COMP424 Final Project Report
Bozhong Lu
260683363

Technical approach :
My first approach was to apply the Alpha-Beta pruning algorithm. And I decided to use an offensive strategy to face my opponents.

First of all , I need to generate a game tree for searching (**the minimax method**) . In order to do that , I used the **boardState.getAllLegalMoves**() method to get all the legal moves that can be processed from the current state . And then I used the **boardState.clone**() method to clone the current state of the board . After that , I generated the game tree by processing all the legal moves on the current cloned states with both recursions and for loops . After successfully generating the search tree , I applied the Alpha-Beta pruning algorithm to search through the game tree in order to to keep tracks of the best leaf values for both myself and the opponent player . I initially decided the depth of my search to be three in order to prevent timeout errors . And after all , I returned the optimal scores together with the index of the moves as the result.

Secondly , in order to give each board state a heuristic , I have to define certain ways to evaluate them . This is where the **evaluationFunction method** comes in . After playing the game multiple times against the RandomPlayer , I realized that the central pit of each quadrant plays a very important role in order to form a line of five . So I named all of them goldPits whose coordinates are [**1 , 1**] [**1 , 4**] [**4 , 1**] [**4 , 4**] , and I give those board states occupying these four pits an extra heuristic of 100. Meanwhile , I evaluate each pit by exploring its surrounding pits. The more lines it forms by placing a piece inside , the higher score it get evaluated . Also, a line of three will result in a higher score as compared to a line of two and the same rule applies for all cases which one line is longer than the others.

In order to examine how valuable a board state is after a move has been processed, I created a helper function (**explore**). This function helps evaluating a board state by exploring the surrounding pits of a move in four directions. It takes the x-coordinate , y-coordinate , colour of a piece , boardState and direction as inputs. It ranks a board state by constantly exploring in the same direction until there is a piece that has different colour with the current player. Every time a piece in a certain direction has the same colour as the current player, the score of a board state will be increased by five. Then the recursive call starts to explore the next pit in the same direction. Once a pit of a different colour occurred, the current recursion stops. And the function starts to explore in a new direction off the move.

After completing all the preparation work, my final attempt is to choose the initial move. Assuming our opponents are optimal, we are guaranteed to get an optimal result from our minimax method. So, we will process the move returned from our minimax method. However , if there are errors occurred during our search process such as timeout error, we will process a random move.

Motivation :

Since this is a 2-player, perfect information, deterministic board game, I decided to find a strategy that maximizes utility. Since Alpha-Beta pruning is an algorithm which is both complete and optimal assuming the game tree is finite and our opponent is optimal, I decided to implement it in order to find the maximum utility out of our game. Moreover, compared with minimax algorithm , Alpha-Beta pruning greatly increases efficiency of the program by discarding useless paths without affecting our final result. The above advantages make Alpha-Beta pruning the perfect algorithm for solving our board game.

Pros and cons :

cons: The design of the utility function is not optimal.

The search can not reach the leaf nodes at once . We have to self-decide how deep we would like to search.

The opponents' AI might not be optimal, which can cause us not getting the optimal result.

If the branching factor is too big, we can not directly get our final result.

pros: Useless branches can be eliminated which will result in less search time . Meanwhile, a deeper search can be performed.

Optimal results are guaranteed to be found against an optimal opponent.

Keeps track of the best leaf value of our player (alpha) and the best one for our opponent (beta).

Future Improvement :

Several improvements can be made to the program for better efficiency and accuracy.
First of all , we can use **Monte Carlo Tree Search** to simulate future game states .By applying Monte Carlo Tree Search, we can greatly improve the program's efficiency meanwhile increase performance with number of lines of play. More importantly, with huge branching factor, our search can go much deeper which greatly improves the program's accuracy while moves are always being made in time. Also, we can use Deep Reinforcement Learning to learn how to value moves and board states.

Secondly , utility function can be optimized by continuously experimenting on the program. For example, by experimenting different weights on the goldPits, we might be able to increase our chance of winning. Furthermore, we are able to idealize our **evaluationFunction**() method by adding more important features that affect the chance of winning. For example, we can decide how much weight a certain board state will lose, if multiple pieces of the opponent connected with each other. We can also decide how much weight a certain board state will lose, if the opponent's piece land on one of the goldPits. After experimenting different weights on different features, we'll be able to improve the accuracy on evaluating board states.

Ultimately, reducing the time complexity of the program can let it run more efficiently. Meanwhile, our search can probably go much deeper, which is great gain. The approach to this can be reducing the amount of recursion calls, replacing high-cost functions with low-cost functions.