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.

RoPaSci360 singleplayer is a fully observable, deterministics, 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 oktens on the board. In this singleplayer 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 of 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.

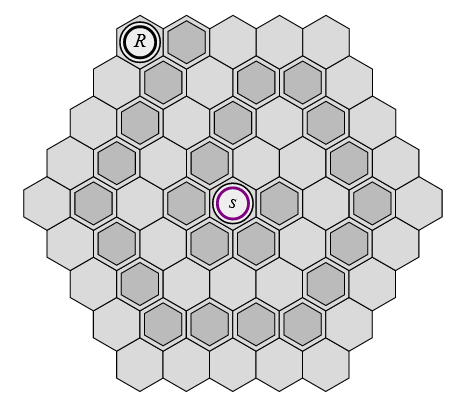
To formulate a solution to the game each Upper token controlled by the AI would need to find and follow a path whereby it defeats all defeatable tokens corresponding to its own symbol. This is how the game can be abstracted into a search problem, where from our current state we must search for opposing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

1. 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 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. Breadth first search expands it’s search at a depth by depth basis prioritizing the oldest neighbours in the queue (FIFO).

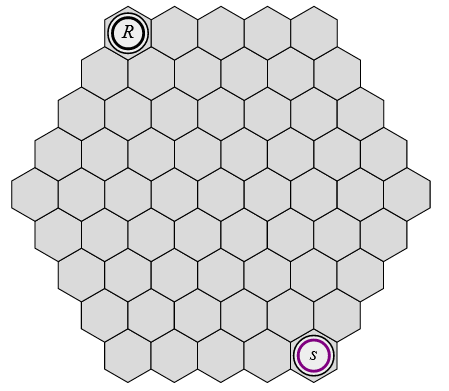
Specifically applying the algorithm to our problem; each node in the search tree represents a possible position reachable from the root/initial position of the search. Since for each token, any slide move takes one turn, so the cost of each move will only be one. As a result the depth of a hex position in the tree represents the number of turns required or path cost to reach that position. The shortest possible path to a position will be that position’s first appearance within the search tree which is the basis of how we search for nearest defeatable tokens. The first located defeatable token will be returned in our BFS.

It’s time complexity and space complexity can be considered to be high however the maximum depth is at worst is 36 and branching factor to 6 for the game due to the limited size of the playing board. The highest possible depth/path cost is 36 for the case when the nearest defeatable 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 however the branching factor will be limited to 1 while in the spiral. In the worst case branching factor usually never goes over 5 once assuming we store all previously visited nodes and therefore cannot traverse in reverse. But even 5 is rare as we are searching at a depth by depth basis and therefore neighbouring nodes are likely to be already explored resulting in less branches.

Image: Depth of 36, Branching Factor of 1 (count this again to check thanks)

1. 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.

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. Traversing a node a second time will not help find a shorter path as it already was traversed in a previous depth so at the cost of memory we save a lot of time by only traversing outward. Theoretically, the biggest BFS avoiding all previously traversed nodes would traverse all of the hex positions in the board. A case where this happens is if the upper token is on one corner and the nearest defeatable lower token is on the opposite corner of the board. This would mean it would only need to traverse through 61 hex positions at most.



However, for every turn, each upper token has a BFS search performed to find it’s nearest defeatable token. Therefore for every new upper token, each turn needs to perform another BFS search of up to three per turn under the specification which means the time complexity would linearly increase in the worst case where all tokens have defeatable tokens on the board of the correct symbol. 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. Lower tokens are more varied to how much it affects runtime as adding a new lower token does require at least one more BFS to be performed. 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. The amount of time added by each lower token will vary based on how many lower tokens are already close to it as well and if there are multiple potential upper tokens available to remove it from the board.

For every move required to reach a lower token (it’s distance), the exponentially bigger a single BFS would be as each move is a whole new depth to search. The base of this exponential growth depends on the branching factor, which as previously mentioned would at maximum be at 6. This is for only the case of a root node not at the edge of the board. However normally it is expected in the search for the branching factor to only be 2 - 3 branches per node since previously traversed neighbouring nodes are ignored.

THE QUESTIONS ARE TO BE REMOVED FROM REPORT