Group members:
Huicheng Zhuang A99075768
Richard Ying A10954259

CSE150 PA2 REPORT

1.Design Description:

Problem 1:

Just a simple implementation of minimax algorithm. The function takes two parameters, one is “state”, which is just the next state generated by taking an action from the current state and the other is a boolean value “isMax” telling the function if it is handling a max player or min player. Then the function uses recursion to go back and forth between max and min. Once it encounters a terminal state, the utility is calculated by subtracting the stones in opponent’s goal pit from the stones in the player’s goal pit. Then the utility value is returned and the max or min level of recursion will handle the value returned based on how it compares to other values returned by other children. Therefore, we will get best value and best move accordingly once the search is done.

Problem 2:

A modified implementation of the minimax algorithm in problem 1 with appropriate checks and breaks for alpha and beta values. In addition to the self, state, and isMax parameters, there are a and b parameters which compare the bestValue to the next state’s alpha for max and beta for min. If beta is less than alpha in isMax for example, it means the other children are definitely smaller and this is the best move for our player, thus pruning unnecessary traversals. We also implemented a transposition table using a Python dict, storing each unseen state for the player and its associated utility. We also check whether a state has been previously seen, and if so, simply return the utility for that state.

Problem 3:

For this problem, just add an evaluation function using the heuristic function given in the write-up. Each of the attributes is calculated as following ways.

stonesInMyGoal and stonesInOpponentGoal are calculated by getting the elements with player_idx and opponent_idx in state.board

stonesOnYourSide and stonesOnOpponentSide are calculated by summing up values on the corresponding row.

2.Problem 4 description and evaluation:

The custom agent is based off the recommendations in the assignment guideline. We began by modifying the alphabeta algorithm in problem 2 to accommodate a depth limited search instead of a simple depth first search.

The iterative depth limit is set manually by hardcoding. For each depth, a current best move is calculated, and we compare it to the previous best move. If it is better than the previous one, we replace the best move with current best move and this process is repeated until the iterative depth limit is reached. During the iteration, every time the search method is called, we check for “self.is_time_up()”. If it returns true, which means it has reached the time limit, we stop and calculate the expected utility value of current state instead of going any deeper.

For the actual game playing, if we want the player to make quick decision even for a huge game board, we just set the depth to be shallow and it will produce a result within half a second every time. However, there is a tradeoff between speed and winning rate, as we expected. If the depth is too shallow, the custom player will sometimes even lose to a random player for a small game board. And based on our observation, an ideal depth that balances speed and winning rate is around 15, which will take nearly one second to generate a result and still being able to win the game at an acceptable rate.

Neither of us has any experience of playing the mancala game, so we barely have any idea of what a good move would look like, but we still have tried some different evaluation methods. For example, if there is a “1 stone pit” with a “non-1 stone pit” next to it counterclockwisely, then we just let the player pick the move at the non-1 stone pit since it leads to a situation that moving that “1 stone pit” next round will probably capture all of the stones on the opposite side. However, this does not work as good as the given function, and I think it is because making such a move causes an outstandingly low utility for the player playing for the other row and it will definitely eliminate that possibility immediately.

3.Maximum Board Size To Handle:

Minimax:

When M = 2, the minimax player can make a decision in reasonable time, which I think should be no more than 5 seconds, with N up to 9. For N = 9, it takes nearly 5 seconds at the beginning and it becomes faster, as expected, as the game proceeds.

When M = 3, minimax can only handle N up to 2 and once N goes beyond 2, it will take a very long time to calculate a move.

Minimax player is not able to handle a board with M >= 4, even if N = 1.

AlphaBeta:

Still, a reasonable time is considered as 5 seconds.

When M = 2, can handle up to N = 17

When M = 3, can handle up to N = 3

When M =4, can handle only N = 1

Not able to handle M >= 5

Custom:

Based on our implementation of the custom player, it can actually handle any board size since we use an iterative deepening search and we set the depth limit manually. The lower the depth limit, the faster it will be. And the higher the limit, the higher the winning rate.

4.Custom Agent Battle:

The left column represents the depth limit of player0, the upper row is depth limit of player1. Numbers in the rest represents times player0 defeats player1 out of 10 games.

M = 3, N = 4, TIMEUP = -1

	5	10	20
5	**10**	**0**	**0**
10	**10**	**0**	**0**
20	**10**	**0**	**0**

This is apparently not a first-player friendly game setup. It looks like the player playing for the upper row always have advantage. We can see that player0 got beaten by player1 sometimes even it searches deeper.

This happens maybe because that the game is actually complicated as it has 3 pits each row and up to 4 stones each pit. So the actual perfectly rational move may appear in a very deep position, which means that going 20 levels down does not differ a lot from going 10 levels down since they are both predicting a result that is far away from their reach.

6.Member Contribution:

Huicheng Zhuang:

I did problem 1, 3 , part of problem 4 and part of the report. I also took a look at the alpha-beta pruning part, and I think it is much more difficult than just minimax and the evaluation. What I learned from this assignment is that there is always a tradeoff between speed and rationality, so making a perfect agent is extremely hard.

Richard Ying:

I did problem 2 and part of problem 4, as well as part of the report. I helped verify the validity of the other algorithms. I learned the different implementations of game trees and specific differences in the results of changing different parameters in the custom game tree. For example, although a high max depth such as 25 performs better in games, it takes much longer.