Unbeatable Tic-Tac-Toe: Applying AI to Virtual Gaming

CS51: Final Project

Kristina Hu
Hana Kim
Alexander Solis
Stephen Xi

Demo Video

http://youtu.be/frbVzduyXJ8

Overview

We implemented the Minimax algorithm to create an unbeatable AI for TicTacToe where a player will only ever lose or at best tie — all necessary files are contained within a folder called “submission”. Once inside, another folder called “code” contains a README file for detailed instructions on running our code. The user can play the game directly from the terminal window, where the board and additional instructions are displayed. To run the program, cd into the directory containing the file tictactoe.py and run from the command line: python tictactoe.py. A classic game of TicTacToe results, with alternating turns between the user and AI.

The Minimax algorithm works by calculating a score for each index in the array of all possible moves for a given player. Wins (for the AI) are given a positive value, losses a negative value, and draws a value of 0. The algorithm traverses each open position and simulates all possible outcomes, adding up the values of these outcomes. Because the AI wants to optimize its move against the user (i.e. the AI wants to win or draw), it selects the largest score from the array generated by the algorithm. We account for different game states by adding a depth variable, which calculates the total number of moves for a given game simulation. Games that result in a win earlier in the game are weighted more heavily than games that take more moves to win.

Planning

We planned to fully implement the Minimax algorithm for TicTacToe, adding additional features if time permitted.

Please see our Report Folder inside the submission folder for our annotated draft and final specifications.

Although our implementation has evolved since the draft spec, we were able to achieve all of our major milestones for the project. Initial planning for the project went well — we set very reasonable deadlines for our milestones and modularized the project accordingly

to accomplish those milestones. We also identified our priorities for the project, clearly differentiating realistic goals from “reach” goals.

As for our specific milestones, we successfully implemented the Minimax algorithm, which is able to optimize the AI’s decisions by always resulting in a win or draw for the computer. We also implemented a Board class and accompanying helper functions to facilitate user interaction with the board. Additionally, we were able to include enough abstraction in our code so that it could easily be applied to more complex games of perfect information, such as Connect Four. That being said, if we had more time we would have implemented Connect Four using the outline we created with Tic-Tac-Toe. Although we would have used a list of lists rather than a dictionary to represent our board and would have reimplemented many of the functions differently due to the differing rules and board sizes of Connect Four, all of our signatures and modules would be easily applicable to the new game. Implementing Tic-Tac-Toe gave us an extensive understanding of the minimax algorithm and how to design an interface suitable for its application to a board game. Therefore, we could draw on these for Connect Four and only need to alter some of the function’s implementations to accommodate for the different game rules.

Design

Although we had some familiarity with Python, it was a largely new language for our group. Therefore, some of the big challenges of our project were learning how to apply Python to the Minimax algorithm as well as how to implement an interactive board. We decided to create a board class, complete with various methods that analyze configurations of the board. We also included a helper functions class which contains the minimax algorithm itself in addition to functions allowing the player and AI to make moves on the board. With regard to the minimax function, we created two separate methods: one private and one public. The private method calculates the scores for all possible combinations of moves in a given turn while the public method calculates the maximum of those scores.

We decided to contain the visual aspect of the board within the terminal window because creating a GUI would have been less relevant to the class curriculum and might have taken time and energy away from the actual implementation of the algorithm. Through the command line, the user can easily select a spot on the board by typing in the number associated with that spot.

We also added a separate text file called minimaxdata.txt — this contains the hard-coded scores for all possible moves for the AI on the first turn. We made this design choice to decrease the runtime of our program, as there are *many* more combinations of game configurations early on in the game. We load the data within this text file on the first move of the AI. This design choice was made in lieu of alpha-beta pruning since both achieve the same goal of decreasing runtime.

We found testing and debugging to be fairly straightforward for our project. Because our algorithm revolved around the quantification of possible moves, we debugged using print statements. In this way, we were able to look at the calculated scores of each move and track the progress of the algorithm. Beyond that, a lot of our tests simply involved playing games since our board was able to visualize the actions of the AI.

Implementation

Implementing the Minimax algorithm proved to be much more difficult than we anticipated. Although many online templates for our algorithm exist online, we found all of them to be unsatisfactory due to conflicts in design. In fact, our algorithm ended up very different from the original Minimax algorithm. The original Minimax algorithm states that only three different scores exist: 1 for a win, -1 for a loss, and 0 for a draw. We found this problematic because it failed to account for the depth of the game — a win 8 turns into a game is much different than a win 5 turns into a game. We instead used two methods to resolve this problem. First, we created a depth variable that added to the score as the game progressed. Next, we took the inverse of that score so that earlier scores would be weighted more heavily in relation to later scores. Thus we ended up implementing an algorithm that performed like Minimax but added an additional layer of optimization.

Reflection

This group project was an incredibly informative learning experience for all of our team's members. The most important thing we learned from it was how to work on a technical project as a group. While all of our members had completed CS50 partner projects, this was our first experience working in teams larger than two. Doing so involved much more organization and planning. In order to facilitate group work, we modularized our project into functions that could be worked on individually. Before programming, we collectively brainstormed all of the helper functions we imagined would be useful in the implementation of our tic-tac-toe program and minimax algorithm. We then divided up these helper functions equally amongst our members and worked on them independently. This system enabled us to each take ownership of our work and significantly increased our efficiency since we were able to work on different parts of the project simultaneously. Another aspect of group work we learned was using Git to manage our different code and openly communicating with one another to schedule meeting times and check in points.

Other learning experiences from this project include learning how to pick up a language from online documentation sources, extracting out common code, and translating an algorithm into an actual working program.

Although we outlined a design and implementation plan with our two specs, we came across unexpected turns that led to adaptations on our part. One of these surprises were the corner cases we came across in our minimax algorithm during extensive testing. These arose when the computer would chose to block the player's move despite having an available winning spot, essentially weighing blocking over winning. In order to fix this, we created a helper function is_about_to_win, which checked all of the cases where a player is about to win and then returns the move needed to win. We also had to account for the corner cases of a player choosing (1 and 9) and (3 and 7). These arrangements presented problems because the computer's first move was unable to determine if it should try to block or win. In order to solve this issue, we improved upon our minimax program by explicitly checking for these cases to ensure robustness.

One of the most pleasant surprises we encountered was how helpful it was to decide on and implement our helper functions first. Figuring out which helper functions would be useful for our actual board functions made implementing our board and minimax algorithm significantly more efficient as we had a slew of available tools to pull from. In also helped us to deconstruct the minimax understanding and fully flesh out our understanding of the algorithm. Furthermore, it allowed us to work independently on different functions which led to an incredibly more efficient work set up.

Advice for Future Students

We encourage future CS51 students to choose a clear application for their algorithm and to spend time at the very beginning really understanding how the algorithm works. We also suggest that they focus more on learning and implementing the algorithm rather than spending a significant amount of time on other features. We recommend this approach because the implementation of an algorithm is the most relevant portion of the project to the term's coursework and thus was by far the most rewarding aspect for us to accomplish. Overall, scheduling, setting appropriate goals, and focusing on the problems relevant to the coursework led to an incredibly rewarding and challenging learning experience.