CSC 442 Project 1 - TicTacToe

Name : Tushar Kumar
NetId : tusharku

September 30, 2018

Abstract

In this project I have implemented three variants of TicTacToe and tried to build an agent which applies Minimax algorithm to look ahead in the gamepplay and "think" what the best move at the current situation would be. I will be discussing regarding the design of the my game, what heuristics I chose, statistics of different gameplays and also the basic object design of my code. The three variants I worked on are 3*3 TicTacToe, 9-Board TicTacToe and Ultimate TicTacToe.

1 Introduction

Tic-Tac-Toe (also known as noughts and crosses or Xs and Os) is a game for two players, X and O, who take turns marking the spaces in a grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game [1].

Our goal is to fuse in intelligence to a computer and demand it to play TicTacToe and more importantly, play it well. One way we do it is by using the Minimax algorithm, where we have the computer(referred as agent henceforth), go over all the available moves one by one at any point in the game. For each available move, the agent tries to predict what the opponent would do once the agent chooses that move. It makes the assumption that the opponent also(out of all possible moves), will choose the move that's best for them. The agent repeats the same process, and keeps on going till it reaches at a point where game is over(either win/loss/draw), then back tracks this information and each point where it had a choice, tries another possible move from the list of available moves, and checks the outcome. Once the entire tree of possible moves and outcome is completely explored , the agent picks the best move based on their predicted outcome. With this basic idea lets move over to talking about the basic design of my code before we delve into each variants.

2 Design

The basic design can be understood from the Class Diagram 1 and Sequence Diagram 2. In order to be able to apply different kinds of strategies like simulating moves by picking random move on board(to enable simulating games) and to play manually against the agent, I created

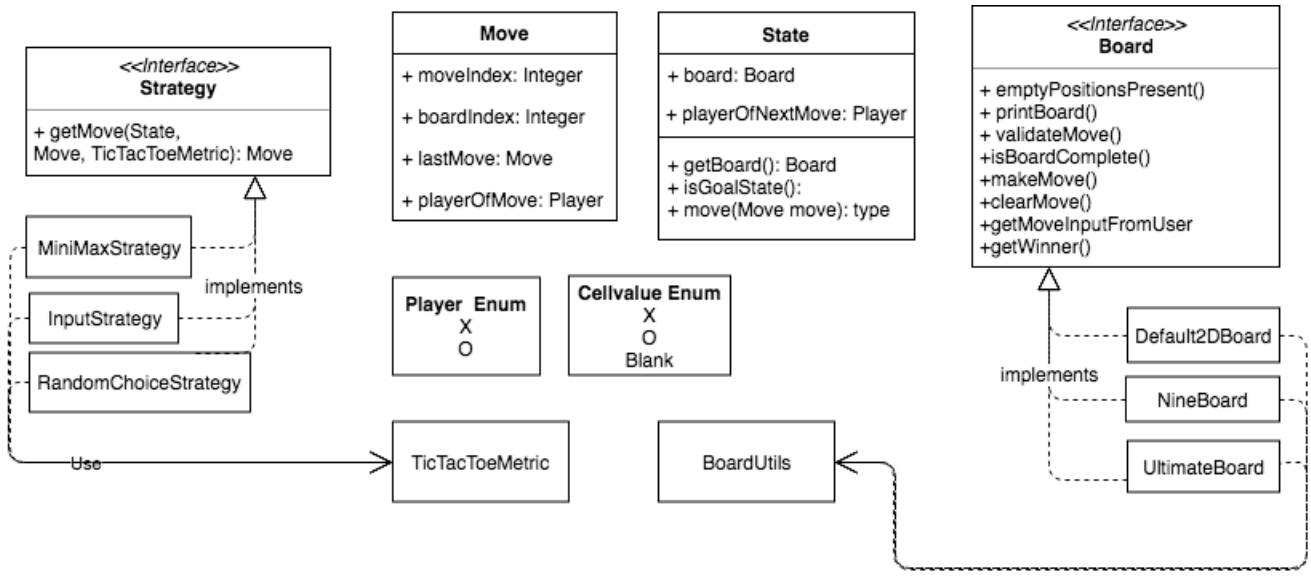

Figure 1: Class Diagram of TicTacToe Project

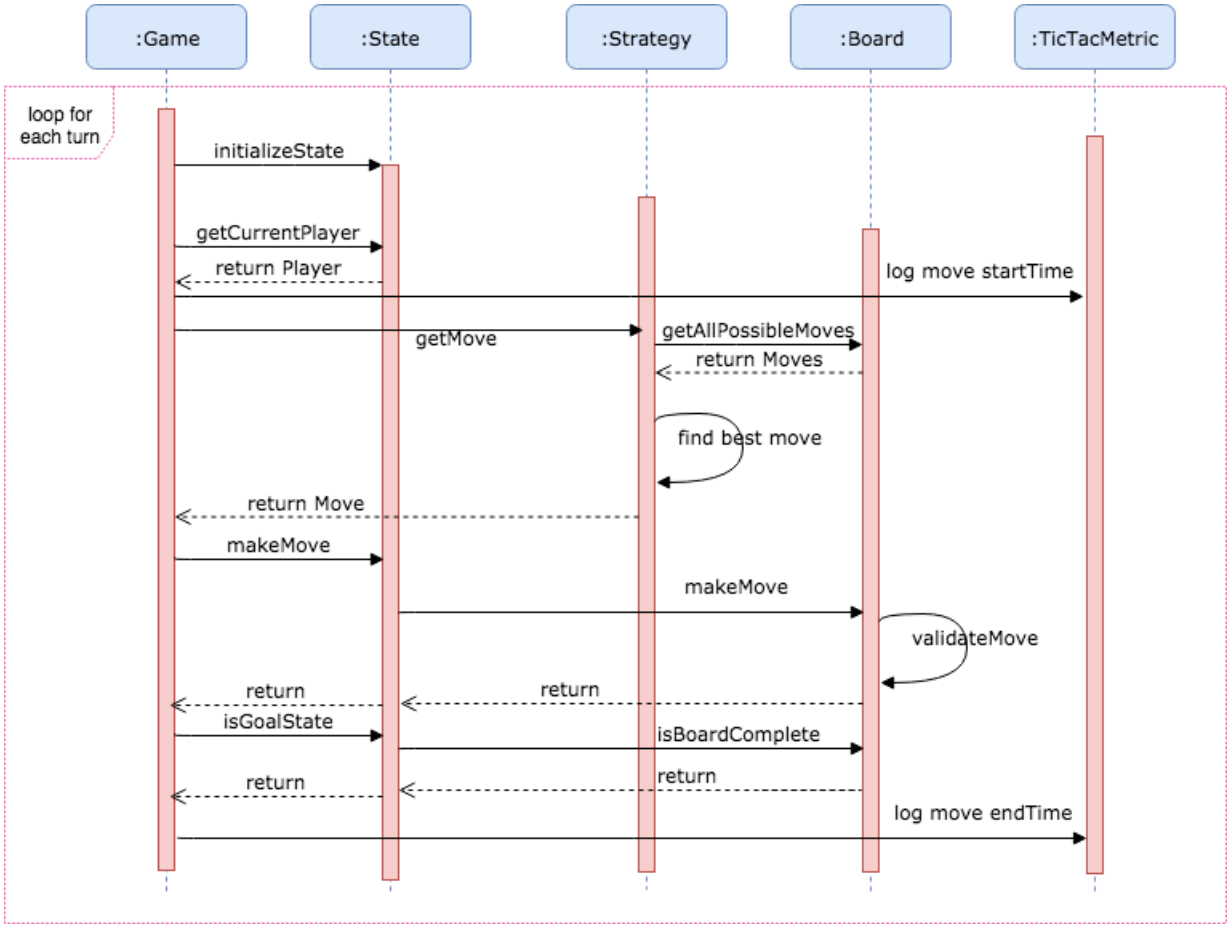

Figure 2: Sequence Diagram of TicTacToe Project

different strategies following the obvious Strategy pattern for picking moves. Furthermore, every game has a board class of its own and using inheritance (like UltimateBoard is just another child class of NineBoard) allowing child boards to use most of it's parent boards functionality and add its own custom behavior on top of it.
The Game class is the one which handles actual game play but it is loosely coupled with the kind of board, the game is being played on. This allowed me to have the same basic game play structure for all types of game. Similarly drafting out the basic Board behaviors into an interface allowed me to have zero dependency between State class and board and simplified my code.

2.1 Key Design and High Level Implementation Details

1. Strategy Pattern for Picking Moves(MiniMax Strategy, RandomChoice, UserInput)
2. Interface for Board and each implementation present for different game.
3. Utility class for all utilities that game requires.
4. Exception classed for bad input to game or invalid move demanded by current player
5. Logger class to log all statistics like time taken for agent to think, and number of states it looks at in the process.
6. Enum for Player, Piece and Command Line Arguments
7. Exception handling for all erroneous inputs
8. Throwing descriptive errors for bad command line arguments
9. Added facility to automatically play game against a random opponent(who picks moves randomly)

2.2 Possible Improvements in Overall Code Design

I will talk about the possible improvements for actual game play in later sections but from an overall high level design of the project perspective, I feel I could have generalized the classes further so that a 9Board can actually be used to play the same game on say 16Board or even generalize the 3*3 tictactoe board to be able to play N*N board. Unfortunately, right now thats not the case, even though my methods accept gridSize and other parameters in the hope that I would be able to generalize but I have never tested that out and I am pretty sure it would not straight away work out of the box. Infact, I know it wont.

3 Implementation

I have implemented Minimax(with heuristic functions and alpha beta pruning), which basically recursively calls itself till either the maximum depth(provided as input) has been

reached or the game is complete. Once it looks ahead through all the possible moves available on the board it figures out the best possible Move according to the gain of the agent. The first player can either be X or O, but my implementation does make the assumption that X will always be the maximizer. I also tried with different heuristic functions and logged different statistics(TicTacToeMetric class does this job) in order to understand the game better and how its performing.
In order to ensure that I don't have to calculate the available positions everytime(specially for complicated boards) , I store a cache of all currently possibleMoves and keep on updating that cache whenever a move is made. I also implemented a clearMove function for each board which is just like the undo version for MiniMax. This would help in ensuring that all the moves that minimax tries while its looking ahead, they don't impact the state of the board, because minimax would keep on clearing them once they get the utility value from them and update the local best possible move.

4 Basic Tic Tac Toe

The board of basic TicTacToe is the Default2DBoard class in my code. That class pretty much does the entire weightlifting for this version of game. Even though the basic version works without using any depth restrictions and without using any alpha beta pruning or heuristic, I still played with these parameters to understand better the game and how my agent was doing. I have attached a snapshot of one of my games in Figure 3 , and I also demonstrate how my program handles all kinds of erroneous inputs gracefully through Figure 4. This graceful exception handling is common for all game variants.

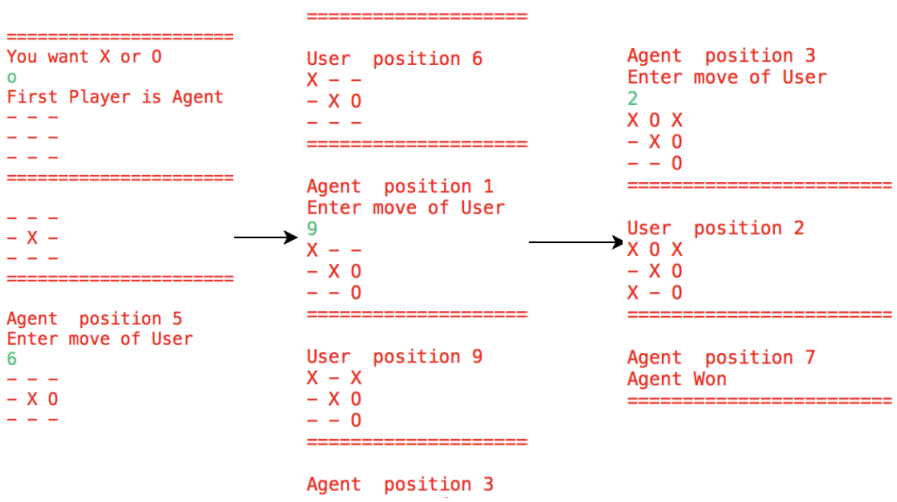

Figure 3: Game play of Basic TTT

4.1 Heuristic and Depth

As mentioned before, Basic TicTacToe will run even without heuristics and alpha beta pruning but I wanted to see the impact of allowing very few depths coupled with my heuristic

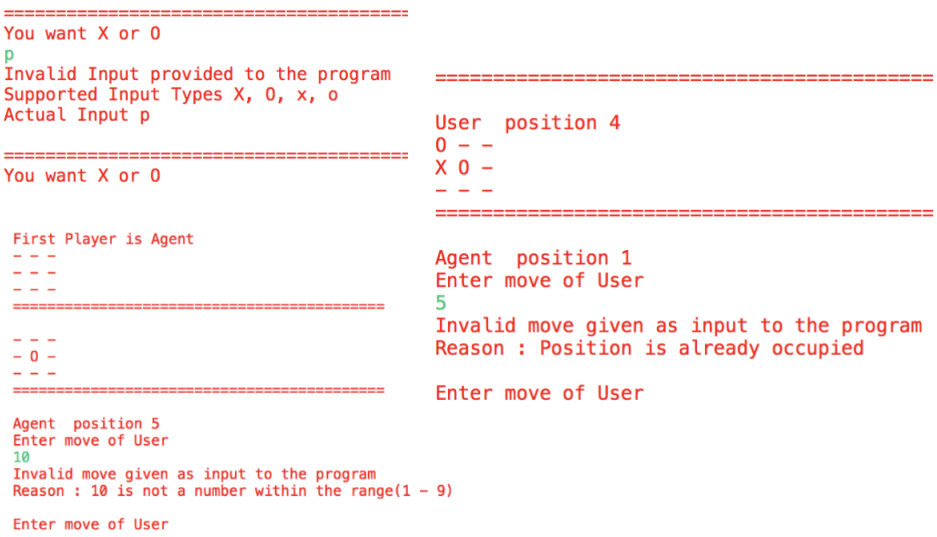

Figure 4: Exception Handling of Basic TTT

function being applied to Basic TTT. Hence, I simulated 1000 games of TTT, with minimax competing against an opponent who picks board positions randomly. I ran this simulation with no heuristic and varying depths, followed by with heuristics and varying depths.

4.2 Analysis

In this section I will briefly talk about my analysis after applying the Minimax algorithm to Basic TTT.

With just the basic minimax algorithm, I was able to build an agent that will never lose. Most of the games that I played manually it was able to either draw or beat me. The time taken to play the game is also decent enough. By decent enough, I mean that while playing with the agent, I don't see any visible delay(caused by it playing the entire game by assuming my optimal moves and then figuring out best moves). I wish the same could be said by the agent regarding my moves.

Nevertheless, I wanted to see if whatever small time it takes can further be optimized. Obviously we could just cut off the state space search at come fixed depth, and I tried the obvious. As you can see in Figure 5 that in the absence of any heuristic(I just return 0 when I reach that depth), so if its not a win/loss state till that depth, then the agent assume it will be a draw. Now, granted that its not a amazing thing to do because for the agent this means that utility of the move that causes it to lose or the move that causes it to win is the same after it has reached that depth. Which is why we see the abysmal 350 number of losses in a 1000 game simulation at a max depth of 1. Since all the agent is doing is looking ahead 1 move. And the only reason that number is JUST 350 is because the opponent was choosing randomly otherwise am sure it might as well have been 100% loss.

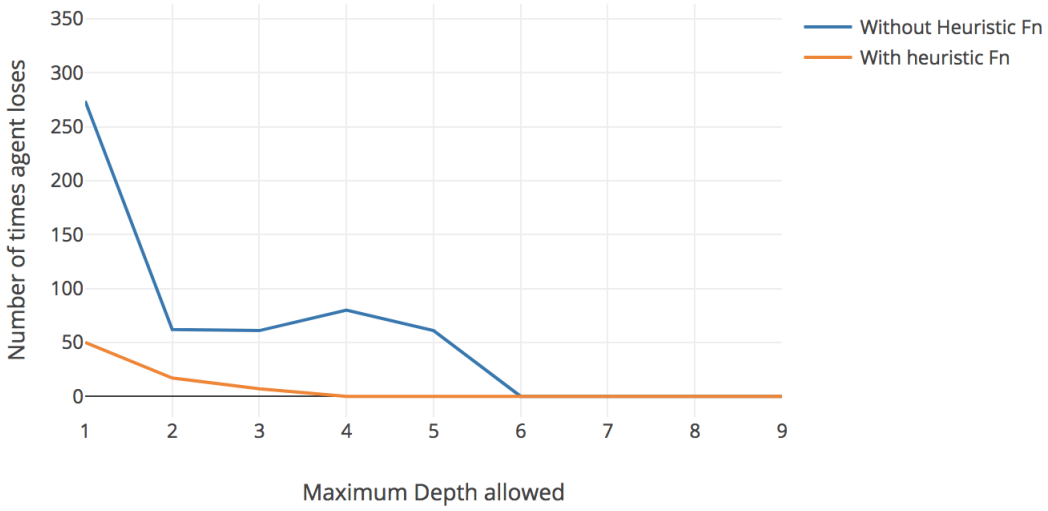

Figure 5: Plot of Loss rate(against a random opponent) vs Maximum allowed Depth of Minimax

4.2.1 Heuristic Function

The only way to better estimate utility when the board is not yet complete was to be able to make a guess regarding the quality of current state of the board. This way if the agent has two states, both of which are neither at a win or loss state, it can use the heuristic to compare these two and pick the one that is of better quality according to the heuristic.

These are the following heuristics I tried:

a) Sum of number of continuous two pieces of same type followed by a blank(in row/ column/diagonal). Utility is the difference between sum for X and sum for O, since in my code, X is always the maximizer.

b) Same as a.) but if the sum is across diagonal then give more score. This was based on the intuition that a centre diagonal piece, gives the added benefit that placing one piece nearby in any remaining corner will give you two winning positions.

c) Difference of game winning positions between X an O. By game winning positions, I mean, positions where if I place my piece, then I win the game. Like if there are three X's on three corners(Top left, Top right, Bottom Right) of the board(and assume for now that rest of board is empty), then there are 3 game win positions , Top Center, Center Right, Center,

I found that a combination of first and third working best by simulation and also by manually playing against the agent. With these, I was able to achieve a 0% loss rate at a depth of 4.

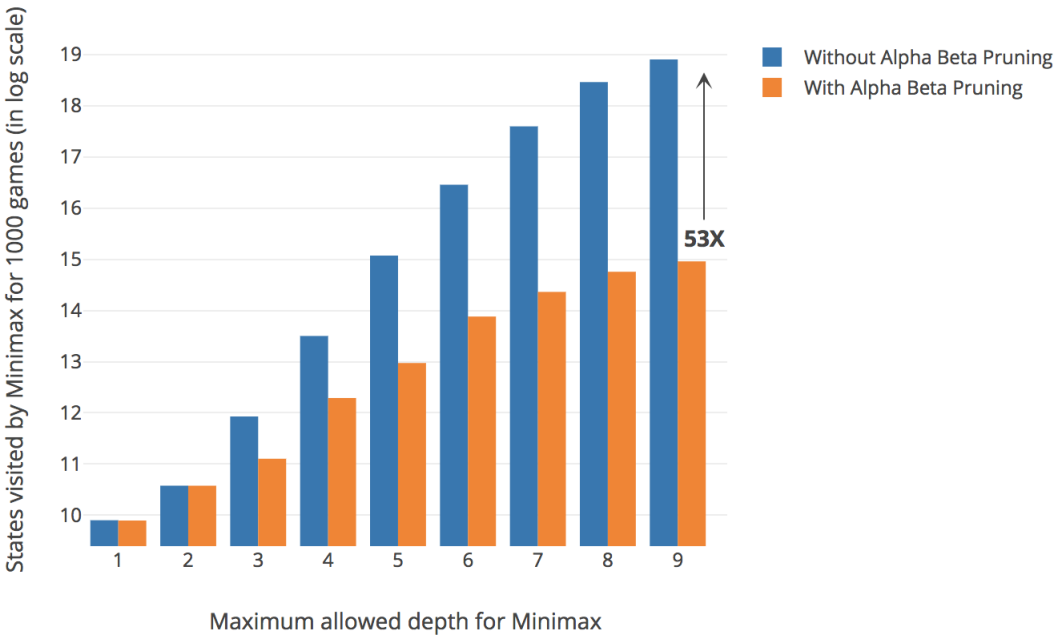

Figure 6: Effect of AlphaBetaPruning

4.2.2 Alpha Beta Pruning

I also implemented alpha beta pruning to ensure that I don't visit states/subtree of states which are guaranteed to have worse utility than the minimum utility I could achieve till that point. I added two metrics to measure the benefits of alpha beta pruning on the game:.

a) Total number of states visited by the agent. This is basically the number of board states the agent created in order to mimic the game and look ahead to find the best moves. I plot this number for 1000 games in total in log scale in Figure 6. As one can see at depth of 9, the agent looks at 53 times lesser number of states with alpha beta pruning.

b) I also tracked the time the agent took for "thinking" for a move. This basically encapsulates the entire time it used to mimic the game play till maximum allowed depth and then find the best possible move. I add these timings over 1000 games and plot the time in seconds in Figure 7. Alpha beta pruning leads to improvement by factor of 10 at depth of 9.

The reason I only plotted till depth 9, because post that the game will behave the same and have the same metric, because the maximum depth that basic 3*3 TTT can ever have is 9. Which is exactly when minimax algorithm is run for the first move itself. Since it will take 9 moves to completely fill the board, the maximum depth that ever can be reached is 9.

4.2.3 Key Implementation Details

a) **Board taken as a N*N 2D array**

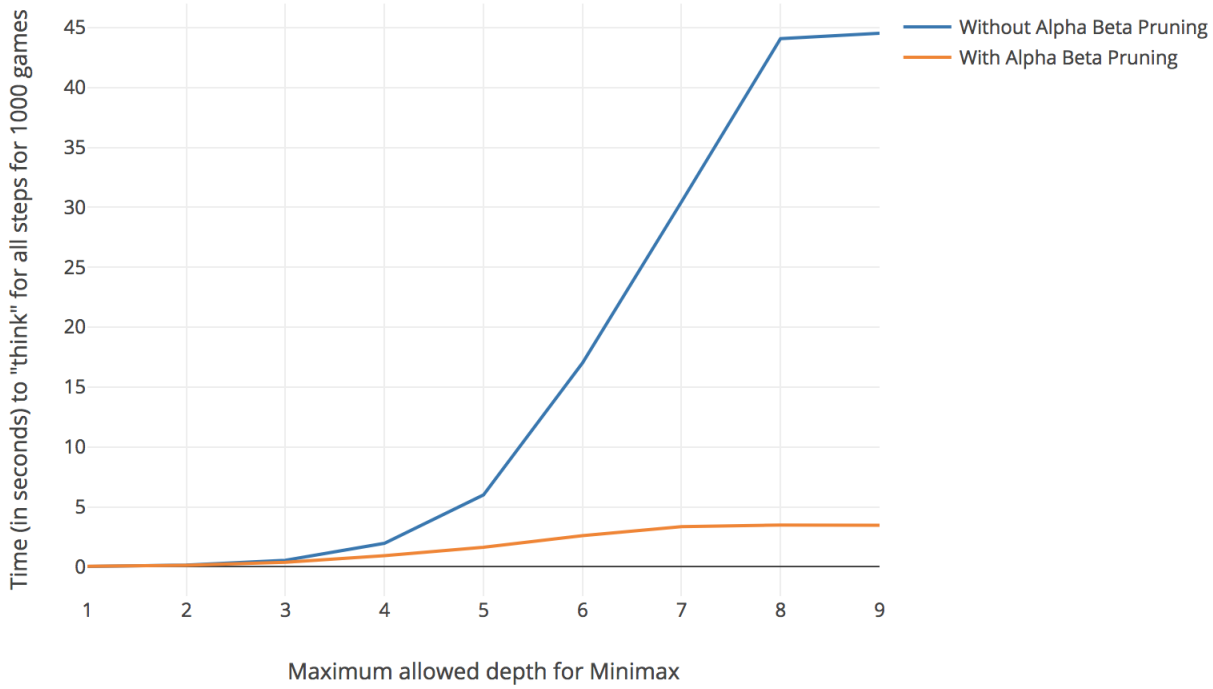

Figure 7: Effect of AlphaBetaPruning on Time

b) **Metric logger class to log statistics**

c) Added **alpha beta pruning** and **heuristic function** mentioned in 4.2.1 a) and 4.2.1 c.)

d) Assumed **X is always the maximizer** but **first player can be anybody**

e) Ensured that **agent never loses even when restricted to depth 4**.

4.2.4 Key Observations

To summarize, these are the key observations I made with my analysis of Basic TTT:

a) **Alpha beta pruning leads to a reduction of 53X(at a depth of 9) in terms of the board scenarios/tree nodes** the agent will check to find the best move. Even if two board states are exactly same, they will count as different because they were reached through different paths.

b) From time perspective, **Alpha beta pruning lets the agent think 10 times faster**.

c) An okay enough heuristic function will cause significant improvements with respect to a heuristic which treats every state as same quality. This is true even if the agent only looks ahead two moves(it will at least prevent a loss most of times).

d) One can notice that the **loss rate suddenly goes up after depth 3 and then goes to 0 at depth 6** in Figure 5. This is an interesting outcome caused by a concoction of depth that I am allowing the agent to look upto, bad heuristic function and the fact that opponent is picking moves randomly. I talk about why this happens in detail in **Appendix A.1** (for the interested reader).

e) **Minimum game moves it takes the agent to lose the game at depth 1 is 5 and after that, for all other depths, the agent only loses at move 7** or greater, if at all it loses.

5 Nine Board TicTacToe

Nine Board TTT is played on a 9*9 board having 9 Basic TTT grids within itself. The game play is same as Basic TTT except for one caveat:
Each players turn is restricted to one board only. That board position is equivalent to the move position of the earlier player. So player A moved to board 5, position 2, then player B must make a move ONLY in board 2. In case board 2 is completely filled , then player B can move in any board. First player can move anywhere.

This is a pretty **interesting variant of TTT** and there are multiple reasons I feel so:

a) **Draw is very rare**. Unlike TTT where you can quite often, be able to ensure that the game at least ends in a draw, its very rare that Nine Board game will not end up in a win/lose scenario. This is evident even when I simulated like 1000 games with minimax playing against random move picking opponent and there were zero draws whereas in the same environment, Basic TTT had about 50 draws. My bet would be on the rules of the game and presence of nine boards which will ensure that entire game ends in a draw if and only if ALL 9 boards are ending in a draw. I also had Minimax play against Minimax, for the Basic TTT and all 1000 games were draw. However, for NineBoard, the story was different. About 500 times, one agent won, and the other half number of times the other agent won. There were ZERO Draws!.

b) **You need to think MORE!!**. Restrictions of the game force you to think smartly. You must always think of the big picture or the entire board here. If you make moves just so that the current 3*3 board cannot be captured by the opponent, without thinking which board you are directing the opponent to , you will lose, like me.

c) Even though the game is played on a 9*9 board , its much **smaller in terms of branching factor and state space than a 9*9 tic tac toe**. This is because even though the first player has 81 move options, post that, every player only has ≤ 9 possible moves, and as the game progresses the branching factor rapidly decreases. In fact excluding the 81 branches for first move, average **branch factor is 6**(computed by averaging possible Move sizes for every move for 1000 games)

The gameplay of Nine Board TTT in my project is similar to the basic TTT. In this also I ensure all the exception handling is in place, and I also make sure that the players conform

to the rules. If a player enters a board number that does not match the last players move, I throw an exception and ask the player to enter again as displayed through Figure 8.

```
------------------
X 0 -   - - -   - - -
- - -   - - -   - - -
- - -   - X -   X - -


- - -   - - -   - - -
- - X   - - -   - 0 -
- - -   - - -   - - -


- - -   - - 0   - - -
0 - -   - - -   - - -
- - -   - - -   - - -


------------------
Agent board 6 and position 5
Enter move of User
89 1
Invalid move given as input to the program
Reason : 89 is not a number within the range(1 - 9)

Enter move of User
a 1
Invalid Input provided to the program
Supported Input Types : Board Position and Move Position should be integer between 1-9
Actual Input :  board position given was a and move position given was 1

Enter move of User
6 1
Invalid move given as input to the program
Reason : Current move not in the board index according to last move

Enter move of User
```

Figure 8: Exception Handling of NineBoard TTT

5.1 Analysis

5.1.1 Heuristic Function

Because I thought for heuristic functions for 3*3 board, for Nine Board, I only tried two combinations of heuristic functions:

a) Average of Utility value of all Nine boards. For each of the 3*3 boards, I calculate the utility by count of continuous of two X's vs count of continuous two O's and also add to that the difference of game winning positions. In the end I just sum the utilities for

each board. This is part of reason I kept the maxUtility and minUtility 1000 and -1000 respectively, as I did not want the utility of a non winning board to ever exceed the max/min utility. I even verified this over 1000 runs of game to ensure that maxUtility never ends up being greater than 1000 because of my custom heuristic function.

b) Initially I thought to give more importance to the current board where I am supposed to move, and make sure that I ensure a higher utility is obtained by my move for the current board. So I used a weighted sum of the utility value of all 9 boards using quantity α where $\alpha \leq 1$ and it just represents weight of current board. So utility would be given by :

$$\alpha * (CurrentBoardUtility) + (1 - \alpha) * (RestOfBoards)$$

I ended up using the first heuristic itself because even with second heuristic on trying different values of α I found that 0.1 is what works best(which is equivalent to all boards having same weight), so I might as well simplify the heuristic by just taking the sum.

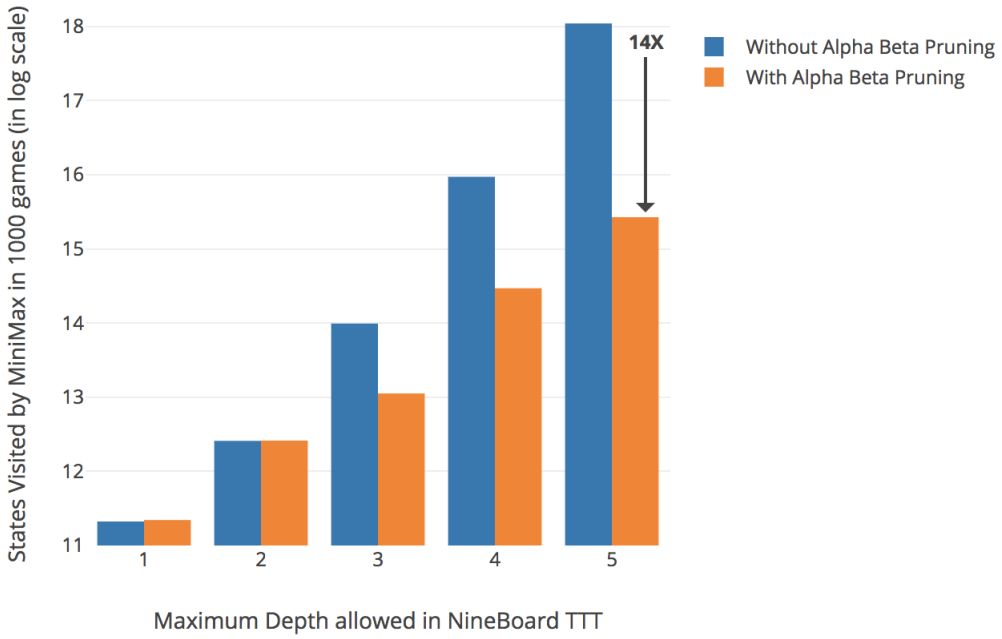

Figure 9: Effect of Alpha Beta pruning for NineBoard TTT

5.1.2 Alpha Beta Pruning

Same as basic TTT, I made sure to use alpha beta minimax to improve the efficiency of search. I analyzed the same two metrics for Nine Board TTT also.

a) Total number of states visited by the agent. Figure 9 demonstrates the benefit of alpha beta pruning on the NineBoard TTT game. Like before, these metrics were generated by playing 1000 games against an opponent whose strategy of picking moves is random.

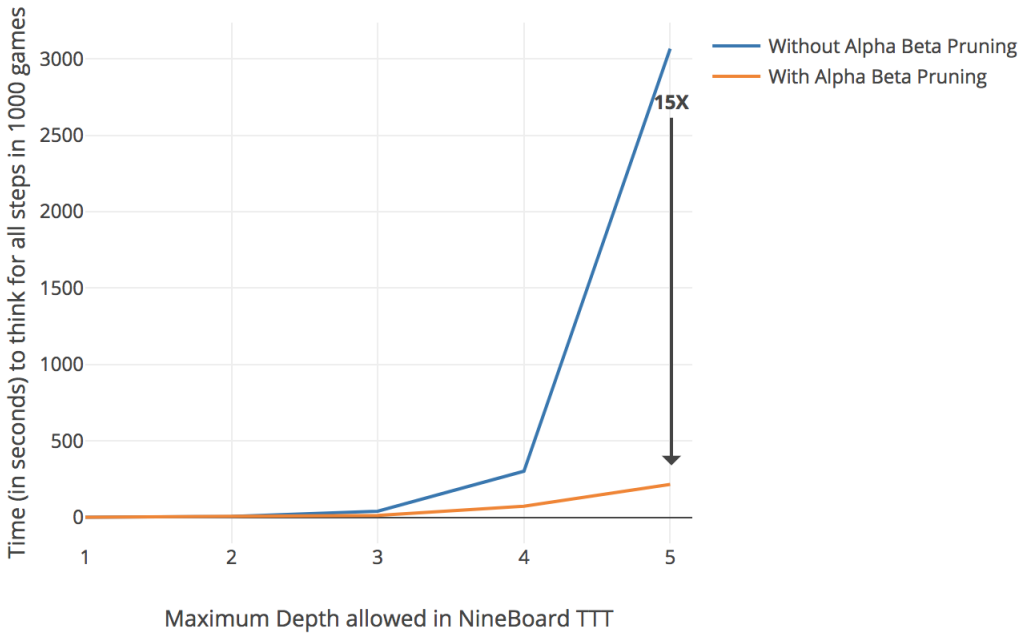

Figure 10: Effect of Alpha Beta pruning on Time for NineBoard TTT

b) Time the agent took for "thinking" for a move. Figure 10 demonstrates the benefit of alpha beta pruning on the thinking time of the agent for NineBoard TTT game. As one can see, there is a tremendous benefit that alpha beta pruning gives here improving by almost 15 times.

5.1.3 Caching to Improve Efficiency

There were two ways where I thought of using caching to improve the time it takes for the agent to think of a move:

a) **Cache the utility of the board**

Throughout the game, minimax would be referring to the heuristic function so many times to get the utility of all nine 3*3 boards, because my heuristic function requires summing them up together. So I thought that why not cache the board and utility mapping and this way whenever I see the same board again, I just return the cached utility value. While the idea seemed promising , it turned out that because I used the array type to store the board, in order to store anything in map, I had to convert it into a representation which could easily/uniquely be hashed based on its contents, like a string. Because of this additional computation, the cached solution was actually taking more time. Hence **even though I implemented it, I did not end up using it**.

b) **Cache the winner of board**

So one of functions that minimax would be calling at every step is to determine the winner of the board, and in games like NineBoard, it means that entire board has to be iterated over to get that value. I thought a better way would be to instead keep a state of player winning the board by checking as soon as a move is made. Because then I know that only possibility for a winner, is the board where that particular move was made(as if there was already some other board winning, the game wouldn't have moved forward or reached this point), so I right away reduce the need to check eight 3*3 boards. If I find that there is a winner of this board then I just maintain that state. Then during my utility calculation, I can just see the state to know which player won the game, instead of calculating this for entire 9 boards. Ofcourse, I also update the cache as soon as a move is "cleared" or is "undone" by minimax, and that caused that board to no longer be in a winning state. Adding this strategy gave me a reduction of 40% (Figure 11) in time taken for a move, when measured at a depth of 5 for NineBoard TTT.

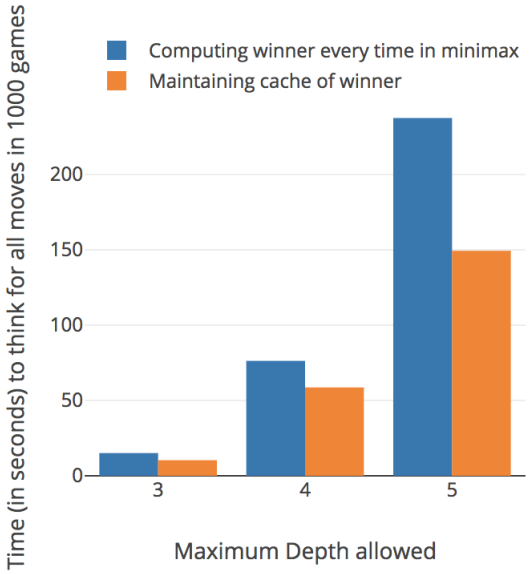

Figure 11: Effect of Caching Winner of Board for Minimax in NineBoard TTT

5.1.4 Key Implementation Details

a) **Board stored in program as 2D N*N array**

b) **Metric logger class to log statistics**

c) Added **alpha beta pruning** and **heuristic function** mentioned in 5.1.1 a)

d) **Cached winner of board** for Minimax to ensure (costly)winner calculation is optimized as explained in 5.1.3.b

e) Assumed **X is always the maximizer** but **First player can be anybody**

f) Ensured that **agent never loses against a random opponent even when restricted to depth 4**.

5.1.5 Key Observations

To summarize, these are the key observations I made with my analysis of NineBoard TTT:

a) **Alpha beta Pruning leads to a reduction of 14X(at a depth of 9) in terms of the board scenarios/tree nodes** the agent will check to find the best move.

b) From time perspective, **Alpha beta Pruning lets the agent think 15 times faster**.

c) **Caching winner of board** optimizes the minimax step and **leads to 40% improvement in move think time**

d) Playing against a random opponent loss goes to 0, at depth 2 itself, but thats just because of a random opponent will be significantly easy to beat (easier than beating a random opponent at Basic TTT)

e) When playing manual games, **I have not been able to beat the agent even once**(in 20 games).

6 Ultimate TicTacToe

Ultimate TTT is played like NineBoard on a 9*9 board having 9 Basic TTT grids within itself. The game play is same as NineBoard TTT except for one caveat:
In order to win the game, you must win three boards in a row(horizontally/vertically/diagonally).

The gameplay of Ultimate Board TTT in my project is similar to the NineBoard TTT with same level of graceful exception handling. In Figure 12 , I demonstrate the winning condition found by the program and it declaring the winner. A keen reader would have noticed that even though the user moved to position 7, which should restrict the agent to only moves in board 7, but the agent moved in board 5. Why did the agent do so ? Well that's because the agent knows one additional rule of the game.

Once the outcome of a local board is decided (win or draw), no more moves may be played in that board. If a player is sent to such a board, then that player may play in any other board.

Which is why the agent chose board 5 because placing its piece at position 5 on board 5 would immediately lead to its win.

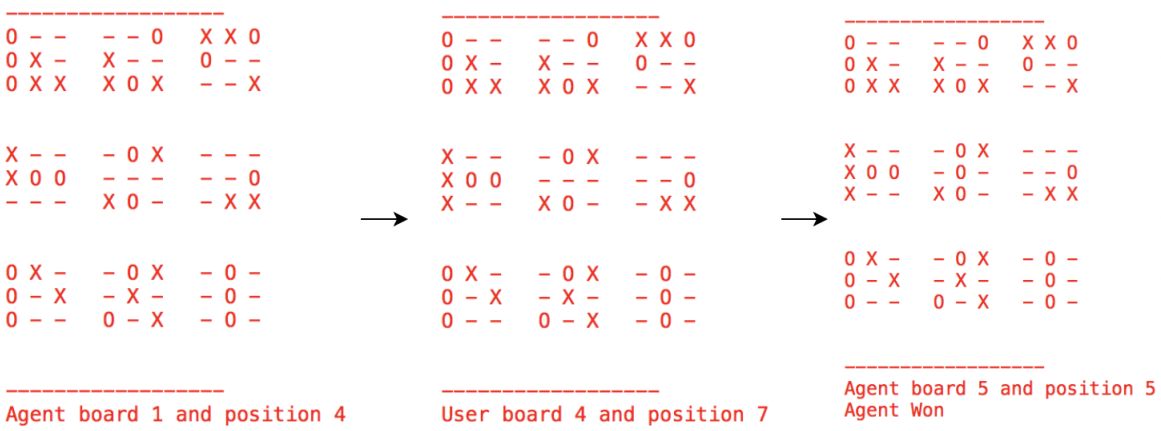

Figure 12: Gameplay of UltimateBoard TTT

6.1 Analysis

6.1.1 Heuristic Function

These are the following heuristics I tried for Ultimate TicTacToe :

a) Initially I thought, why not just do the same like Nine board TTT. For each of the 3*3 boards, I calculate the utility by count of continuous of two X's vs count of continuous two O's and also add to that the difference of game winning positions. In the end I just sum the utilities for each board. However, this obviously does not make sense when you think about it. I can actually be winning, 3rd, 6th and 7th board completely but still end up losing the game, because the opponent might be winning the 2nd, 5th and 8th board(3 in a row) and hence I might lose even though the heuristic would have found out that Player X and Player O both are winning three three boards each hence they both are equally winning and hence would have returned a heuristic value of 0.

b) I thought of ensuring that the heuristic function never becomes a linear combination of the boards. The reason being that as soon as it becomes a linear combination, then it would never care of the order in which boards are being won. I had to make sure that winning board 3,4,5 is lesser value than winning 1,2,3. Because of this, I introduced a non linearity by using Max and Min functions while calculating heuristic values for rows and columns. I demonstrate the calculation of utility of a Row below, (column is also done the same manner):

```
method getUtilityOfRows {
    maxUtility = 0
    minUtility = 0
    boardIndex = 1
    for(i in [0,3,6])
        rowUtility = 0;
        for(j in [0, 3, 6])
            CellValue[][] tempBoard
```

```
            //Bunch of code will set tempBoard
            CellValue winner = getWinnerOfBoard(boardIndex)
            if winner is X
                rowUtility = rowUtility + 100
            else if winner is O
                rowUtility = rowUtility - 100
            else
                set gameWinPositionsForX;
                set gameWinPositionsForO;
                rowUtility += 2*(gameWinPositionsForX -
                ↪  gameWinPositionsForO) +
                ↪  getUtilityUsingContinuousValues(tempBoard)
            boardIndex++
        maxUtility = Math.max(maxUtility, rowUtility);
        minUtility = Math.min(minUtility, rowUtility);
    return Math.abs(maxUtility) - Math.abs(minUtility);
```

Basically what I do is calculate utility value of first row of 3*3 boards. Here, if maximizer is winning a 3*3 board I add 100 else if minimizer is, then I subtract 100 from utility of that row of 3*3. If nobody is winning a board then I use my heuristics for BasicTTT to calculate utility of that 3*3 board. Once I got the values for this row of 3*3. I move onto next row of 3*3(there will be 3 rows of 3*3 boards). I keep updating the best row of 3*3 board that I have seen for maximizer and the best row of 3*3 board that I have seen for minimizer. In the end I return the difference between those two values. The intuition behind this heuristic is that If I am winning 2nd and 3rd board, and my opponent is winning say 4th and 8th Board, while calculating the utility, assuming I am maximizer, I would get first row with a value > 200(I am winning 2 boards in that row). The opponent would get same values for 2nd and 3rd row(close to 100) since it is winning only 1 board in each of those rows. Hence the utility of entire UltimateBoard in this state is 200 - min(100,100) = 200 - 100 = 100 And hence is in my favour which it indeed should be. If I would have just added all utilities it would have become 200 -100 -100 = 0. Hence I believe the non linear approach is a bit better. The same methodology is used for utility of columns of 3*3 boards also.

6.1.2 Alpha Beta Pruning

Same as NineBoard TTT, I made sure to use alpha beta minimax to improve the efficiency of search. I analyzed the same two metrics for Nine Board TTT also. I kept the maximum depth only till 4 and simulated 100 instead of 1000 games like before, to avoid waiting for huge time when alpha beta pruning is disabled.

a) Total number of states visited by the agent. Figure 13 demonstrates the benefit of alpha beta pruning on the Ultimate TTT game. These metrics were generated by playing 100 games against an opponent whose strategy of picking moves is random.

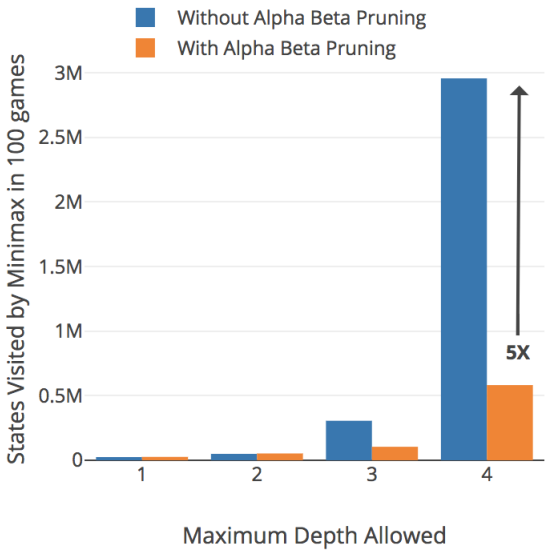

Figure 13: Effect of Alpha Beta pruning for Ultimate TTT

b) Time the agent took for "thinking" for a move. Figure 14 demonstrates the benefit of alpha beta pruning on the thinking time of the agent for Ultimate TTT game. As one can see, there is a benefit that alpha beta pruning gives here by improving it by almost 5 times.

6.1.3 Caching to Improve Efficiency

Similar to NineBoard, I implemented a caching mechanism to ensure that the computation of winner of the board does not perform avoidable operations again and again. I order to find if a board is winning, I look at my cache. And after every move, I check if this move led to this board being won by a player and I update my store/cache.

a) **Cache the winner of board**

Adding this strategy to Ultimate board gave me a reduction of 55% (Figure 15) in time taken for a move, when measured at a depth of 5 for Ultimate TTT.

6.1.4 Key Implementation Details

a) **Board stored in program as 2D N*N array**

b) **Metric logger class to log statistics**

c) Added **alpha beta pruning** and **heuristic function** mentioned in 5.1.1 a)

d) **Cached winner of board** for Minimax to ensure (costly)winner calculation is optimized as explained in 6.1.1.b

e) Assumed **X is always the maximizer** but **First player can be anybody**

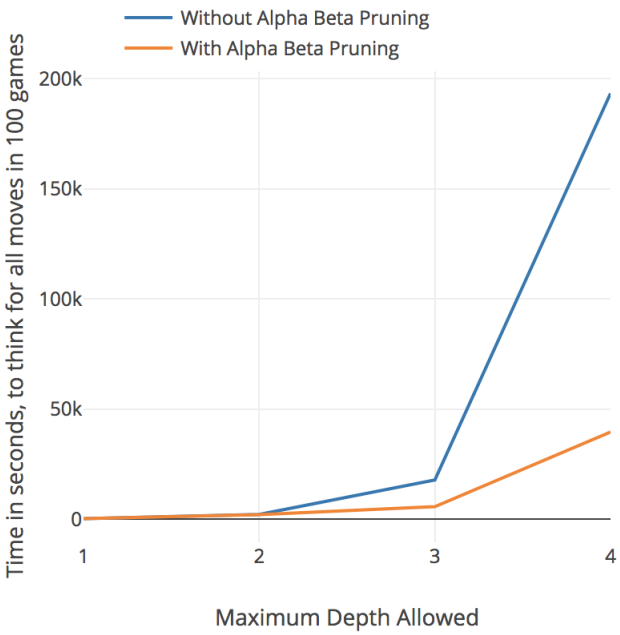

Figure 14: Effect of Alpha Beta pruning on Time for Ultimate TTT

f) Ensured that **agent never loses against a random opponent even when restricted to depth 3**.

6.1.5 Key Observations

To summarize, these are the key observations I made with my analysis of Ultimate TTT:

a) **Alpha beta pruning leads to a reduction of 5X(at a depth of 4) in terms of the board scenarios/tree nodes** the agent will check to find the best move.

b) From time perspective, **Alpha beta pruning lets the agent think 5 times faster**.

c) **Caching winner of board** optimizes the minimax step and **leads to 55% improvement in move think time**

d) Playing against a random opponent loss goes to 0, at depth 3 itself, but same as NineBoard, that's again because a random opponent will be significantly easy to beat (easier than beating a random opponent at Basic TTT)

e) When playing manual games, **I have never been able to beat the agent even once**(in about 10 games) but that's not really an amazing advertisement because I don't know how play this variant well.

7 Possible Future Improvements

Even though, I attempted to include most of the functionalities and optimizations that would improve the game and help my agent play better, there were some that are still missing from

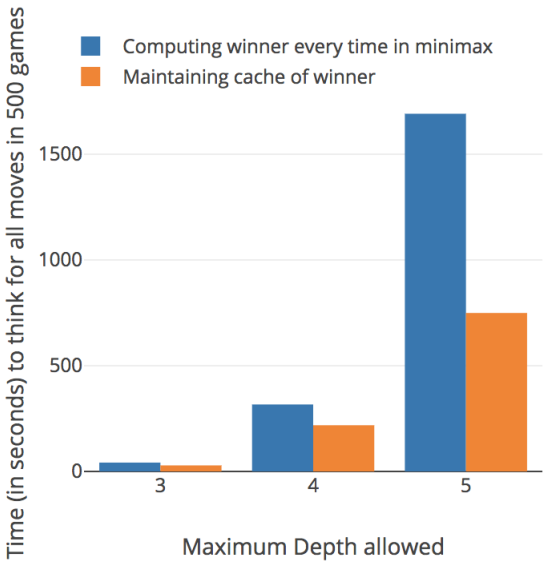

Figure 15: Effect of Caching Winner of Board for Minimax in Ultimate TTT

the game which can be termed as possible future improvements. The reason that some of them were not implemented was because they would either require too much memory, or might not really improve the game to the extent that does justice to the extra computations it takes or simply because I thought about them too late in the development process.

7.1 Cache the utility value of all states for 3*3 grid

The textbook talks about book-keeping in the chapter. And I feel that there is an opportunity to store the utility of value of the board. I believe that instead of using arrays to store the board, if I used something like Lists, then maybe I could reap in the benefits of a cache of utility value. I mean come to think of it even if I just store the basic 3*3 grid and its utility value, there are just $3^9 = 19683$ different states (since at every position we have three choices). Moreover a lot of them aren't even legal, so the number will further reduce. And having that cache will definitely improve the thinking time of agent.

7.2 Agent should try to win as early as possible

When the agent sees two moves, which lead to a win, both of them will have the same utility. But, one might lead to a win in say 3 moves, and other in 6 moves. As of this version, the agent just picks the first move with maximum utility, so it depends on luck whether the game finishes in 3 or 6 moves. But that's not really "human" behavior. If we were playing we would attempt the move that lets us win the game earliest. So instead of every move having a utility value, we can also associate every move with a depth value(at what depth of the tree will this move result in a win) and then I define the best move as the one which has highest utility value and if two moves have same utility value then I pick the move which

will ensure the game finishes earlier.

7.3 Agent should keep fighting a lost battle till the end!

This is kind of like the same situation as above, but now if the agent knows that no matter what move it chooses, it will lose. At that point also, it should not just choose any move, it should choose the move that delays the loss to the maximum time. Because agent sees game at a depth of K and figures out the moves the opponent will do, but the opponent might not be of the same caliber, and even though according to the agent its a sure loss situation but its a sure loss situation ONLY if the opponent plays the exact moves that the agent thinks it will. If the opponent misses a good move, then the game can still be saved.

References

[1] Wikipedia contributors, "Tic-tac-toe," Wikipedia, The Free Encyclopedia,https://en.wikipedia.org/w/index.php?title=Tic-tac-toeoldid=858341794 (accessed September 21, 2018).

Appendices

A. Loss rate of Basic TTT vs Depth (No heuristic case)

Just to recap, we saw in Figure 5 that without a heuristic function, when we run the BasicTTT game for varying depths, the number of agent loses suddenly jumps at depth 4 and 5 (larger than what was at depth 2,3,6,7).
Lets answer the easy question first. Why it goes to 0 at depth 6?
That is simple, because the agent is able to look ahead till 6 moves and hence avoid all moves which causes them to lose, well in advance. The interesting part is why the loss increases at 4 and depth 5 ? To analyze this , I found the moves it take for the agent to lose a game. For depths of 2 and 3, the minimum number of moves it would take for agent to lose is 7. This makes sense, because unless the opponent has two winning positions, the agent can always look ahead minimum of two steps and avoid simple losses. However, when the opponent has two winning positions(which the agent could not predict because I did not allow it to look at that much of depth), the agent has no way of getting to a draw and it WILL lose!. And it takes minimum of 6 moves of game for anybody to build two winning positions, so a minimum move of 7 for any loss is perfectly understandable.

Now comes the interesting part, I noticed that as soon as I changed depth to 4 or 5, the minimum number of moves it takes for me to lose is 5. This is a bit weird! , why should I lose earlier, when I am actually allowing the agent to look further. But that precise statement which I just made, is the answer, "THE AGENT LOOKS FURTHER".

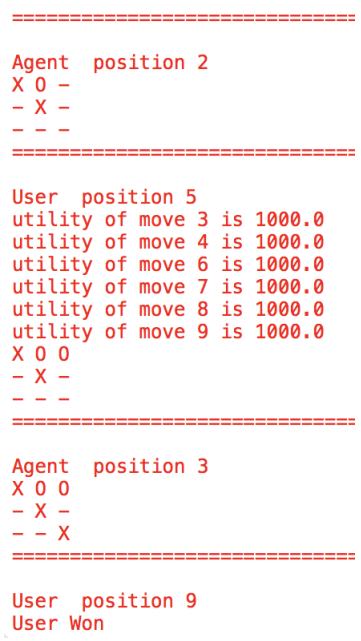

Figure 16: Why loss at depth 4 is more than depth 2

Let's look at the Figure 16, here Agent = O, User = X , depth is 4 and heuristic function is that all states are equal unless its a win/lose. So the agent, having a depth of 4 looks ahead 4 moves and realizes that no matter where the agent places the next O, its bound to lose the game. So Since all states are equal(X winning) , it picks the first possible move. And the agent loses at move number 6 of the game itself. This explains why the minimum moves to loss changed but it still does not explain why the agent lost more games at this depth than at depth 1.

The answer is that the opponent is a RANDOM opponent. At depth 2, agent would have seen that if I place O anywhere other than 9, I lose. Because the agent can only look at depth 2 and has unfortunately, a very bad heuristic function, for it, rest all scenarios would have been a draw. It would have placed O at 9. Now had it been a human opponent or a minimax opponent playing against the agent, then it would have placed their X at say 7 and created a destined loss situation for agent. But the opponent is random and the chance that it will pick first 7 and then pick a 4 or 6 is 1/15 and hence not that likely to happen which results in the not actually losing. This is why loss at depth 2 is actually better than loss at depth 4/5.