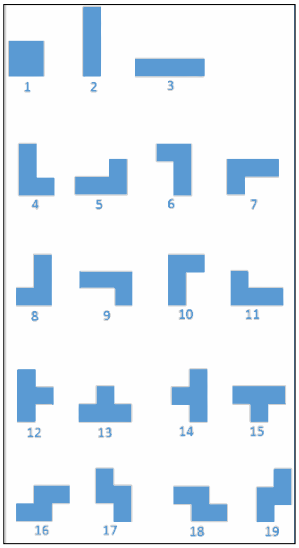
Tetromino Coursework



Abstract:

The Tetromino Coursework proposed a challenge to fill a grid of size up to 1000x1000 with 19 different Tetromino shapes, with 2 constraints: 1. The amount of each Tetromino shape used is determined by the proposed "perfect solution" and 2. The solution must mimic the "perfect solution" as much as possible and will be reviewed for its accuracy and the code's running time. The provided solution includes two "direction prioritised depth first searches", and a final optimising function for increased accuracy.

Evaluation:

Using a deep copied version of the target matrix where a blank space is 0 and a block is 1 (defined as new_tar in the code), the solution aims to find neighbours, not the most appropriate ones, but the most prioritised ones – explained more in Main.py. To top off the accuracy at the end, I have traversed new_tar again to check for 4 unoccupied spaces (1's), and if the appropriate Tetromino can be added, and finally 3 unoccupied spaces as well as one completely blank space (3 x 1's and 1 x 0), where the result would return 1 "EXCESS PIECES" but would still fill in 3 spaces.

The initial solution containing *CHECK* and *CHECK2* without *Final*, was solving accurately for the first 60% of the board, but then the function couldn't use pieces that it found using the direction prioritising searching method. Therefore, *Final* was added which took the remaining Tetromino pieces and traversed the most abundant remaining Tetromino coordinates to look for any possible spaces. These two strategies combined, create the complete solution which focuses both on accuracy and speed, and it returns with quite good data:

Matrix Size	**10x10**		**100x100**		**1000x1000**	
Density	**Time (s)**	Accuracy%	**Time (s)**	Accuracy%	**Time (s)**	Accuracy%
0.4	0.00049	84.00	0.05759	88.30	7.03278	88.49
0.6	0.00073	78.00	0.07973	88.31	9.33175	88.51
0.9	0.00088	79.66	0.11878	91.41	11.1171	91.75

Table 1: Running Times and accuracy of different densities and sizes

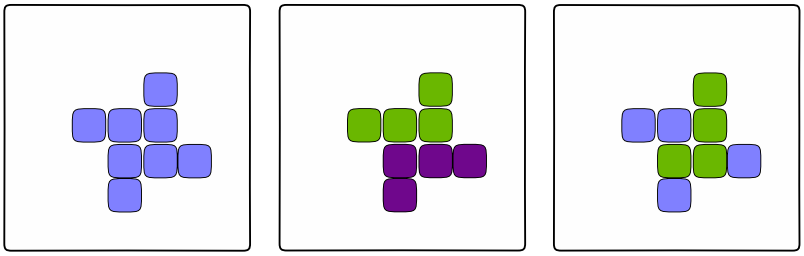
Main.py:

Tetris and its components:

Initially within the function *Tetris* [line 4-17], a matrix M is required from *Tetris* which is the proposed solution matrix [line 8]. This must be in the form of a list of tuples, i.e. [(a,b) , (0,0)] where 'a' is the Tetromino ID (1-19), and 'b' is the pieceID, which determines which order the tetrominoes are placed, which has been done by a **global variable** called 'PID' [line 219]. Another useful matrix to have is the target matrix, this has been copied into *Tetris* and has been renamed **new_tar**. This is a matrix where instead of each (a,b) term, blank spaces are a 0, and spaces where parts of a Tetromino exist are 1 [line 6]. This has proven very useful identifying places to add Tetrominoes.

CHECK, CHECK2:

The solution includes 11 functions including Tetris 3 of which are used to determine which tetrominoes should be used: *CHECK* [line 18-50], *CHECK2* [line 52-84] and *Final* [line 265-287]. Both *CHECK* and *CHECK2* use a prioritised direction to find neighbours, i.e. *CHECK* would find neighbours in the order: 1. Left 2. Right 3. Down, and if there wasn't a neighbour there, it would then check its **predecessor node**, (if predecessor node has no more neighbours, then move on), thus prioritising shapes 3, 9, 15, 7, etc. CHECK2's priority is 1. Down 2. Left 3. Right, thus preferring shapes 2, 8, 4, 19, and so on. This produces a **semi-Greedy algorithm**, as it wants to put certain tetrominoes in and if it can't, moves on.



Perfect Solution CHECK's Solution CHECK2's Solution

Figure 1: Demonstration of "Direction prioritised depth first search"

This is a **depth first search algorithm** because *CHECK* and *CHECK2* don't consider predecessors until they must. The algorithm is purely trying to go the direction that it prioritises. Breadth first search was considered but was not utilised because it would find more arbitrary shapes but would also create many small gaps between each Tetromino filled section.

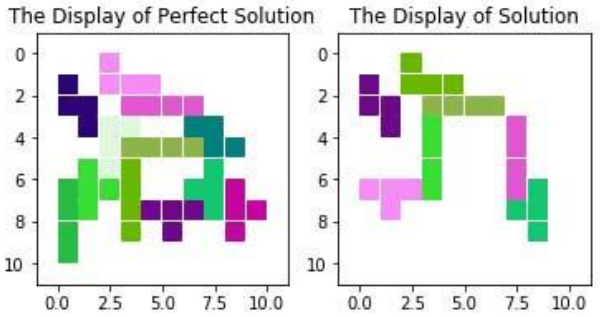


Figure 2: 100x100 using *CHECK* and CHECK2 only

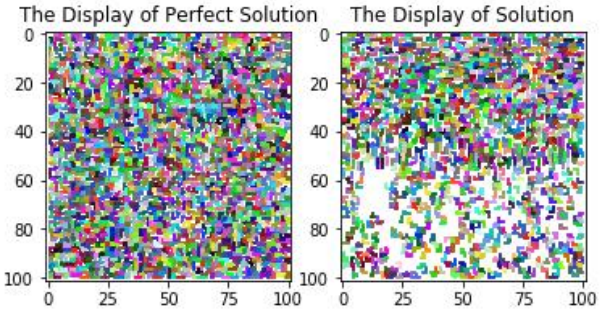


Figure 3: 100x100 using *CHECK* and CHECK2 only

While this method seems good at first, a large flaw is the implementation of **limit_tetris**. This is the collection of tetrominoes used in the perfect solution and constrains the solution to be purely based upon those pieces as well. In the example on the left, limit_tetris has a limited amount of tetrominoes, and once *CHECK* or *CHECK2* find differently orientated tetrominoes which limit_tetris don't have enough of, it cannot place that shape. Hence creating an accurate top 50% of the solution and much worse bottom 50%.

The top half will also be separated into sections. The top section will be shapes from *CHECK* and the lower section in the top half will be from *CHECK2*. This is expected as *CHECK2* will be able to use some tetrominoes which have not been placed and then find them with the second **direction prioritised depth first search**.

Final:

After *CHECK* and *CHECK2*, what's left are still many pieces in limit_tetris and an average of 40% of the solution matrix not filled. The remaining spaces have been found by the initial functions but not enough limit_tetris tetrominoes are provided to append those pieces, hence the lack of pieces near the bottom 50% of the solution.

Final is the last optimisation needed to fill in the remaining parts. Firstly, the remaining available tetrominoes are identified and put into a **greedy priority queue**. This has been done by sorting limit_tetris by the most abundant piece remaining [line 269]. A list of set of coordinates of all 19 tetrominoes is created to cross reference PID and Tetromino ID. Sets of coordinates are used as they have no order and therefore when it comes to appending the tetrominoes into the solution matrix, if both sets contain the same coordinates, they are the same (figure 4). This make it easier to identify tetrominoes instead of immutable tuples or ordered lists.

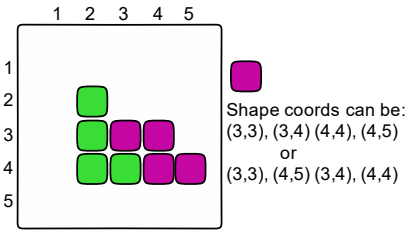


Figure 4: Sets of Coordinates

CHECK and *CHECK2* returns new_tar as a matrix of not only '1's and '0's, but also with '3's where the 3 is where a piece has been placed and fixed and therefore should not be touched. The function *Final* will call **X** as a variable which is either 4 or 3. When **X** = 4, *Final* is trying to find the most abundant shape that will fit into the solution where there are currently 4 '1's. When **X** = 3, *Final* is again trying to find the most abundant shape that will fit not into a 4 connected '1's but will accept 3 connected '1's and one blank space where the perfect solution doesn't have a block.

Final then traverses new_tar from top left to bottom right again to find coordinates which are 1s. When a first '1' is identified at new_tar[y][x], *Final* checks the coordinates of the **most abundant tetromino** from limit_tetris, referenced to [y][x] as its **origin** point (0,0). *Final* also has a counter = **S** [line 275] which adds up the values of new_tar[y][x] where (y,x) are the coordinates of the most abundant shape and new_tar[y][x] should return a 0, 1, or 3.

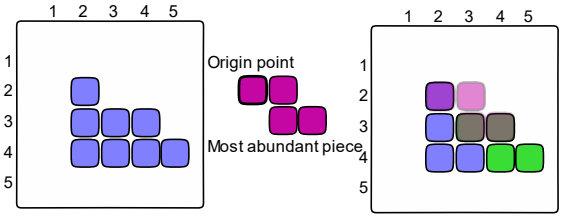


Figure 5: Sets of Coordinates

In figure 5, the blue blocks are '1's and green are '3's. *Final* would be trying to put the most abundant piece into coordinate (2,2). There is overlap of 2 blocks in the green area, meaning S + 3 + 3. The white blank space is 0, and the one blue block is 1. This gives S a total of 7. *Final* equates the called value of X and the value of S. if S = X [line 284], which is either 3 or 4, then the function decides that adding this tetromino would be beneficial.

Traversing through new_tar:

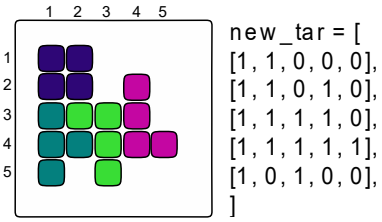


Figure 6: Visualising new_tar

Using the function: *locate* [line 190-197] the solution traverses through the matrix from the top left corner, scans the rows from left to right then the next row when finished, ending up at the bottom right corner. As new_tar is a matrix of mainly '1's (existing Tetromino needing placing) and '0's (no Tetromino), *CHECK* and *CHECK2* can easily identify the shapes that it wants to place.

Notation in new_tar exists as new_tar[y][x] where the cartesian plane for the solution provided has positive y going downwards and positive x going to the right, with the origin at the top left corner.

Within *CHECK* and *CHECK2* when new_tar[y][x] == 1, and a piece can be placed on 4 coordinates all with value of '1' in the matrix, the Tetromino is appended and all the '1's become '0's. This is to prevent the algorithms finding the same coordinate and **overlapping**. This also is a smaller version of a **reduction algorithm** as after each piece is found, new_tar has less '1's to consider.

Initially, **recursion** has been used to find tetrominoes in *CHECK* and *CHECK2,* but this direct implementation quickly led to hitting **maximum recursion depth limit**. Therefore, a less direct method has been used where *CHECK* dips in and out of the function *checkaround* [line 90-128], allowing *CHECK* to find its Tetromino and then come out of the function. This means the function *CHECK* can find a **4-node traversal tree** and continue to find the next tree without having stacked recursion.

Placing tetrominoes:

Every time a Tetromino has been found and needs placing, the coordinates of the piece are added to a list called 'remove' [lines 237, 224]. There are 2 **placing functions**, both firstly check if 'remove' has 4 elements. If it does, it takes the associated Tetromino ID (called into the function) and the global PID in the form (TID, PID) and changes the items in **M** with coordinates from 'remove' into the new form. Once this is done, the values within new_tar are also changed, as this insures new_tar information **only changes** when actual tetrominoes have been placed and are fixed. The function *appending* is used for *CHECK* and *CHECK2*, and each time a piece is placed, the new_tar value changes from '1' to '**3'**. *Final* uses its own placing function: *Append,* which subtracts 10 from each new_tar coordinate when a block is placed – making the new number on '**-9'** on new_tar.

Running Time:

The running time of all algorithms combined gives a linear running time – constrained by: **Big Oh (n)**, (upper asymptotic time) where the combined algorithm is **3 times** the time of just *CHECK* and *CHECK2*. The solution has kept the time down by using linear or constant time algorithms, such as more '**if'** statements than **'for'** loops.

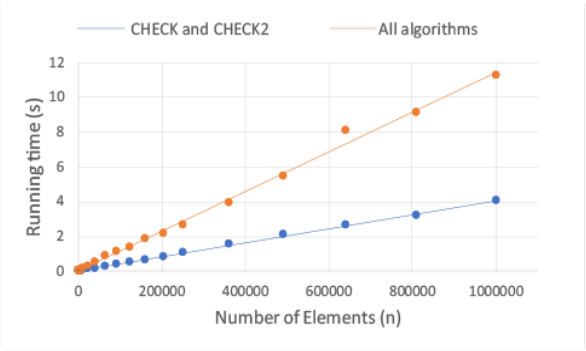
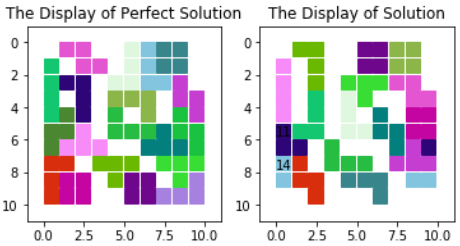


Figure 7: Total Running Time vs elements

Errors:

A common error found was the boundary error. This error comes up because either new_tar is trying to access negative coordinates or place a shape with coordinates within negative boundaries. As you can see in figure 8, there are 2 pieces with the wrong TID and PID. Their numbers are also shown on the left edge of the solution. Both pieces have wrapped around because of the negative coordinates that the algorithms are trying to append. The solution is to simply define boundaries clearly. This has been done by checking if new_tar[y+a][x+b] (where (a,b) are the coordinates within 'remove'), is greater than 0, whilst also being less than the length of the target solution and the height of the target solution [line 278].



WARNING: 2 pieces have a wrong shapeID. They are labelled in image of the solution, and their PieceID are: [11, 14].

Figure 8: Boundary Error