1 4 9 5

UNIVERSITY OF
ABERDEEN

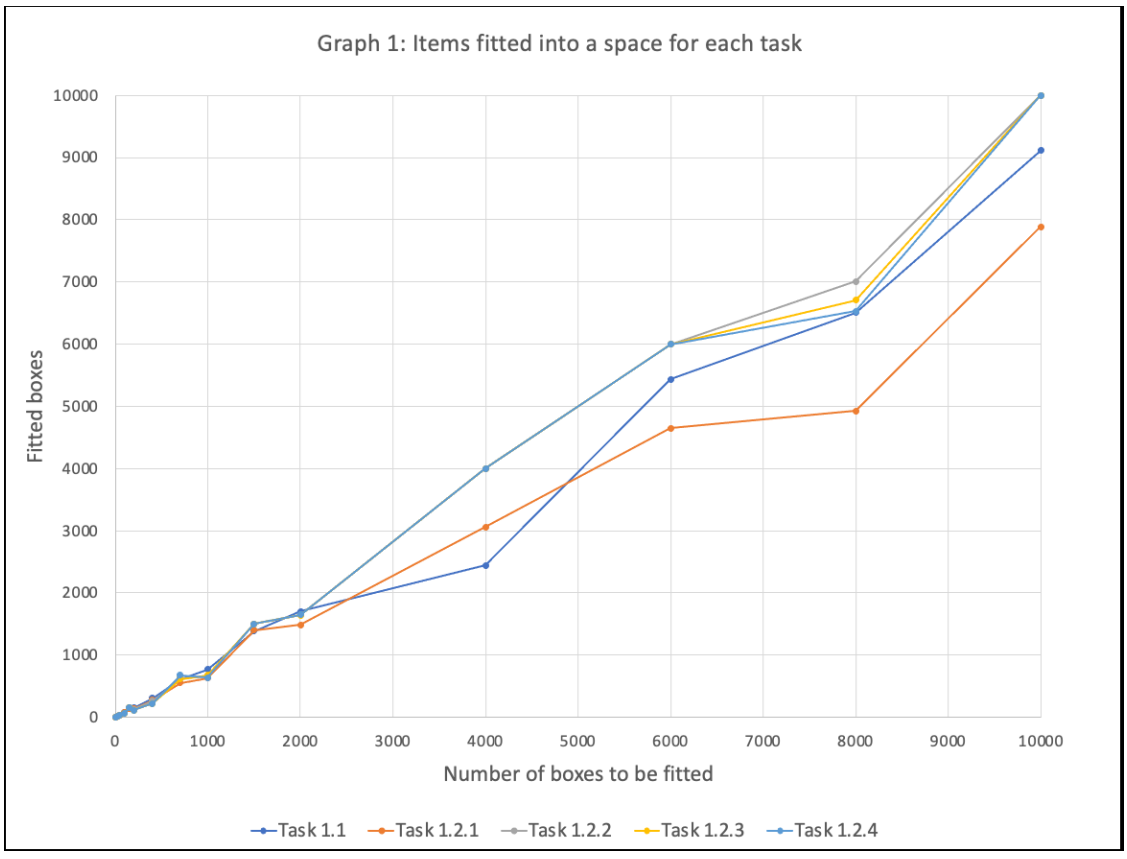
Evaluation of different approaches
while inserting boxes into area
Author: Wiktoria Szitenhelm

Task 1.3 – 2D part

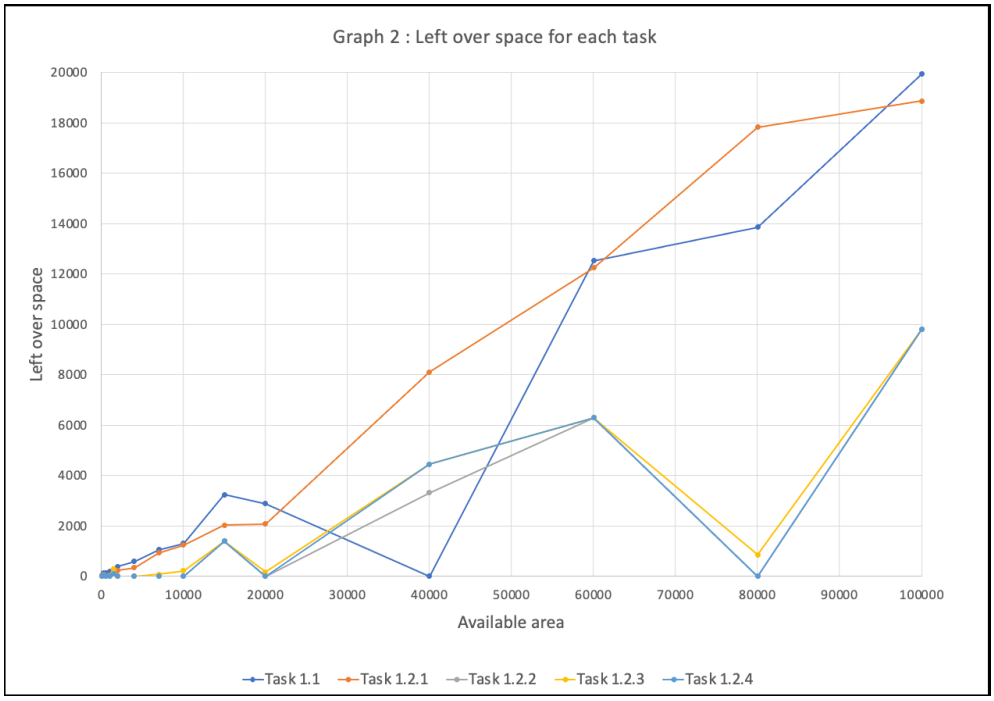
To determine which of the approaches is the most efficient in terms of fitting the most items into a space, of how much space is left over for each approach, and in terms of runtime I ran multiples instances of the problem with different sizes of inputs. Statistical analysis supports confidence in my results.

I started with sample input that has 10 boxes and area equal 100. While changing the size of the input (increasing number of boxes) I was proportionally increasing the size of the area to keep my results as reliable as possible. The largest input was equal to 10000 boxes with area size equal to 100000.

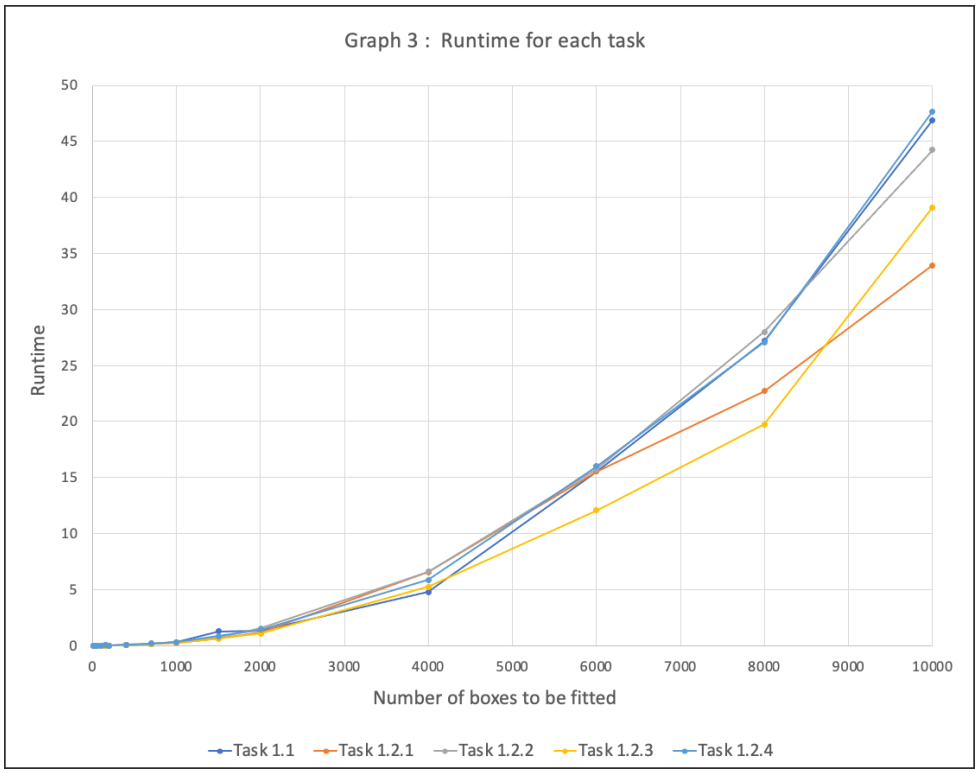
Starting with analysing items fitted into a space, from the Graph 1 it can be easily seen that approaches in both Task 1.1 (no sorting) and Task 1.2.1 (sorting by x coordinate) fitted the smallest number of boxes. The rest of the approaches were equal for the number of boxes less than 6000. Exceeding 6000 boxes approach in Task 1.2.2 (sorting by y coordinate) dispelled doubts and became the most efficient in terms of fitting the most items into a space.



In terms of how much space is left over, Graph 2 shows that again approaches in Task 1.1 (no sorting) and Task 1.2.1 (sorting by x coordinate) are the least efficient. The efficiency of the rest is similar, but for the bigger numbers approach in Task 1.2.4 is the most efficient.



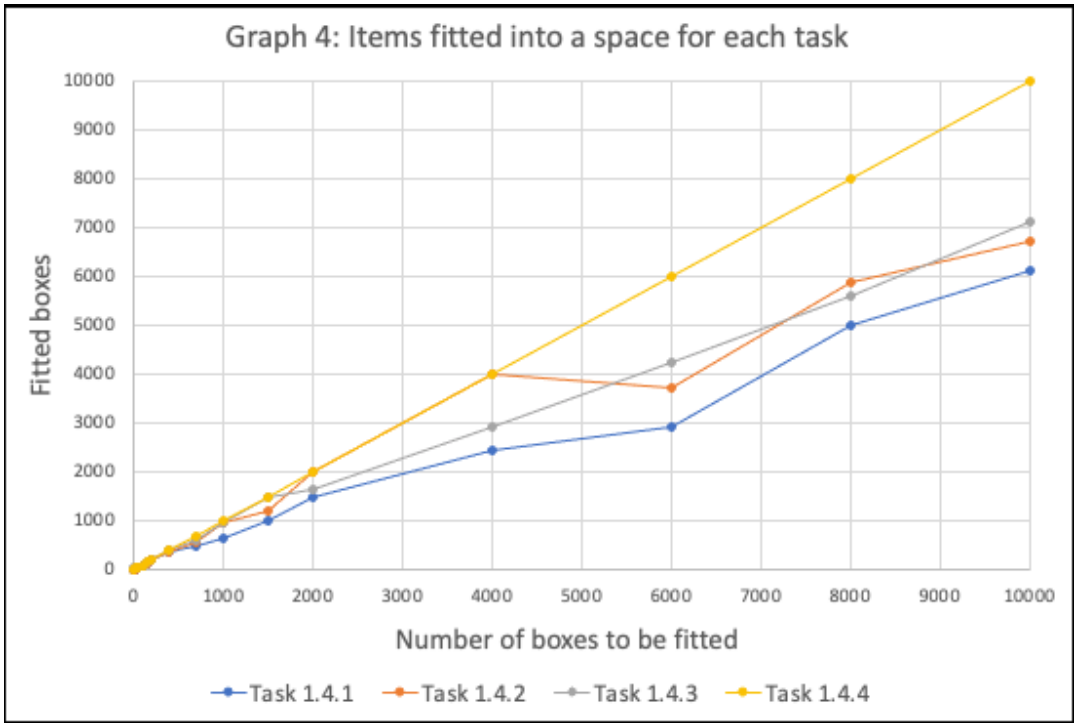
Analysing runtime for each approach.
Based on Graph 3 I can say that for small inputs the difference of the runtime is not big, but for the bigger inputs approaches in Task 1.2.1 (sorting by x coordinate) and Task 1.2.3 (sorting by the area) are the most efficient.



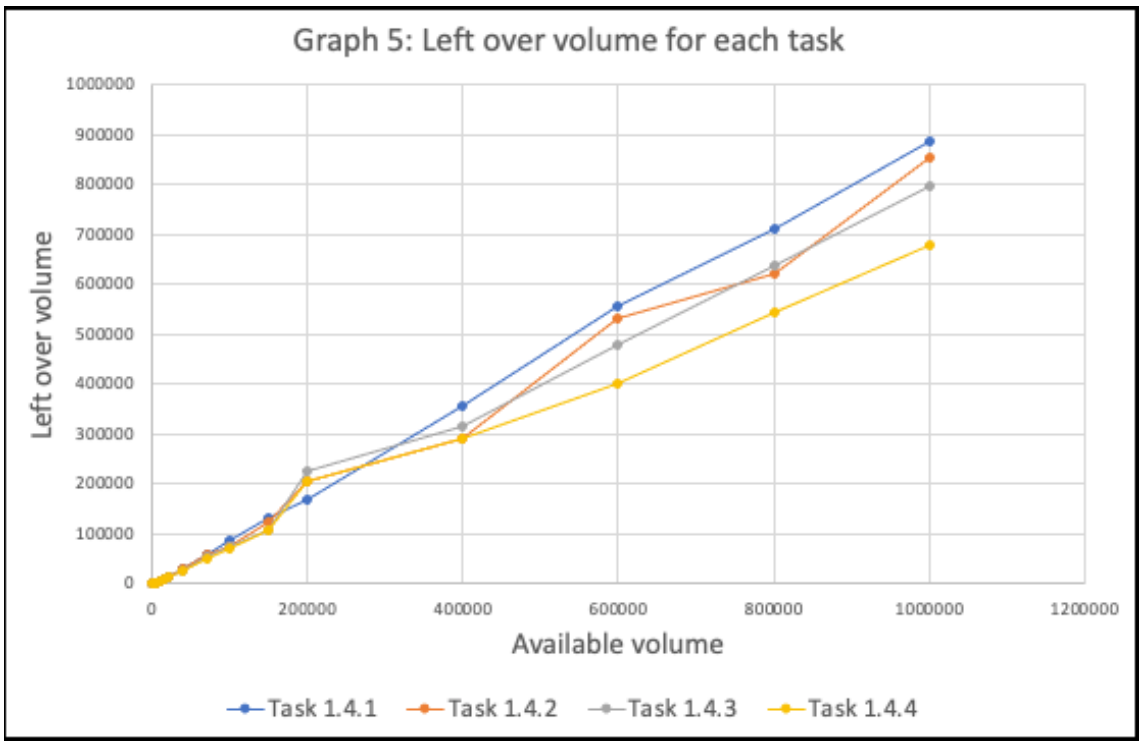
Having carefully analysed all three graphs I set that approach used in Task 1.2.3 (sorting by area) is the best because it is very close to the best approaches in terms of fitting items into a space and in terms of space left over and at the same time has one of the smallest time complexities.

Part of Task 1.4 – 3D part

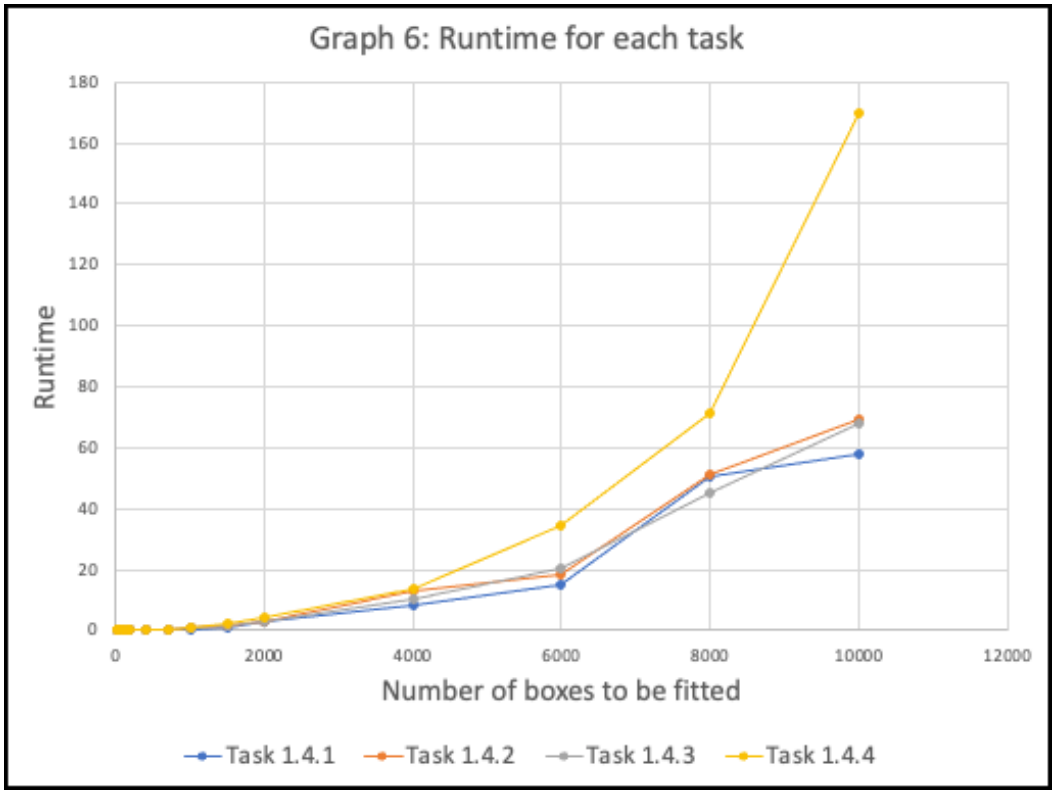
Graph 4 shows the huge difference between all the approaches used to fit items into a space. Approach used in Task 1.4.4 (sorting by the area and then rotating) is the best in terms of fitting items and even for the big inputs by this approach all the boxes are able to fit.



Similarly for the left over volume, approach used in Task 1.4.4 is the most efficient because by fitting the biggest number (sometimes equal to the number of all the boxes) of boxes leaves the least volume.



Graph 6 shows that determining the best approach in 3D volume is difficult because the most efficient approach in all the terms despite the runtime, is the worst in terms of the runtime. Because the runtime for the rest of the approaches is similar, the best approaches are used in Task 1.4.2 (sorting by y coordinate) and Task 1.4.3 (sorting by the area) having low time complexity and being efficient with regard to number of fitted boxes and left over volume.



Task 2.4

Starting with approach used in Task 2.1. Dijkstra Algorithm implemented without priority queue has the worst-case runtime equal to O(V^2) where V is number of vertices. For each vertex (V) all connected edges need to be checked in order to find the minimum cost that connects a vertex to V so it needs to do V number of calculation and every calculation takes O(V) time. Implemented Dijkstra Algorithm could be improved by using Binary Heap and Priority Queue what would effect in worst-case runtime equal to O(E log V) where E is number of edges. Space complexity is equal to O(V). Task 2.2 by recursive approach has a big worst-case runtime because it goes through all nodes (V) for all nodes so the worst-case runtime is equal to O(V*V). By using dynamic programming based approach for the recursive algorithm of Task 2.2 in Task 2.3 worst-case runtime is much smaller. Some nodes being a neighbour of many nodes occur couple of times so by using caching code avoids counting the same values couple of times and is able to reuse computed caching values. The main aim is to optimize the program by reducing the repetition of values and by smart caching of recursive functions program does it. The worst-case space complexity is slightly bigger because of keeping track of the computed values.

Task 2.5

From the plotting the time taken for different numbers of chains and different chain lengths it was easy to see that the difference between runtime was huge. It's all because of the very effective strategy – caching that gives most of the benefits of dynamic programming. As said in the lectures slides this strategy trades off computation time with storage space, but for the big output, runtime of Task 2.3 was not two or four times but many times smaller than runtime of Task 2.2 so if storage space is not limited it's worth it.