COMP30024 Project A Report

David Le and Ella McKercher

Semester 1 April 1st 2024

1 Implementation

For project A, 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 preprocessing of distances prioritizes certain paths, especially those with fewer obstacles and in turn, lower distances. It also enables the algorithm to evaluate heuristic values efficiently as the distance does not need to be calculated upon each iteration.

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. 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

A^* search keeps every node in memory, so worst case space complexity is based on the instance where the maximum amount of nodes can be held in A^* search's priority queue. The space complexity would then be $O(b^d)$, with d being the depth of the least-cost solution and b the branching factor. See Time Complexity Analysis for A^* search's branching factor.

Both BFS and A^* search keep every node in memory, so worst case space complexity is based on the instance where the maximum amount of nodes are held in each respective algorithm's queues. BFS's space complexity would then be $O(b^d)$, the same as A^* search but with a branching factor of 4 (there is a maximum of 4 possible branching opportunities from each square of the Tetress board).

1.2 Time Complexity Analysis

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

In this case we presume 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 each move (due to obstacles). h would then represent our current heuristic (see Heuristic), which takes the upper bound of a sum of the shortest distance to a segment and the size of the segment.

In terms of the Tetress board, we estimate that the branching factor for our A^* search is $(19 \times 4 \times R)$, 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.

The time complexity of our breadth first search algorithm would be $O(b^d)$, exponential, as BFS generates nodes by a branching factor of b at each depth (d) of the search. Considering the Tetress board, the branching factor for BFS would be 4, as there are maximum 4 possible moves from each square (up, down, left, right).

A* search's time complexity dominates BFS as A* search's branching factor is greater. Therefore the overall time complexity of our program is approximately $O(b^{\epsilon d})$ with a branching factor (b) of 76R where R is the number of empty squares that are adjacent to a red token, d is the depth of the least-cost solution and ϵ is the relative error in our heuristic.

2 Heuristic

Our heuristic is computed by summing the minimum number of moves to clear the target's row or column with the number of previous actions that have been taken. The minimum number of moves to clear target row/column is calculated by tracking the This is then multiplied by 1.001 to break any ties between the minimum number of estimated moves. If the target has been removed, the heuristic will be the number of previous actions, i.e. h(0) = length of previous actions. If the target has been removed, the heuristic will be the number of previous actions, i.e. h(0) = length of previous actions.

Our heuristic is consistent *and* admissible, two conditions needed for optimality in this context. Our heuristic is equal to the estimated move from a successor node to the target, plus the step cost of reaching that successor node, and according to the consistency theorem (page 95),[1] this indicates that our heuristic is consistent. A consistent heuristic will always be admissible, as a consistent heuristic does not overestimate the cost of reaching the target, therefore our heuristic is also admissible. Our A* search will always find the least cost path, if there is one.

3 Removal of All Blue Tokens Problem

If the goal state was to have no remaining Blue tokens, the time complexity of the problem in terms of our solution would be: $O(B \times b^d)$ where B is the number of Blue tokens on the board. The problem changes from one target block to many, with all the blue blocks on the board becoming targets to remove. Our solution would have to implement a repeated call of our A* search that updates until no blue tokens are remaining.

4 Previous Djikstra Implementation Attempts

Both BFS and Dijkstra's algorithm were tested as preprocessors for A* search for this project. The "dijkstra" function in our program calculated the shortest distance from each square on the board to the target square, and stored those values in a dictionary ("dist") to be used by search. Dijkstra used a priority queue to identify the smallest distance (element) in the queue.

Comparison of Dijkstra's algorithm and Uniform-Cost Search (UCS) demonstrate that the main difference between the two search techniques is the termination sequences - UCS searches for a goal node, whilst Dijkstra's continues until the priority queue is empty (all nodes have been explored). The text states that Dijkstra's algorithm is the origin of UCS, so we will assume this comparison is appropriate (page 110).[1] From that comparison, it can be assumed time and space complexity of Dijkstra's algorithm is roughly $O(b^{1+\frac{C^*}{\epsilon}})$ where C^* is the cost of the optimal solution and ϵ is the cost of every action. With this particular problem, each action has the same cost, so dijkstra's complexity can be approximated to be $O(b^{d+1})$.

The rationale for favoring BFS over Dijkstra's was based on complexity analyses. In Tetress, step costs are equal, and this reduces the differences between Dijkstra's ($\sim$ UCS) and BFS down to when each algorithm terminates. Dijkstra's algorithm terminates once it has found the lowest cost node at the depth of the target, whereas BFS will terminate once it has reached the target. This means that Dijkstra's algorithm can perform unnecessary work compared to BFS (page 85).[1]

References

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