Tetriling Project

This project involved filling a random target space similar to figure 1 with Tetris pieces as accurately and quickly as possible. This target space could also vary in size from anywhere between 10 by 10 and 1000 by 1000 of varying densities. This problem requires the algorithm to be equally efficient at varying sizes and that the algorithm should avoid using high running time algorithms like exponential and polynomial running times. Due to this it was recognised that getting a perfect solution by repeated trial and error wouldn't be ideal and therefore the first step to designing an efficient algorithm was recognising that iteration would not be the solution.

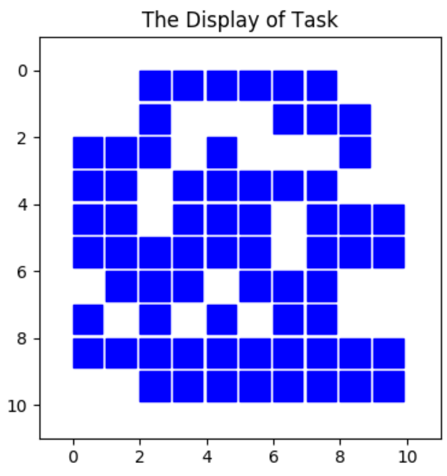

Figure 1: Initial target space demonstration

The next decision made was the method of placing shapes. It was determined there were two main methods that one could place a shape. The first is to evaluate every possible shape as a whole, and place whole tetrominoes in one go and the second method was to build the tetrominoes from one cell outwards to create a collection of four conjoined positions that make up a valid tetris piece (joined at each edge). The second method was determined to have a better running time because for each position on the target space that would need checking with a function, the function would only have to iterate through four steps as it decides the optimal position for each unomino (single cell within a tetromino). Whereas the placing entire shapes would require checking almost 100 different shapes for optimal positioning, due to the number of different rotations of each shape and the consideration of different origins for each shape. It would also be easier to create a weighting system. Therefore, the designed algorithm uses the second method "The crawling method" to place shapes.

Due to the decision that the perfect solution was not always attainable it was decided that using a greedy algorithm to place these shapes would be the most effective way to increase accuracy while keeping running times low. This is usually done with a weighting system to give certain directions priority and so the designed algorithm was focussed on finding the optimum method to give the correct positions the best weightings. The main method used to give weightings throughout the algorithm was using number of possible neighbours for a shape as shown in figure 2. For example, if an unomino is connected on all four sides it has a value of 4 but if it is only connected to on other unomino it would have a value of 1 and the algorithm would focus on prioritising unominos with lower values because they were would in theory be more difficult to fill.

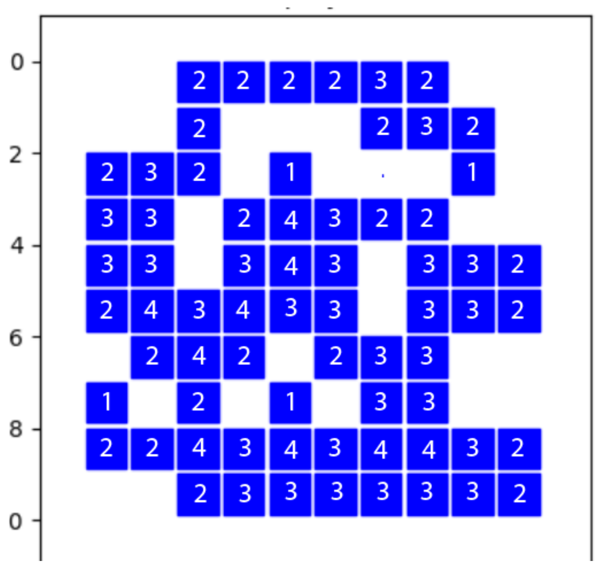

Figure 2: Demonstration of the concept of neighbouring nodes

The proposed Algorithm

The set up

The algorithm required some initial set up to use as references for when calculating directions and weightings for the crawling method. The first step involved creating a labelled graph (figure 3). This would allow each point in a graph to have a relevant number that can be referred to much more easily and quickly then using i and j components of an array(where i: column and j: row). This also changes the input for the algorithm into a constant number that can be referred to as number of elements.

The next step involved creating Adjacency list using these element numbers as method of referring to each neighbour. This is done by creating a list of lists where each list is represented by the specific element and contains values for each connected element. This allows access to each element's neighbours by referring to it in the code as AdjacencyList[element]. This adjacency list also allows easy identification of number of connected nodes through "len" command and easy manipulation of connected elements. This will make it easy to update the adjacency list throughout the placing of shapes.

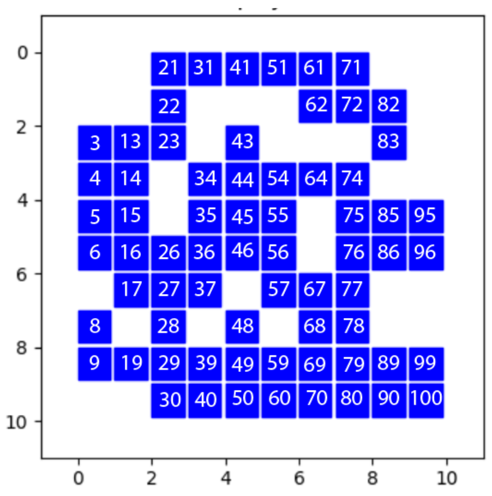

Figure 3: Demonstration of labelled target (note blank spaces = 0)

For placing shapes due to the use of "The crawling method" it becomes important to be able to identify the shape id of the shape about to be placed. This must be calculated from a list of elements that could appear in any order with any origin point for the shape. Therefore, it was important create a way to transfer these element values into a set of shape IDS and coordinates to place into the solution. The first step was to convert elements into coordinates - this was done mathematically, so that the algorithm didn't need to search through the labelled target to find the elements numbers which would have had a running time of "n" but instead using division and modulo the i and j values were found with constant time reducing algorithm running time. This could also be reversed too if needed and used elsewhere for further evaluations due to it being written as a callable function.

Once a list of coordinates is found these need to be converted to be based around a (0,0) value to compare with a set of stored values which identify shape IDs. Therefore, the values for i and j for each coordinate are translated by subtracting the values for i and j of one unomino in the to be placed tetraomino from the other unominos. This would make it easier to create a dictionary of values to which the coordinates can be compared to because all the value can have coordinates around the origin point, but this would still be equal to almost 100 different shapes in the dictionary due to different starting origins. Therefore, to reduce the number of shapes. The shapes are translated so the left most pieces are the origin by determining the unomino with the lowest i value and subtracting its values from itself and the other unominos. This reduces the number of different shapes in the dictionary to about 25 shapes, thus reducing storage space and number of shapes that the to be placed tetromino must be checked against.

The final issue with finding shape IDs was that the elements were received in a random order and therefore each different position for the tetromino would have 4! different ways of arranging the coordinates. This could be avoided by using a sorting method for each tetromino that would be placed however, a faster method would be to create a set which doesn't require any ordering of the lists and would compare the different list. This is much cleaner code to write as it is unnecessary to have the lists be ordered and requires less code to be written but also uses constant running time.

This method now allows identification from a dictionary containing keys that are represented by each shape ID. These keys would contain a list of the 1-4 different sets of tuple variables that represent the tetrominos. These could be compared to each different shape set of coordinates that represent shapes to be placed to find shape IDS which are transferred to another function to place the shape. This whole process has running time of n where n is equal to number of shape IDs stored in the dictionary due to having to loop through each key in the dictionary to find the related set of coordinates. However, in relevance to the total algorithm this would have a constant running time that is as low as possible, while wasting little memory space.

<u>How the tetromino positions are found</u>

The designed algorithm iterates through the number of elements till it finds the first empty place in the target place it is then designed to focus on packing the pieces as close together as it can. It does this by checking the first unomino's list in the adjacency list to find a list of all the connected neighbouring elements. It then finds the number of neighbours of the neighbouring elements and appends this value and the neighbouring element in a list of two values into a blank list called listOfPositions. It then finds the neighbouring element with the fewest neighbours in this list. Following this it appends this element to a list containing the original unomino called listOfPlacedElements.

Once this second unomino is found its important to update the adjacency list so the original element isn't considered a neighbour as it is an already placed position and therefore invalid. This is done by accessing the neighbours of the first unomino and using remove() to remove all instances of the initial unomino from each off its neighbours in the adjacency list lists. The listOfPositions would need to also be updated as each value which is connected to the original element would now have one less neighbour, so the listOfPositions are compared with the initial placed unomino's neighbour and a value 1 is subtracted from number of neighbours if it was connected. It is important to check if it is a neighbour to the placed element as the listOfPositions values are saved to create a big list of all placeable elements when placing third and fourth piece.

The function then recursively calls itself twice more to try and place the rest of the unominos by passing the new starting element, the list of positions to place and the list of placed elements. It is important that the list of positions is passed through because now it contains positions around previous elements that are also adjacent and potential directions.

The final issue that arises from placing shapes in this way is that if a mistake is made and the code cannot place four unominos then the algorithm has to undo the changes to the adjacency list. This is redone using the listOfPlacedElements and the labelled target. The algorithm looks through listOfPlacedElements and for each placed element and checks the labelled target for connected nodes and adds them back to the adjacencyList. Its very important the labelled target is making changes and not the adjacency list.

These steps are repeated for each of the elements in the whole grid and in the target space would give a solution like shown in figure 4. This solution at this size usually produces similar levels of accuracy either missing 4 blocks or 8 blocks. This is due to it never placing excess blocks. With a little more time, the code could have been optimised to fill empty blocks of threes with pieces so that there would be one excess increasing accuracy unfortunately the time was spent testing alternate methods and optimisation methods.

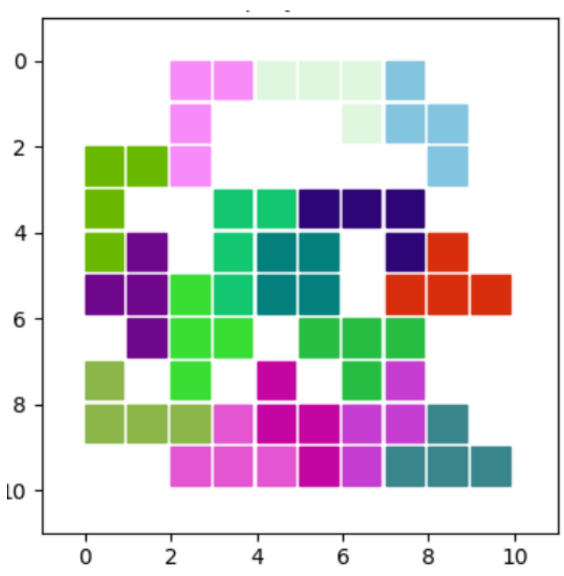

Figure 4: The solution for the above problem using the designed algorithm

Some examples of these optimisation methods included adjusting the recursive place function so that the last piece couldn't shut off a singular piece by forbidding the placement of the fourth unomino if it had only one neighbour. Doing this seemed to have little effect and would instead lead to the gaps being gaps of two instead of ones as the algorithm would end up isolating multiple pieces instead of just one piece. There seemed to also be a slight reduction in accuracy across the different sizes and speed and therefore this was disposed of as unnecessary. Some tests were also run to try and workout whether it would be justifiable to create recursive function that always prioritise the starting unomino being a node with only one neighbour. This was implemented as a started with the function filling only the first iteration of one neighboured elements, but the effect again seemed more detrimental and occasionally by seeming large amounts and therefore this was also not used because adding another recursive function would be quite detrimental to the speed of the algorithm and therefore was also scrapped in favour of the algorithm with a better success rate in the performance test.

The proposed algorithm is generally very efficient and achieves accuracies above 90% in times under 20 seconds at all sizes required and is very consistent at a variety of sizes the main cause in changes in accuracy is the changing density. There is also the issue that at smaller sizes one random extra mistake causes a much larger percentage drop which can explain the occasional much lower accuracies. As shown in figure 5 the graph also has a running time of n squared approximately, which is to be expected as n is equal to number of rows/columns.

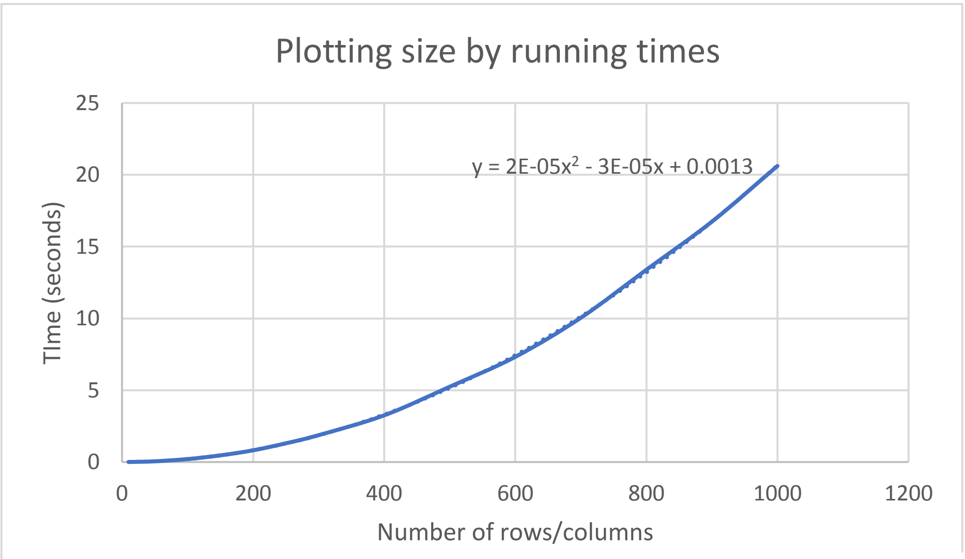

Figure 5: Graph showing running times