COMP30024 Project A Report

David Le and Ella McKercher

Semester 1 April 8th 2024

1 Implementation

Our solution revolved around a relaxation of the problem - specifically, being able to place pieces of size 4 (the tetromino's size) to fill the target row/column without taking into consideration the shape of the tetrominoes. We opted for BFS preprocessing and A* search. Before running A* search, the distances between every empty square and each empty square on the target row or column was stored in arrays using BFS - this pre-processing of distances prioritizes certain paths, especially those with fewer obstacles and in turn, lower distances. This enabled the algorithm to evaluate heuristic values efficiently as the distance does not need to be calculated upon each iteration when nodes are generated. For A* search, heuristic values were calculated by identifying shortest distances to segments of the target row/column, adding this to the size of each segment, dividing this by 4 (the tetromino size), and then taking the ceiling of this value to determine the number of moves.

Notable data structures used in our program are nodes and priority queues. Each node is a data class that stores its corresponding board and a list of previous place actions taken. Min priority queues were implemented through Python's in-built binary heap, provided in the "heapq" module. Our implementation of A* search involved popping the the lowest cost node from the priority queue. After expansion, the node was checked to have either removed the row or column. If it had, the search terminated. If it did not, the node was expanded with its children nodes generated (those with valid moves) and pushed into the priority queue. This process would then be repeated until the priority queue was empty or a solution had been found.

1.1 Space Complexity Analysis

Both BFS and A* search keeps every node in memory, so worst case space complexity is the maximum number of nodes held in their queues. The space complexity of the BFS pre-processing would be $O(n^2)$, where n is the length of the board. The space complexity for A* search would then be $\mathrm{O(b^d)}$, with d being the depth of the least-cost solution and b the branching factor. In terms of the Tetress board, we estimate that the branching factor for our A* search is $(19 \times 4 \times \mathrm{R})$, where 19 represents all the possible tetromino shapes, 4 represents the maximum different orientations of each tetromino, and R is the number of empty squares that are adjacent to a red token.

1.2 Time Complexity Analysis

For our A* search algorithm, the worst case scenario would be that every possible move must be iterated through: $\mathrm{O(b^{\epsilon d})}$ where b is the branching factor, d is the depth of the search, and ϵ is the relative error in our heuristic. Relative error is calculated by $\epsilon = \frac{(\mathrm{h^* - h})}{\mathrm{h^*}}$ where $\mathrm{h^*}$ is the actual cost of the solution, and h is the estimated cost of the solution (page 98).[1]

Consider the initial case where no previous actions have been made. Then, $h(k) = g(k) = \sum_{i\in I}\lceil \frac{m_i+s_i}{4} \rceil$ (see section Heuristic below for explanation). In this case we presume $\mathrm{h^*}$ would represent the worst case for our heuristic, which would be if each tetromino placed could only occupy one square in the empty segment maximum (due to obstacles) as opposed to a tetromino being placed on multiple blocks of that segment. Then $\mathrm{h^*}(k) = \sum_{i\in I}(\lceil \frac{m_i}{4} \rceil + s_i)$. Thus, inputting this into the relative error equation, we would get that:

$$\epsilon = \frac{\sum_{i\in I}(\lceil \frac{m_i}{4} \rceil + s_i - \lceil \frac{m_i+s_i}{4} \rceil)}{\sum_{i\in I}(\lceil \frac{m_i}{4} \rceil + s_i)} \tag{1}$$

An example of the computation of this for a specific board with two contiguous segments of sizes 2 and 3 respectively, and shortest distances to the segments were 7 and 5 respectively (i.e. $m_1 = 7, m_2 = 5, s_1 = 2, s_2 = 3$ and $I = \{1, 2\}$). Then, in this example, $\epsilon = \frac{4}{9}$. Thus, the time complexity of A* search is given by $O(b^{\epsilon d}$, where $b = 19x4xR$ (as discussed in section 1.1), d is the length of the optimal solution and ϵ is given by equation (1) above. In contrast, the time complexity of the BFS pre-processing would be $O(V+E)$, as we visit each ”vertex” (i.e. square) once and explore its neighbouring vertices (squares). So, where n represents the length of the board, $V = n^2$ and $E = 2n^2$ (as the board is ”wrapped” around). So, the time complexity of BFS is $O(n^2)$.

The overall time complexity would be the time complexity of the BFS pre-processing added to the time complexity of the A* search algorithm. So, $O(n^2 + b^{\epsilon d})$, where b and ϵ are the values stated before for A* search.

2 Heuristic

Our heuristic is computed by $h(k) = f(k) + (1+p) \cdot g(k)$, where $f(k)$ represents the number of previous actions taken, $g(k)$ represents the number of moves to clear the remainder of the target's row or column and $p = 0.001$. Let I be the set of empty contiguous segments in the target row/column, let m_i be the distance to that segment (as determined by the BFS pre-processing) and s_i be the size of that segment. Then, $g(k)$ is calculated as $g(k) = \sum_{i \in I} \lceil \frac{m_i + s_i}{4} \rceil$.

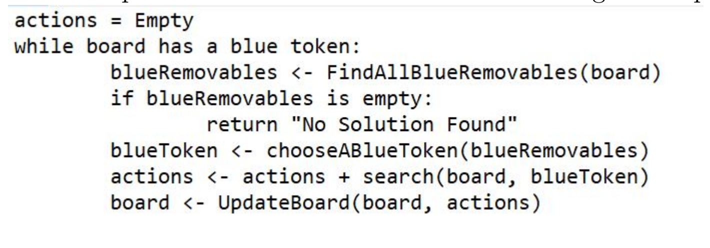
$g(k)$ is multiplied by $1+p$ to ensure that ties are broken within the heuristic function, prioritising nodes that have fewer moves remaining to remove the target. For example, if $f(k) = 1$ and $g(k) = 3$, then $h(k) = 4.003$, whereas, if $f(k) = 3$ and $g(k) = 1$, then $h(k) = 4.001$. Between these two cases, the latter would be chosen since it has a lower heuristic value, which is what we want to prioritise since the goal is simply to find one optimal solution. The value of p was chosen to be miniscule to not have an effect on $h(k)$. Specifically, since it was known that number of moves would not exceed 150 moves for part A (as specified in document), we chose p such that $p < 150$.

Since the time complexity of our heuristic function is $O(b^{\epsilon}d)$ (as discussed in section 1.2), where $\epsilon \leq 1$, then this time complexity would be less than that of an uninformed search method, which would have time complexity of $O(b^d)$, thus demonstrating how our heuristic speeds up the search.

Our heuristic is consistent *and* admissible, two conditions needed for optimality in this context. It is consistent as the step cost from going to one node to its children node is equal to the actual step cost (i.e. a placement of a tetromino). Since our heuristic function determines the minimum number of blocks needed to be covered to reach the shortest path to each segment and cover all blocks of that segment (then dividing this by 4 to calculate the number of moves), it will always be less than the actual cost. In contrast, the actual cost may fill the same blocks, and more additional surrounding blocks due to the shape of the tetrominoes. Thus, as $\sum_{i \in I} \lceil \frac{m_i + s_i}{4} \rceil \leq h^*(k) \leq \sum_{i \in I} (\lceil \frac{m_i}{4} \rceil + s_i)$ and $h(k) = \sum_{i \in I} \lceil \frac{m_i + s_i}{4} \rceil$, then $h(k) \leq h^*(k)$, as required for admissibility.

3 Removal of All Blue Tokens Problem

Removing one target blue token would take $O(b^{\epsilon}d)$ (as described in more detail in the heurstic section). If the goal state was to have no remaining Blue tokens, the worst time complexity of the problem in terms of our solution would be: $\mathrm{O}(\mathrm{B} \times \mathrm{b}^{\epsilon d})$ where B is the number of Blue tokens on the board. This worst case scenario would occur when all blue tokens are on distinct rows or columns, or positioned in such a way that the optimal removal of that token would not remove any other blue tokens. Our solution would have to implement a repeated call of our A* search that updates until no blue tokens are remaining. A simple psuedocode of the process is shown below:



```
actions = Empty
while board has a blue token:
        blueRemovables <- FindAllBlueRemovables(board)
        if blueRemovables is empty:
                return "No Solution Found"
        blueToken <- chooseABlueToken(blueRemovables)
        actions <- actions + search(board, blueToken)
        board <- UpdateBoard(board, actions)
```

In our pseudocode, it is important to choose a blue piece that is removable. This can easily be determined based on the heuristic value and bfs pre-processing. It is also important to update the board every time to make sure an updated board is performed search on. The pseudocode also keeps track of the placements of tetrominoes.

References

[1] Russell, S. and Norvig, P. (2016). Artificial Intelligence: A Modern Approach. 3rd ed. Pearson Education.