The Tetriling Reassembly

Clara Arcos CID 01348932

1. Introduction

The project brief was to design a program that could find a perfect tiling solution to an arbitrary finite polyomino region by using a given set of tetrominoes. The maximum target shape size that the algorithm would be tested against was a 1000x1000 elements, with a maximum density of 0.8. It would be tested based on two different factors; time performance and accuracy, which was defined as the sum of missing squares and excess squares as a percentage of the total number of squares. A maximum running time of ten minutes was set. Therefore, the main goal was not only to solve *the tetriling reassembly*, but also optimise the running time and accuracy. The programming language to be used was python.

2. Background

In order to gather information that would aid the initial design stages, online research was carried out. However, there were not many relevant articles relating to this topic. Some articles on how to perfectly tile rectangular shapes with a density of 1 were found, but did not prove to be very helpful in solving the proposed problem. Some literature on how to design algorithms to solve the popular *Tetris* game has been published. Even though these findings were not entirely related to the project brief, they inspired some ideas that would initiate the early stages of development.

Online communities and forums such as *Stack Overflow* were consulted for specific coding problems and doubts.

3. Method

Development started with hand-written diagrams that would help understand the problem itself, as well as initiate the first ideas for the design. Basic algorithms were written and tested with target sizes of 10x10, 100x100, 500x500 and 1000x1000 to identify their strengths and weaknesses. After being optimised, if problems such as very high running times or failure to tile the target shape within the time limit persisted, the algorithms were discarded, and a new idea was written and tested.

4. Discussion

4.1 Tree DFS Algorithm

The first approach taken was a brute-force solution based on searching a tree, where each node was a different tetromino placement, using depth-first search. This approach had been taken with the mindset of always finding the perfect tiling solution, and therefore its accuracy was 100%. However, it turned out to be extremely slow, as every branch of the tree had to be explored and the algorithm would start backtracking if a perfect tiling was not generated.

This algorithm searches the grid starting from the top left corner and checks if any of the pieces fit in the current square. It then places a random fitting piece and repeats the process until all pieces have been placed or none of the pieces left can be placed. Once it is done placing the pieces, it checks for any missing squares in the solution. If missing squares are found, the algorithm will start backtracking and deleting all problematic pieces, replacing them with other tetrominoes until the perfect solution is found.

This approach had an acceptable running time for very small instances of the problem and, as mentioned before, it had an accuracy of 100%. However, it proved extremely slow once the target size went past 10x10 even after optimisation, and was therefore discarded.

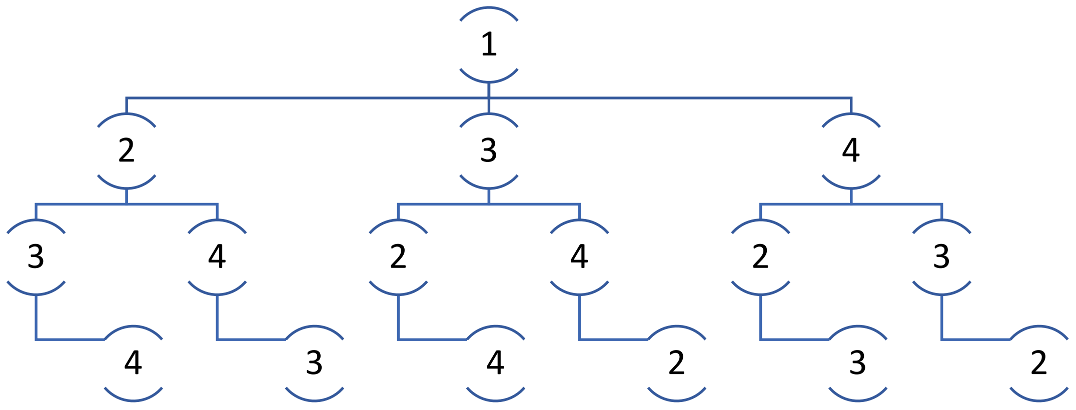


Figure 1: *Example of a tree with 4 pieces. The algorithm starts with piece 1 and explores each possibility with a depth-first search before it backtracks and moves on to the following node.*

4.1.1 Edge Finding optimisation

One of the optimisations that led to the design of the final program was the implementation of the edge finding method. As this technique plays a big part in the final solution, details on how it was implemented can be found in section 4.2, ‘Edge Finding Algorithm’.

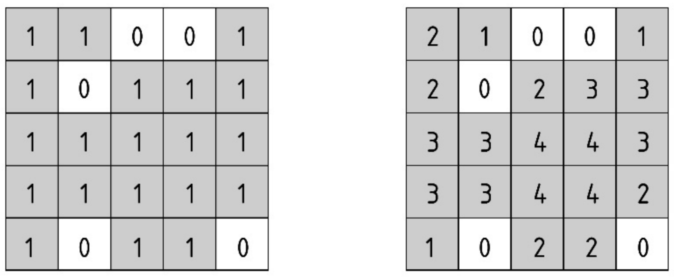
Before this method was implemented, pieces were placed randomly without accounting for edges or missing squares. Instead, the algorithm waited until the end to check for missing squares in the solution and then started backtracking. However, if there was a position where a piece fit perfectly or almost perfectly, then the algorithm should have been able to identify it as soon as possible. This discarded the branches of the tree where this piece would have been placed anywhere else, reducing the search and therefore saving time.

Even though this optimisation reduced the running time of the DFS approach, the main tree structure was kept and therefore the speed of the algorithm was still too slow to be a valid solution to the project brief in bigger target shapes.

4.2 Edge Finding Algorithm

As the edge finding optimisation in the DFS algorithm did not prove to be effective, the tree structure was rejected. An approach which was entirely based on the edge finding technique was used instead.

This algorithm works by calculating a *neighbours matrix* from the target shape. This is illustrated in *Figure 2*. It then creates a *score matrix* for all the possible placements for each shape, obtained by summing the total value of neighbours that the shape would contain when placed in each square of the *neighbours matrix*. Once the *score matrix* is generated for each shape, the algorithm will start placing the tetrominoes that encapsulate the lowest sum of neighbours, which can be observed in *Figure 3*.



1	1	0	0	1
1	0	1	1	1
1	1	1	1	1
1	1	1	1	1
1	0	1	1	0

2	1	0	0	1
2	0	2	3	3
3	3	4	4	3
3	3	4	4	2
1	0	2	2	0

Figure 2: *Example of a target shape and its neighbours matrix.*

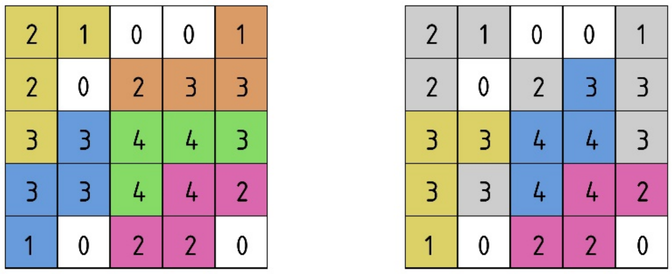


Figure 3: *Example of a perfect solution and a low accuracy solution. It can be observed that pieces with a lowest sum of neighbours fit more perfectly. (The L shape's sum in the first picture is only 8, as opposed to the sum in the second picture which is 10).*

4.2.1 2D Convolution

Both the *neighbours matrix* and *score matrix* were calculated by using 2D convolution. To generate the *neighbours matrix*, this method uses a kernel (a 2D matrix of weights) that moves along the target shape, multiplies each of the values in the *target matrix* that are contained within the kernel at that moment with their corresponding weights in the kernel, and returns the sum of these products, placing it in the central square. This can be observed in *Figure 2* by comparing it to *Figure 4.1*. A similar approach was taken to generate the *score matrix*, the main difference being that the kernel was an array defining each different shape and it was being compared against the *neighbours matrix* in order to find the score of each placement.

```
[[0,1,0],
 [1,0,1],
 [0,1,0]]
```

```
[[0, 0, 0],
 [0, 0, 0],
 [0, 1, 0],
 [0, 1, 0],
 [0, 1, 1]]
```

Figure 4.1: *Kernel used to generate the neighbours matrix.*
Figure 4.2: *Example of a shape kernel (L). The starting point must be in the centre of the matrix.*

It was found that generating all the scores for a shape and storing them in an array using convolution was less time-consuming than having a function generate the scores for each shape at each position, as this function would have been called multiple times for every square and score recalculation. On the contrary, storing them in an array removes the need to call the function for every individual square. In addition, the *convolve2d* function from scipy itself is faster than going through every square with the initial *get_score* function.

4.2.2 Reduction of Score Recalculations

The second main optimisation was reducing the number of times scores would have to be recalculated. Initially, scores would be updated each time a piece was placed, resulting in a higher running time. This problem was solved by simply placing more pieces at the same time before recalculating the scores.

If the target shape is less than 1000 squares, the algorithm will only place one piece at a time in order not to compromise accuracy. A size between 1000 and 100000 squares will result in the algorithm placing a number of pieces corresponding to a hundredth of the size, as it was found to be enough for an acceptable running time after numerous tests. When the size is above 100000 squares, it will place a number of pieces corresponding to a fiftieth of the target size. The number of placements increases in this last case because, as the target gets bigger, the algorithm needs to reduce the number of recalculations even further.

4.2.3 Numpy Structured Arrays

While testing the algorithm, it was discovered that appending and deleting elements from lists was a bottleneck. This is because Python dynamically allocates more memory when new elements are added into a list. Numpy can be used to allocate all required memory at once and avoid overhead generated by Python lists. This is done by specifying to numpy the exact format of the data to be stored so that it can allocate the exact amount of memory needed.

This idea was implemented in the code by turning the list that stored the possible placements into a numpy array. The *dtype* function was used to define the data type of the array. Specifically, the data types used were *uint8* for the shape ID and the score, and *uint16* for the row and the column, where *uint* stands for unsigned integer and 8 and 16 are the number of bits used.

5. Conclusion

Even though the first attempt proved to be extremely slow in comparison to the final solution, it was helpful in the definition of a new idea and served as the testing grounds for the final algorithm.

The final submitted solution implemented an edge finding method along with some optimisations that greatly reduced the running time while keeping a high accuracy. The main optimisations performed were the use of convolution for the generation of the neighbours and the score matrices, the reduction of score recalculations by placing multiple pieces at a time, and the use of numpy structured arrays to reduce overhead created by Python lists.

The average running time for a 1000x1000 target shape with a 0.8 density was of 2.9 minutes, keeping a consistent accuracy of about 95% for the biggest target sizes and about 90% for very small instances of the problem.