Tic Tac Toe

Kimaya Bedarkar, Vanshika Kapoor

July 24, 2020

1 Introduction

Tic Tac Toe is a classic simple strategy 2 player zero sum game where each player takes turns placing noughts and crosses in a 3 by 3 board. The first player to place 3 noughts or crosses in a row, column or diagonal wins. We chose this problem since we wanted to explore game theory and AI algorithms in game theory. We aim to look at particularly the minimax algorithm and what variations of tic tac toe can be played using the minimax algorithm.The following report is divided in three parts:

- 3x3 Tic Tac Toe
- NxN Tic Tac Toe
- Ultimate Tic Tac Toe

2 3x3 Tic Tac Toe

The first part of our project aims at building an unbeatable Artificial Intelligence agent that plays this game.

2.1 Algorithm

Our AI agent uses the minimax algorithm to play the game. We draw a tree of every move possible at every state of the board. Based on the rules of the game, we assign a score to each of the boards at the leaves of the tree. Several heuristics can be used to evaluate the board according to a set of rules. A board at the leaf of the tree will either have the AI agent winning, the opponent winning or a tie. All boards in which the AI agent wins should have the same maximum score. All boards in which the opponent wins should have the same minimum score. The boards having ties have should have a value between the maximum and minimum scores. Various heuristics can be used to order the boards having ties. One example of such a heuristic can be :

$$\begin{aligned} score = &(number\ of\ rows/columns\ or\ diagonals\ having\ two\ in\ row\ of\ AI\ agent) \\ &-(number\ of\ rows/columns\ or\ diagonals\ having\ two\ in\ a\ row\ of\ the\ opponent) \end{aligned}$$

Our program assigned the winning boards a score of 1000, the losing boards a score of -1000, and all tied boards a score of 0. However this value is adjusted by adding or subtracting the depth to account for a ”slow” or ”fast” win.The minimax algorithm traverses this game tree and, at each node, makes the best choice possible for the player. A “maximizing” player refers to the player who is trying to maximize their own score and a “minimizing” player refers to the opponent who we assume is trying to minimize the maximizer player’s score. Thus, our program recursively traverses the game tree based on these assumptions and outputs the best move.

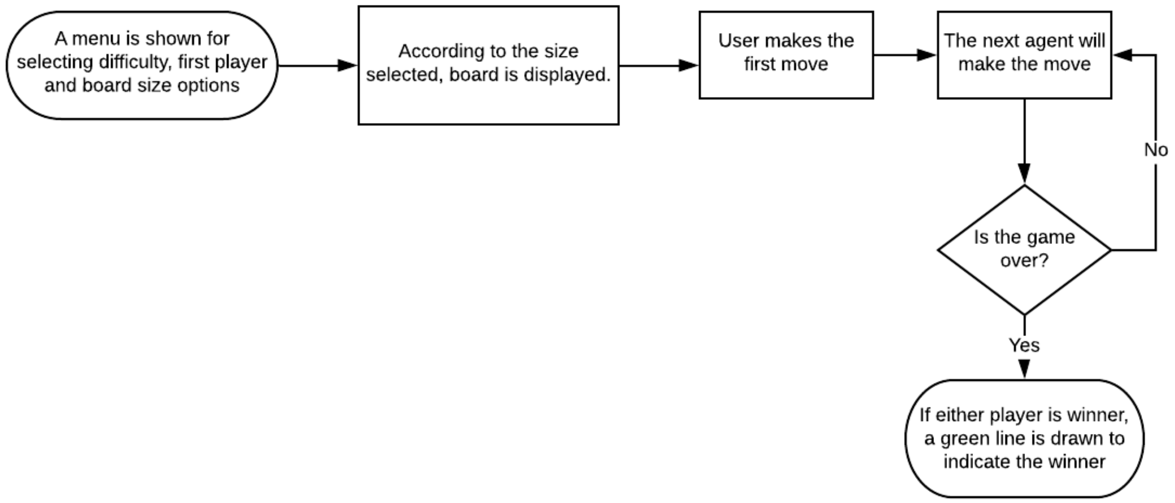
A menu is shown for selecting difficulty, first player and board size options
According to the size selected, board is displayed.
User makes the first move
The next agent will make the move
No
Is the game over?
Yes
If either player is winner, a green line is drawn to indicate the winner

Figure 1: Flow of control

2.2 Performance of the AI agent

Due to, the relatively small state space ($3^9 = 196839$) our agent can traverse the entire tree and converts the game into a completely deterministic one. Since we assumed the opponent to be playing using a strategy of minimizing our score, if this assumption is incorrect, the opponent can only make a move which gives us a higher score than the score achieved by the opponent making the assumed move. Thus, it can be proved that this AI agent is unbeatable , and each game can only result in AI winning or a draw.

2.3 Changing the difficulty of the game

As stated above, when our AI agent traverses the entire tree, it is unbeatable. However, instead of traversing the entire tree, we can alter our program to only traverse the game tree up to a fixed, finite depth only. When we reach that depth, we can either give all non-leaf nodes the same score of 0 or evaluate them using a heuristic. If we have reached a leaf node, their score according to the previous rules will be returned. In our program, we have gone with the former approach. The options for lesser difficulty alter the program to only traverse the tree up to a limited depth.

Difficulty	Maximum Depth
1	2
2	3
3	4
4	5
Max	-1

Table 1: Mapping of difficulty to maximum depth for 3 by 3

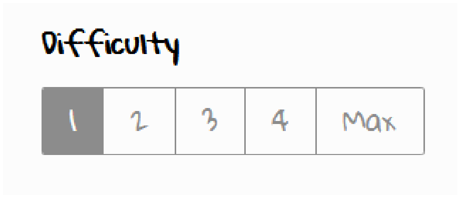

Figure 2: Difficulty option from the game

3 NxN tic tac toe

We tried a variation of tic tac toe where we modified the board to sizes of 4, 5, and 6. We modified the rules to state that the first player to place ‘n’ noughts or crosses in a row wins. All other rules remain the same.

3.1 Algorithm

We followed the minimax algorithm since the game essentially remains the same with only the game space increasing. Since the size of the game space increases significantly, it is apparent that we can no longer traverse the entire tree for bigger boards.

Board size	Game space
3 by 3	$3^9 = 19,683$
4 by 4	$3^16 = 4,30,46,721$
5 by 5	$3^25 = 8,47,28,86,09,443$
6 by 6	$3^36 = 1,50,09,46,35,29,69,99,121$

Table 2: Increase of game space with board size

3.2 Optimizations

3.2.1 Pruning

The first optimization that can be implemented is pruning the recursion tree. Pruning is a technique in machine learning and search algorithms that reduces the size of decision trees by removing sections of the tree that provide little power to classify instances. Pruning reduces the complexity of the final classifier, and hence improves predictive accuracy by the reduction of over-fitting. [2] Our algorithm uses a pruning technique called alpha-beta pruning. Alpha-beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, etc.). It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further.

When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

3.2.2 Memoization

Memoization is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. [3] If we hashed each game state to the best move, we can save some computations by not having to traverse the tree for repeated moves. However, since we have implemented alpha-beta pruning, we have to heed the fact that some nodes of the tree have not been explored because when we reached a particular node and had a better score stored in the alpha-beta variables, that node was not explored. Thus the hashed values may not be true and can be only considered if we have the same alpha-beta values as we had for the previous computation of the board state. Furthermore, the depth at which we are in the tree is also significant. We might encounter the same board state at different depths in the tree and it might have different answers according to the depth that we are at. Therefore, to implement memoization, we need to hash not only the board state but the alpha-beta values and the depth too. The effect of this is two-fold. The complications in implementing memoization are increased and the effect that memoization has on reducing the time complexity is also reduced since now, we have to check to see not only whether the same board state has occurred previously but whether the same board state along with the same alpha-beta and depth values has occurred previously. Due to these two reasons, we decided to not implement the memoization in our program, but the task can be taken up as a part of future work.

3.3 Efficiency of the Algorithm

It was noticed that even after implementing pruning, the algorithm if applied to boards of size bigger than three, gave a timeout error while running in the browser. Therefore, it was necessary to limit the search only up to certain depths for boards of bigger size.Thus, our artificial intelligence agent is not unbeatable for boards of higher sizes.The performance however could be further optimised by trying out various heuristics to rank the nodes upon reaching the maximum depth. In our program we give all nodes a score of 0 unless they are a leaf node. We have used the following maximum difficulty to depth mapping in our program.

Board size	Maximum depth for max difficulty
3 by 3	-1
4 by 4	9
5 by 5	7
6 by 6	5

Table 3: Max difficulty to maximum depth mapping for different board sizes

4 Ultimate Tic Tac Toe

Ultimate tic-tac-toe is a board game composed of 9 tic-tac-toe boards arranged in a 3 by 3 grid. Players take turns playing in the smaller tic-tac-toe boards until one of them wins in the larger tic-tac-toe board. Compared to traditional tic-tac-toe, strategy in this game is conceptually more difficult and has proven to be more challenging for computers. [4]

Rules :

1. Each small 3 by 3 tic-tac-toe board is referred to as a local board, and the larger 3 by 3 board is referred to as the global board.The game starts with 'X' playing wherever they want in any of the 81 empty spots. This move "sends" their opponent to its relative location. For example, if 'X' played in the top right square of their local board, then 'O' needs to play next in the local board at the top right of the global board. 'O' can then play in any one of the nine available spots in that local board, each move sending 'X' to a different local board.

2. If a move is played so that it is to win a local board by the rules of normal tic-tac-toe, then the entire local board is marked as a victory for the player in the global board.Once a local board is won by a player or it is filled completely, 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.

3. The first player to win 3 local boards in the same row, column or diagonal in the global board wins.

4. If any move "sends" the player to a local board which is already won or tied, the player must make a move in the sequentially next available local board.

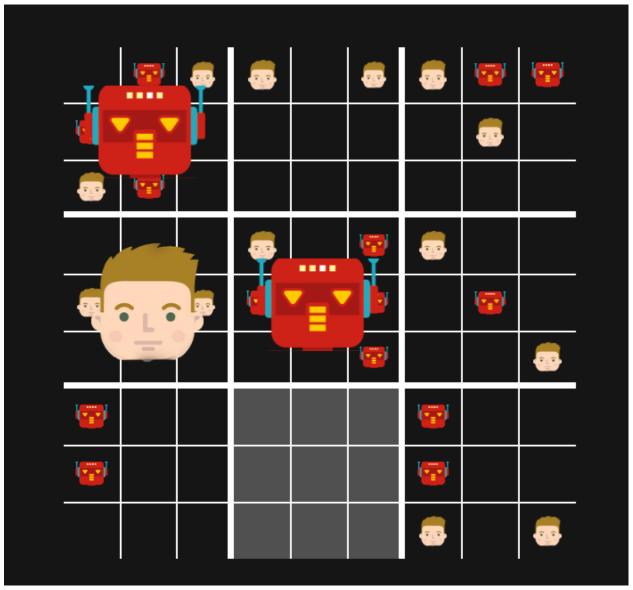

Figure 3: An incomplete ultimate tic tac toe board

4.1 Algorithm

Ultimate Tic Tac Toe is still a simple strategy, zero sum game. Therefore, the same strategy of making a tree and traversing it can be followed. However the game space is very huge and it is impossible to traverse the entire tree.Therefore we are forced to limit the recursion to a certain pre-decided depth. So we have to use a heuristic to decide the score associated with each board state once we reach a particular depth. The heuristic we have used gives the board state a score of

$$\begin{aligned} score = 100 \times ((&number\ of\ local\ boards\ won\ by\ AI\ agent) \\ &- (number\ of\ local\ boards\ lost\ by\ AI\ agent)) \end{aligned}$$

Difficulty	Maximum Depth
1	1
2	3
3	4
4	5
Max	8

Table 4: Max difficulty to maximum depth mapping for different board sizes

4.2 Performance

The algorithm is able to traverse the tree up to a depth of thirteen. The browser starts giving a timeout error for depth 14 onward. The performance of the AI agent was measured by how many intelligent agents the AI was able to beat. The performance of the agent at difficulty level other than max is sub optimal since the search is limited to fairly low depths. At maximum difficulty the search is limited to a maximum depth of eight. At maximum difficulty the AI agent was found to beat or tie with most of the intelligent agents it was tested against.

5 Software

We used Javascript for coding the algorithm. We used HTML and CSS to render the pages. The framework for the HTML and CSS files was taken from **this** open source project on github. We used github to collaborate and github pages for hosting our web app **here** . We used Microsoft Visual Studio code to code in Javascript.

6 Future work

In the future we would like to improve the processing speed to be able to play at higher depths by :
implementing memoization along with alpha beta pruning,
using better evaluation functions to order the nodes of the tree in such a manner that alpha beta pruning is more effective. (Move Ordering Alpha-Beta Pruning)
We also aim to make the app fully compatible with Smartphones.
We would, also like to obtain statistical data to test and analyse the performance of the algorithm at different depths and using different heuristics. This can be done by first using an agent who plays randomly and then using an intelligent agent.
We would also like to compare performances of an agent trained using neural networks and our algorithm based agent.
Furthermore, we would also like to look at a concise mathematical proof of why the minimax algorithm for 3 by 3 tic tac toe is unbeatable.

References

[1] *Minimax Algorithm*, available **here**

[2] *Decision tree pruning*, available **here**

[3] *Memoization*, available **here**

[4] *Ultimate tic tac toe*, available **here**