CSC 242 Project 1 --- Tic-Tac-Toe

Hecheng Sun

I. Introduction

For this project, we need to use the knowledge of state-space search from chapter 3 and adversarial search from Chapter 5 to design two basic programs. The first one can play the standard tic-tac-toe with a human player and the second one can play an extended version of tic-tac-toe, called "9-Board Tic-Tac-Toe". The two programs I designed have very similar data structures and algorithms. In fact, the second one is just a modified version of the first one so it can handle the more complicated rules and game trees. For each program, I will talk about my design process, my design logic and my self-evaluation of the final performance.

II. Standard Tic-Tac-Toe

I started my coding by something simple --- a working human vs human standard tic-tac-toe program. I chose a 2D array of integers to represent the state of the game and used "system.err.println" to roughly draw out the looking of the game board.

```
public void draw(){
    System.err.println(" "+squareConvert(state,0,0) + " | "+squareConvert(state,0,1)+" | "+squareConvert(state,0,2));
    System.err.println("-----------");
    System.err.println(" "+squareConvert(state,1,0) + " | "+squareConvert(state,1,1)+" | "+squareConvert(state,1,2));
    System.err.println("-----------");
    System.err.println(" "+squareConvert(state,2,0) + " | "+squareConvert(state,2,1)+" | "+squareConvert(state,2,2));
}
```

Two players take turn by an int value change from +1 to -1 or from -1 to positive 1. For each choice, I wrote a squareConvert method to decide which square to update

with “X” or “O”. Then, the basic I/O was done in a short amount of time, including all the illegal input handlings, plus an endgame Boolean value to decide when to quit the loop and end the game. After that, I wrote a simple transition method to translate the 1-9 moves to the correct position in the 2D state array and update it with the move. At that point, my program could update the “X” and “O” on the board but had no way to tell who has won. Therefore, I wrote a terminal test method to check who has three moves in the same row, column, diagonal or reverse diagonal. Now, my program can work for two humans and know who won or draw at the end of a round. Next, for a AI vs human program, I need to let the AI know which square to choose for each ply. According to the textbook, I wrote an action method that returns all the applicable action the AI can do at any point and finally implementing the alpha beta search algorithm to trace the game tree. At first, I tried to modify the pseudocode from the textbook page 170, but I realized that what we need to return here is not only a utility value, but a move. So, I tried several ways to do this and finally found a way to use an array of two integers to carry around both the utility and the move that the AI need to do at the end of the search. Because the standard Tic-Tac-Toe only have a maximum of 3^9 = 19683 possible states (empty, belong to X, belong to O for each of the 9 squares), we don’t need a heuristic function here. However, the 9-Board version must have it, and it became the most important modification I need to make for part 2.

```
X or O?
x
 1 | 2 | 3
-----------
 4 | 5 | 6
-----------
 7 | 8 | 9
It's your turn: 5
It's AI's turn

 O | 2 | 3
-----------
 4 | X | 6
-----------
 7 | 8 | 9
It's your turn: 4
It's AI's turn

 O | 2 | 3
-----------
 X | X | O
-----------
 7 | 8 | 9
```

III. 9-Board Tic-Tac-Toe

Most of the code in my 9-Board version is similar and pretty much identical with the standard version. The focus of this part is to handle the heuristic and some extra parameters and a few more loops. At first, I needed to change the illustration of the 9-Board. Now, a 3D array is needed to represent states, with an extra parameter to be the location 1 to 9. Initially, I just used a 9*3*3 3D array like it should be, but then during the process of my modification I realized it was a nightmare to handle the input of 1-9 but must store them into an array with index 0-8. This discrepancy caused me lots of glitch and pain in programming. And, in the end, I decided to use

a 10*3*3 3D array and disregard of the zero index and use it from index 1 to index 9 to match the input. Then, with some little tweaks of I/O and the terminal test method, the program became “playable”, but the AI couldn’t choose the right one because I need to change the alpha beta search algorithm as well.

As we know, the 9-Board version has about 3^ (9*9) ≈ 4.43 * 10^38 possible moves, which would take forever it we just used the normal minimax algorithm. Therefore, I need to come up with a heuristic algorithm to estimate a score for each player at any time during a round. I had no idea how to do this so I went to a TA session and got my answer there. Starting with leaves, if we get a winning case, we return infinity with the appropriate sign. If it’s not a leaf, a simple but effective way to estimate the chance of winning is to count the possible wining rows, columns, diagonals and reverse-diagonals. The basic idea is this: if there is one square of a row/column/diagonal/reverse-diagonal belongs to the enemy, you won’t win your game by this line. Therefore, just ignore it. If there is no enemy territory in this line, then you are good. If you have one territory in this line, you get 10 points. If you have two territories in this line, you get 100 points. Adding up all three rows and all three columns plus two diagonals for one 3*3 location, with a loop that takes from location 1 to location 9, we then will get an overall heuristic score for one of the two characters. We do it again, now we get the score of the other character. What we need to do is to find the difference between these two scores and apply the appropriate positive or negative sign to it, depending on we are on a human level or

an AI level in the game tree.

```
X or O?
x
 1 | 2 | 3   1 | 2 | 3   1 | 2 | 3
-----------  -----------  -----------
 4 | 5 | 6   4 | 5 | 6   4 | 5 | 6
-----------  -----------  -----------
 7 | 8 | 9   7 | 8 | 9   7 | 8 | 9

 1 | 2 | 3   1 | 2 | 3   1 | 2 | 3
-----------  -----------  -----------
 4 | 5 | 6   4 | 5 | 6   4 | 5 | 6
-----------  -----------  -----------
 7 | 8 | 9   7 | 8 | 9   7 | 8 | 9

 1 | 2 | 3   1 | 2 | 3   1 | 2 | 3
-----------  -----------  -----------
 4 | 5 | 6   4 | 5 | 6   4 | 5 | 6
-----------  -----------  -----------
 7 | 8 | 9   7 | 8 | 9   7 | 8 | 9
Location Number (1-9): 1
It's your turn at 1: 5
It's AI's turn

 1 | 2 | 3   1 | 2 | 3   1 | 2 | 3
-----------  -----------  -----------
 4 | X | 6   4 | 5 | 6   4 | 5 | 6
-----------  -----------  -----------
 7 | 8 | 9   7 | 8 | 9   7 | 8 | 9

 1 | 2 | 3   1 | 2 | 0   1 | 2 | 3
-----------  -----------  -----------
 4 | 5 | 6   4 | 5 | 6   4 | 5 | 6
-----------  -----------  -----------
 7 | 8 | 9   7 | 8 | 9   7 | 8 | 9

 1 | 2 | 3   1 | 2 | 3   1 | 2 | 3
-----------  -----------  -----------
 4 | 5 | 6   4 | 5 | 6   4 | 5 | 6
-----------  -----------  -----------
```

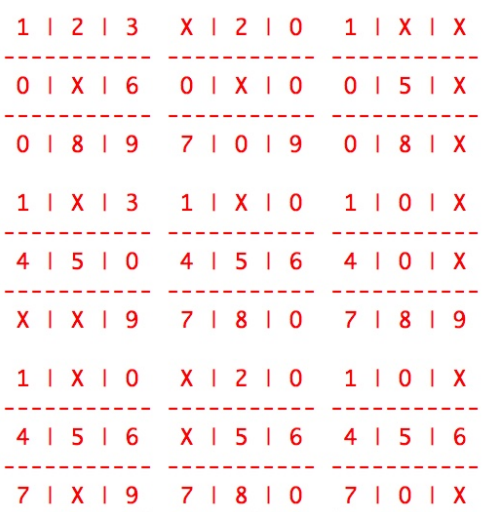
```
public int heuristic(int who){
    int out = 0;
    for(int boards = 1; boards <= 9; boards++){
        for(int r = 0; r <= 2; r++){//all three rows
            boolean worth = true;
            int score = 0;
            for(int j = 0; j <= 2; j++){
                if(state[boards][r][j] != who && state[boards][r][j]
                    worth = false;
                }
                if(state[boards][r][j] == who){
                    score++;
                }
            }if(worth)
                out = (int) (out + Math.pow(10, score));
        }
```

IV. Evaluation

The results are good. My standard version program can do at least a draw, and if human made a bad move, the program will take advantage of it and possibly leads to a victory if the move is bad enough. For the 9-Board version, since testing is much harder and more time-consuming than the standard version, I can't test all the possibilities. I managed to beat it 2 or 3 times during my testing and refinements,

but if my fixes work as I intended, it now should be able to beat any normal human player. The fact is, for the at least 15 times of testing (about 15 minutes per round…) to my final build, I never have won once. I would say this is a pretty solid achievement, considering how many possible states the game have. Without some 9-Board Tic-Tac-Toe mater helping me to test it, I can't guaranty this program is unbeatable, but if you do, you must be really good at 9-Board Tic-Tac-Toe or encountered some glitch I didn't find. I used various tactics and tried to avoid any stupid mistakes during the long and boring testing process, but I always would be "checkmated" by the AI (I define checkmate in the 9-Board game as directing your opponent to a location that all the choices your opponent can make in that location leads to a winning location for you).

If there is something I can do if I have more time, it would be to improve the heuristic algorithm. This heuristic algorithm definitely is not the best. Counting the lines that could lead to a victory is good idea for rough estimation, but it cannot differentiate different situations that have the same score but different effect. For example, for two lines that both have two of your territories, one with the missing piece that have the value leads to another location for you victory can make you opponent not be able to "block" this line, because it will make you to complete the line in the other victory location and win the game. While the one can't lead to another winning location is less effective against your opponent.



Disregard about X has won this game. You can see, for O, location 3,5,8 are more effective than location 1 even though they all have the same score of 100 for their lines. This is because if X tries to block O in location 5, it will make human to be able to make a move in location 6, which the human will choose 8 in location 6, leading to a loss for AI. However, in location 1, if X choose 1 in it, X will not only be able to block a 2-in-a-line of O but even get a 2-in-a-line for itself.