RoPaSci360 single player is a fully observable, deterministic, episodic, static, discrete environment for an AI to process. The problem could be broken down into states where each state provides the symbol, position and player of all the tokens on the board. In this single player version of the game; our goal as the player or AI is to play as the Upper and defeat all lower by moving all tokens simultaneously per turn. The initial state of the game is given as the input and the goal is a state where all the Lower tokens are defeated. The path cost of a certain solution is the number of turns taken in order to achieve the goal state.

To formulate a solution to the game each Upper token controlled by the AI would need to find and follow a path whereby it defeats all defeat-able tokens corresponding to its own symbol. This is how the game can be abstracted into a search problem, where from our current state we must search for opposing tokens from each of our tokens in order to know what action to take each turn.

Thinking more specifically of what happens in AI; since all Upper tokens move per turn, the problem can be broken down to search problems for each Upper token. The current position of a token is the root of the tree and each node in the search tree represents a possible reachable position. Since for each token any slide/swing move takes one turn, the depth of a node in the tree represents the number of turns required or path cost to reach that position. This is assuming we use a complete and optimal search algorithm.

Our program uses breadth-first search to solve the problem. We chose this algorithm as it would be complete and be optimal for finding the shortest path from the root node to the nearest possible goal node while being uninformed. Since it searches by a depth-by-depth basis prioritizing the oldest neighbours in the queue (FIFO); the shortest possible path to a position will be that position's first appearance within the search tree. Alongside this search, we will store previously traversed nodes in order to simplify the search as search loops would be expected due to the hex system of six neighbours.

The time complexity and space complexity of this search is considered high but due to the limited size of the playing board, the amount of computation required for a single BFS is limited. The highest possible depth/path cost is 36 for the case when the nearest defeat-able token requires the furthest possible path in the gameboard which would be in a spiral utilizing block tokens or undefeatable lower tokens. In this case fortunately the branching factor will be limited to 1 while in the spiral creating a linear search problem. In the worst branching factor case assuming we store all previously visited nodes; 6 can be reached but at most only once in a search. More usually, neighbouring nodes are already explored resulting in less branches therefore a branching factor of 2 - 3 is more expected.

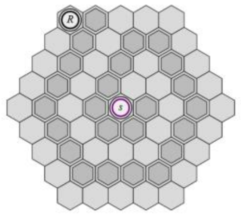

Image: Depth of 36, Branching Factor of 1.

Breadth first search runs at a time and space complexity of O(b^d). However due to the limited size of the game's board and by the game's design we can limit the exploration by storing an array of all previously visited nodes as mentioned earlier. Theoretically, the biggest BFS avoiding all previously traversed nodes would traverse all the hex positions in the board. A case where this happens is if the upper token is on one corner and the nearest defeat-able lower token is on the opposite corner of the board. The search would need to traverse through all 61 hex positions. The depth would go up to 8 and the branching factor would vary from 1 - 3.

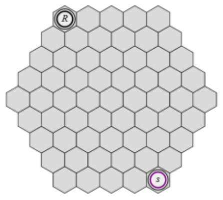

Image: BFS that explores the entire board.

For every turn, each upper token has multiple BFS searches performed to find its nearest defeat-able token. The number of searches per upper token matches the amount of defeat-able lower tokens on the board. This is to ensure the Upper token will head towards the Lower token of lowest distance first. Therefore, for every new upper token, each turn needs to perform at least one more BFS search. Under the specification, there can be up to 27 BFS per turn for the case where three of the same symbol upper tokens have 9 defeat-able lower tokens to search for (e.g., 3 upper rocks and 9 lower scissors; 3 x 9 searches). The space complexity would not be heavily affected as one BFS would only need to be performed at a time no matter how many upper or lower tokens are on the board.

While the in the specifications, it is stated that all Upper tokens move at once, in our solution each Upper token has its next action solved (with BFS) separately to form the final solution. As a result, the ordering how upper tokens are stored affects the output if the program. For the extremely rare situations where the order does matter and can make the problem unsolvable, our solution will repeat turns but with different orders in a permutation loop. Since there are a maximum of 3 upper tokens, a turn at maximum may be repeated up to 6 times (3! = 6). These permutations will be iterated only when needed, therefore for almost all cases a turn will only use one permutation.

Lower tokens are more varied to how much it affects runtime but adding a new lower token does require at least one more BFS to be performed per turn. The distance of this token from upper tokens will affect the run time as the more turns required to reach the token; the more BFS needed. It will also depend on how many upper tokens are present and can defeat this lower token. Each upper token would need to perform a search to this lower token. The amount of time added by each lower token will also vary based on how many lower tokens are already close to it as well. But in essence; for each lower token, another BFS will be performed per turn for every compatible upper token.