 (2006).

Appendix A: Notation

notation	**meaning**
B	set of all possible bounding boxes of S
H	fixed height of C, equal to ∞ if the height is free
H_S	the maximum height of a rectangle in S ($\max\{h_i : R_i \in S\}$)
S	set of rectangles in the plane that need to be packed
W_S	the maximum width of a rectangle in S ($\max\{w_i : R_i \in S\}$)
n	number of rectangles
p	the number of possible configurations for a rectangle
r	boolean indicator whether rectangles can be rotated or not
R_i	rectangle i
w_i	width of rectangle i
h_i	height of rectangle i
(x_i, y_i)	position of rectangle i
O_n	the largest amount of bottom-left placements possible for a rectangle in Bottom-Left placing
A	the area of the placement by a chosen fast algorithm (e.g. NFDH)