Assignment 1 Part A Report

1. How did you formulate this game as a search problem? Explain your view of the problem in terms of states, actions, goal tests, and path costs, as relevant.

To formulate this game as a search problem, we looked at the initial state, how this changed over the game, and what the terminal state was. The state of the game is defined by the position and type of tokens on the board, being tokens of upper and lower, and blocker tokens. The actions in this version of the game are to slide or swing upper's tokens to a new tile, as those are the only tokens that can move. This game has an explicit goal, destroying all of lower's tokens, i.e., lower having no tokens remaining. In terms of the path cost of the solution, the cost is defined by the total number of turns taken to win the game.

2. What search algorithm does your program use to solve this problem, and why did you choose this algorithm? Comment on the algorithm's efficiency, completeness, and optimality. If you have developed heuristics to inform your search, explain them and comment on their admissibility.

Our program uses A* as the search algorithm. This algorithm uses an evaluation function to estimate the cost from the current node to the goal, and from the initial node to the current node. A* itself is complete and optimal, given non-infinite search space and an admissible heuristic.

In our implementation, our heuristic was inadmissible, hence the search is not optimal. We chose an inadmissible heuristic because it ran faster than admissible heuristics we trialled. It is not an admissible heuristic because it does not account for swing actions, which can reduce the cost to reach a lower node. The program creates a list of the distances from all upper nodes to their corresponding lower nodes and returns the maximum of those distances. The corresponding lower nodes are the closest node to an upper node that the upper node can eliminate. Some nodes also have their position in the heap queue devalued if they have less upper tokens remaining. This avoids exploring states a solution may not be able to be reached from (i.e., if an upper rock is eliminated and there are still lower scissors remaining).

The time complexity of A* is derived from $O(b^{d+1})$ for best-first search, where b is the branching factor and d is the depth of the solution. With our heuristic, the time complexity of our A* algorithm is $O(b^d)$, where b is the branching factor, and d is the error in our heuristic function multiplied by the length of the solution. A* is a time-efficient search algorithm, however the implementation with an inadmissible heuristic in this case decreased the efficiency of the algorithm.

The space complexity of our algorithm is the same as best-first search, as we must store all the generated nodes in memory, hence the complexity is $O(b^d)$, where b is the branching factor and d is the depth of the solution.

3. How do the features of the starting configuration (the position and number of tokens) impact your program's time and space requirements? For example, discuss their connection with the branching factor and search tree depth and how these affect the time and space complexity of your algorithm.

The maximum number of moves for one token including additional swing moves is 9. If another token is added there are 9 * 9 combinations of these moves. Therefore, the branching factor is 9^n, as new nodes are generated for each possible move combination. Generalized, the branching factor is m^n, where m is the maximum number of moves available for each token, and n is the number of upper tokens. In this context, the starting configuration significantly impacts the time and space requirements of the search, as the number of tokens on the board is the exponential factor in this expression. On the other hand, the number of opponent tokens will not impact the branching factor, as the branching factor as shown only depends on the number of moves available to the upper tokens.

There is an inverse relationship between tree depth and the number of upper tokens, because as the number of upper tokens increases, it takes fewer turns to reach a solution against the same number of lower tokens. For example, if there is one rock upper and two scissor lowers, then introducing another rock upper may allow them to find a shorter solution. Our program however does not benefit much from this as it is not guaranteed to find the optimal solution, so having more upper nodes creates more branches that may reach non-optimal solutions. This means the time complexity of the program suffers, and computation takes longer. The position additionally affects the depth as the further an upper token is from a lower token, the more paths it can take to that token. Furthermore, with the non-admissible heuristic this may generate more non-optimal solutions. The space complexity worsens equally with the time complexity, as it is equal to the number of nodes generated.