University of St.Gallen
Course 3,793
Dr. Mario Silic

Marco Broger
Valentin Schnellmann

Tic-Tac-Toe AI Documentation

A short overview of the code and logic behind a Python-based AI to play Tic-Tac-Toe against.

Setup

As per common practice in software development, the code is split into multiple files: `runner.py` and `tictactoe.py`. The former handles the graphical interface that the user interacts with, running the game and moving it forward, while the latter supplies the logical functions required to play Tic-Tac-Toe.
Since the graphical part is based primarily on the use of out-of-the-box functionality of the Python package `pygame`, a greater emphasis is placed on the discussion of the functions and the algorithm enabling the AI to play the game.

Graphical User Interface (GUI) and Game Execution

As mentioned in the section on the setup, the graphical interface is defined in `runner.py`. Using the popular game development package `pygame`, a GUI can be established very efficiently. In a first step, this is done by creating a game window that serves as a basis for additional renderings of objects.

Start Screen

In an effort to keep the interface comprehensible, the initial screen features a title, as well as two buttons that enable the user to choose a player. In this case of a turn-based two-player game, the user can choose between 'X' and 'O', where 'X' always takes the first turn. The user's choice is registered if the mouse pointer at the time of a mouse click overlaps with a button's area.

Game Screen

As soon as a player has been chosen, the GUI changes to the game screen. This part of the GUI is made up of nine tiles of equal size to display the classic 3x3 playing field of Tic-Tac-Toe.
If the board has registered a move by either player, it renders the player's symbol onto the chosen tile. This is again based on a combination of the mouse pointer's position, the registration of a mouse click, and the area overlap with a tile. In addition, the game screen always displays the current player as a title above the board.

End Screen

When a winner has been determined, the user is confronted with the end screen. The terminal state of the playing field remains to show the final score. A title above the board shows which of the players has won the game.
Finally, a button below the field enables the user to start a new game without having to restart the program.

Game Steps

Besides the graphical implementation of the game, this part of the code also handles the turn-taking that is required. It makes use of the functions implemented in the logical part of the code and calls them in the adequate sections. For example, if it is the AI's turn, `runner.py` calls the function from `tictactoe.py` that is responsible for determining the best move to take. Afterwards, it is responsible for the taking of that step and displaying the move that has been played.

Auxiliary Functions and AI Decision-Making

Almost the entire logic of the Tic-Tac-Toe game is contained in `tictactoe.py`. It is based on helper functions and an algorithm that allows the AI to make informed decisions on which move to choose next. Important concepts to keep in mind throughout are `board` and `action`:

- `board` is the current state of the playing field modeled as a 3x3 array.
- `action` is a coordinate tuple (i, j) that corresponds to a position on `board`. Conceptually it is equivalent to a move that is or can be played, hence the name.

Helper Functions

The functions that are classified as auxiliary are listed and explained in this subsection.

`initial_state()`

This function returns the starting state of the board. This is defined as a 3x3 array where every entry is `None`.

`player(board)`

The purpose of this function is to take in a state of the playing field and return the player who will take the next turn. Since player 'X' always goes first, the function uses the modulo operation to distinguish the cases based on the number of moves played.

`actions(board)`

As mentioned above, an action is a playable move. Therefore, this function takes a board as a parameter and returns the coordinates of all its empty cells in an array.

`result(board,action)`

This function serves the purpose of taking a board and an action and returning a 'hypothetical' board that has been moved forward by said action. At first glance this seems more technical than need be, but by not moving the original board by one move, this function can be used for actually moving the board forward **and** for hypothetical situations that aid in determining the optimal next step.

`winner(board)`

The determination of a winner is implemented in this function. It checks whether any of the win conditions on the board are satisfied and returns either the winner (if there is one already) or `None`.

`terminal(board)`

A board is in a terminal state when a winner has been found or when the board has no more empty tiles. This function takes a board and returns a boolean indicating whether the playing field is in a terminal state or not.

The Minimax Algorithm

The decision-making process of the AI is based on the Minimax algorithm. It establishes a way for the AI to figure out which move it needs by considering all the moves it can take and all of the subsequent of the other player.

`utility(board)`

The Minimax algorithm needs a utility function to distinguish the goals of the different players. In this case, player 'X' aims to maximize the utility and player 'O' aims to minimize it. If player 'X' wins, this function returns 1, if player 'O' wins, it returns -1 and otherwise it returns 0. This function will only be used on terminal boards.

`MaxValue(board)`

As the name suggests, this function is used for the maximizing player. For all the actions the player could take, it considers all the actions that the opponent (the minimizing player) could take. It then returns the value of the best move for the maximizing player without knowing the moves that the minimizing player will play.
Since `MaxValue` is called on `MinValue`, it aims to choose the move that is least disadvantageous (maximum) of the ones that the opponent will consider to win (minimum).

`MinValue(board)`

This function is analogous to `MaxValue` with the exception that the goal is the opposite: to minimize the value. In that sense, it is the inverse of `MaxValue`, which makes sense since the minimizing player player aims to minimize the value.

`minimax(board)`

Finally, this is the function that ties it all together. It makes use of `MaxValue` and `MinValue` to comb through possible game trees and make an informed decision based on the possible outcomes. It distinguishes between 'X' and 'O' since the AI can take on the role of both players. Therefore, it can play both as the minimizing as well as the maximizing player.

Example Case & Visualization

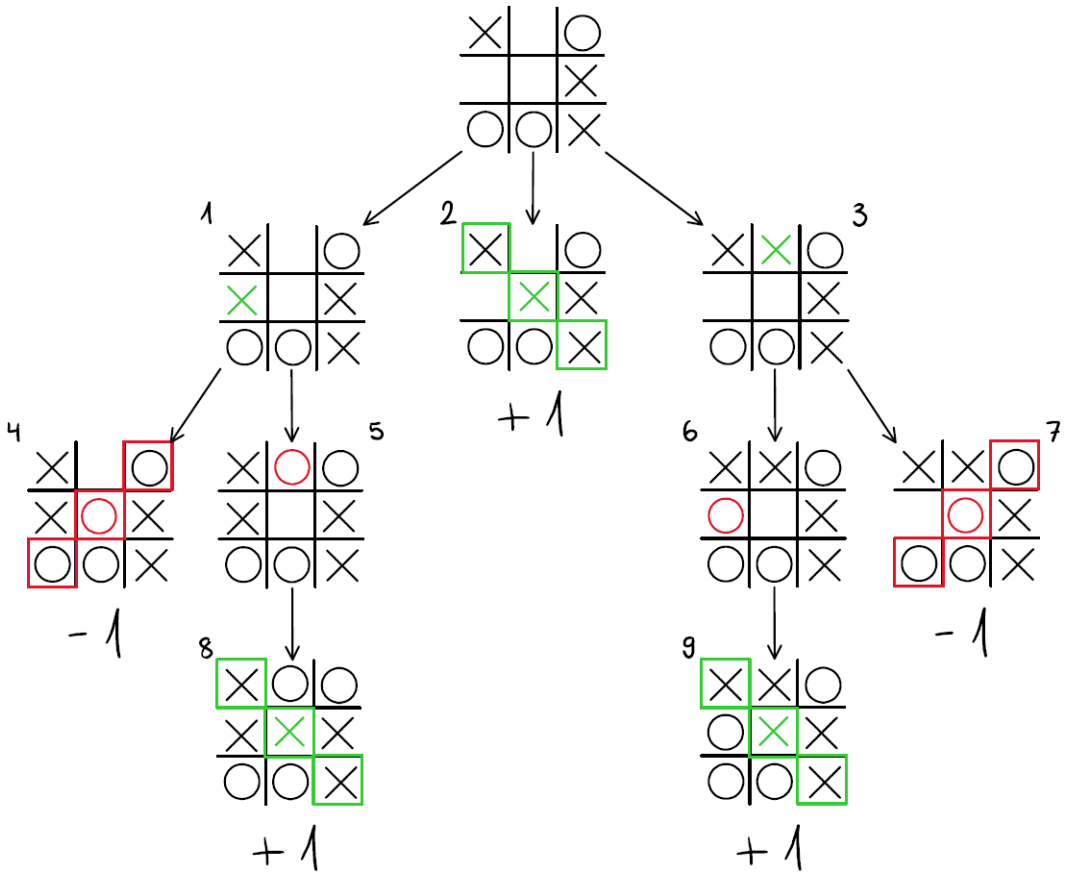

Figure 1: Example case of a game tree that is explored by the Minimax algorithm.

Let's consider the example in Fig. 1 and explore the corresponding branches:

- State 1 is not a terminal state, so it does not have an associated utility. It leads to states 4 and 5, where state 4 returns a utility of -1 and state 5 leads to state 8 with a utility of 1. Since player 'O' decides whether to take the move that leads to state 4 or state 5 and they want to minimize the value, they will opt for the move that ends in state 4. This means that the utility of state 1 will be set to -1.
- State 2 is a terminal state of utility 1.

- State 3 is not a terminal state, therefore it leads to states 6 and 7. State 7 has a utility of -1 whereas state 6 leads to state 9 with a utility of 1. Based on established principles however, the utility of state 3 will be -1 since player 'O' will want to end up in state 7.

Following this logic, player 'X' (the current player in our example) should choose the move that leads to state 2 because it is the maximum value in the utility vector $(state_1, state_2, state_3) = (-1, 1, -1)$.