4x4x4 TicTacToe with AI

CS640 Programming Assignment 3

Georgios Karantonis

Jingzhou Xue

Yernur Alimkhanov

11.26.2019

1. Problem Definition

Tic Tac Toe is a game played, usually, by two players in a shared board in which each player takes turns trying to beat their opponent. Each player has a specific mark, either X or O and the goal is to fill any line of the board with their respective mark; the most common variation of Tic Tac Toe is the one with a 3x3 board, but for the purposes of this assignment, we used a board of size 4x4x4. In this variation, the winner is the player who manages to fill first 4 consecutive cells with his/her mark, where the lines may be vertical, horizontal or diagonal. For the implementation of the game we created two separate AI agents, which act as the competing players and in order to model the game and the players' strategies, we implemented the minimax algorithm with alpha-beta pruning along with a heuristic rule of our own. In the following sections we present our detailed method and as well as the results it yields.

2. Method and Implementation

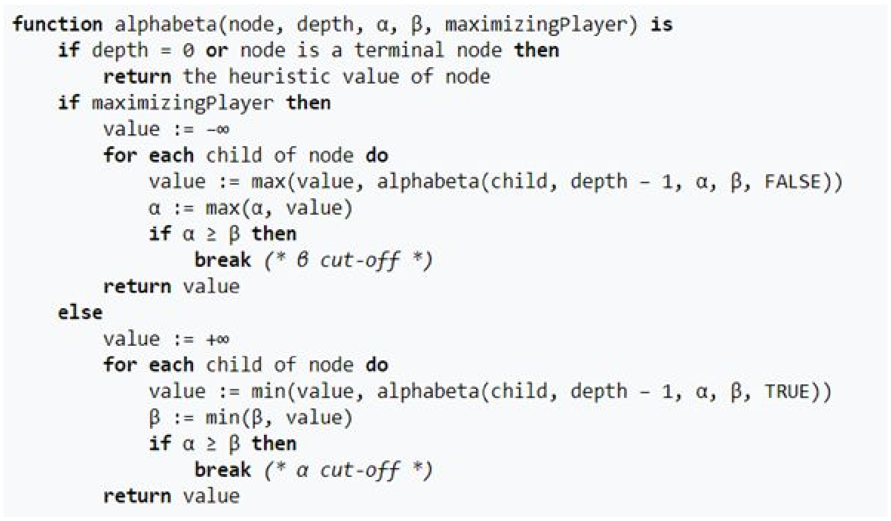
As mentioned, the methods we utilized to model the game and the players' strategies are the minimax algorithm with alpha-beta pruning as well as our own scoring function. More specifically:

Minimax with alpha and beta pruning

Minimax is a decision-based rule for minimizing the possibility to lose; in our case minimax is used to identify best possible move for each player in their respective turn. Minimax is represented by a tree, each level of which performs a different action; either maximization or minimization of the nodes' scores, where the nodes of the tree represent all the possible choices for the player's next move as well as their potential outcomes. The heuristic evaluator is used to assign values to the nodes of the tree,while the minimax method analyzes the nodes' values and finds the best possible move. In other words minimax tries to make a decision for the next move of each player by predicting how this move is going to play out in the future.

The minimax algorithm is a recursive method that goes through the tree in a Depth-First-Search manner. Taking into account the fact that DFS is a greedy algorithm but also the fact that the search space can be very big, we can see that this procedure can prove to be extremely time consuming. It is obvious that in order for the computations to be feasible we have to set a threshold for how far in the future we will allow the agents to look; this threshold is defined by the depth hyperparameter. Also, in order to optimize the algorithm even more we utilize the alpha-beta pruning method, which seeks to decrease the number of nodes that are evaluated by minimax. This algorithm finds the best possible node to expand, by selecting the best move based on the outcomes of the nodes in the predefined depth, and prunes all the rest. It is worth mentioning that even this algorithm is a very good choice to increase the efficiency of our system, it may be affected by the horizon effect meaning that the best choice in the near future does not necessarily lead to the best outcome; on the contrary it may lead to disaster.

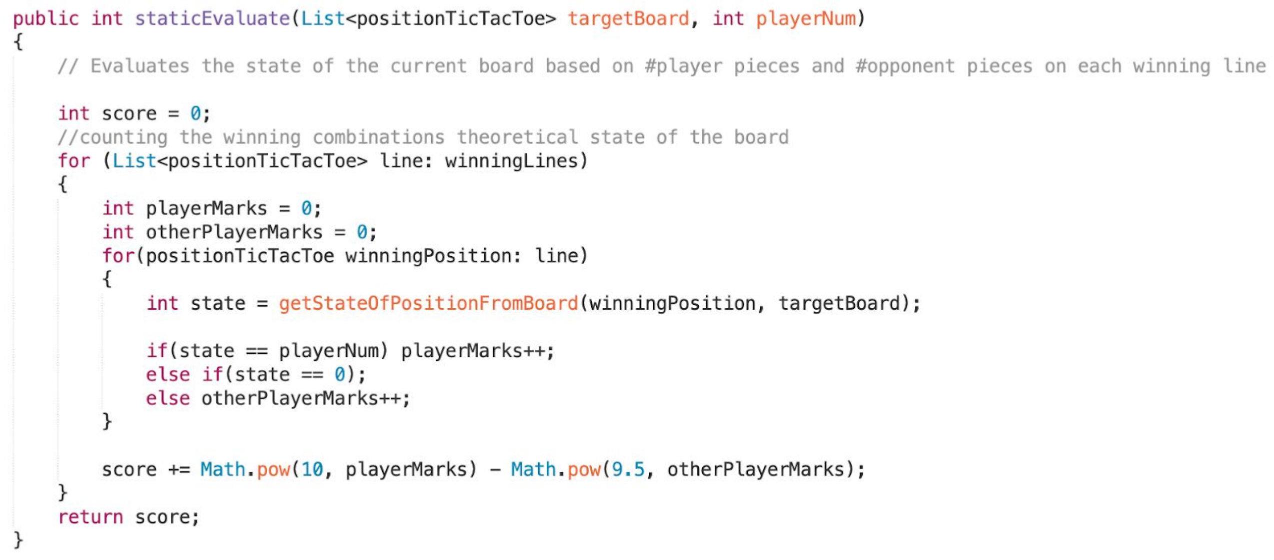
Below we present the pseudo code of minimax with alpha beta pruning:



```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cut-off *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if α ≥ β then
                break (* α cut-off *)
        return value
```

Heuristic Evaluator

The heuristic evaluator is a scoring function that assigns a score to a board that is created after a every potential move. The evaluation function iterates over all the possible winning lines and calculates the respective score, based on the marks that our player and our opponent have on the board. For this assignment, we implemented and tested 3 different evaluation function, but we ended up with the one presented bellow:



```java
public int staticEvaluate(List<positionTicTacToe> targetBoard, int playerNum)
{
    // Evaluates the state of the current board based on #player pieces and #opponent pieces on each winning line

    int score = 0;
    //counting the winning combinations theoretical state of the board
    for (List<positionTicTacToe> line: winningLines)
    {
        int playerMarks = 0;
        int otherPlayerMarks = 0;
        for(positionTicTacToe winningPosition: line)
        {
            int state = getStateOfPositionFromBoard(winningPosition, targetBoard);

            if(state == playerNum) playerMarks++;
            else if(state == 0);
            else otherPlayerMarks++;
        }

        score += Math.pow(10, playerMarks) - Math.pow(9.5, otherPlayerMarks);
    }
    return score;
}
```

3. Experiments

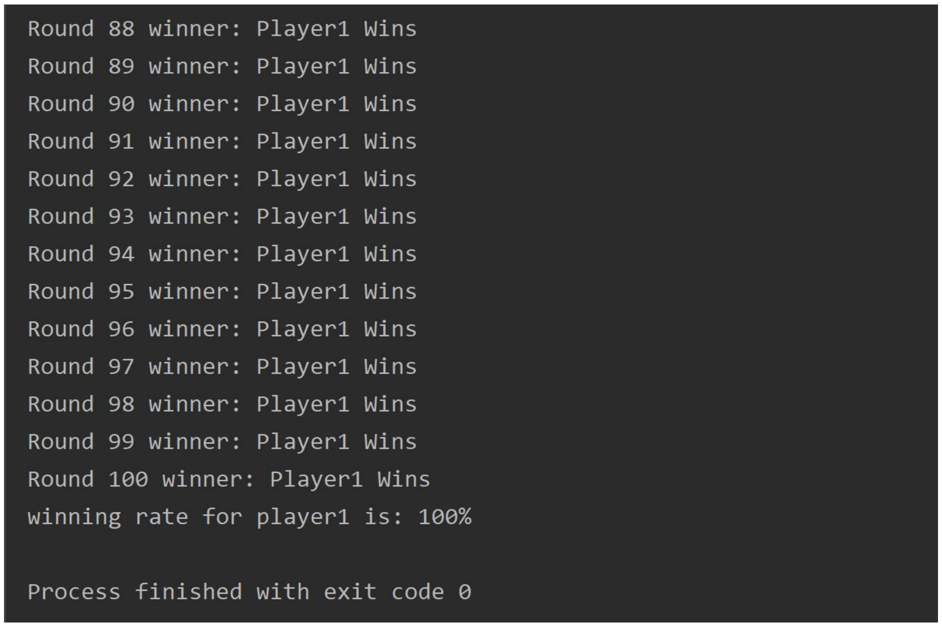
For the purposes of this assignment we conducted several experiments in order to evaluate the performance of our implementation. The goal of these experiments was to see how our agent performs under opponents under different settings' scenarios. With our first experiment we test our agent against another agent that, every time, chooses its next step randomly. Our second experiment was to test our agent against an agent with different depth parameter; we run tests for both cases that our agent has a higher depth value and for the case that the opponent has a higher depth value. Finally, our last experiment was to test our agent against an agent that has the exact same settings with our agent. We present the results of our experiments in the following section.

4. Results

For our first experiments we observed that our agent beat the agent using random selection in all the games played. In our second experiment we saw that the winner was always the agent that explored a bigger depth of the game tree. Finally, for our last experiment where both of the agent has the exact same settings the winner was always the agent that made the first move.

Because our initial heuristic evaluation method was inconsistent, we started to change it
The results of our experiments are presented in the following images:

Our AI agent vs randomized agent

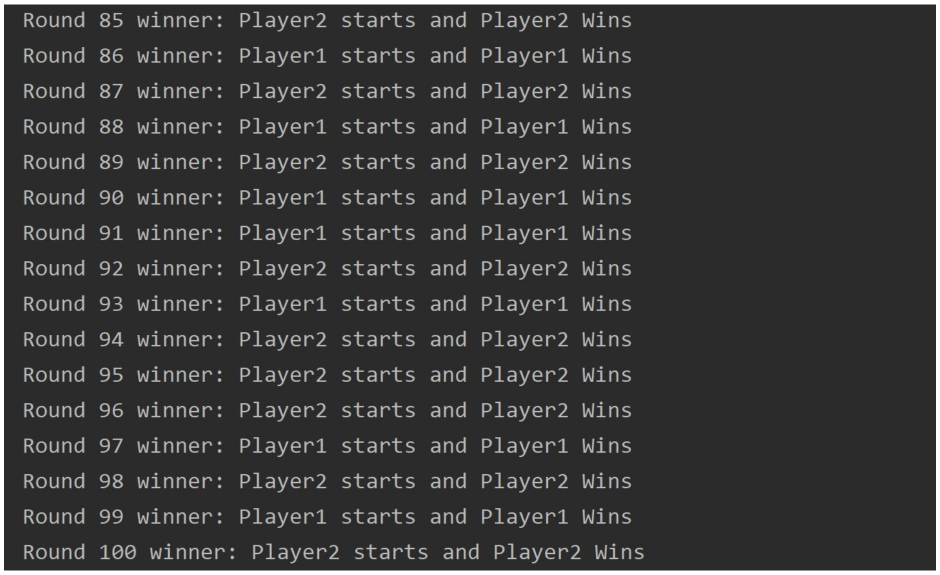


```
Round 88 winner: Player1 Wins
Round 89 winner: Player1 Wins
Round 90 winner: Player1 Wins
Round 91 winner: Player1 Wins
Round 92 winner: Player1 Wins
Round 93 winner: Player1 Wins
Round 94 winner: Player1 Wins
Round 95 winner: Player1 Wins
Round 96 winner: Player1 Wins
Round 97 winner: Player1 Wins
Round 98 winner: Player1 Wins
Round 99 winner: Player1 Wins
Round 100 winner: Player1 Wins
winning rate for player1 is: 100%

Process finished with exit code 0
```

Our AI with depth = 3 vs randomized agent with 100% winning rate (random start turn)

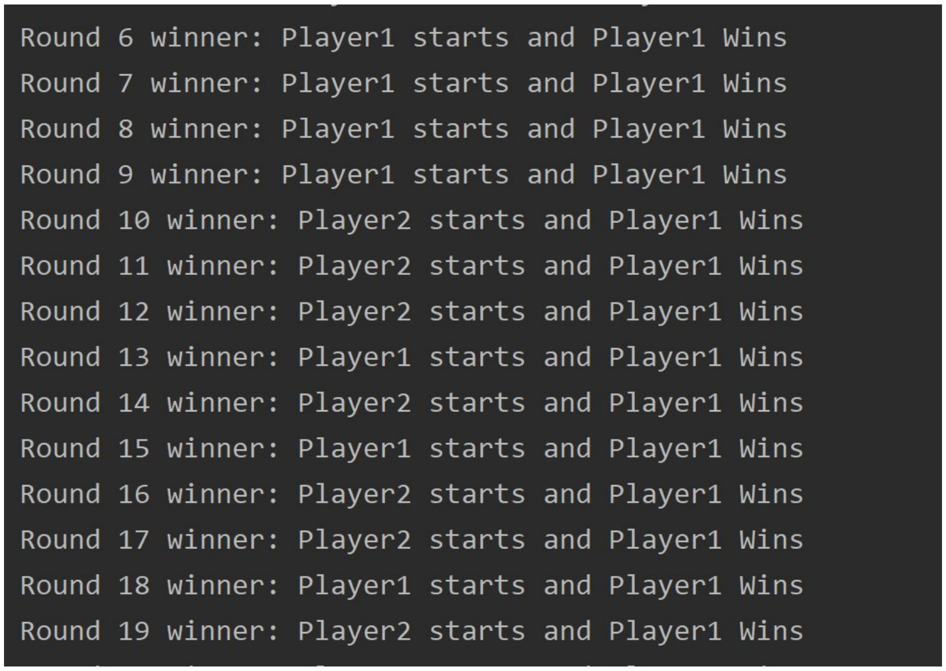
Our AI agent vs our AI agent (same settings)



```
Round 85 winner: Player2 starts and Player2 Wins
Round 86 winner: Player1 starts and Player1 Wins
Round 87 winner: Player2 starts and Player2 Wins
Round 88 winner: Player1 starts and Player1 Wins
Round 89 winner: Player2 starts and Player2 Wins
Round 90 winner: Player1 starts and Player1 Wins
Round 91 winner: Player1 starts and Player1 Wins
Round 92 winner: Player2 starts and Player2 Wins
Round 93 winner: Player1 starts and Player1 Wins
Round 94 winner: Player2 starts and Player2 Wins
Round 95 winner: Player2 starts and Player2 Wins
Round 96 winner: Player2 starts and Player2 Wins
Round 97 winner: Player1 starts and Player1 Wins
Round 98 winner: Player2 starts and Player2 Wins
Round 99 winner: Player1 starts and Player1 Wins
Round 100 winner: Player2 starts and Player2 Wins
```

We use lookahead depth = 3 for both agents and randomized start turn. We get 100% winning rate for whoever makes the first move.Note that using 9.5 in the evaluator would produce this result. However using 9 would result in second player always win.

Our AI agent with depth 4 vs our AI agent with depth 3

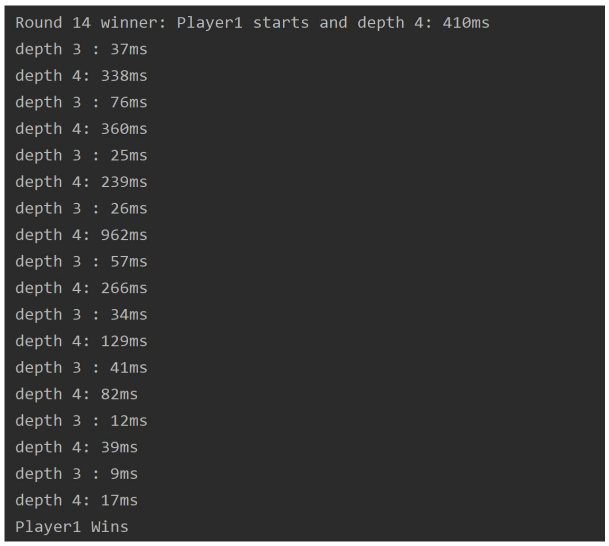


```
Round 6 winner: Player1 starts and Player1 Wins
Round 7 winner: Player1 starts and Player1 Wins
Round 8 winner: Player1 starts and Player1 Wins
Round 9 winner: Player1 starts and Player1 Wins
Round 10 winner: Player2 starts and Player1 Wins
Round 11 winner: Player2 starts and Player1 Wins
Round 12 winner: Player2 starts and Player1 Wins
Round 13 winner: Player1 starts and Player1 Wins
Round 14 winner: Player2 starts and Player1 Wins
Round 15 winner: Player1 starts and Player1 Wins
Round 16 winner: Player2 starts and Player1 Wins
Round 17 winner: Player2 starts and Player1 Wins
Round 18 winner: Player1 starts and Player1 Wins
Round 19 winner: Player2 starts and Player1 Wins
```

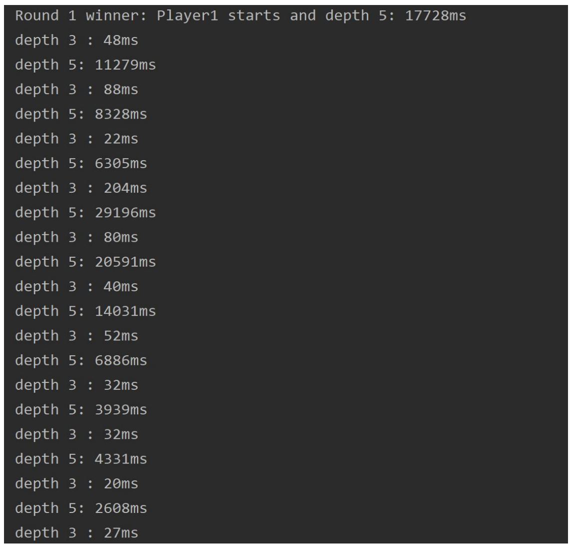
We randomize the start turn. Winner is always the agent with deeper lookahead depth. Note that this is achieved by using 9 as the base instead of 9.5 in the evaluator.

For our algorithm which uses 9.5, it doesn't show this feature. Instead, it is still first turn wins.

Time used to compute one move



```
Round 14 winner: Player1 starts and depth 4: 410ms
depth 3 : 37ms
depth 4: 338ms
depth 3 : 76ms
depth 4: 360ms
depth 3 : 25ms
depth 4: 239ms
depth 3 : 26ms
depth 4: 962ms
depth 3 : 57ms
depth 4: 266ms
depth 3 : 34ms
depth 4: 129ms
depth 3 : 41ms
depth 4: 82ms
depth 3 : 12ms
depth 4: 39ms
depth 3 : 9ms
depth 4: 17ms
Player1 Wins
```



```
Round 1 winner: Player1 starts and depth 5: 17728ms
depth 3 : 48ms
depth 5: 11279ms
depth 3 : 88ms
depth 5: 8328ms
depth 3 : 22ms
depth 5: 6305ms
depth 3 : 204ms
depth 5: 29196ms
depth 3 : 80ms
depth 5: 20591ms
depth 3 : 40ms
depth 5: 14031ms
depth 3 : 52ms
depth 5: 6886ms
depth 3 : 32ms
depth 5: 3939ms
depth 3 : 32ms
depth 5: 4331ms
depth 3 : 20ms
depth 5: 2608ms
depth 3 : 27ms
```

We could see some large time difference between different lookahead depth.
Depth 3 is very quick and efficient. Depth 4 is prefered with acceptable time and higher winning rate. Depth 5 takes too long to compute for one move. Not realistic in a real game.

5. Discussion

The results of our implementation are in agreement with our original expectations of a good Tic Tac Toe playing agent.

When a randomized agent competes with an agent that follows a specific strategy, it is obvious that if the strategy is good enough then the later agent should always win. That is because the randomized agent does not take into account any of the game's available information so the only way to win is by luck. On the other hand, an agent that follows our implementation has a specific strategy which is defined by the, partial, knowledge of the outcomes of its actions. This is the reason why an agent that utilizes this knowledge should be able to outperform a randomized agent.

Another important observation is that a good agent should be able to perform better if it has the ability to look further into the future, meaning that, assuming everything else is fixed, an agent that can search into a larger depth should be able to defeat an agent which only searches to less depth. This is due to the fact that as the depth of the search increases, so does the knowledge of the agent regarding the consequences of each of its possible action. Our implementation manages to capture this intuition, since as we showed in the previous section when one of our agents has a bigger depth it manages to beat every time an agent with a smaller depth. Here we encountered some interesting facts on the heuristic function we had. Originally, we defined $10^{playerMarks} - 5^{oppositePlayerMark}$ as our function to calculate board static score. At this point, it was already beating random AI with 100% winning rate. However, digging deep into the behavior of our agent in every step, we found that our static evaluation tend to encourage attack rather than defense. Blocking the opponent becomes less important in this scenario. Then we modified the function to $10^{playerMarks} - 9.999999999^{oppositePlayerMark}$. Since we value “winning” more than “not losing”, we give playerMarks a little advantage by making the second constant close to 10. This function works quite well and beats our previous agent (with same depth). However, new problem that arises here is that agents does not follow our intuition of lookahead depth. With this heuristic function, an agent with depth 3 outperforms the one with depth 4 with certainty. And agent with depth 2 can even compete with agent with depth 3. This is a question that remains unsolved. However, we solve this problem by changing the function to $10^{playerMarks} - 9.5^{oppositePlayerMark}$.

We had several experiments on the same algorithm with different lookahead depth. It is natural to think that if one could predict further into the future, he/she is more likely to have the big picture under control. For our AI agent, we expect the one with larger lookahead depth to perform better. However, this is not always the case for all the evaluation functions we tried. Some of them follow this intuition which we view them as good agents. Others which perform randomly or even counter-intuitively are probably due to bad evaluation function or in our case long double calculation (as we guessed). Another observation is on the time consumption for different depth. We have tried depth 1 to 6. Agents with depth 1 and 2 take less than 5ms for a step. Agent with depth 3 usually takes hundreds of milliseconds to make a move, very quick and efficient, while the longest time it takes is less than 1000 millisecond. When we choose a depth of 4 for our agent, it may take a few seconds for each step but can also take up to 30 seconds. This one works the best in our case with a realistic amount of time. On the other hand, agents with depth 5 and 6 take too long to finish a game or simply make a move. Thus we think lookahead depths of 3 or 4 are the most suitable for this game.

We have to mention that we tried to improve our results by defining more evaluation functions, but the results proved to be quite unstable.

6. Conclusions

In the scope of this assignment, we implemented an agent that is able to play the Tic Tac Toe game in a 4x4x4 board, using the minimax algorithm with alpha-beta pruning along with a heuristic evaluator of our own. The results of our implementation follow our intuition of how a good agent should perform under different scenarios of this game. Despite the successful results of our implementation, we believe that there is still room for exploring additional ways to improve its performance by focusing on testing and evaluating different heuristic evaluators. Finally, even though the results of our implementation follow our intuition, we believe that they would be more accurate if our agent was tested against agents implemented by different teams, in order to get an idea of how our agent would perform in a real world setting.

7. Credits and Bibliography

Wiki 3D TicTacToe: https://en.wikipedia.org/wiki/3D_tic-tac-toe

Team

Georgios Karantonis
Jingzhou Xue
Yernur Alimkhanov