Project 1.1- Phase 3

Group 17

January 22, 2024

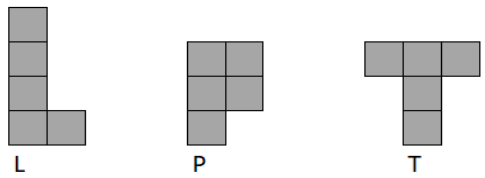
Ron Adar, Sheena Gallagher, João Tavares, Sarper Yuksel, Alexandra Valentina Stan, Julian Nijhuis

Abstract

The three-dimensional knapsack problem is a packing problem in which you are given a container and a set of objects and you are required to find the best way to pack these objects in the container with various constraints. There are a number of subproblems: you are required to find the most optimal way to fit these objects in the container, Each object is assigned a value and the most optimal way to fill the container is determined by the total sum of these values.

1 Introduction

This paper presents a version of how to solve the 3D knapsack problem. The problem is how to fill a container with dimensions: 16.5 x 2.5 x 4 The container must be filled with pentominoes (see fig 1) or parcels (see fig 2).A pentomino is made up of 5 blocks and they are placed in every possible iteration to create shapes. There are twelve possible shapes of pentominoes and we have used three of these shapes (see figure 1) in order to fill the container. The dimensions for the parcels are:



(a) Pentominoes used to fill the container

piece	size	points
A	1 x 1 x 2	3
B	1 x 1.5 x 2	4
C	1.5 x 1.5 x 1.5	5

(b) Parcel dimensions

Figure 1: The different objects used to fill the container

The approaches we explored to solve the task are a dancing links algorithm developed by Donald E. Knuth (2000) and explained in relation to adapting it to a game by J. Chu (2006). The second approach is implementing a dynamic programming algorithm- which involves a recursive function with a database that saves states that have even encountered before. This helps avoid repeat calculations and improves the efficiency of the algorithm.

The research questions answered in this paper relating to the experiments for our algorithms are:

- Is it possible to fill a container without any gaps using A, B and/or C parcels.
- If the values 3, 4 and 5 are assigned to the parcels A, B and C respectively, what is the maximum value the container can store.
- Is it possible to fill a container so there are no gaps with L, P and/or T pentominoes.
- If the values 3, 4 and 5 are assigned to the L, P and T pentominoes respectively, what is the maximum value the container can store.
- How does input order affect the algorithms runtime/score/fill result.
- How do the results from the dancing links and dynamic programming algorithms differ for pentominoes and parcels

2 Methods and Implementation

2.1 Dancing Links

We developed our dancing links algorithm based on the research of Donald E. Knuth Knuth (2000). The algorithm is an exact cover search algorithm that works by disconnecting and connecting columns and nodes from a doubly linked list, where a solution is a state where all columns are disconnected from the header. The algorithm presents a number of problems in order to implement: translating a three dimensional problem into one dimension, disconnecting and connecting columns and rows.

2.1.1 How it works

The goal of the algorithm is to find a combination of shapes in a location, such that there is no overlap between the shapes and they cover all the cells in the container.

Each column represents a cell in the container and each row represents a different placement option of a shape. A node in the row of the column represents a cell in the container the shape occupies. That means that we want to find a combination of rows such that there is no overlap between them and all the columns have exactly one full cell under them.

The algorithm is a recursive algorithm. Each time the function is called, the algorithm chooses the column with the smallest number of nodes and goes to the first node of the column and chooses its connecting row. After a row has been chosen, we hide the columns that are connected to it, disconnect all the rows that overlap with the chosen row, and add the row to the partial solution. We then call the algorithm again with the task of completing the partial solution, if the algorithm fails, we reconnect the rows and the columns and move one node down and again, disconnect the columns and rows and call the algorithm again. If all the nodes under the column have been checked, we return false. If the algorithm finds a solution- which means the header does not point to any columns- the algorithm terminates successfully. If the algorithm returns to the first column and does not find a solution, the algorithm terminates unsuccessfully.

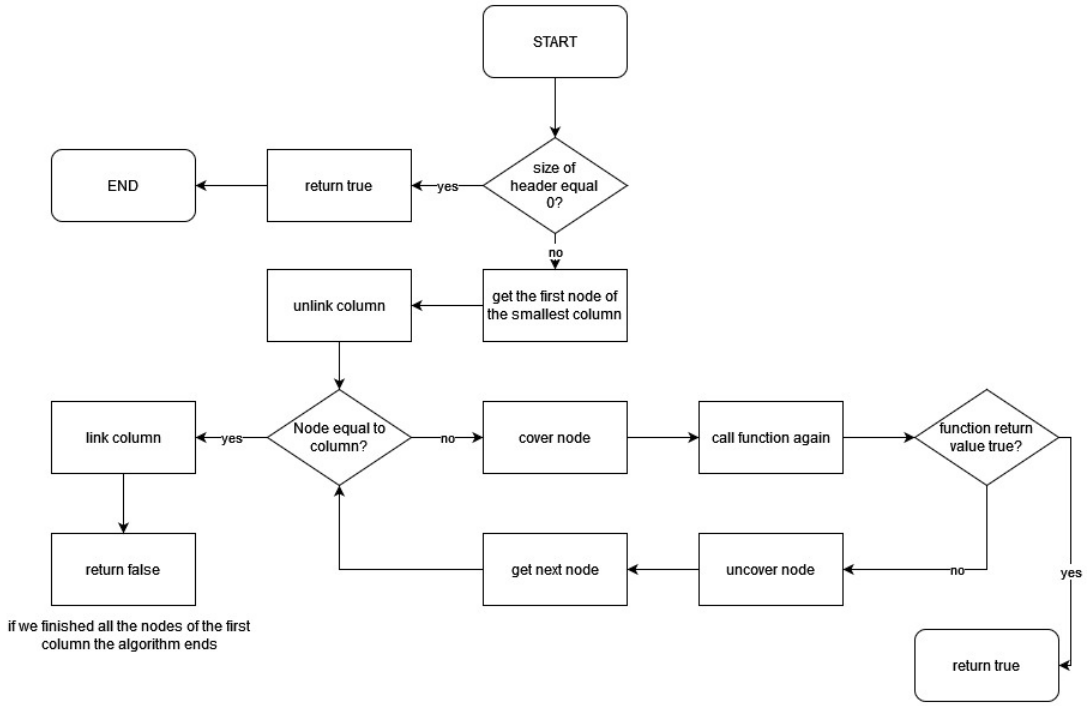


Figure 2: Dancing links flowchart.

2.1.2 Building the data

In order to build the list of possible placements of the shapes, we place each shape in all possible locations inside the container. For each location, we translate it into a one dimensional array of boolean where true indicates an occupied cell and an empty cell is false.

The algorithm requires a header and column head for each cell. The amount of column heads must be equal to the number of cells in the container. After creating the columns, we then go over all of the placement options of the boolean array and add the nodes under the fitting columns.

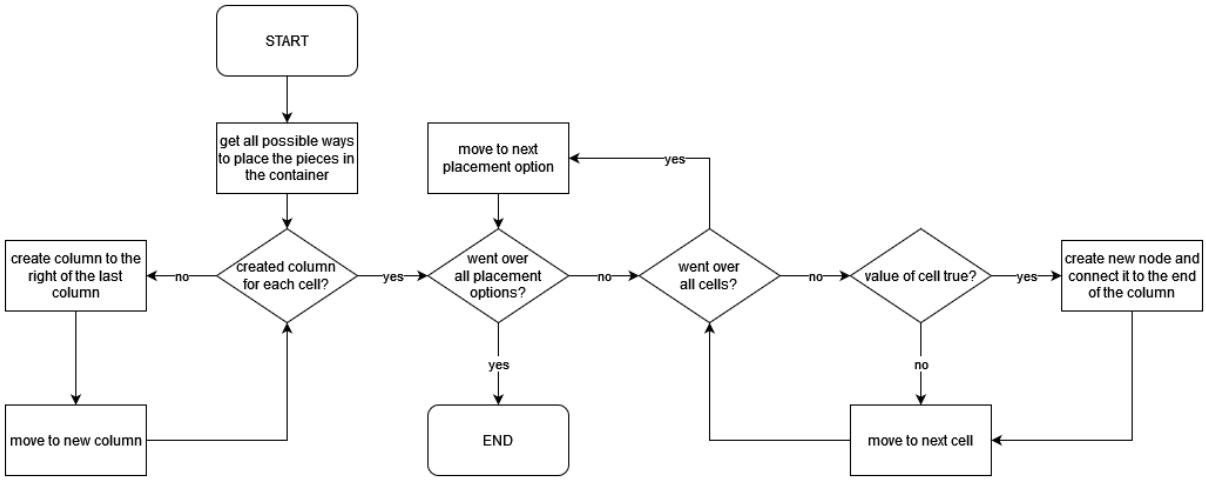


Figure 3: Placing a piece.

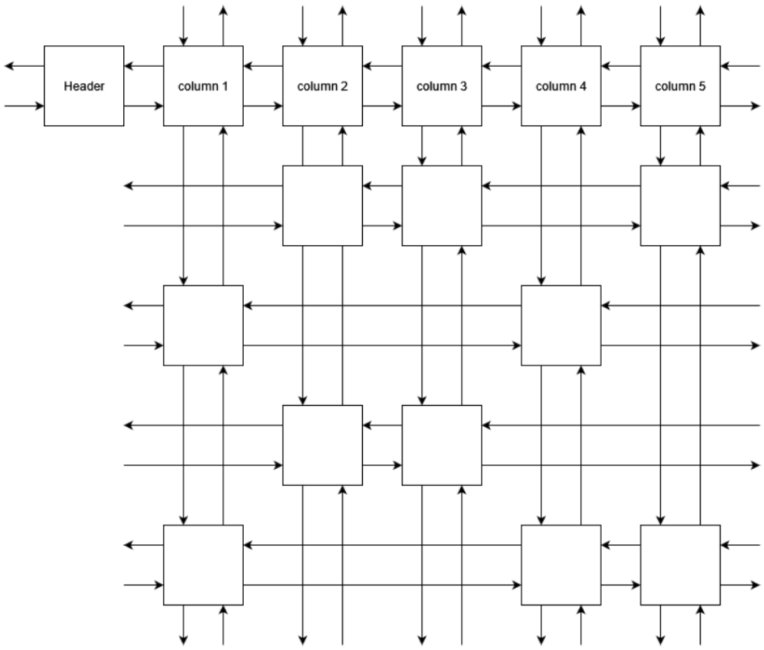


Figure 4: Example of the data structure.

2.1.3 Disconnecting and connecting columns

When we choose a cell, we must disconnect every column that is indirectly connected to the chosen cell by the rows. If we choose the first cell of the first column (shown as green in Figure 5), we then need to disconnect the relevant columns (yellow) and disconnect the rows that have cells in those columns (red).

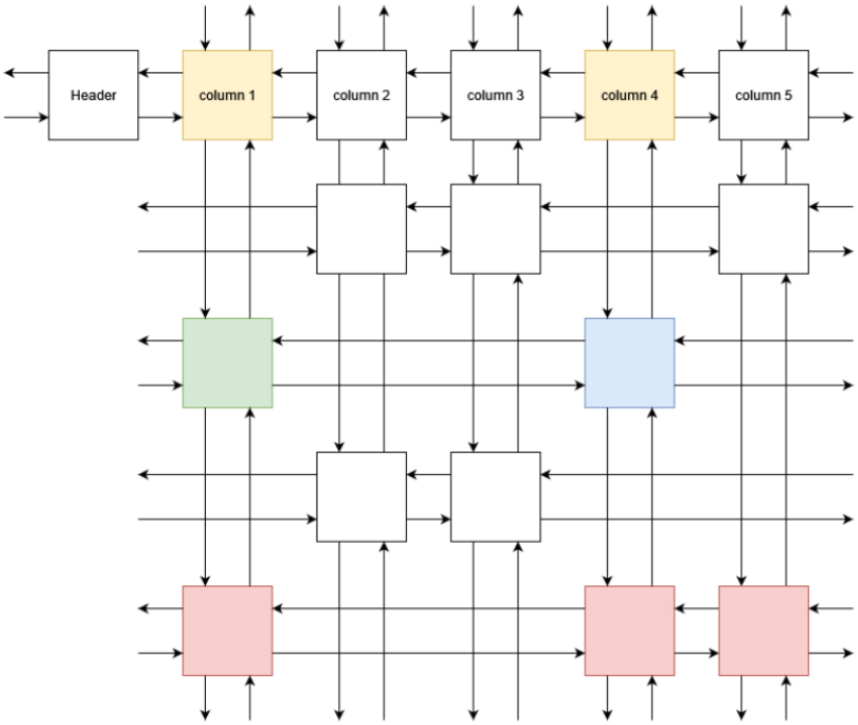
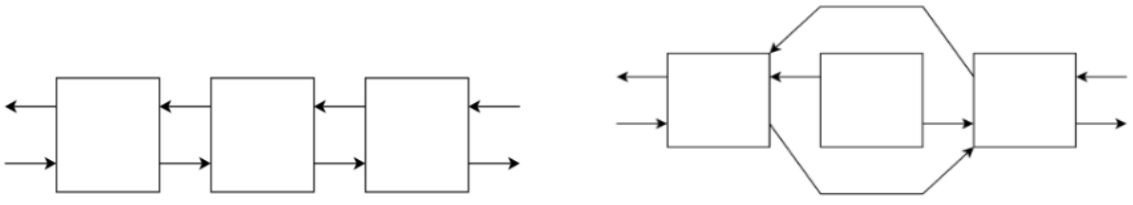


Figure 5: Coloured doubly linked list.

When disconnecting our chosen column, we change the pointer of the columns on either side of the chosen one and make it so they rather point to each other- they go around the column we are disconnecting; but the chosen column will still point to its surrounding columns. In this way, when reconnecting the column, it will "remember" how to reconnect itself. The same thing happens with the rows- but instead of left and right, we disconnect right and left.

If the order of disconnecting was to go down and right- to reconnect we need to do the opposite, go up and left.



(a) Original

(b) After pointer changes

Figure 6: How node becomes covered

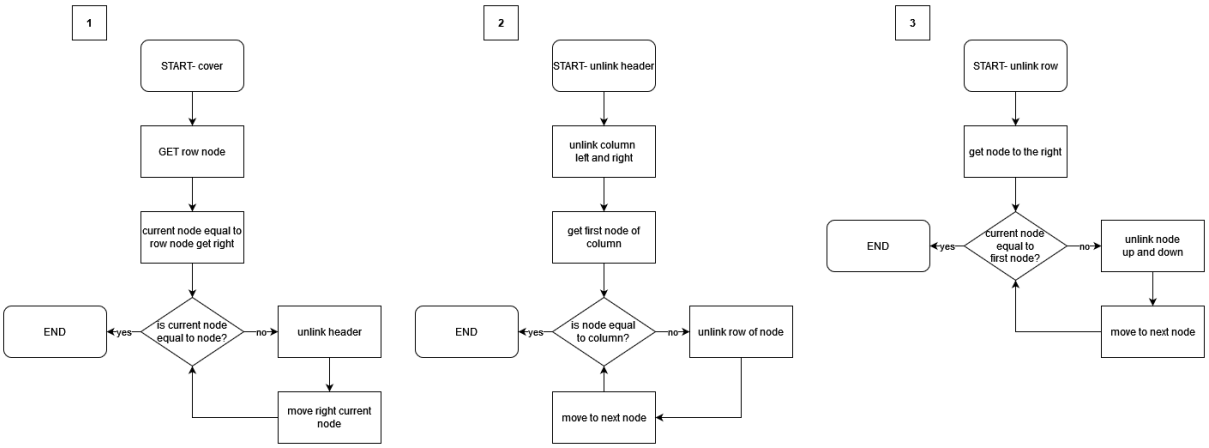


Figure 7: How disconnecting works.

2.2 Dynamic Algorithm

2.2.1 The concept

Dynamic programming is a method that simplifies a complex problem by dividing it into smaller subproblems, solving the subproblems and using the solutions to solve the complex problem. In our case- since we cannot divide the problem into subproblems, we chose to adapt the concept. Instead of solving small problems and building from there, each time we arrive at a conclusion that the state cannot be continued from, we will save it to the memory and if we arrive at it again the function will skip over it. This removes any unnecessary calculations and improves efficiency.

2.2.2 How it works

The task of the algorithm is to fill the box with the given shapes, each time the function is called, it gets the container and the possible shapes to place in it. We then perform a check to see if the state already exists in the dictionary and if so we see what the value of the state is. If the state exists with a false value, we return false.

In case the state does not exist, the function will go from the first shape to the last; and for each shape it will go over all the rotations and try to find one that it can place. If it did not find any shape that is possible to place, the function will save the state to the dictionary and return false. If the function managed to place a shape in the container, it will call the function again with the updated container.

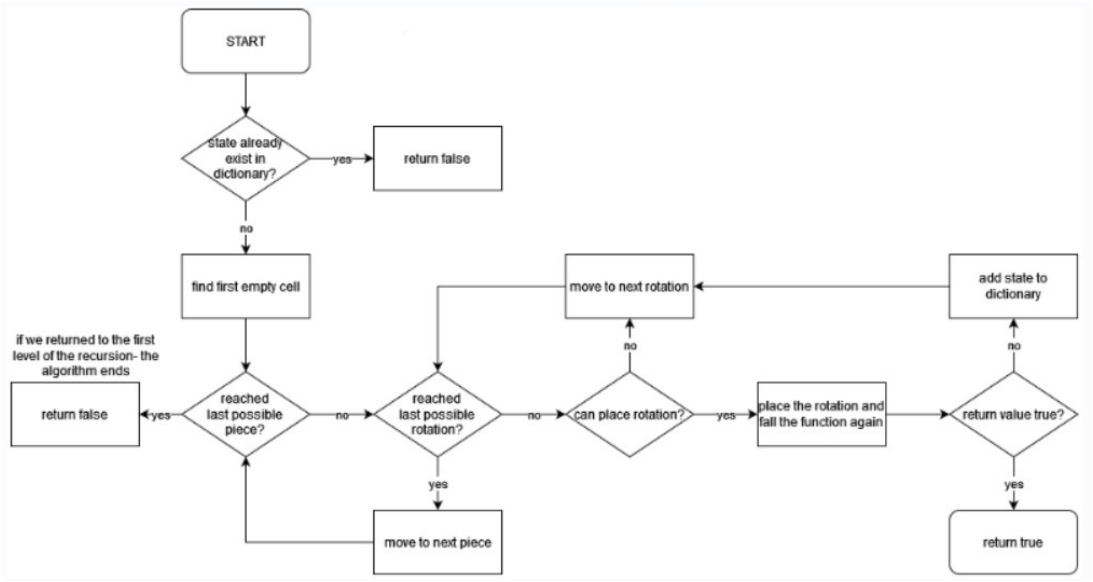


Figure 8: Dynamic algorithm flowchart.

2.2.3 Creating the key

A key in our algorithm is the description of the current state of the container. We create a key by going over the container and saving a full cell as 1 and an empty cell as 0. We do that until we reach the last full cell in the container.

3 Experiments

The purpose of the experiments is to check how well each algorithm can handle the different tasks that were presented for the two sets of objects.

- Is it possible to fill the container without any gaps?
- What is the highest packing value of objects in the container?

Additionally, we wanted to compare the two algorithms that we developed so we added a few questions:

- How much of the container can the algorithm fill?
- How fast can it arrive at a result?
- Will different orders of input result in different outcomes?

3.1 Experiment Settings

- Each experiment was performed 5 times and the average result was calculated.
- The sizes of the parcels and pentominoes stayed constant.
- The value of the parcels and pentominoes stayed constant.
- Each algorithm had a run limit of 40 million function calls.
- The size of the container stayed the same.
- For each experiment- the recorded metrics were: runtime in ms, how much space was full by the end of the run and the bin value.

The dimensions of the container are: 33 x 8 x 5

4 Results

The dynamic programming results for pentominoes are not applicable as it was unable to finish.

order	time in ms	fill result max 1320	score
A,B,C	5283.8	1314	232
A,C,B	4647	1314	240
C,A,B	4044.6	1314	240
C,B,A	4399.8	1314	233
B,A,C	4086.6	1314	226
B,C,A	4055.8	1314	225

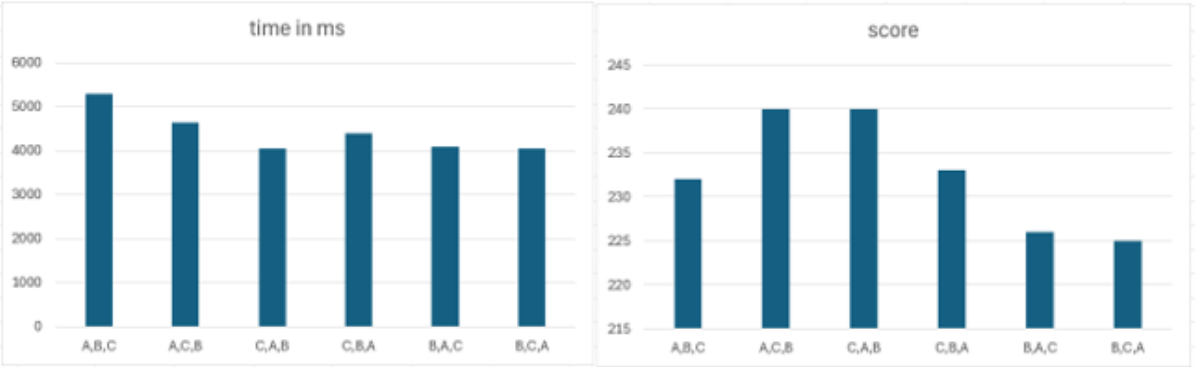


Figure 9: Results of dynamic programming algorithm for parcel experiments

order	time in ms	fill result max 1320	score
A,B,C	314355.6	1314	232
A,C,B	338556.6	1314	230
C,A,B	318780	1314	229
C,B,A	319820	1314	226
B,A,C	304569.4	1314	242
B,C,A	305423.2	1314	237

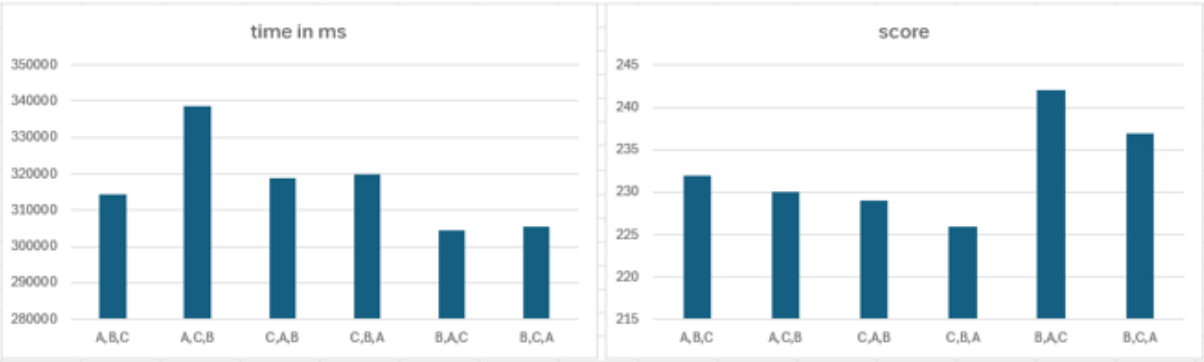


Figure 10: Results of dancing links algorithm for parcel experiments.

order	time ms	fill result max 1320	score
T,P,L	9.3	1320	920
T,L,P	9.5	1320	914
L,T,P	4.8	1320	1039
L,P,T	11.8	1320	817
P,T,L	5.3	1320	808
P,L,T	5.5	1320	1304

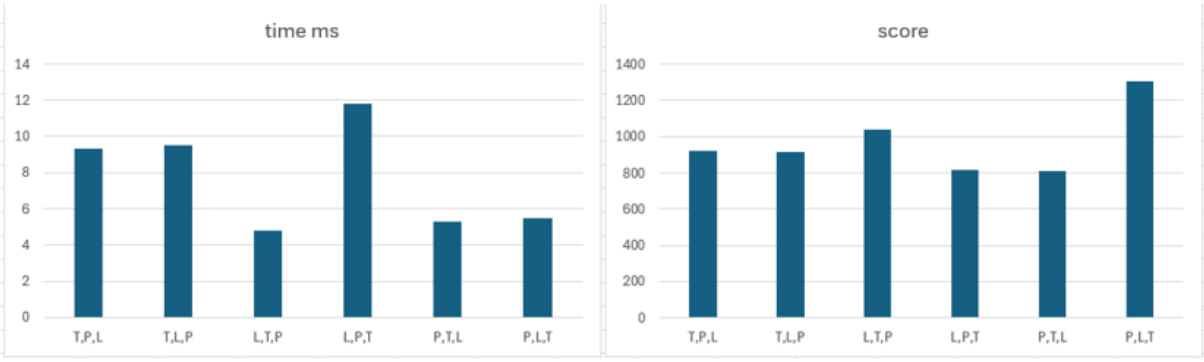


Figure 11: Results of dancing links algorithm for pentominoes experiments.

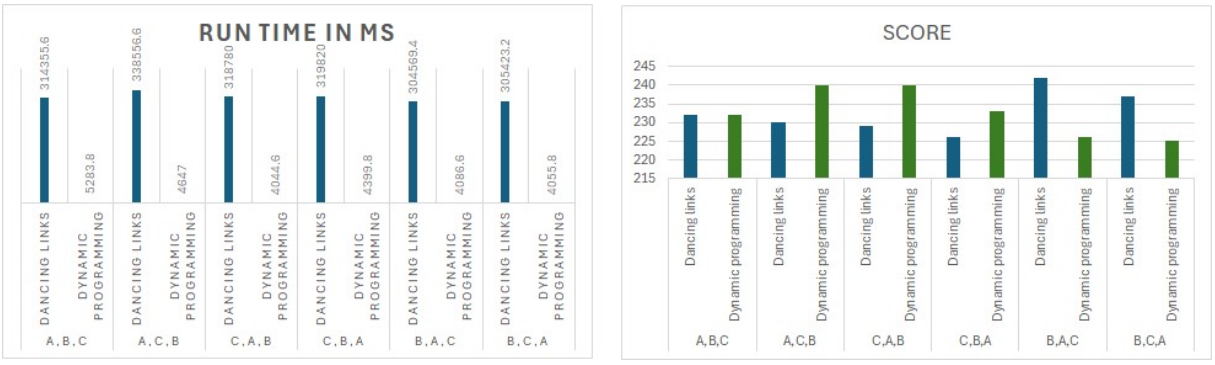


Figure 12: Comparison of dynamic programming and dancing links.

5 Discussion

In the experiments with ABC parcels, the dynamic programming algorithm achieved comparable scores to the dancing links algorithm. The dynamic programming had a shorter execution time and managed to deal with the parcels very efficiently.

In the experiments with TPL pentominoes however, the dynamic programming algorithm was unable to finish the run. The dancing links algorithm conversely, was able to finish within milliseconds.

The conclusions to the research questions showed that the ABC parcels were unable to completely fill the container. The maximum score of both the dancing links and dynamic programming algorithms were the same: 1314/1320. The dancing links algorithm had a better score of 242 compared to the dynamic algorithm that had a score of 240. For the TPL pentominoes, the dancing links algorithm was able to fill the container completely with a highest score of 1304. We have found that input order does affect the algorithm and performance as seen in figures 9, 10 and 11. These tests showed that the fill result is not affected as regardless of the order, it will fill as much of the container as it can. The runtime is also not greatly affected by order; but, the score differed greatly depending on input order. The results for the parcels showed that the dynamic algorithm performed faster than the dancing links algorithm but they both had comparable results. The results for pentominoes are not comparable, as the dancing links was able to finish in milliseconds while the dynamic algorithm was unable to finish.

6 Conclusion

In conclusion, we have managed to create a successful dancing links and dynamic programming algorithm and compared their performances through several experiments. To further our research, we will try to implement a way to include more shapes and create more algorithms to compare the performance.

References

Chu, J. (2006). A sudoku solver in java implementing knuth's dancing links algorithm. In *The. first Harker Research Symposium*.

Knuth, D. E. (2000). Dancing links. *arXiv preprint cs/0011047*.