Tetress AI Report (Authors: Ken Liem and Leo Brooks)

Question 1:

We incorporated an A* search algorithm which calculated the path cost and heuristic cost of each state. As each state was expanded to, it was added to a Priority Queue which was ordered by lowest cost (smallest of path cost+heuristic). The Priority Queue was implemented using the built-in queue library and adaptations of functions for the built-in heapq library to allow for the specific priority comparisons we wanted. Each piece is stored as a tuple of vectors where each rotation of a piece is defined as its own different piece. Since we normalised the piece vectors to go from top-left to bottom-right, each 4-block piece only had 4 possible translations (including none) vertically and horizontally.

The worst case time complexity is O(B^8 * P^2 * L^4) and worst case space complexity is O(B^6 * P^2 * L^4) where B is the size of the square board (number of rows/columns), P is the number of pieces, L is the maximum number of blocks of a piece.

Question 2:

We used two heuristics and weighting between these heuristics. The first heuristic we used was a heuristic that measured the shortest manhattan distance between the target block and a red block. This was calculated by getting a red's blocks row and column coordinates , and subtracting by the target block's row and column coordinates as shown, then absoluting it to ensure we do not have negative distance. Then we added them together and checked if this is the lowest distance so far, completing it for every red block on the board until we found the lowest distance and returned it.

The second heuristic is a heuristic measuring what is the least amount of pieces in order to clear the target/row column (excluding the pathway) called estimatePiecesRemain
. Firstly, we measured how many pieces need to be placed in the targeted column and row. This is calculated by the following aspects:

- If there is 4 empty blocks, +1 pieces
- If there is a gap that is 1-3 blocks. +1 pieces

Then we check if any part of the row or column is blocked. If they are blocked, it means that we have to add additional pieces to clear a way to the blocked path. We repeat the same process as above to find the lowest amount of pieces to clear the path. For targeted columns, we check the columns next to it and the row above and below the blocked path. For targeted rows, we check the rows next to it and the column left and right of the blocked path. After finding that, we add it to the total of the targeted column or row that is blocked. Then we return the minimum between the number of pieces to complete the row and column.

These two heuristics combined together generally speed up the function. The manhattan distance make it so the A* search prefers states that are going in the direction of the targeted block, allowing to prioritise those states and expand upon them. However, it will not make it optimal just alone, as there are states that are more optimal even if they are further, a key example being in test-vis1.csv, a red block being next to the target block will take priority and expanded upon over those that only have 1 piece remaining because the closest block is further away, thus getting a complete, but unoptimal solution. Thus, the second heuristic is used to prioritise states that are making progress in completing rows/columns, allowing to find completed and more optimal states faster. As we wanted the favored states that have fewer pieces remaining to place over the distance to the block, the weighting of the Manhattan distance is 0.25 due to it being usually higher than estimated number of blocks.

Both remain optimal because they are both admissible. For Manhattan distance, the aspect of it not considering external elements such as blocks blocking the pathways, will mean it is always underestimating/equals the true distance. Whereas for the estimate_number_pieces_remain, it can only estimate the exact amount of blocks that the target row or column needs. As such, these two heuristic functions combined will both be admissible and therefore optimality isn't sacrificed.

Thus our heuristic function:
f(n) = number of pieces placed + estimated number of pieces needed + 0.25 * manhattan distance

Allows for the A* search to be faster while still remaining optimal.

Question 3:

There are multiple ways in which we can approach this. Firstly, we could maintain our solution and just keep clearing one "target" block at a time by adding them all into a list and only returning a solution once all blue blocks have been cleared. In this case, this would increase the time complexity to at worst the current search algorithm * the number of blue blocks we have to remove, as we would need to do it several times until we remove all blue blocks, but the space complexity will remain the same. Another way we could see it would be to not specifically target a specific block, but rather target to remove all columns/rows. If that is the case, we would only need to modify it so there is ENUM with every number from 0 to the Board Size and go through each row or column and mark it off in the ENUM till clear and have a separate count so we do not need to double check the ENUM Class. This will then increase the time complexity to the current search algorithm * (BOARD SIZE - rows/columns that do not have blue blocks in them), as we would be checking whether the rows or columns will be the most optimal to remove at the beginning, we can also check whether if there is any blue blocks on the row/column, allowing us to ignore them during the search and reduce time complexity.