Project Part A Report

In terms of a search problem, the game of *RoPaSci 360* was formulated as a starting node of a state of the game board, with a searching algorithm used to go through potential successive board states to find the goal.

Initial State

StartingNode => state of the board based on the starting input.

i.e. a class object of the upper and lower coordinates

`self` is an instance of the `Node` class.

```
self.upper = [["s",2,-1],["r",-3,2]],
self.lower = [["s",1,-1],["p",-1,1]],
self.block = [["",0,0]]
```

Actions

Possible change of the board due to movement of upper tokens

i.e. movement of all upper tokens in one of 6 directions. Given a coordinate (r, q), the slide actions could be

```
(r + 1, q)
(r, q + 1)
(r - 1, q + 1)
(r - 1, q)
(r, q - 1)
( r + 1, q - 1)
```

There are also swing movements possible considering the close spacing of nearby upper tokens. The new board due to the movements of the upper tokens is considered an action.

Goal Test

0 lower tokens remaining on the board

i.e. `self.lower` is empty

Path Costs

Actual number of turns required to complete the game and form a solution.

The corresponding solution is the set of nodes, or board states, that lead to the solution of an upper win.

The algorithm we chose to use to solve the game problem was A* search. This was because it is an informed search algorithm, which is efficient. An informed search algorithm doesn't necessarily get the most optimal solution; however, that isn't necessary, as we just need any solution. A* search also doesn't expand the search tree as wide, which is essential in a game with multiple different tokens with multiple possible moves.

A* searches efficiency in time is quite acceptable, as we have chosen an appropriate heuristic. It's more than acceptable to meet the 30 second limit.

It's efficiency in space isn't great due to it having to keep all the nodes in memory. However, this isn't an issue for this problem as the nodes we keep are small in size and there aren't many of them. Moreover, space isn't a limiting factor in this problem.

A* search is complete in this case as we have a finite number of branching nodes due to us stopping the search from accessing nodes we have already looked at.

A* search is optimal as we can only search the neighbouring node once the current node has been complete.

The heuristic we used to guide our search is an application of Manhattan Distance. We look at the node and the cost for the node to reach the goal state. This is done by calculating the Manhattan Distance between all upper tokens and their respective lower "goal tokens" (token that can be defeated by the upper token). If an upper token has more than one lower goal token, the heuristic cost is additive: the distance from the original position to the closest goal token, from the latter to the next closest goal token, etc.

Because of this A* returns the most optimal solution in the case of one upper token.
In the case of multiple upper tokens it should also return the most optimal solution. This is because our heuristic is admissible, in that it never overestimates the path to the solution. This is because of our use of Manhattan Distance.

The position of tokens doesn't necessarily affect the program all that much. It just means that the search will take slightly more time and space depending upon the complexity of the positioning. The greater factor is the number of tokens. In the case of one lower token the search will always return the most optimal solution as the search will follow the best path. In the case of multiple lower tokens the search is still quite fast as there are now multiple goals involved in the heuristic. In the case of multiple upper tokens, the search has to take into account upper tokens landing on one another, and so the search complexity in both time and space is increased.