CO1406 – Algorithms and Data Structures

Tutor: Panayiotis Andreou

Assignment 1 – Container Packing Algorithm

Serdyukov Antoniy – G20982067

Deadline 21/03/2023

Report

This report aims to analyze the requirements and complexity of the algorithms in general and functions of said algorithms in particular. Additionally, several heuristics will be discussed that were chosen to significantly speed up the algorithm.

Requirements Analysis

Problem outline: The algorithms is supplied with a set of boxes and a container. It should determine a possible solution/solutions (if any).

1. Datatypes

The first decision that had to be made, related to the data structures datatypes. Since the algorithms deals with a “real” example, it is safe to assume that the negative numbers would not be used. Thus, by using unsigned types, we can double our limits. From now on, every type is unsigned.

Furthermore, the data types themselves must be considered, i.e., the sizes of the box, container, stack and counters. It was decided that the size of the container will be stored in an unsigned short, and unsigned char. A bigger data type was not chosen because, a) it is unrealistic to have containers/boxes that are bigger than the Earth, b) if the container is big enough, and the box is small enough, it might take all the time in the Universe to calculate that: for example, container of type int and box type short, then there would be 4,295,098,369 boxes that would fit in the container, and iterating through all of their positions and rotations is infeasible. Thus, the size for container was chosen to be short and type char for box. Since, the smallest box size is 1, the container should be able to fit 4294836225 boxes, therefore number\_boxes should be int. This means almost 4 Gb of boxes, 8 Gb if the short is used instead of char. Nevertheless, the code is structured so that it could be scaled to bigger numbers, in case we need to accommodate more accurate dimensions, like cm, mm and so on.

For the stack, the same type is used as in the container (short) – automatically, with template – because the stack is storing the coordinates of the placed box and cannot exceed the size of the container. Additionally, the stack contains the pointer to the box. The alternative would be to store the box itself, which would be 4 times less space, but if the box size were to change – this changes accordingly. Therefore, the stability here is chosen over space complexity.

1. Possible solutions

During development of this algorithm there were identified two approaches to solving this problem. Since the overarching approach cannot be changed (only backtracking, no recursion), there were several things to consider, i.e. how to place/remove boxes and how to choose the next position of the box.

Regarding the first, the box could be “manually” placed: the algorithm needs to check every square unit that it is empty (does not collide with any box) and then again put in each unit the box’s name to indicate occupation. Therefore, it takes n2 where n is area of the box. The same time complexity needs for removing the box – n. Therefore, the total time complexity is O(n3).

The other approach is mathematical. By testing box-to-box collision, the worst-case scenario is n, where n is the number of boxes in the stack. Removal of the box is O(1) – simple stack.pop(). Therefore, the total time complexity is O(n).

Thus, the second approach was chosen.

Complexity Analysis

“loadPackerProblem” and “solveProblem” functions.

The first function’s complexity depends on the number of boxes and size of the container. First, it needs to initialize n arrays of the container, where n is width of the container, and then write n2 chars into each array – O(n3) complexity. After that it need to read and write m boxes on to array of boxes – O(m) complexity. As a result, final complexity is O(n3), O(m).

As for the second, we need to distinguish between solving for one and all solutions. All the cases need to be simplified, i.e. the boxes are 1x1. In the first case, the worst-case would be that for every box, we have to move every other box on every cell of the container: n\*m! where n is number of boxes and m is number of possible box placements in the container. At the same time it means that during this process (not considering worse case where we do not find a solution), we would find all the solutions.

Thus, the final time complexity if O(n\*m!).

Heuristics

Two heuristics were used that could answer the corresponding two questions: how do we chose the box, and how do we choose where we place the box.

The answer to the first one is sorting the boxes in descending order by their area. This allows to quickly reduce the size of the container, thus leaving a much simpler problem. Additionally, if the big boxes were to be placed last – more boxes would have to be moved around, increasing run time by orders of magnitude. Finally, this allowed to have definite understanding where the algorithm is in terms of finding all of the solutions, i.e. if at the start the biggest box is at the top-left, and right now it is at the bottom-right this means that we have scanned all the solution space (provided, of course, the rotation was used).

As for the second one, a modified “Improved-Bottom Left(BBLT)” was used (Terashima-Marín, 2008). In the paper the algorithm places the box at the available place and slides it down (until hit) and to the left moving along(!) the contour lines of the already placed boxes. In this algorithm however, the box is placed at the left-top corner and slide to the right and then down – “scanning the first available spot which would be the closest to the top. Essentially, these two algorithms are mirroring each other, and would produce largely same result.

If the box is popped from the stack, it is moved 1 unit to the right and then “mBBLT” is applied.

Reference list

Terashima-Marín, H. *et al.* (2008) “Generalized hyper-heuristics for solving 2D regular and irregular packing problems,” *Annals of Operations Research*, 179(1), pp. 369–392. Available at: https://doi.org/10.1007/s10479-008-0475-2.