CSE150 PA2 Write Up

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

Problem 2:

Problem description: Description of the problem we're solving? -1

We need to implement a minmax search with alpha-beta pruning and a transposition table in order to let the agent to play the games on a larger board faster than the agent which implements only a minimax search.

Algorithm description:

We implement the transposition table as a dictionary which has the state of the board(represented as a string) as the **key** and its corresponding utility as the **value**. We implement two methods: "add_transposition(state,utility)" and "is_in_table(state)".

"add_transposition(state,utility)": This method will add the visited state to the transposition table.

"is_in_table(state)": This method will check whether the new state is in the transposition table or not and return a boolean to indicate the result.

We implement the minmax search with alpha-beta pruning as the two methods, "maxValue(state,alpha,beta)" and "minValue(state,alpha,beta)". In each of the two methods, it will call the other one in order to get the value of the correct utility for each node belongs to the max or min player. After getting the value of the utility back from the method, it will compare the utility with alpha or beta to determine whether to prune or not.

Problem 3:

Problem description:

We need to implement the state evaluation function for the EvaluationPlayer agent for the agent to play the next move based on the best result returned from the function. The function will return the length of the longest streak on the board divided by K.

Algorithm description:

We implement four methos: "checkRow(index)", "checkColumn(index)", "checkDiag(index)", "checkReverseDiag(index)" to calculate the longest length on rows, columns, and the two diagonals. Particularly, we first find all the stones in a color on the board. Then check each of them in the four directions. And then we calculate the maximum among the results of the result those four methods, which is supposed to be the longest streak on the board. And then we divide it by k to get the final return value for the state evaluation function.

Problem 4:

Problem description:

We need to implement an agent as clever as possible and will be able to play m,n,k game up to m = 19, n = 19, and K = 6 in a limited amount of time.

Algorithm description:

Based on problem 2 and problem 3. I implement the minmax search with alpha-beta pruning. We have a heuristics function to evaluate the utility of the middle node. Good! For the first move, I take the move at the center of the board. Then, I use the transposition table and the heuristics function to return the desirability of the middle state. Then we use the alpha-beta pruning to get the valid value and the desire move. The basic idea is like above. We make some improvement to our agent. We will discuss them in detail in the next question.

2. Describe the approach you used in Problem 4 and other approaches you tried / considered, if any. Which techniques were the most effective?

The basic idea is in Problem 1. Here, I will focus on how we improve it.

1. **The start move:**
   If we are the one who take the first move, we make it to put the stone on the center of the board. Since the center of the board is the best location.

2. **The modification of the evaluation function.**
   For the evaluation function, I return the steak only when the steak has to potential to achieve K-th connected. For example, for a board:
   ```
   State(4,(
   (0,2,1,0),
   (0,1,2,0),
   (0,0,0,0),
   (0,0,0,0)
   ), last_action=Action(2, (0,1)))
   ```

   I will count it as steak of one, not steak of two. Although there two stones in a line, it is obviously that the direction it is taken can never achieve a steak of 4 to win the game. However, the stone on the position (1,1) has a potential to achieve a line of 4 if we get the position (0,0), (2,2) and (3,3). So I count it as steak of 1.

   The algorithm: We search the whole board 4 times. First time, we focus on the row checking; then we check column; last the two diagonals. Since it always go for one direction, it will not check the same stone for the same direction more than once. Also, for each check, when we meet self.color, we increase both "count" and "potential". The blank, 0, will only increase "potential". The opponent's color is block and will set both "count" and "potential" to 0. We will only document the count when it has a potential greater or equal to K. Meanwhile, every time we find a max value, we will put it as a "limit" to the next find. It will help us rule out part of the board that is impossible to have a line longer than "limit". For example, for the following board:
   ```
   (0,1,x,x),
   (0,1,0,x),
   (x,0,0,0),
   (x,x,0,0)
   ```

   When we check the column, we already get a steak of 2. So it will be a limit when we check the diagonal from top-left to bottom-right. So, the place where I marked 'X' will not be visited in the check, for they at most can have a steak of 2. However, we are looking for the steak that has a length more than 2. It will help us save some time.

3. **The preference of diagonals and unblocked steak in evaluation function.**
   In the evaluation function, we set a preference to diagonals.
   When we are checking rows and columns, we count each stone as 1; when we checking the diagonals, we set each stone has a value of 1.1. So we are likely to move in diagonals.

   In the evaluation function, we set a preference to unblocked steak.
   It means, whenever we check a steak, we also check its previous location and its next location. If one of them is a free block (0), we will add value of 0.5 to the steak.
   So If a steak is blocked on one sided, it will earn 0.25 value; If it is free on both side, it will earn a value of 0.5, and we are likely to take this move. If it is blocked on both side and still manage to achieve a steak of K, it will remain its original value.

4. **Use the evaluation function to implement heuristics function:**
   To sum up, the evaluation function has the following property:
   a) It will ignore the steak that has no potential to achieve a steak of K and win the game.
   b) It prefer move in diagonal
   c) It prefer the steak that is free on both side

   Our heuristics function is quit simple. It is like:
   H(state) = E(state in color of agent) – 0.5*E (state in color of opponent)

   To get the final desirability of a state, we will use the result of the evaluation function of the opponent to counter the evaluation function of the agent. For example:
   K = 5
   (0,0,0,0,0,0),
   (0,0,1,1,0,0),
   (0,0,2,0,0,0),
   (0,0,0,2,0,0)
   (0,0,0,0,0,0)
   Assume the agent is 1 and is going to take the next move. It can take two possible move according to E(state in color of agent) (both of them are (3 + 0.25 + 0.25)/5 = 3.5/5=0.7):

(1)	(2)
(0,0,0,0,0,0),	(0,0,0,0,0,0),
(0,0,1,1,1,0),	(0,1,1,1,0,0)
(0,0,2,0,0,0),	(0,0,2,0,0,0)
(0,0,0,2,0,0)	(0,0,0,2,0,0)
(0,0,0,0,0,0)	(0,0,0,0,0,0)

However, the E (state in color of opponent) is different for two stages.
For the (1), it is (1.1+1.1+0.25+0.25)/5 = 0.54
For the (2), it is (1+0.25+0.25)/5 = 0.3. Because it cannot achieve a steak of 5 in diagonal at all! The only possible way to get a steak of 5 is to moving in rows, and it only has one stone in the row.
So If we apply the function:
   H(state) = E(state in color of agent) – 0.5*E (state in color of opponent)
For (1), we get value of 0.7 – 0.54/2 = 0.43
For (2), we get value of 0.7 – 0.3/2 = 0.55
So, our agent will know that (2) is a better move. And in fact, it is.

5. **If the next move is a WIN move, return it.**
   It is not hard to implement and easy to understand. Basically I have a Flag to indicate it is the win move so I can return it without further searching.

6. **If the time is out, return the current best move.**
   It is not hard to implement and easy to understand. Same, I have a Flag to indicate the time out.

7. **We will not consider the move that is TOO FAR!!!!**
   Consider a 19*19 board and your opponent has just finish his first move. Normally, we have to expand the remained (19*19 – 1) states. But it is a waste of time!
   Whenever our agent gets a move, it will check whether there exists a stone in a 3*3 square with the center of the current possible move. If there is no stone in the 3*3 square, our agent will simply give up expanding the move.
   As a result, no matter how big the board is, out agent will only expand the move that is next to the existing stones. It will save us a lot of time of expanding useless moves.

8. **We will predict and block the opponent when we think it is necessary.**
   Each time we take a move, we will pay a lot attention to the opponent's next move. So, it is the same as the depth-2 of the DFS tree. Whenever we detect that the opponent's second move, we will get the E (state in color of opponent). If this value is larger than the block limit that we set for this game according to the M, N and K. This is an Alert Move and we will set up an Alert Flag. When we have more than one Alert Moves, we will keep the one with the highest value of E(state in color of opponent), since it is likely to be the most dangerous move. After finishing searching the 2nd depth of the tree, if the Alert flag is On, Then we will get that Alert move back and take this move before the opponent to block it.

Great work here!

Above all, I think the 7, 8, and the evaluation function is effective.

3. Evaluate qualitatively how your custom agent plays. Did you notice any situations where they make seemingly irrational decisions? What could you do/what did you do to improve the performance in these situations?

First, when we run the program at the beginning, we sometimes find the agent will put the stone on the line that is already block by the opponent and is impossible to line in K stones. To improve it, we add the variabel "potential" in the heuristics function. We will consider a move is valid only when it has a potential to achieve K stones in a line.

Second, our agent sometimes will ignore the fact that the opponent is going to win and fails to block the opponent. To fix it, we set a flag to indicate whether to block opponent. When blocking, we put the opponent's good move (calculated by the heuristics function) in the Alert list and take that move before the opponent to block it.

4.What is the maximum number of empty squares on the board for which the minimax agent can play in a reasonable amount of time? What about the alpha-beta agent and your custom agent, in the same amount of time?

Assume that the reasonable amount of time is 10 seconds.

For Minimax,The maximum number of empty squares is 6(with K = 2 and a 2 X 3 board)

For Alpha Beta, The maximum number of empty squares is 16(with K = 3 and a 4 X 4 board)

5.Create multiple copies of your custom agents with different depth limits. (You can do this by copying the p4 custom player.py file to other * player.py and changing the class names inside.) Make them play against each other in at least 10 games on a game with K > 3. Report the number of wins, losses and ties in a table. Discuss your finding.

Game index	*Board size*	*K*	*Time limit*	*Winner (3 competitions)*
Game 1	*4 * 4*	*4*	*3s*	*Draw*
Game 2	*5 * 5*	*5*	*3s*	*Draw*
Game 3	*7*6*	*5*	*5s*	*Draw*
Game 4	*8*8*	*6*	*5s*	*Draw*
Game 5	*8*8*	*8*	*7s*	*Draw*
Game 6	*10*9*	*7*	*10s*	*Draw*
Game 7	*13*15*	*10*	*10s*	*Draw*
Game 8	*14*17*	*10*	*15s*	*Draw*
Game 9	*17*19*	*14*	*15s*	*Draw*
Game 10	*19*19*	*5*	*20s*	*Draw*

I find that it is always a draw. Since I am using the exactly the same function for both agent, they can exactly predict what action will the other agent takes. As a result, they can always block each other and result in the tie.

Great!

6.A paragraph from each author stating what their contribution was and what they learned.

Author 1: Xueyang Li
Login: xul008@ucsd.edu

I mainly implemented problem 1 and part of problem 2. And I also helped implement problem 4. After I am done with implementing problem 1 and 2, I understood the minimax search algorithm and alpha-beta pruning much better. And also I learned a new way to write recursive algorithm that it can be written as two functions and they will call each other until they hit the base case. Also I got more familiar with python as well. For example converting the state of the board to a string and storing it in the transposition table using only this one line of python code:

“transposition_table[''.join(str(i) for i in state.board)] = utility”

Author 2: Zhengkun Tan
Login: z2tan@ucsd.edu

I focus on optimizing and improving the problem 4. Through optimizing the problem4, I have a better idea of how to design and calculate the evaluation function. How to rule out the useless move. In addition, I participate in the test of the code. While testing, I find some corner case and fix them in my agent. Besides, I realize that implementing a good agent is not easy. Meanwhile, I find I really join programming with python. I have a better understanding of python.

Author: Qianran Fu
Login: qfu@ucsd.edu

The main contribution that I made was constructing the code for Problem 2,3. The alpha beta pruning with transposition table was a little hard to understand at first. However, after went through some pseudocode online, I started to get what the algorithm is about. Basically, the first part of the code was the transposition table,which has two method add and check. Those two methods ensures that if the state was checked before, then it wound't be checked again and if not it will be added into the transposition table. The idea is kind of like the visited and unvisited node. The second part is the alpha-beta algorithm which is partly based on the pseudocode given in the lecture. The third part is to initialize a maximizer and start the algorithm from it

Great report, nice job on your custom agent!

59/60