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

The full two-player version of RoPaSci360 is a game played on a hexagonal board of 61 hexes, where the player's token and the opposing player's token are both simultaneously moved each turn in order to remove 9 tokens of the opposing player. 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 throw actions (placing a token onto the board), slide actions (direct move one hex away from the original hex) and swing actions (a slide action from an adjacent upper-token).

What search algorithm have you chosen, and why?

We implemented our version of the minimax algorithm to select actions throughout the game. In our version of the minimax algorithm, it runs at a very short depth to not overwhelm the computer, and because the enemy is making choices every turn, and so we have to adjust to every enemy choice, and not merely run on long term predictions. We chose this algorithm since it matched our coding abilities, and since it seemed naturalistic. In a realistic case, a human player playing the game would look at their opponent's tokens and predict what their opponent would do to maximise its gain. Then, they would make their own move based on maximising their own benefits, while also minimising the enemy's benefits.

How have you handled the simultaneous-play nature of this game?

In order to consider the simultaneous-play nature of RoPaSci360, our AI keeps track of the position of the enemy's tokens to have knowledge of how to act against the enemies. This lets us generate potential enemy actions, based on the current boardstate. To judge every potential enemy action, we used an evaluation function to predict what the enemy would be likely to do. Then based on what we predicted the enemy team's action would be, our player then generates a list of possible actions/boardstates for itself. The player's generated moves/boardstates are then evaluated against the enemy's predicted boardstate, using an evaluation function. The highest scoring player-generated move/boardstate is executed. Hence, the player avoids any action that increases the enemy's chances of winning.

It does this every turn to keep up with the constantly changing conditions.
A lot of the strategy is contained in the evaluation function, as this is the function that stops the AI merely picking random moves.

What are the features of your evaluation function, and what are their strategic motivations?

About our evaluation function:

One feature was tracking the average distance between our player tokens that are stronger than the enemy's weaker tokens. This was made so if our team had a scissors-type token, it would be able to pursue the enemy's paper type tokens, (or any other type matchup). If a possible generated move, and so board-state, was found to decrease the average distance between the player's strong tokens, and the enemy's weak tokens, it would score higher, because that would mean the player is closer to possibly eating an enemy.

Another feature was inspecting the tokens and their types on the board, to see if the tokens our team had were sufficient to beat the enemy's tokens. For example, if the enemy had paper tokens, but our team had no scissor tokens, it would generate a scissors token to throw onto the board, so our team would have something that is able to eat the enemy's tokens.

When generating possible board states, it would also evaluate if a token from our team ate a token from the enemy team, or if one of our tokens was eaten (either cannibalised by its own team, or eaten by the enemy). Board states and moves that would cause our player token to eat the enemy would be scored highly, while board states where our player loses token, would be scored negatively, so the AI avoids picking detrimental moves.

It also has a "danger" detection". It is less likely to pick board states that cause it to stand within immediate vicinity of an enemy token that can eat our player tokens.

Performance evaluation:?

We manually evaluated the AIs, by watching it perform against other AIs.

Judgement of the AIs consist of:

- If it contained no bugs/errors
- If it won, draw, or lost. If a feature was implemented, and it was more likely to win, then it was good.
- How quick it would win. If a feature was implemented, and it lowered the number of turns, then that feature was good.
- Human judgement. Having a human watch the AI play the game turn by turn. If the human senses the AI is doing something daft (e.g. not eating a token right next to it), then the human would go back to adjust the evaluation functions/add more features.

Our program did win against the random AI player, and the greedy AI player, but mostly draws against other people's AIs. It seems against other people's AIs, our AI plays more "defensively", as it tends to run away from the tokens, rather than seek to attack.

The approach to versions of our AI, it evolved gradually. We generally added features one by one to our program, where we thought it needed improvement. Since playing the versions of our own AI's against each other often resulted in a draw, the features that decreased the amount of turns to play the game, or lead to a win, against Greedy and Random player, were kept.

Other aspects:

There are cases in which our algorithm doesn't do a minimax search to find the optimal action necessary since it requires less strategy. Our first case of this is the situation where there are no tokens on the board. The algorithm randomly throws a token onto the board without entering the minimax function.
Our second case was when there were no player tokens on the board since it has been beaten by the opposing player's token. In this case, the AI throws a token with the token type that has the capacity to beat the opposing player's token.
Our last case is when the opposing player has no token on the board at the end of the turn due to being beaten, the player randomly slides a token on the board.

We also implemented efficient generation of possible moves by only looking at valid actions. This is so that we would not waste time and memory on creating invalid future actions through the minimax algorithm. Furthermore, a throw is only used when there are insufficient token types on the board to beat the opposing player. This was to reduce the chances of our tokens being beaten unnecessarily, or throwing too many tokens. Our most useful case of not using minimax is to directly throw a token on the enemy's token if it is weaker and within throw range.