CSE150
Program Assignment 2 Report

Yujia Li A98064697

yul200@ucsd.edu

Part 1: Description of the problem and the algorithms used to solve problems 2-4

Problem 2: Alpha-Beta Pruning and Transposition Table

This problem asked for implementing a Alpha-Beta Pruning algorithm based on Minimax algorithm from problem 1. Here is a short version of pseudo-code:

```
Max_value:
        if terminal test is true, return utility
                val <- negative infinity
                for each child in Action do
                        val <- max(val, min_value(child, a, b))
                        if val >= b then return val
                        a <- max(a, val)
        return val
Min_value:
        if terminal test is true, return utility
                val <- positive infinity
                for each child in Action do
                        val <- min(val, max_value(child, a, b))
                        if val <= a then return val
                        b <- min(b, val)
```

Here the max_value part calls the min_value part, and the min_value part calls the max_value part, these two functions recursively call each other to generate values to determine the best next step for the current player. The difference between Alpha-Beta Pruning and Minimax Algorithm is that Alpha-Beta Pruning uses two values, alpha and beta to help minimizing the branches it needs for further search at each iterative calls:

Alpha value is the best backed-up value found so far for MAX;

Beta value is the best backed-up value found so far for the MIN;

With the help of checking with backed-up alpha and beta values, Alpha-Beta Pruning algorithm runs faster than simple Minimax algorithm.

Also, we use transposition table to fastener the running time of the algorithm

further. The idea here is that transposition table is an open list data structure used for storing the utility value of a state that has been explored before, even if it is not a terminal state. So when another branch search for the same state, the transposition table can return its utility value in constant time, which save a lot time. We implement it using hash map.

Problem 3: Evaluation function

The problem 3 asked for a naïve implementation of evaluation function of a state. A more developed version of this evaluation function will be used for problem 4. The evaluation function calculate a value of a state, which represent how much chance the state would lead to a win for the current player. A state with higher value should have higher chance to lead to a win and should be on further exploring. In this problem, we define the value of a state based on the length of the longest streak on the board of the current player, and the return value is just the length divide by k, the win number. We implement it in this way:

First, using a double-for loop run over the board checking for all locations where the current player is on, and then call a count function, start to count the number on its streak;

Second, in the count function, check for eight directions(up, down, left, right, up-left, up-right, down-left, down-right), if the current location is a start of a streak. Only if it qualifies for a start, we count the length of the streak and store it in a array;

Third, order the array which stores all the lengths of streaks on the board, and return the largest number in the array;

Fourth, calculate the value and return back to the evaluation function call.

Problem 4: Create a Custom agent

We are asked to implementing a agent by using Alpha-Beta pruning, transposition table and evaluation function from previous questions. We are asked to implement a iterative deep version of Alpha-Beta pruning, we can improve the evaluation function to make it represent the value of states more accurate, and we can add more methods to make the agent has more chance to win the game with less steps. A detailed explanation and analysis about what we implemented and how we

did it is addressed in Part 2 below.

Part 2: Description and analysis of all the approach used for problem 4

The technics we used to implement Custom agent in problem 4 are iterative deepening Minimax search with Alpha-Beta pruning, transposition table, and move-ordering based on the evaluation function.

1) Adding iterative deepening to Minimax search with Alpha-Beta pruning. It really helps when the board size is large. As the depth becomes larger, the more accurate next step will be returned based on the Minimax search with Alpha-Beta pruning. But with limited time allowed, iterative deepening helps to generate an best next step based on all the searches done, while Minimax search cannot return any results till it searches the whole board with all conditions.
2) Transposition table helps save the a lot running time. The time takes for the Custom agent to play on a board doubles without the using of transposition table.
3) We modified the evaluation function from problem 3 to make it reflect the "value" of each state more accurately. The idea here is not only checking how many of the current player are at the longest streak, but also checking if there is more space for it to continue the streak. The evaluating rule we use here is that, if the opponent has been put at the end of the streak, the streak will not be count, as long as it is still smaller than K value. It helps to cut off the situation, where exist a pretty long streak of the current player on it, but its two ends are taken by its opponent. It will be returned as a good move with a high value from problem 3, but it is actually useless.

We did notice some situations that the Custom agent make irrational decision. For example,

100		120		121
000	→	000	→	000
000		000		000

Here at step 3, we think Player 1 should place on (2,1) instead of (1,3), since there is already a 2 on the left side. But as we test it on the simple Minimax algorithm, it also returns step 3. So we conclude that maybe the place we think is more rational to put the next step and has more winning chance, is actually not the best next step, based on calculation. After all the implementation and modification, as explained before, we think our Custom agent plays rational.

Part 3: Time analysis of agents used in problem 2-4

The speed of a agent depends on several factors: implementation of the algorithm, size of the play board, the max time allowed for thinking, and even how the opponent plays. Here we constraint some of the factors, and see how different agents preforming.

Each of the table below stores the maximum number of empty squares the agents can fill out in a given game with a fixed size table and a fix opponent. We figured that it is impossible to just set up a time limit, say 50 seconds, and let all the agents start to play on a huge board, so they do not run out of empty spaces and end the game. The reason is that as the board becomes large, it takes incredibly long time for Minimax agent to start play, even just one step, but if we give it a timeout value, it may just generate a random step, which will not reflect the real time it takes to calculate its steps.

First Agent	Second Agent	Size of Board(M)	Size of Board(N)	Timeout	Number(K)	Steps Played	Real Time
Minimax	Random	3	3	-1	3	3	10.521
AlphaBeta	Random	3	3	-1	3	3	0.239
Custom	Random	3	3	-1	3	3	3.225

Table 1

First Agent	Second Agent	Size of Board(M)	Size of Board(N)	Timeout	Number(K)	Steps Played	Real Time
Minimax	Random	4	3	3	3	3	8.386
AlphaBeta	Random	4	3	-1	3	3	3.604

Custom	Random	4	3	-1	3	3	3.528

Table 2

First Agent	Second Agent	Size of Board(M)	Size of Board(N)	Timeout	Number(K)	Steps Played	Real Time
Minimax	Random	4	4	10	4	3	44.105
AlphaBeta	Random	4	4	10	4	4	19.285
Custom	Random	4	4	-1	4	5	9.202

Table 3

First Agent	Second Agent	Size of Board(M)	Size of Board(N)	Timeout	Number(K)	Steps Played	Real Time
Minimax	Random	5	5	10	5	5	55.124
AlphaBeta	Random	5	5	10	5	5	55.286
Custom	Random	5	5	-1	5	5	7.749

Table 4

Note:

1. For Minimax-Random and AlphaBeta-Random, there are some random moved played by Minimax agent and AlphaBeta agent due to exceeding max time limit.

2. We set the maximum time (function feel_like_thinking) of CustomPlayer to be 1 second. This constrains the max time for each move of CustomPlayer to be less then 1 second.

As the size of the board becomes bigger, the real time Minimax player takes to calculate one step becomes incredibly long, so we stop testing any larger board here. Based on the data we have so far, we can construct a more concise table showing the approximately maximum number of empty squares the agents can fill out in a given amount of time, with a fix opponent.

Agent	Time	Approximately steps
Minimax	10.5	5
AlphaBeta	10.5	132.063

Custom	10.5	9.787

Table 6

Note: Table 6 is constructed based on Table 1. As we fix the time to be about 10 second, and we just multiply the steps AlphaBeta and Custom agents might get up to.

Agent	Time	Approximately steps
Minimax	55	5
AlphaBeta	55	11.407
Custom	55	34.375

Table 7

Note: Table 7 is constructed based on Table 3 and Table 4. As we fix the time to be about 55 second, and we just multiply the steps AlphaBeta and Custom agents might get up to.

It is not hard to see from our data that:

1) Minimax agent play the slowest;

2) With a small board, like 3 by 3, AlphaBeta agent can play really fast, but as the board become bigger, its speed drop down dramatic, though it is still faster than Minimax agent, it takes a good amount of time to make an action.

3) The speed of Custom agent is consistent, since we set our own maximum time inside the algorithm implementation to be one second. Custom agent could be slower than AlphaBeta agent in small board, but as the size of boards become larger, Custom agent plays faster.

Part 4: Relationship between win/lose condition and depth limits of Custom agent

There are several facts controls the winning of Custom agents, one of them is the max depth it can search. Different max depth would return different evaluations, which making differences of the next step. The table below shows the win, lost or tie conditions of different Custom agents play against each other on different size boards.

Index	First Agent Depth	Second Agent Depth	Size of Board(M)	Size of Board(N)	Timeout	Number(K)	Steps Played	Result
1	5	15	6	6	10	6	36	Tie
2	20	25	6	6	10	4	11	Second
3	5	15	6	6	10	4	11	Second
4	5	25	6	6	15	4	11	Second
5	2	25	7	7	15	5	49	Second
6	2	25	6	6	15	4	18	Tie
7	20	25	7	7	20	5	49	Tie
8	20	25	8	8	6	20	64	Tie
9	2	25	8	8	6	20	64	Tie
10	30	30	8	8	6	30	64	Tie
11	20	25	9	9	20	6	51	Second

Table 8

Based on our results from Table 8, we have several finding:

1) If the win number K is large, like equaling to the row or column number of the board, it is really hard for the two players to break up a tie to win the game. On the other hand, with low K value, the player with higher maximum depth has better chance to win, like test 11 from Table 8.
2) If the two players have about the same maximum depth, it is easy for them to end up in a tie, like test 6 and 7 from Table 8.
3) As the board size become larger and larger, if we keep the maximum depth of both players the same, it becomes harder for the players to break a tie, like test 2, 7 and 8 from Table 8.
4) Generally, it is hard for the two players to break a tie since they both use the same algorithm and optimization.