Literature Review

Shevtekar, Malpe, and Bhaila[1], discuss the merits of using minimax to backtrack through a game tree to model a 2 player game and touch upon its failings including redundant iterations and unlikely situations. We adapted this approach to create our adversary for Tic-Tac-Toe.

The website [2] discusses the importance of adding a depth modifier to a Tic-Tac-Toe MiniMax algorithm to prevent "fatalistic" decisions, in which a naive algorithm chooses a decision that does not prolong the game. We adapted this to our minimax algorithm to favor moves at shallower depths.

Introduction

For this project, our group wanted to create an adversary for Tic-Tac-Toe on both standard and non-standard board: this entails both the classic 3x3 board, but also variants like a 4x4 board. While Tic-Tac-Toe is a fairly simple game, implementing our adversary for these larger boards requires significant optimizations to our base code for a 3x3 board. In addition, we wanted our adversary to play optimally, where we define optimal as picking the best move that progresses the adversary to a winning board state the fastest. Solving these problems would improve computational efficiency, which becomes significant as the number of possible board states increases by O(3^(nxn)) for an nxn board. Hence, this solution would allow us to scale to larger boards while preserving reasonable computational efficiency and runtime. Finally, we wanted to experiment with the difficulty of the adversary and other GUI options.

Initial Goal and Outcome

Our main goal was to create the perfect adversary such that a game would always result in a win or a tie. To this end, we were partially successful in achieving our goal and our adversary will always tie with a perfect player for the 3x3 case. Furthermore, we are able to guarantee that the adversary is optimal in choosing the best move, taking the shortest path to its final board state. We also wanted to be able to scale up for nxn boards. For this, we struggled with finding a reasonable runtime for the 4x4 case. A secondary goal was to create an easy-to-use user interface to play the game. We were able to achieve this with a Gui that displays the game and allows the user to change the difficulty of the adversary, choose to start as either 'x' or 'o', and play on game boards of different sizes.

Methodology

For the game board, we created a class that acts as a model and contains all the necessary basic methods for facilitating a tic-tac-toe game, including creating an empty 3x3 board, placing the X's and O's on the board, ensuring they are placed on empty spots in the board, checking for

wins for each player in each row, column, and diagonal, and checking if the game has terminated.

We used Pygame to implement the UI. We used Pygame to draw the board and the positions of the letters on the board. The UI lets the user select X or O and the adversary difficulty level. We display a start button, visual indicators for whose turn it is, and indicate when the game is over. Finally, we display the winner.

For the algorithm, we used minimax to model and select maximal board states. We created a score function that would return 1 (X winning final board), 0 (a drawing final board), and -1 (O winning final board). We adjusted for depth so that solutions that achieve a winning state in fewer turns are preferred. We generated every possible board state from a given board state every minimax iteration because it's computationally feasible for Tic-Tac-Toe.

We intended to follow the model-view-controller paradigm to ensure our project is easy to debug and potentially expand our solution, but because of how interconnected the controller and UI became we condensed it all into one file. The controller manages our game logic, handles the user and AI's inputs, keeps track of the current player, handles mouse clicks and user input, and updates both the visual game state and model based on those results.

Our Solution

We used a standard minimax approach to create our adversary. Our implementation of minimax would generate all possible subsequent board states given a board state and include a depth modifier to favor solutions found at shallower depths. Each board is scored as follows: +100 to winning board states, +0 to tied states, and -100 to losing board states. The algorithm would recursively iterate through the game tree, alternating between maximizing and minimizing nodes and propagating values back up the tree. Our depth modifier would increase by +1 per minimax iteration and, upon reaching a final board state, subtract its total depth to the score if it is maximizing and add if it is minimizing. In addition, we include a depth limit to preemptively stop searching once that depth is reached.

For our 4x4 board, as mentioned above, we struggled significantly with runtime due to the (nxn)! increase in the number of possible game states. In particular, this made the runtime for finding the initial best moves very lengthy but would be reasonable after the first few turns. To address this, we created a heuristic for our larger boards that would look for empty adjacent spaces to a marked tile. These spaces would then be appended to an array and a space from that array would be randomly returned. In the event of an empty board or a marked tile with no empty adjacent spaces, a random tile would be returned. This heuristic would allow the algorithm to place subsequent moves relative to a marked tile and skip over searching the game tree for the first few moves.

Our solution works well for a 3x3 board. Since the 3x3 board space has a relatively small amount of possible game states, the algorithm is able to search the entire game tree and return the best board state. Further, with our depth modifier, we can guarantee the optimal move as well. Hence, our adversary is guaranteed to either win against or tie with a human player for a 3x3 board. However, our minimax solution is unable to generalize to 4x4 and larger boards due to the o(n^2) increase in runtime for these larger boards. Our heuristic attempts to address this but fails to guarantee the best or optimal move since it no longer uses minimax most of the time. Due to this, our main future improvement would be to implement pruning.

Results:

We ran 100 trials using our heuristic on a 4x4 board to examine runtime (quantified by highest move time) and win rate against a random player (player that chooses moves randomly) as a function of different heuristic/breadth limits and depth limits. We found that increasing the heuristic limit and increasing the depth limit significantly increased the highest move time. Notably, at a depth limit of 17, a heuristic limit of 6 was two orders of magnitude faster than a heuristic limit of 8, which itself was two orders of magnitude faster than a heuristic limit of 10. While, for a heuristic limit of 12, a depth limit of 6 was significantly faster than a depth limit of 8, 10, and 17. Finally, we examined our win rates against a random player and found a higher heuristic limit correlated with a higher win rate, which aligns with our expectations as below that limit we would run minimax that is guaranteed to find the optimal move. We also modified the breadth limit and examined its effect on win rate against a random opponent. We found the win rate would increase to a point before tapering off again, with a sweet spot around a limit of 10.

Figure 1:

Free Space Lim (Breadth)	Depth Lim	highest Move Time	Wins/100 Against RDM
12	17	>60	N/A
12	10	>60	N/A
12	8	>60	N/A
10	16	19.46014214	72
12	6	9.152496099	71
8	17	0.3214011192	68
10	17	20.69679403	67
9	15	0.3120071888	64
10	15	20.46193194	62
6	17	0.008286952972	59
12	5	1.463224173	58

15	3	0.04261517525	58
15	5	4.525607109	57
8	15	0.2932941914	56
4	17	0.0004518032074	53

Figure 2:

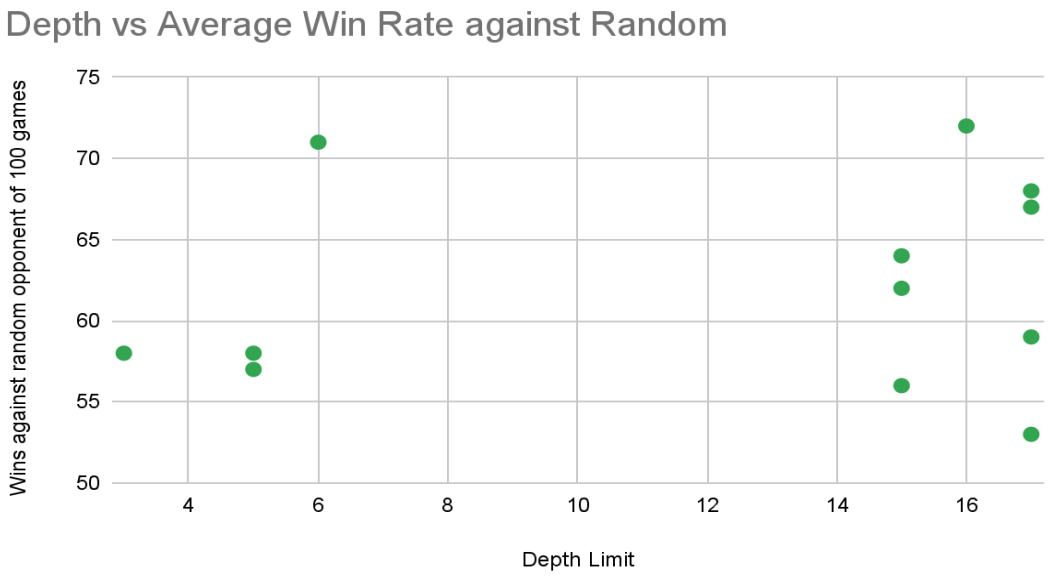
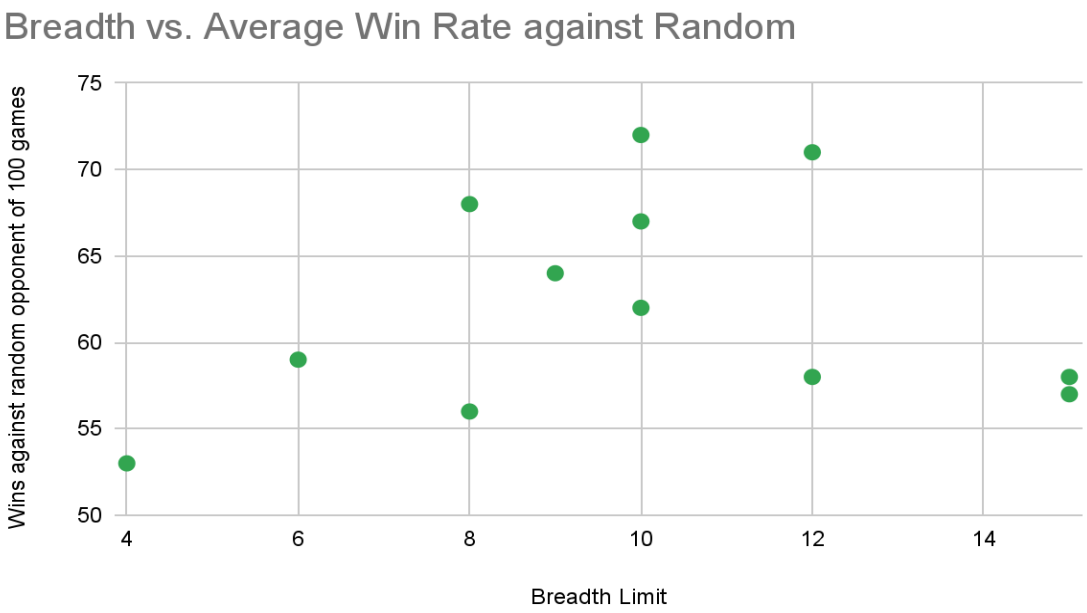
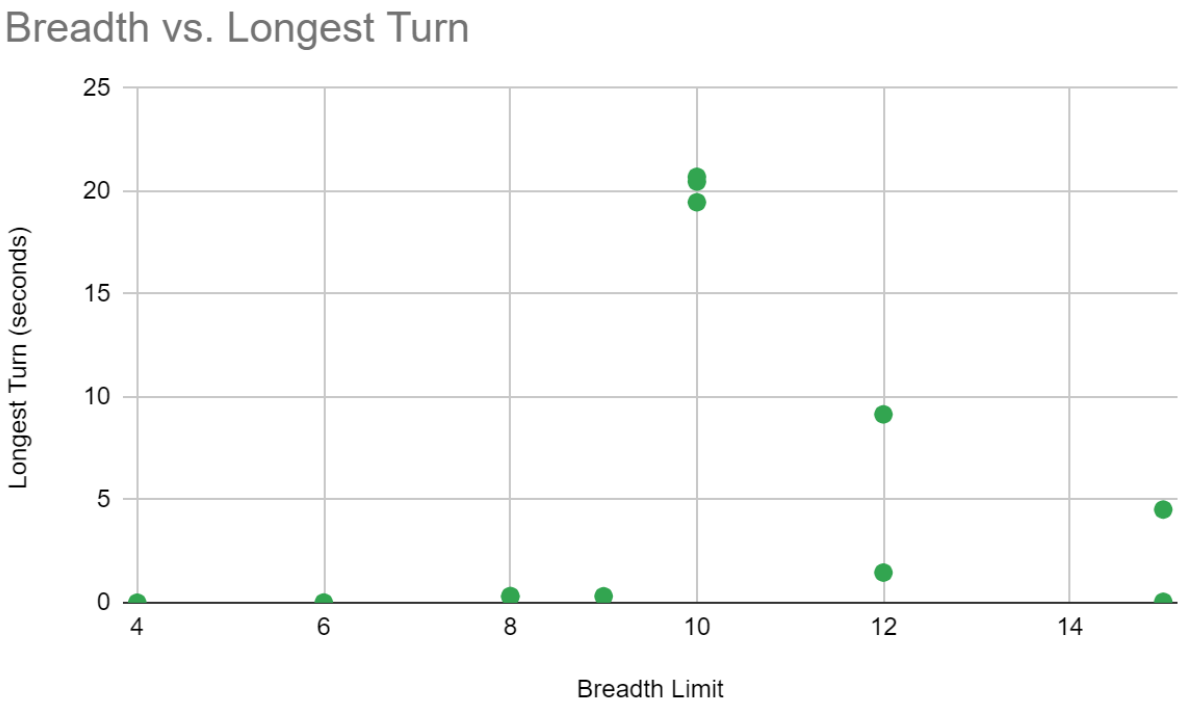
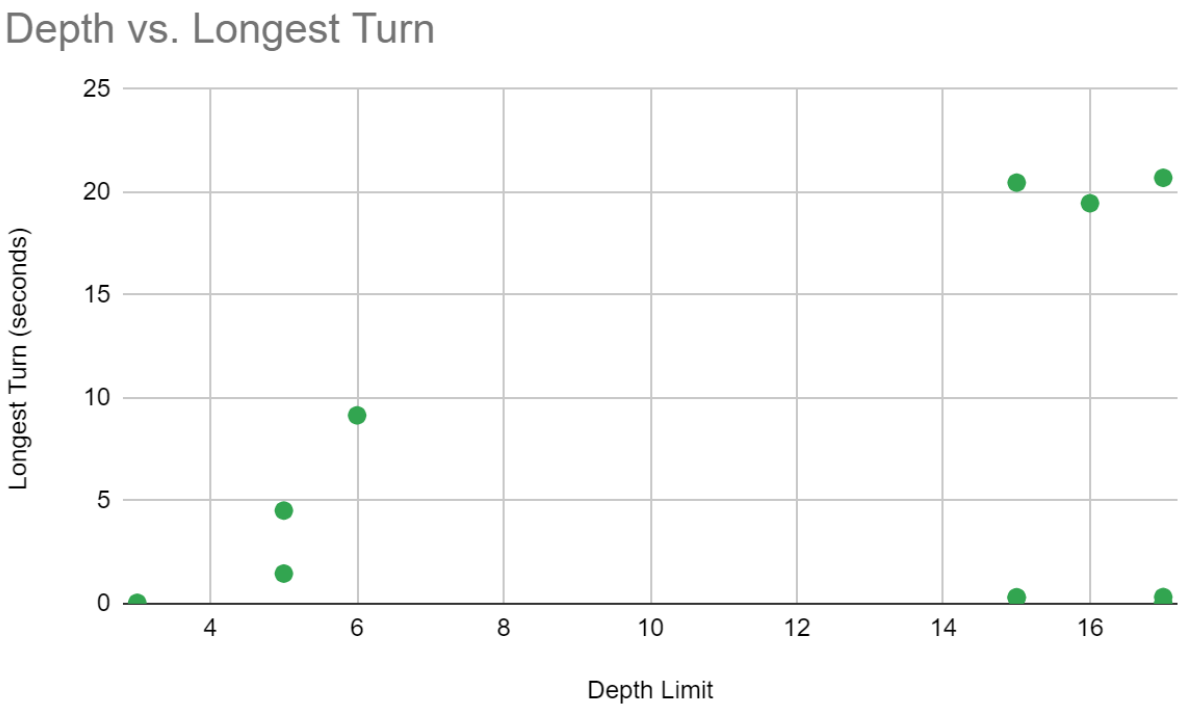


Figure 3:


Breadth vs. Average Win Rate against Random
Wins against random opponent of 100 games
75
70
65
60
55
50
4
6
8
10
12
14
Breadth Limit





Bibliography:

1) Shevtekar, S. S., Malpe, M., & Bhaila, M. (2022). Analysis of game tree search algorithms using Minimax algorithm and alpha-beta pruning. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 328–333. https://doi.org/10.32628/cseit1228644
2) Fox, J. (2021, November 10). *Tic Tac Toe: Understanding the minimax algorithm*. Never Stop Building - Crafting Wood with Japanese Techniques. https://www.neverstopbuilding.com/blog/minimax

Contributions:

Rida: minimax algorithm, board, evaluation
George: minimax algorithm, evaluation
Samirah: report, poster (slides)
David: report, GUI, controller
Kyle: report