COMP424 Final Project

Zirui He 260766420

Technical approach:

My first approach is the Alpha-Beta pruning algorithm, which can be turned into an offensive strategy to face my opponent.

First of all, I need to generate a game tree for searching which is involves the **minimax** method. In order to achieve that, I used the **boardState.getAllLegalMoves()** from **PentagoBoardState** 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. Then, 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 keep tracks of the best leaf values for both myself and the opponent player. Here, my player wants to maximize the score while the opponent intends to minimize the score. I initially decided the depth of my search to be three in order to prevent timeout errors. After all recursions, the optimal scores together with the index of the moves in form of an array will be return.

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 **nicePoints** whose coordinates are **[1 , 1] [1 , 4] [4 , 1] [4 , 4]** located in the center of each board, and I give those board states occupying these four pits an extra heuristic of 100. Meanwhile, every pit is explored by its surrounding pits. That means the more lines it forms by placing a piece inside, the higher score it gets evaluated. For example, 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.

On top of that, I created a helper function **explore** in order to examine how valuable a board state is after a move has been processed. This function aims to evaluate 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 with different colour for 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 work above, 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. Therefore, we will process the move returned from our minimax method. However, if there are errors occurred during our search process such as timeout error, the player 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, for minmax model we used in the game, some useless branches increase the complexity of the model. So, to avoid this, Alpha-Beta pruning comes to play so that the computer does not have to look at the entire tree. These unusual nodes make the algorithm slow. Hence by removing these nodes algorithm becomes fast [1]. The above advantages make Alpha- Beta pruning the perfect algorithm for solving our board game.

cons:

- The design of the utility function is not optimal.
- The search cannot reach the leaf nodes in one time. 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 cannot 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) which guaranteed a best score.

Future Improvement:

There are several improvements can be made to the algorithm for better efficiency and accuracy. First, we can use **Monte Carlo Tree Search** to simulate future game states. Essentially, MCTS uses Monte Carlo simulation to accumulate value estimates to guide towards highly rewarding trajectories in the search tree. In other words, MCTS pays more attention to nodes that are more promising, so it avoids having to brute force all possibilities which is impractical to do. At its core, MCTS consists of repeated iterations (ideally infinite, in practice constrained by computing time and resources) of 4 steps: selection, expansion, simulation and backup [2]. By applying Monte Carlo Tree Search, we can greatly improve the program's efficiency meanwhile increase performance with number of lines of play. Also, we can use Deep Reinforcement Learning to learn how to value moves and board states.

Another possible improvement is to optimize the utility function by continuously experimenting on the program. For example, by experimenting different weights on the nicePoints, 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 nicePoints. After experimenting different weights on different features, we'll be able to work out a better solution with higher accuracy.

Ultimately, we can always go for better structure and reduce the time complexity of the program. Meanwhile, our search can probably dig 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.

Reference

[1] Alpha beta pruning in AI - great learning. Retrieved April 13, 2021, from https://www.mygreatlearning.com/blog/alpha-beta-pruning-in-ai/

[2] Wang, B. (2021, January 11). Monte Carlo TREE Search: An introduction. Retrieved April 13, 2021, from https://towardsdatascience.com/monte-carlo-tree-search-an-introduction-503d8c04e168