Ultimate Tic Tac Toe

Summer Project 2014

(Programming Club)

By

Anand Singh Kunwar

Pranav Vaish

Saransh Srivastava

Acknowledgments

First and foremost, we would like to thank our project mentor, Chirag Gupta, for guiding us through this challenging project through his valuable suggestions and knowledge about the subject. Also, we express our gratitude towards the Programming Club coordinators for appreciating our idea and giving us the chance to work on it.

Contents

Introduction

This project aims to design an Artificial Intelligence (AI) for a turn based, two player game called Ultimate Tic Tac Toe. As must be obvious from the title, this game is inspired from the well-known Tic Tac Toe. We discovered this game on Quora and were gravitated towards this interesting modification of a solved, stale game. Many a time, one of us would run out of opponents to play with, which led us to this idea of developing a computer-based opponent for the game.

Jumping to the main point, the game is described as follows:

This is a modification of the famous Tic tac toe game. It is a turned based, two player, zero sum game. It contains 9 normal tic tac toe games arranged in a 3×3 grid, basically a tic tac toe board where every square is another tic tac toe game. The first player (say A) gets to make his mark anywhere on the 81 vacant positions. Depending on where A makes his mark in the small tic tac toe puzzle, the second player (say B) has to make his own mark in the corresponding position of the bigger grid. For example, if A makes his mark in the top left corner of any of the nine puzzles on the board, B has to make his mark somewhere in the puzzle at the top left corner of the big grid, and so on. Whoever wins any of the nine small puzzles, makes a bigger mark on the big puzzle. Also, if A marks a mark on say the center position of any small puzzle, and the puzzle at the center of the large grid has already been won, lost or drawn, B has a chance to make his mark wherever he wants. The objective of the game is to win the larger tic tac toe game by winning appropriate smaller ones.

Besides making the actual game, our objective was simple: Design an AI that is efficient, competent, challenging and fast enough in order to not be an annoyance. We looked a lot on the internet on the various techniques used for AI design, while thinking of methods of our own. We came across an implementation of this very game (with a little change of rules) on khanacademy (see references, page 9) from where we scraped up some ideas to implement. We also came across a standard theorem/algorithm for zero-sum two player games, the Minimax algorithm. We implemented the aforementioned algorithm along a method called ‘alpha-beta pruning’ to speed the process up.

Usage Details

The github repository for the project is split into three main branches: master, minimax and monte_carlo. The master branch was originally intended to contain just the game UI, while the minimax and monte_carlo branches would contain the two AI implementations. But it happened so that the minimax and monte_carlo branches are complete in themselves.

The python file in each branch is the game, for example, the file ‘minimax.py’ can be run to play versus the minimax bot. The ‘data’ folder contains the files used for I/O.

The Project

UI Design

Taking advice from our mentor, we decided that we will make the game UI in python, using the pygame module which can be easily learned and is adequately powerful for us to make a satisfactory UI.

For storing the state of the game, we used a simple four dimensional matrix, where the first two indices would denote the position of the grid in the main grid, and the rest two indices would denote the position within that grid. So, (1, 1, 1, 1) would correspond to exactly the middle square on the board.

We made two functions that would check the state of the game and make appropriate actions. The check_winner function takes a normal tic tac toe grid and returns the winner, if any. The check_win_main function does the same thing for the main grid.

We created the images for the UI elements of the game viz. the symbols (Xs and Os) and the board ourselves using Adobe CS. We imported them into the python program using pygame methods.

When this was all done, the two player game was ready and playable.

Monte Carlo

This is the first AI implementation. We stumbled upon the idea on khanacademy.com, in the work of Matt Dunn-Rankin (see bibliography). It plays as many random games as it can in a fixed amount of time and finally makes the move with the highest probability of winning.

First, we tried to do it in python itself. We used functions and methods in the ‘time’ module to account for the time taken for each move. The AI was the player ‘O’ and we used a function ‘monte_carlo’ to generate a list of all possible moves and then used a random function on that list for 1 second to explore all possible moves. It worked, but we discovered that the bot was not doing as well as we expected it to, due to python being slow and hence it being able to play very few games in the desired time limit.

We overcame this problem by coding our AI in C, and using files for I/O between C and python. For calling gcc from within python, we used the ‘subprocess’ module and used the ‘call’ function to execute external commands. We also studied basic python syntax and file handling from thenewboston.org (see references) in order to correctly integrate the game in python with the AI in C.

Shifting the AI code to C was relatively straightforward. We re-wrote all the functions in C and used the ‘time.h’ library for keeping track of the time. The performance of the approach improved dramatically by this, which was proved when we submitted this implementation on a hackerrank contest (see

references) where it stood at the 5th position amongst 187 submissions from across India (submission by the handle ‘vashious’).

Most of our time was spent on debugging and playing with the AI to make sure it was bug-free and making correct logical choices.

Minimax

Our mentor guided us and led us to study the minimax algorithm, a standard approach for AI design for zero-sum two player games. A zero-sum game is basically a game in which, say, if the score for a move made by a player is v, the score for the same move made by the opponent is (-v). So the sum of the scores remains zero, hence the name.

We started studying it from Wikipedia first, then moving on to other references we found on the internet (see references). We also had to study basic trees in order to understand the algorithm.

The minimax algorithm is best described in the Wikipedia page itself:

A **minimax algorithm** is a recursive algorithm for choosing the next move in an n-player game, usually a two-player game. A value is associated with each position or state of the game. This value is computed by means of a position evaluation function and it indicates how good it would be for a player to reach that position. The player then makes the move that maximizes the minimum value of the position resulting from the opponent's possible following moves. If it is **A**'s turn to move, **A** gives a value to each of his legal moves.

A possible allocation method consists in assigning a certain win for **A** as +1 and for **B** as −1. An alternative is using a rule that if the result of a move is an immediate win for **A** it is assigned positive infinity and, if it is an immediate win for **B**, negative infinity. The value to **A** of any other move is the minimum of the values resulting from each of **B**'s possible replies. For this reason, **A** is called the *maximizing player* and **B** is called the *minimizing player*, hence the name *minimax algorithm*. The above algorithm will assign a value of positive or negative infinity to any position since the value of every position will be the value of some final winning or losing position. Often this is generally only possible at the very end of complicated games such as chess or go, since it is not computationally feasible to look ahead as far as the completion of the game, except towards the end, and instead positions are given finite values as estimates of the degree of belief that they will lead to a win for one player or another.

This can be extended if we can supply a heuristic evaluation function which gives values to non-final game states without considering all possible following complete sequences. We can then limit the minimax algorithm to look only at a certain number of moves ahead. This number is called the "look-

ahead", measured in "plies". The algorithm can be thought of as exploring the nodes of a *game tree*. The *effective branching factor* of the tree is the average number of children of each node (i.e., the average number of legal moves in a position). The number of nodes to be explored usually increases exponentially with the number of plies. The number of nodes to be explored for the analysis of a game is therefore approximately the branching factor raised to the power of the number of plies. It is therefore impractical to completely analyze games such as chess using the minimax algorithm.

The performance of the naïve minimax algorithm may be improved dramatically, without affecting the result, by the use of alpha-beta pruning. Other heuristic pruning methods can also be used, but not all of them are guaranteed to give the same result as the un-pruned search.

(Courtesy Wikipedia)

It took a considerable amount of time to understand this recursive algorithm and then understand the alpha-beta pruning part of it, which was quite necessary for our project as the game tree was quite large.

We started by implementing this algorithm on a standard tic tac toe game, just for our understanding. We found a simple heuristic evaluation function for the standard tic tac toe on the internet and implemented the algorithm based on the same function. The point of using a heuristic evaluation function and not just an end-game evaluation function was to avoid the look-up of the entire game. So, we added a depth variable to the minimax function which made sure that we didn't end up searching through the entire game tree.

We then moved on to implementing the algorithm on the Ultimate Tic Tac Toe. After the standard tic tac toe implementation, it was just a scaled-up task for us to do. We used a structure to store the game tree, with every node storing its heuristic value and the pointer to its child. We used a modified version of the evaluation function that we employed in the standard tic tac toe. The evaluation function is:

Evaluate all the 8 possible lines (three rows, three columns and two diagonals) independently. Score them as follows:

100 (or -100) for each line with three 'X's (or 'O's)

10 (or -10) for each line with two 'X's (or 'O's) and an empty square

1 (or -1) for each line with one 'X' (or 'O') and two empty squares

0 for each line with a tied square

0 otherwise.

We used the check_empty function to populate an array of all possible moves and finally used the minimax function to look up the best possible move for the game, setting the depth target (lookahead) to 10 moves.

Our first implementation was with numerous bugs and we took about a week to fix all of them. Then, we just had to integrate the C code with the python game, which was done without hiccups.

Shortcomings to Work On

While the UI is aesthetically simple, it sometimes becomes hard to keep track of the grid you have to play in. An indication of the square to play in would be helpful.

Overall UI improvements and heads-up display elements like score, current player etc. would make the game more presentable.

The minimax lookahead could work better if it were dynamic, according to the state of the game.

Conclusion

This project led us to learn quite a few things. Starting with python programming and using pygame to make simple games, we learned about object oriented programming, lists, trees and several standard game theory algorithms. Also, it was fun playing the game against the AI, forming our own strategies, as it was the game itself that grabbed our attention at first.

Future Work

Future work that could be done on this project is quite a lot, apart from the UI improvements already mentioned. This game could be ported to make an Android or facebook app. Also, as our mentor suggested, better techniques involving machine learning could be used to generate much more effective heuristic evaluation functions.

The game could also be extended to more than two players, as the minimax algorithm is extensible to n players.

Another possible idea is that the game could be changed to something even more recursive, as in even more puzzles inside the smaller puzzle. The players can zoom in to play inside the inner grids.

References

Github repository - https://github.com/anandsinghkunwar/pransa-tictactoe/

Khanacademy project - https://www.khanacademy.org/cs/in-tic tac toe-ception-perfect/1681243068

Hackerrank challenge - https://www.hackerrank.com/challenges/ultimate-ttt

Python and pygame tutorials – http://thenewboston.org

Minimax and alpha beta pruning - http://cs.ucla.edu/~rosen/161/notes/alphabeta.html

Miscellaneous problems - http://stackoverflow.com/