COMP30024 Project Part A Report

Team Name: Admin

Team Member: Yifeng Pan (955797) & Ziqi Jia (693241)

1. **How did you formulate this game as a search problem?**

The state in this question contains the position of all the upper player's tokens on the board at the start of the game turn. These tokens are represented by different symbols of rock, scissors, or paper. The state also includes the symbol and position of each of the lower player's pieces and the position of any block, however, remains static for the purpose of part one of the project.

The actions of the problem can be interpreted as the action the upper player could perform for each token during each turn of the game. According to the game rules, there are three actions that the upper player can take: Slide, Swing and Battle. each of which will change the state of the game. Sliding grants tokens the ability to move to the next position by one hex in all directions if possible, while swinging expands the token's movement range to include its first friendly neighbour token's slide movement range, and battle action will grant the token to defeat (or get defeated) other tokens to clear pathway or achieve goal. For example, change the position of any tokens in the upper player's tokens set, or battle with the lower player's pieces to remove a token.

The goal test can be formulated as, are all the lower players' pieces removed? If the lower player has any pieces left, the goal has not been achieved. However, for the first part of the project the problem is guaranteed to have a solution, which means the goal test can also be formulated as, for each upper token the player has, is all the lower tokens on the board it is capable to defeat got defeated? If any of the upper player's token still got remaining lower pieces it could defeat, then the goal has not been achieved.

If looking at each individual upper token present on the board, the path cost is the number of actions it requires to complete all its goals. However, the actions the token takes are not counted directly but through the incrementation of game turns, which each turn all the upper tokens could perform movements simultaneously. Therefore, the actual path cost of the game becomes the number of turns needed to achieve all goal states. Since no turn is special, all the turns can be regarded as having the same weight 1.

2. **What search algorithm does your program use to solve this problem, and why did you choose this algorithm?**

The best first search strategy was first implemented to solve the problem. It is realized through an evaluation function to calculate all of the Euclidean distance with hexagonal coordinates of the upper player's tokens next possible position to the nearest goal, which is also determined by the distance calculation function, and the position with the lowest distance to the destination will be marked as the most preferable for traversing and take that position as move action until the target token is reached by the token. It is easy to implement and cheap in terms of time complexity. However, the completeness of strategy is found to be incomplete and could not guarantee a solution even though there is one.

$$h = \sqrt{(\mathrm{r1} - \mathrm{r2})^2 + (\mathrm{q1} - \mathrm{q2})^2 + (\mathrm{r1} - \mathrm{r2})(\mathrm{q1} - \mathrm{q2})}$$

For improvement, the previous algorithm is modified into a variant of the informed A* search algorithm. First, for a given token, the nearest goal will be determined. Then, all the possible moving positions in each turn and using a heuristic function that sort them according to their distance to the target position. Then, the nodes expanded are marked as visited and depth-first traversal will visit the unexplored nodes and mark it as explored recursively. The visit starts from the node with the lowest distance to the goal until the goal is achieved or the last node does not have any branches marked as unvisited. If it is in the latter situation and the goal is still not found, the pathfinding will resume from the last node that has unvisited nodes and repeat

the process above again. The algorithm will terminate when all the nodes are explored and return the path from start to goal.

The original A* search algorithm will avoid expanding nodes that are already expensive. However, it is not possible in the context of this problem since some routes may require taking a 'detour' over the obstacles. If avoid expanding expensive nodes, it will result in a straight line to the goal and trapping in a dead end, just like the previous algorithm. Instead, the algorithm will avoid expanding the expanded nodes so that it does not trap itself in a loop. Therefore, the variant is complete since it will traverse all the nodes once and return a valid route if it exists. However, the efficiency is not impressive since the searching algorithm's behaviour can be similar to the depth-first search of exponential time complexity O(b^m). But since the heuristic function gets the shortest straight-line distance from start to goal which cannot be overestimated, it is admissible, and therefore the algorithm will be optimal as it will always try the shortest possible route to the goal first, and then try the alternatives in ascending orders of distance to the goal.

Implementing the A* algorithm in a depth-first search manner has a preferable characteristic of returning a clear path containing start and goal. A stack is used to record and update the searched path. A new node is pushed to the stack when it is newly traversed. When the algorithm needs to return to the last node with unvisited nodes, all the visited nodes before the returning point will be popped from the stack.

Although the position of the target pieces is known, the upper player's tokens do not know which path each other upper tokens might take in the turn, so the path found is only used for the upper tokens to move their position by 1 in the next turn. After all the upper tokens have moved in each turn of the game, the path of every node needs to be searched again since the old path might become unavailable or there will be a new shorter path found. The tokens will stick to the old path if it is not available for consistency and only will change when the new available route is of smaller path cost or the old route becomes unavailable.

3. **How do the features of the starting configuration impact your program's time and space requirements?**

Our search strategy requires traversing every node on the board starting with the position of the upper player's token in every state of the game. The algorithm is very demanding on the time complexity O(b^m), where b is the maximum branching factor of the search tree and m is the maximum depth of the state space. But since the board given is of a fixed size and the node has a fixed number of adjacent nodes, the maximum depth is a constant, and so is the maximum branching factor, which makes the time complexity acceptable. The advantage gained from such undesirable time complexity is the low memory demand, O(bm), i.e., space linear in length of action sequence.

Such behaviour means that, unless the board size got changed, the upper bound of time taken for the algorithm to return a path remains constant regardless of the starting configuration. However, as increasing the number of upper tokens, the memory required to return a path will not explode drastically and it means for this algorithm it can handle the situation where many upper tokens are present in a memory-efficient way.

As comes to the actual time and space usage, the more separated the upper pieces are from lower pieces, the more time it will take to complete the entire game and the time would generally increase in a linear manner, since the larger separation means the greater number of pathfinding needs to be done. And the separation here not only refers to the linear distance separation, but also the extra moves required to reach the goal caused by all kinds of obstacles, as the lower player's pieces and blocks limit the upper player's moves. It may generate the extra cost to goal, which means that for N extra path costs, the extra time cost will be N*(b^m). However, each upper token only finds the path to its nearest goal, therefore for each token the memory to store a path is only dependent on the number of moves to reach goal, and it grows linearly as increasing the number of upper tokens.