Evalutaion of AI Agents in Ultimate Tic-Tac-Toe

Miguel Alonso
Virginia Tech

Nicholas Raines
Virginia Tech

Pierre Sarabamoun
Virginia Tech

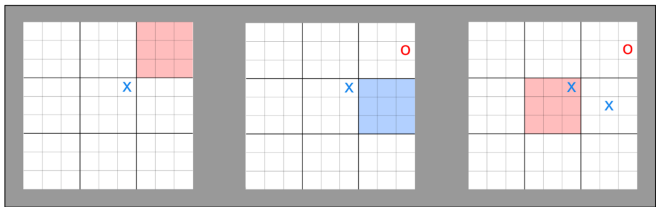

Fig. 1: Gameplay Progression

I. INTRODUCTION

A. Research Problem

We want to create an environment and AI agent to play ultimate tic-tac-toe (UTTT). Unlike tic-tac-toe which is a trivial game that is easily played game-theory optimally, UTTT complicates the game significantly. UTTT's game board is similar to a 3x3 tic-tac-toe board, but each cell of the 3x3 board is another 3x3 tic-tac-toe game, the winner of which decides whether the outer board is an 'X' or an 'O'. Additionally, when a player places an 'X' or an 'O' in one of the inner boards, the next player's move has to be in the corresponding cell of the outer board. For example, if a player places an 'X' in the top-right square of the inner board located at the middle of the outer board, then the next player would have to place an 'O' somewhere in the outer board's top-right square (Figure 1). If this would redirect the player to a board that has already been completed then the player is allowed to place their next move in any of the inner squares that they would like. The following link provides a visual explanation of the rules of the game.

II. APPROACH

The project was segmented into three portions, environment setup, user and AI player setup, and more AI player and UI implementation. These portions were split up and assigned one to each member of our group.

A. Enviroment Setup

The environment setup consisted of creating a game loop that executed on a well tested UTTT class. This UTTT class needed to represent a current game state with the ability to modify the current state or generate a potential other state. The environment itself contained information about the state including:

Var Name	Var Type
game_grid	2D 9x9 Array
simplified_grid	9 Element Array
active_board	Int
valid_moves	Array of Tuples
winner	Int
player_turn	Int

Fig. 2: Game State Variables

- game_grid: An array of arrays, each of which represents one board on the full game grid. Each spot in the array contains a 0 if empty or the player ID of the player that moved in that square (1 or 2 in this case)
- simplified_grid: An array showing the winner of each board (0 if not won and stores the player ID if it is won by that player)
- active_board: The board the current player is restricted to play in (None if they can play anywhere)
- valid_moves : An array of all moves the current player is allowed to play according to the rules of the game (elements are board,cell tuples)
- winner: 0 if on going, -1 if draw, or player ID of player that won
- player_turn: The player ID of the player to act

Fig. 3: Game State Descriptions

While there were helper functions implemented the only ones used by future parts were "make_move()" and "generate_next_uttt()" which served similar functions but the former modified the current game state while the latter returned a new deep copy game state. Once that was completed a game loop was implemented that kept making random moves until "winner" was non-zero and returned the winner as well as the final game state.

B. UI

This portion of the project began with very much needed visual upgrades to the program. Firstly, up until this point, all game states were outputted to the terminal in an ASCII representation of the game grid as well as displaying your valid moves. The UI that was setup made it so that when the code was executed a window opens up with the game grid displayed and constantly updating based on the moves played. The "active board" became highlighted so the user could visually see what moves are considered valid instead of having to read through the printed "valid moves" array. When a player won a board that board was highlighted in a color unique to that character so the game state was easy to understand at a single glance. Finally, user inputs were adjusted to be taken by having the user interact with the UI by clicking on the desired square instead of having to type

the board and cell he wanted to make the marking in which was by far the largest quality of life change implemented to our code.

III. AGENTS

User and AI Agent Setup began with the creation of the "Player" class which is a parent class with a "make_move" function. From this parent class all supported player classes inherit its structure. Three agents were created during this part of the project, they were "RandomPlayer", "UserPlayer", and "MonteCarloPlayer" each of which with their respective make_move function. The game loop was then modified to take in command line parameters on the execution of the code to determine which players would be pitted against each other, this also added the functionality of the user pinning two non-user agents against each other to see how the game would pan out.

A. Random Agent

The RandomPlayer's make_move simply went through all valid moves and returned a random index as the move they wanted to make. The UserPlayer's make_move prompted the user through the terminal to choose one of the valid moves to make. Finally, the first real AI agent, MonteCarloPlayer, was a bit more complex.

B. Monte Carlo Agent

The MonteCarloPlayer object took in an additional parameter from the user in the command line to determine how many games would be simulated. This is because the way the MonteCarlo algorithm works is by simulating X number of games of random moves from each of your possible moves and marking how many of the X simulations you have won, whichever valid moves yields the highest number of wins is the move the agent decides on. So the MonteCarloPlayer class uses two helper functions to implement it's make_move function but successfully runs the MonteCarlo algorithm with the instructed number of simulated games.

C. MiniMax Agent

An additional player was implemented to support another AI agent, the "MiniMaxPlayer," with Alpha-Beta pruning. This player's make_move used a heuristic function to evaluate all potential game states and score them, having the player make whichever move would yield the game state with the highest score. The heuristic took into account many different board configurations to determine which move was the best. It can be broken down into two parts: the current individual board and the larger ultimate board state. Firstly, the current board awards different values according to the player's value. For example, if the player blocks two in a row from the opposing player, this will be rewarded higher than a move that is placed to the side, creating no additional benefit. Thereafter, the larger game state will be considered in a similar fashion to the current board. Where an emphasis is placed on winning boards and setting up winning conditions. These can be viewed in player.py under the functions uttt_heuristic, board_heuristic, and row_heuristic. Our

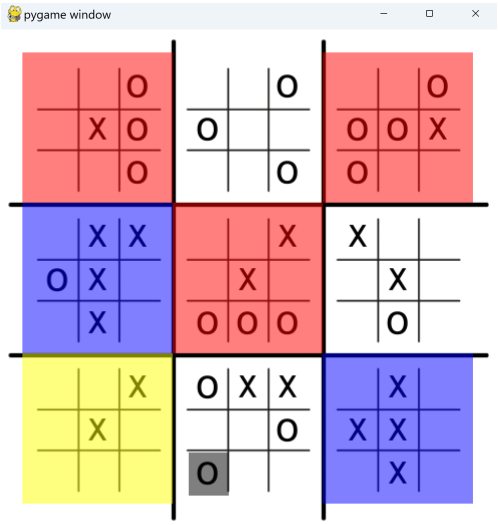

Fig. 4: Video Deduplication

implementation uses alpha-beta pruning to reduce runtime while providing smart moves. We tested different max depth values and determined the optimal value that balances smart moves without sacrificing the user experience.

IV. RESULTS

The first part of our results is the UI and how the user interfaces with it. The UI displays the current game state, showing the whole grid, and highlights the boards that you have won in a color unique to you and the boards your opponent has one in a color unique to them. It also highlights the grid in which the next move MUST be played in as yellow. The UI also highlights the previous move in a grey box for clarity. If the user is a player in the game they just need to click on a cell to make their move in it. Below is a screenshot of our UI in the middle of a game.

The most important part of our results however is the performance of our two AI-agents, the Monte Carlo agent and the Minimax agent. Attached at the end of the report are the results (2 sig figs) of sets of 250 games that were played in the corresponding player combinations. MC## means Monte Carlo agent simulating ## number of games for each potential move.

The first thing we tested was whether or not there was an advantage to going first, to do that we pinned an AI agent against itself and found that P1 had a 40% chance of winning as opposed to P2's 32%, this is a sizable advantage. Our next step was comparing every agent to the random agent (giving the random agent P1 advantage). From that we found that the Monte Carlo agents won a VERY strong majority of the time (25 sims $\rightarrow$ 95%, 100&250 sims $\rightarrow$ 99%). The Minimax agent however did not show the results we were hoping for when pinned against the random agent (winning only 80% with P2 disadvantage into random agent).

The interesting thing about the Minimax agent however, is that even though it performed worse than expected against the random agent, it outperformed the Monte Carlo 25 sim

agent (winning 63% as P1 instead of the expected 40% and winning 51% as P2 instead of the expected 32%). Once the number of simulations for the Monte Carlo agent go up to or above 100 however, it starts consistently outperforming the Minimax agent.

V. FUTURE WORK

This project can be expanded upon in many ways. The core mechanic of our project has been the evaluation of our AI agents, such as random, minimax, and Monte Carlo. These agents can be expanded upon to improve their performance. This includes improving heuristics, tuning hyperparameters (max search depth), and differing evaluation criteria. Additionally, there are other agents we could implement, such as expectimax search. When using our Monte Carlo agent, the user must specify how many simulations should run. This can be done proactively throughout a game to vary the agent's ability to make moves. Finally, the heuristics for our minimax agent can always be improved due to the vast number of considerations that need to be calculated.

Further, the user interface can be improved to allow for a better user experience. Functionality such as replaying moves, games, and backtracking can be implemented to better assess our AI's performance.

VI. RUNNING INSTRUCTIONS

To run our Ultimate Tic Tac Toe Implementation there are a couple requirements such as python 3.10 and installing all dependencies (eg. pygame). We have created a simple user interface to interact with our game. To run our application:

A. Usage

python game.py [-h] [-p1 PLAYER1] [-p2 PLAYER2] [-window DISPLAY_WINDOW] [-games GAME_COUNT] [-delay DELAY]

- -p1: Player 1 Agent (player, random...)
- -p2: Player 2 Agent (player, random...)
- -window: Display GUI (True or False)
- -games: Game Iterations (Number)
- -delay: GUI Update Delay in ms (Number)

Possible Agents include player, random, minimax, monte-carlo#. The # indicates the number of simulations to run such as monte-carlo10, monte-carlo100.... When increase the number of simulations there is an impact on performance speed.

Example Invocation:

- *python game.py -p1 player -p2 minimax -window True -games 1 -delay 100*

This indicates player 1 will be the user, player 2 will be our minimax agent, it will run one game, and there will be an update delay of 100ms.

Our implementation has been uploaded to a public repository and can be found at:

- https://github.com/NickRaines/UltimateTicTacToe.

Players	Player 1 Win	Draw	Player 2 Win
MC 100 vs MC 100	99 (40%)	70 (28%)	81 (32%)
Random vs MC 25	6 (2.4%)	8 (3.2%)	236 (94%)
Random vs MC 100	1 (0.4%)	1 (0.4%)	248 (98%)
Random vs MC 250	1 (0.4%)	1 (0.4%)	248 (98%)
Random vs Minimax	32 (12%)	18 (7.2%)	200 (80%)
MC 25 vs MC 100	56 (22%)	58 (23%)	136 (55%)
MC 25 vs MC 250	42 (18%)	53 (20%)	155 (62%)
MC 100 vs MC 250	82 (33%)	70 (28%)	98 (39%)
MC 25 vs Minimax	113 (45%)	9 (3.6%)	128 (51%)
Minimax vs MC 25	157 (63%)	9 (3.6%)	84 (34%)
MC 100 vs Minimax	153 (61%)	12 (4.8%)	85 (34%)
Minimax vs MC 100	112 (45%)	17 (6.8%)	121 (48%)
MC 250 vs Minimax	168 (67%)	6 (2.4%)	76 (30%)
Minimax vs MC 250	103 (41%)	10 (4.0%)	137 (55%)

Fig. 5: Simulation Results

VII. LESSONS LEARNED

An important lesson we learned throughout this project is the value of planning. It's vital to take the time to carefully plan out each milestone, timeline, coordination, and goal. Without proper planning, it's very easy for a team to lose track of time and provide sub-optimal results. A well-thought-out plan ensures the team hits milestones while staying motivated throughout the project. After initial brainstorming, our team scoped our project with key deliverables that were attainable within our timeframe. We coordinated our efforts so that each team member was not blocked by another member's progress. These tasks were split to focus on the strengths of each individual because no project can be successfully completed without the input and effort of all members. Working together, each member brought their unique skills and knowledge to the project. This increased productivity was due to each member being familiar with their tasks. The last key to our success was clear communication. It was essential that we openly communicate our progress and troubles so that we could help each other if needed. Overall, this project provided many valuable lessons that can be applied to future projects.

REFERENCES

[1] Powell, Scott, and Alex Merrill. "ULTIMATE TIC-TAC-TOE." (2021).
[2] CHEN, PHIL, JESSE DOAN, and EDWARD XU. "AI AGENTS FOR ULTIMATE TIC-TAC-TOE." (2018).
[3] Siegel, Sebastian. "Training an artificial neural network to play tic-tac-toe." ECE 539 Term Project (2001).