Write Up for project 1

**Introduction**

In this write up, I’m going to explain our group’s design for basic tic tac toe and advanced tic tac toe and do some analysis of the performance. Since our group members are not graduate students, we didn’t implement ultimate tic tac toe.

**Part1: basic tic tac toe**

Our design for basic tic tac toe follows mostly the state space search paradigm, but we didn’t follow strictly the code frame presented in class and in the book. The structure and mechanism are the following.

**Class “State”:**

State represents any state in the gaming process, Also, it’s the node for the search process while the game proceeds. There are three most notable features in this class. One of them is an integer array “arrangement”, which represents the status of each position on the board. The number 1 in a position in this array represents an “X” in the corresponding position on the board (index less by 1 of course), and 0 represents “O”, -1 represents empty. The second is an ArrayList of State, which is the ArrayList of its successors. The third is the value “minimax” which is the utility value of this state and will be determined once the minimax method is called in the game class.

the first construction is for the initial state and the second is like the transition model discussed in class which takes an action and return a new state.

Additionally, the method “emptygrid”, which returns the arraylist of available moves, is like the “Action” function described in class that gets all the applicable actions from this state. Functions of other variables will be discussed later.

Other variables: “turn” indicates whether this is a max state or min state, “XorO” indicates whether next player to play is X or O, lastAction record the action that led to this state. These are useful when we implement “Utility” method and their usages are obvious

**Class “TTT”**:

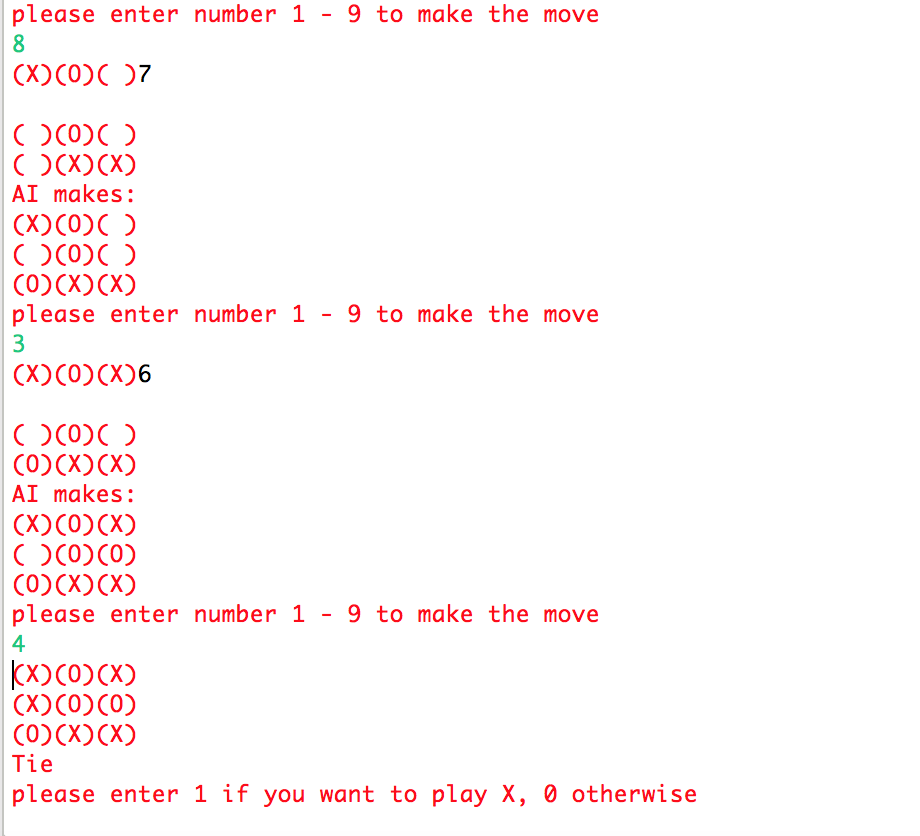
This class contains all the functions that are needed by adversarial search and state space search which I will discuss one by one.

First is the alpha-beta search, using the method “absearch”, “MaxValue” and “MinValue”. We modified the pseudo-code from AIMA pg.170 so they look very similar. We start by calling absearch, if the start state is a min state we call MinValue and vice versa. A notable detail is that in methods MaxValue and MinValue, we assign the minimax value of current state and it’s stored in the “children” list of the previous state, so the entire game tree will have been built after the initial state is processed by “absearch”.

Method “Ttest”, “getWinner”: “Ttest” method implements terminal test which is need by minimax algorithm. if there is a winner, then it returns true, or if there is no empty grid on the board, which means a tie, then it also returns true, else false. For the getWinner, we just check each possible line and see if there is a line occupied by the same symbol.

Method “Start” implements the gaming process which follows state-space search paradigm. Once user decides to play “X” or “O”, we initialize the game tree. Once the user makes a move(action), we check if it’s legal and update the current state by finding the corresponding successor in “children” list in that state. Then let the agent search for the best applicable action by looking for the successor of the current state that has the best utility value, update the current state to that successor and make the move. Repeat this process until the end of the game.

Notice that when we initialize the game tree, we also prune some branches for the Min states assuming that Min player will always play optimally. But obviously human players don’t always make the optimal decision. When we test our program, sometimes we go into a less optimal state whose successors hasn’t been created (pruned). Thus, we add a variable “complete” to the state class which would be set to 1 if all of its successors have been looked at. If a state is not complete, we run the absearch on this state again. This problem led to the only major bug in this part of the program. Other things just worked after they are coded.



(Instance of a tie)

**Performance:**

Agent always win or tie. To be more scientific and precise, and more efficient, we could have counted the number of nodes the agent has visited. But later we notice the agent always response instantly, we decide to save ourselves from those efforts.

**Part 2 Advanced tic tac toe**

The structure for Advanced tic tac toe is very similar to basic tic tac toe. So I will only point out the difference.

**Class SuState**

Arrangement array for advanced tic tac toe is a 2-D array whose first index indicates the index of mini board and second indicates the content in the corresponding position on that board.

getThisArrang(): get the arrangement of the mini board that has just been played, which is used for heuristic calculation and terminal test.

availMove(): get the available moves from this state. The detail of implementation is presented in the comment above it.

**Class SuperTTT**

Notice: “SuperTTT” is not the “super tic tac toe” described in the prompt, this is the advanced tic tac toe.

We implemented the heuristic-depth-limited-alpha-beta-pruning-minimax algorithm in this class. We set the cut-off test in all methods related to a-b-pruning search (MaxValue, MinValue). At the beginning of the search we pass in depth 0 as parameter and increment it by one at each level and do the heuristic evaluation at depth 7.

**Heuristic function:**

We implemented heuristic function by only looking at the arrangement of the board that has just been played in a state. Because the features of this board directly decide how the player from last state should play. The high-level idea is that we look at every three grids in a row (vertical, horizontal and diagonal) and assign value according to whether the arrangement of this row is good or bad for the agent, and how good or bad it is.

We consider the combinations of X, O and empty (denoted E) in every three grids in a row, consider agent play O:

1: there are three Xs. This is the worse because X already won, so the value assigned is -20000;

2: there are two Xs and an empty space: this is also bad because X is one step away from wining, so the value is -200

3: there is one X and two empty space: not as bad but still bad, because X still has chance, so the value assigned is -5

4: there is two Xs and one O: not bad at all at the first glance, because X has no chance to win in this row, but 2 Xs still give other rows chances, so the value is -1

switch the X and O for above 4 conditions, we assign absolute value of those values.

If the agent plays X, just get the additive inverse of the values above.

Add the values of all rows, columns and diagonals, we get the total heuristic value of this state.

There are three methods contribute to the heuristic function: “Heuristic”, “hHelp” and “mapping”. We notice that if we let the empty grid denoted by -4, adding three numbers of a row we can get distinct number for distinct combination, so the mapping function is to map a number to a value, the hHelp function is to get the value of a single board and the Heuristic function is to sum up the value of all nine boards

Things to be noticed in game playing process: although we search to the depth 7, when we go to depth 5 we search again, so that the minimax value can be more precise. The reason for us to choose depth 7 is that depth 8 is too slow. The reason for us to choose depth 5 to research is that we like this number.

**Performance:**

Our program with this heuristic function beats this online AI: <http://ultimatetictactoe.creativitygames.net> almost every time, and this one: <http://www.stratigery.com/gen9.html> with up to difficult level 6. However, we can’t beat one of my friends.

Later, inspired by few words from my friends, I added the heuristic value of all 9 mini-boards instead of the one that has just been played, and the performance was highly increased.

Speed and space: our program responses relatively fast. The branching factor is at most 9 and depth is 7. If my calculation is right, the time and space complexity is O(9^7) each time we renew the tree.

**Conclusion:**

Both of our two parts perform very well. If there is a tournament for advanced TTT, we are definitely going to take part in. A lot of technical details about coding are considered literal and understandable by me so I didn't explain in this write up, but feel free to question us