# 2D Optimal Packing

Andrew Greensweight
*Computer Engineering*
*Stevens Institute of Technology*
Hoboken, USA
agreensw@stevens.edu

Brandon Tai
*Computer Engineering*
*Stevens Institute of Technology*
Hoboken, USA
btai2@stevens.edu

Roni Rahman
*Computer Engineering*
*Stevens Institute of Technology*
Hoboken, USA
rrahman2@stevens.edu

***Abstract*—The two-dimensional bin packing problem is concerned with the optimal containment of some number of smaller rectangles within a large boundary rectangle. This problem is just one sub-classification of the knapsack problem, which is concerned with maximizing the value of items in a one-dimensional container. Herein the problem is defined to be a 2D knapsack problem with no rotation where the containment bound is a large rectangle, where the sub-items to be packed are smaller rectangles and all lengths are integer; the objective is to find the best container for all objects which minimizes the difference between the length and width of the boundary rectangle. Other authors in this field consider variations where each sub-item can be assigned a weight independent of its area and for sub-item rotation, which add complexity but remain similar to the aforementioned problem statement. Although the two-dimensional bin packing problem has no general solution, there are several approaches that have been devised and published - guillotine cuts, where the containment boundary is subdivided into yet smaller rectangles; integer linear programming, a type of mathematical optimization problem that attempts to define the packing problem in terms of an object problem and constraints; and binary tree corner searching, where sub-items attempt to be placed at corners (either to the right or below) where space is available. The binary tree corner search approach is the most widely cited and is used for the defined problem.**

***Index Terms*—optimal packing, knapsack problem, growing containers**

## I. Introduction

Knapsack problems are defined as problems in which "a set of items are given, each having an associated value and size, and it is desired to select one or more disjoint subsets so that the sum of the sizes in each subset does not exceed (or equals) a given bound and the sum of the select values is maximized" [1]. The nomenclature "knapsack" stems from the simple example in which a hiker has to fill up his or her knapsack by selecting and packing among various objects to achieve maximum comfort. Mathematical formulation of this problem can be determined by numbering the objects from 1 to *n* and introducing a vector of binary variables $x_j$ (where j = 1,...,*n*) with the following meaning [2]:

$$x_j = \begin{cases} 1 & \text{if object j is selected} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Additionally, we can complete the mathematical formulation for the simple hiker knapsack problem by defining the following parameters:

- "Comfort" level for the hike given by each object, *j*: $c_j$
- Size of the object: $w_j$
- Size of the knapsack: $s$

Selection for placing in the knapsack must then satisfy the following constraint:

$$\sum_{j=1}^{n} w_j x_j \leq s \tag{2}$$

Satisfying this constraint maximizes the objective function [2]:

$$\sum_{j=1}^{n} c_j x_j \tag{3}$$

At their core, knapsack problems, also referred to as packing problems, are optimization problems centered around the objective of maximizing material utilization and minimizing unused space [3]. Packing problems are encountered in many industries, with each industry requiring unique constraints. For example, industries such as glass, paper and wood are concerned with the packing of clearly defined shapes, whereas an industry such as ship building is concerned with packing irregular shaped items [3]. This project is concerned with packing two-dimensional rectangles into a growing container, with constraints placed on the orientation of the rectangle being placed. Additionally, this project is constrained by the requirement to pack a predefined set of rectangles which may be identical in terms of their dimensions. The objective is to pack all rectangles defined in the set into a growing container while minimizing the area and "wasted" space of the final container.

The aforementioned constraints were selected for the purpose of addressing real world applications in which the orientation of the placed items, or rectangles in this case, is critical. For example, in FPGA design there is a specific orientation of submodules such that the final board layout is usable. Because FPGA designs are inherently dependent on the number of combinational look-up tables and registers, and FPGAs can support varying numbers of these parameters, the problem of fitting submodules onto the FPGA can be modeled as this packing problem. However, the number of look-up tables cannot be swapped with the number of registers - in effect, this is equivalent to placing a restriction on rotation.

## II. Problem Statement

The optimal bin packing problem is an extension of the well-known one-dimensional knapsack problem. As a refresher, the one-dimensional knapsack problem is concerned with maximizing the value of "items" to be placed within a "knapsack," where the knapsack and each of the items have a finite capacity. When the problem is extended into the second dimension, there are additional complexities that can be taken into account - such problems usually involve either a fixed container, and determining the optimal subset of items to place within the container; or a fixed number of items to be placed in an unknown sized container, where the required size of the container to do so is in question. The former has applications similar to that of the one dimensional knapsack problem, while the latter has applications where all items must fit into the container (such as in critical shipping applications or FPGA design). Further consideration is restricted to this latter case.

This characterization of the two-dimensional knapsack problem is not a formally solved problem; despite this, one of the most popular approaches to this problem is to consider the problem using a binary tree [4]. In short, after sorting an array of rectangular items to be placed in descending order of size and placing the first initial rectangle, which serves as the root node, the next rectangular item can either be placed at the (top) right corner of the previous, or at the bottom (left) corner of the previous, which become left and right children of the node, assuming that the enclosing boundary is large enough to contain the next rectangle in the proposed direction. If there is not enough space, then a check is performed to grow either rightward or downward so that the overall shape of the boundary is as square as possible. The placement and expansion of the boundary continues until all rectangles are placed and the final boundary is reported. This slight simplification of the problem allows the tree to be purely binary. In addition, rotation of the rectangular items is not performed; while the mapping of some arbitrary two-dimensional axes into rectangular items may at first glance suggest that 90 degree rotation can be performed, restricting the rotation is common in order to simplify the problem space and due to applications where the horizontal and vertical dimensions are not the same and would not make logical meaning to do so.

## III. Results and Discussion

For any set of rectangles input into the algorithm, the rectangles are first sorted. This is achieved by determining the longest side length of each rectangle and sorting them in descending order of their longest side length. If two rectangles have the same longest side length, they are sorted by the order in which they are input into the algorithm, not by the length of the second dimension. This is important because it reduces the time complexity of the sort function to solely check the rectangles on one dimension. To initialize the system, the first rectangle of the sorted list which has the longest side is used to calculate the size of the container. The container size is set equal to the size of the first rectangle for placement. The largest rectangle is then placed in the top left corner of the container to complete the initialization process. For any given rectangle that has already been initialized, only two possible growth locations will be considered for the next rectangle in the sorted list (Fig. 2). The algorithm checks the right and down positions, denoted by the green circles, to determine if either placement of the rectangle would result in the container not growing. If this is not possible, the algorithm proceeds to grow the container in the direction resulting in the new container being as square as possible. Given the binary decisions being made by the algorithm, a binary tree is deemed to be the best data structure to implement the algorithm with each rectangles position and ID being stored in the down or right child of each node.

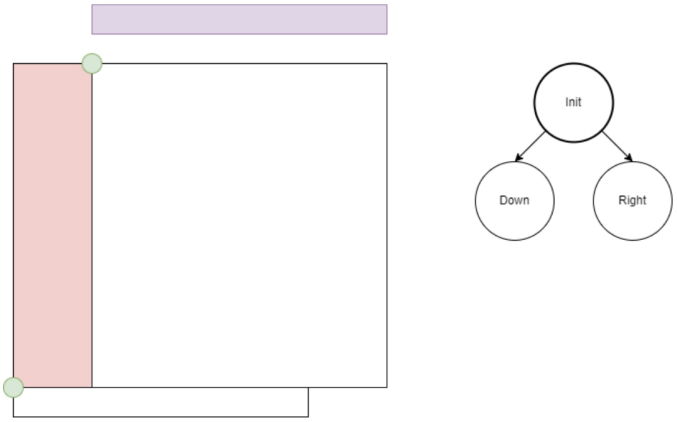


Fig. 1. Binary tree algorithm scaffold. (Left) visual representation of packing positions on the initial item (pink rectangle, vertical), with possible down position (green circle, bottom left) or right position (green circle, top right) attachment points. The position of the next item to be packed (purple rectangle, horizontal) will be stored at either attachment positions of the initial item based on the decision made by the binary tree (Right), where the only two choices from the initial item (init) are down or right.

Five test cases were evaluated using the algorithm. Each test case was read from a text file with each line in the file containing, left to right, a block ID, width and height. For each test, a sorted list of rectangles followed by the coordinates they were placed in and the final size of the container is printed. Results are considered to be optimal if the final container is the smallest possible container that can contain all the objects from the list of rectangles. These test cases discussed here were chosen so that they are small enough in scope to verify and to stress test the edge cases of the algorithm when applicable. They are considered to be not optimal if a different arrangement of said rectangles would result in a smaller container. The first case is a simple two block test in which optimal packing was achieved (Fig. 2). The algorithm placed the boxes in a 3x3 container with no empty spaces as would occur had it placed the boxes in a linear arrangement as a brute force approach would. This arrangement is considered optimal as no other solution exists which would result in a smaller container.

The second test case (Fig. 3) consists of a 3x3 rectangle

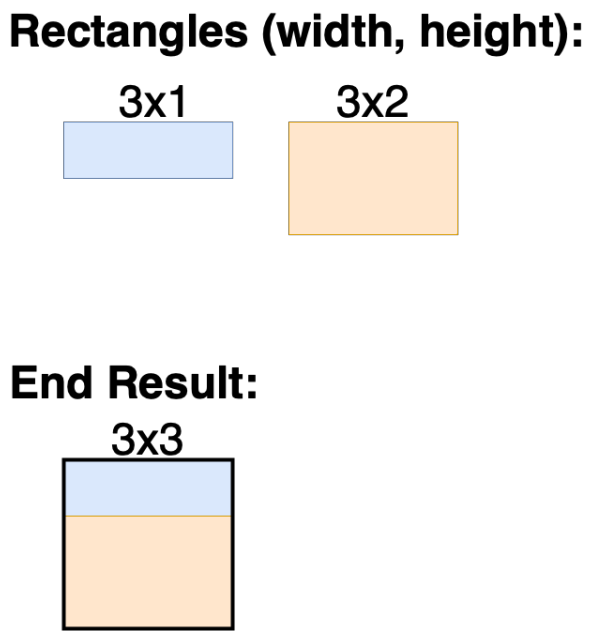


Fig. 2. Test case 1. Optimal Placement of two rectangles, one rectangle of size 3x1(blue), and the other rectangle of size 3x2 (orange).

along with two identically sized 2x2 Optimal Placement of two rectangles, one rectangle of size 3x1(blue), and the other rectangle of size 3x2 (orange). In this case optimal packing was not achieved as the algorithm placed the third rectangle (blue) on the bottom right resulting in a 5x5 container whereas an optimal placement (on the right, under the green rectangle) would result in a 5x4 container. This is because the algorithm first checks the right child of the purple rectangle which is already occupied by the green rectangle, thus it places it in the down child of the tree, attempting to keep the container as square as possible - thus, the corner where the purple and green rectangles meet is not checked.

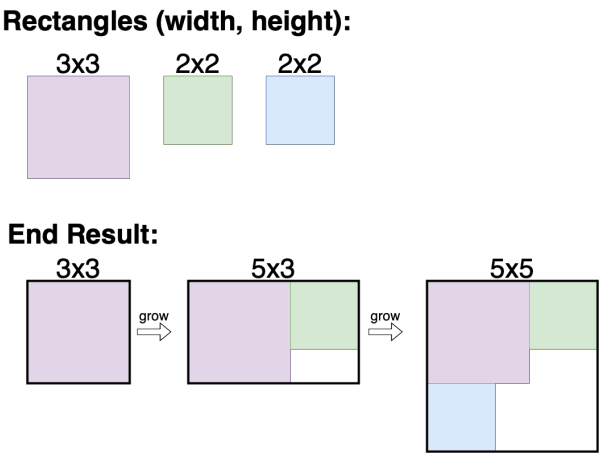


Fig. 3. Test case 2. Non-optimal Placement of three rectangles with one rectangle of size 3x3 (purple), and the two other rectangles of size 2x2 (green and blue). Intermediate steps between the placement of the initial rectangle and the final result and their associated container growth is shown.

The third test case (Fig. 4) features three rectangles of differing sizes with the longest being 5x1 (blue) which is placed first resulting in a 5x5 container. Next the 2x4 rectangle (green) is placed in the down location as the right location would result in container growth. The final rectangle of size 3x3 (purple) is placed in the right child of the green rectangle as once again this would result in the container not needing to grow. This result is optimal as all rectangles were packed in the initial 5x5 container with no other possible arrangement resulting in a smaller container.

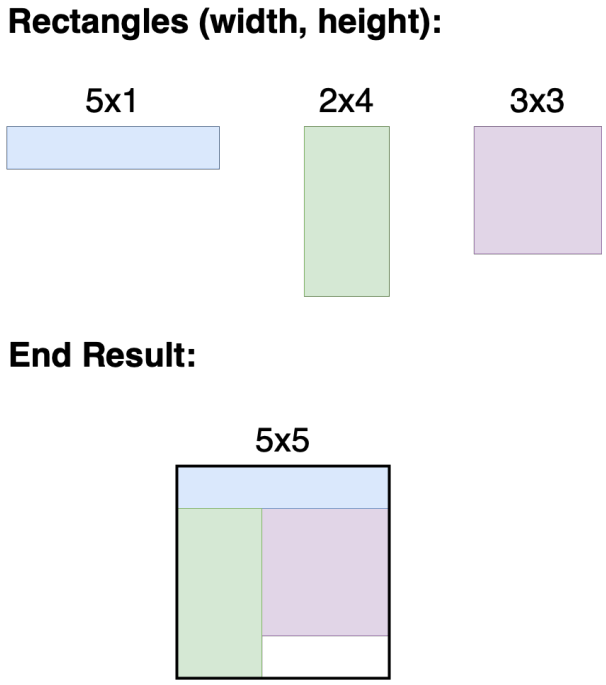


Fig. 4. Test Case 3. Optimal Placement of three rectangles with one of size 5x1 (blue), another rectangle of size 2x4 (green) followed by a final rectangle of size 3x3 (purple).

The fourth test case (Fig. 5) features seven rectangles with many of them being close in size to each other. In cases where ties are more prevalent, since the algorithm does not do any sort of tie-breaking, non-optimal placement is more likely to occur. This resulted in non-optimal placement as placement below the light blue box would result in a reduction of the height of the container by one square to 9x6.

The fifth test case (Fig.6) consisted of rectangles of randomly generated size. Random.org was used as the random number generator for the width and height dimensions. The algorithm placed the rectangles fairly well as shown in (Fig. 7). In this instance, the placement of the larger rectangles along the top of the container greatly limited the possible options for placing the remaining rectangles. As shown in the other test cases, the algorithm favors a square final container size.

Our calculated time complexity for the algorithm by evaluating given numbers of rectangles with randomly generated widths and heights results in an approximate time complexity of O($n^2$). Most of the literature encountered pointed to a

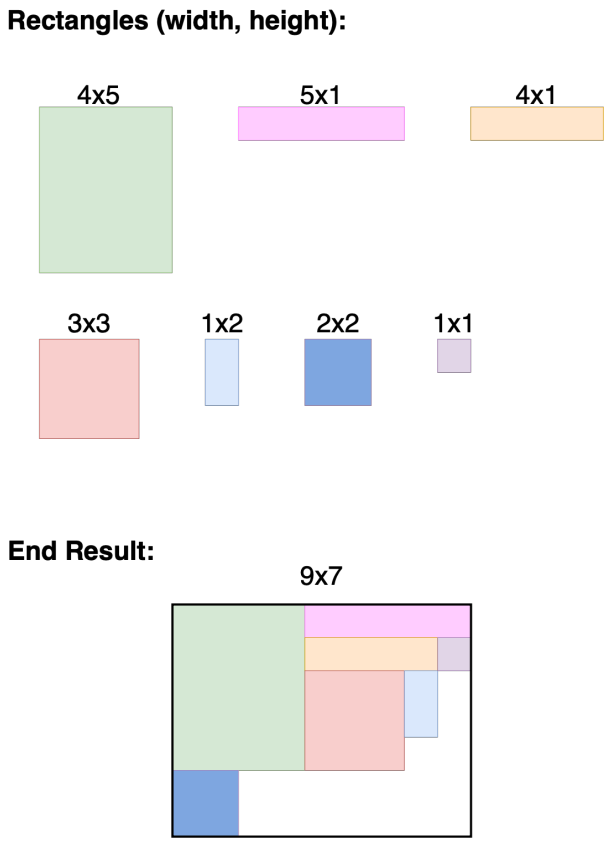


Fig. 5. Test Case 4. The Non-optimal placement of seven rectangles of sizes 4x5 (green), 5x1 (pink), 4x1 (orange), 3x3 (red), 1x2 (light blue), 2x2 (blue) and 1x1 (purple).

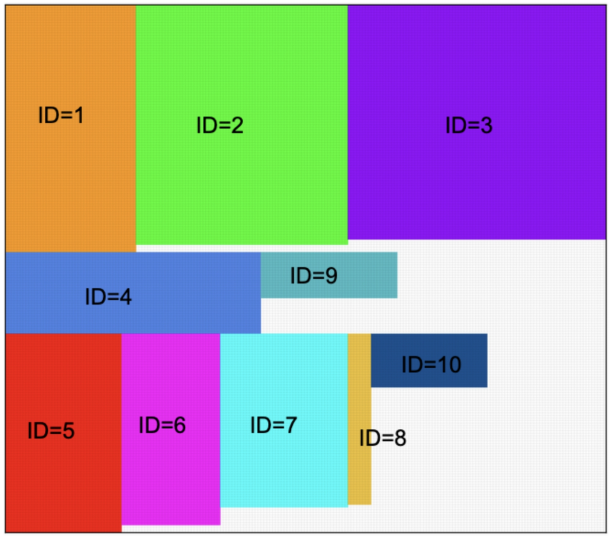


Fig. 6. Test case 5. Randomly generated test case where rectangle of ID 1 is 8x67, 2 is 40x78, 3 is 45x97, 4 is 88x32, 5 is 44x68, 6 is 34x75, 7 is 47x18, 8 is 40x21, 9 us 89x92, and 10 is 73x97 in size.

naive worst-case complexity of O($n^4$) to O($n^5$), though with more optimized approaches having complexity of O($n^3$) and even sometimes O($n^2$) [4]. The estimate of O($n^2$) appears to be due to the simplifications made in the approach, such as removing the ability to rotate the rectangle and sorting based on one check for the largest dimension for each rectangle placed. Of particular note, 24 seconds was required for 100,000 rectangles; 240 seconds for 200,000 rectangles (doubled elements but a factor of 10 for time); 516 seconds for 500,000 rectangles; and 7,400 seconds for 1 million rectangles. As previously discussed, various assumptions were made to reduce the computational complexity of the algorithm which resulted in O($n^2$) complexity.

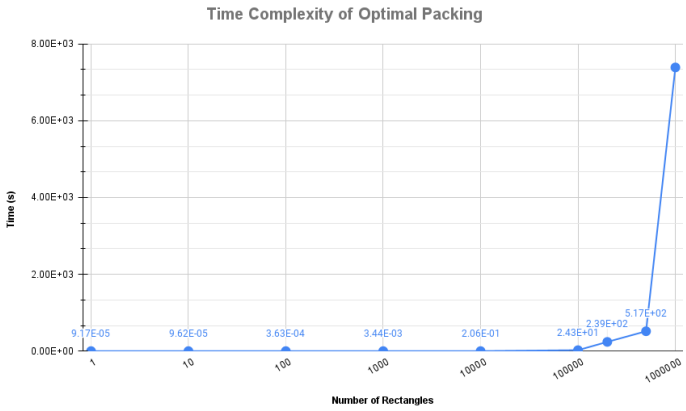


Fig. 7. Time complexity analysis of the implementation of the optimal packing algorithm. The data are presented with the number of rectangles as the independent variable in log-10 scale and the total time of execution as the dependent variable. It can be seen that the execution time begins to noticeably ramp up when the number of rectangles to pack begins to exceed 100,000.

## IV. Conclusion

Based on the test cases observed, it is apparent that the simplifications made to reduce the overall computational complexity of this algorithm can result in the algorithm returning an almost perfect solution, but ends up falling just short of the mark. In particular, when there are rectangles very similar in size or sharing the same size, the boundary will be grown to accommodate the rectangle in excess of the minimum amount to grow. This arises due to only sorting the rectangles by their longer size, and could instead perhaps use the shorter side to further sort rectangles by their shorter size to tiebreak when necessary; the accuracy of the algorithm would improve, though at the cost of increased computational complexity. Similarly, allowing the rotation of the rectangles could allow for some of the unused space to be filled in better. Once again, however, the problem that the bin packing attempts to model may not physically allow for rotation (where the width and height represent parameters in different physical dimensions), and the trade-offs for allowing it are the same as adding the additional check to the pre-sorting of the rectangles.

## References

[1] S. Martello, P. Toth, "Knapsack Problems: Algorithms and Computer Implementations", John Wiley and Sons, Inc. ISBN: 978-0-471-92420-3, 1990.

[2] F. Clautiaux, R. Sadykov, F. Vanderbeck, and Q. Viaud, "Combining dynamic programming with filtering to solve a four-stage two-dimensional guillotine-cut bounded knapsack problem," Discrete Optimization, 2018.
[3] E. Hopper and B. Turton, "An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem," European Journal of Operational Research, 1999.
[4] B. Chazelle, "The bottom-left bin-packing heuristic: An efficient implementation," IEEE Transactions on Computers, 1983.