Created by : Shelanah Rahman [student id: 995900], and Ayda Zhao [student id: 1081566]

The single-player variant of RoPaSci360 is a game played on a hexagonal board of 61 hexes, where the player's token (upper tokens) are all simultaneously moved each turn in order to defeat all the lower tokens on the board. Each token has a symbol of rock, paper or scissors and follows the ruling of the traditional game, "rock, paper, scissors". The possible moves are slide actions (direct move one hex away from the original hex) and swing actions (a slide action from an adjacent upper-token). We formulated this game as a search problem as it had the defining properties of a search problem: ostates, actions, goal tests and path costs.

The states were the current and complete board state which were defined by the positions of all the tokens on the board. The board state consisted of three dictionaries: the upper tokens, the lower tokens, and the block tokens. Each board state also carried the sequence of upper token's moves leading to the current state, along with the number of lower tokens removed since the initial state. The initial state is defined by the given board state where no token has been moved or removed yet.

The movement of upper tokens to simultaneously slide or swing is considered to be the actions of a search problem. However, certain moves were disabled if they were invalid, disadvantageous or unproductive. Invalid moves could mean that the upper tokens were moving out of the boundaries of the board or moving onto a block. Disadvantageous moves were when upper tokens would defeat another upper token or they would be defeated by a lower token. Unproductive moves were when upper tokens moved to a previously visited hex which could lead to possible loops. In consideration of the possibility of moving more than one upper token simultaneously in a turn, we took the combinations of movement from each token that fulfilled the criteria of a possible move. For evaluating the possible turns (and their preceding sequence of turns), we implemented a priority queue.

This search problem also has the explicit goal to destroy all the lower tokens on the board. This was taken into consideration in the elements within the queue. The elements of the priority queue were formatted as the following: [turn sequence, upper dictionary, lower dictionary, block dictionary, number of lower tokens eaten]. The first 4 elements indicate the sequence of actions to the board state and the board state, while the last element is a measure of closeness to the goal. The priority queue was sorted according to the number of tokens eaten or destroyed. If a particular turn-sequence generated a higher number of tokens being eaten, that turn-sequence and board state would be popped off from the priority queue first, and then future turns for what happens after the dequeued-turn sequence would be considered.

The algorithm that we decided to implement is a breadth-first search (BFS) algorithm which is a highly ideal strategy for this search problem. Originally a distance-closing strategy was considered, where the program worked on minimising the distance between upper tokens and weaker lower tokens. However, due to the presence of block tokens, there was the possibility that upper tokens could get stuck behind a wall while trying to "chase" a lower token on the other side of it. In contrast to a distance-closing strategy, a breadth-first search program could discover a way to effectively navigate to reach the goal, despite the block tokens forming a barrier. This makes the BFS algorithm complete since its brute force method of looking through every possibility ensures that a solution is found eventually. In terms of being optimal, the BFS method is optimal in the case that the cost of all the edges of a graph are equal. In the case of this

search problem, it is optimal since the distance between the next possible coordinate and the current coordinate is insignificant, since all possible moves are weighted the same whether they are slide or swing actions. BFS does have a large time and space complexity of O(b^d) where b is the maximum branching factor of the search tree and d is the depth of the least cost solution.

While our program acts to reduce the number of unnecessary moves required to increase efficiency and reduce memory usage, the program's performance can vary based on the starting configuration. An example could be the distance between an upper token and a lower token, under the assumption where the upper token can beat the lower token. At most, one token has 6 possible moves when only the slide function is implemented and 6 is the *b* variable in the BFS algorithm referred to before. Furthermore, since our algorithm is based on generating combinations, for every increase in turn, the number of possible combinations increases exponentially to make 6^*t* possible combinations. The number of turns (*t*) coincides with the d variable in the BFS algorithm referred to before. Hence, in the case where there is a large distance between the upper and lower tokens, the greater number of turns required to be considered to reach the goal. Hence, the search tree depth is also increased, increasing the time complexity. Moreover, in the case where there are *n* number of multiple upper tokens, the number of possible combinations of multiple upper token moves would also increase greatly. Hence, *b* is no longer equivalent to 6. Instead, *b* equates to the number of combinations of moving *n* upper tokens in a turn. While the depth search is not directly impacted, the branching factor increases, which also increases the time and space complexity.

The specific placement of the block tokens also can impact the program's performance. When the block tokens are placed such that they are barriers on the direct pathway between an upper token and a lower token, they can act to increase the number of turns (n) necessary to reach the goal. This is because the program will then have to navigate a path around the blocks, increasing search tree depth along with time and space complexity. On the other hand if the blocks are surrounding the direct pathway between an upper token and a lower token, the blocks will reduce the possible valid moves considered by the program, reducing the branching factor. Hence, this would increase efficiency and decrease the time and space complexity.