Tic Tac Toe AI with JavaFX

When we say the word **game** in the context of AI we usually don't mean it in the sense of entertainment games, instead we refer to a more general definition of a game:

A game is a multi-agent environment in which agents compete and/or cooperate on some specific task(s) while meeting some specific criteria. An agent is referred to as a player.

With this general definition we could see that a situation, like driving your car to a specific place, is a game. In such a situation, you (as a player) exist in an environment along with other drivers (other players) and you cooperate with some drivers to avoid crashes and reach your destination and compete with other drivers to reach there fast.

We can easily see that entertainment games also fall under the general definition of games. Take chess for example. In chess there are two players (multi-agent) who are *competing* on who captures the other's king first (the task) in the lowest number of moves (the criteria).And because

entertainment games have the same properties and characteristics of any other general game, we can use a simple entertainment game to demonstrate the AI concepts and techniques used in any form of a general game, and that's what this post is all about.

We'll be working with Tic-Tac-Toe as our demonstration game mainly because of its simplicity. It has simple rules, simple actions, and simple ending configurations, which makes it easy to describe it computationally.

At the end of this documentation, you should have a working Tic-Tac-Toe game .

. I'll be focusing entirely on the game representation code and the AI code. I'll not talk about any UI code or any of the tests code, but these are fully commented and easy to understand when read. The whole project including the tests is available at >> https://github.com/7osny/TicTakToe-Javafx-App .

The first thing we need to do is to describe and understand the game we want to build. We could describe it verbally as follows:

- Two players (player X, and player O) play on 3x3 grid. Player X is a human player, and player O is an AI. A player can put his/her letter (either X or O) in an empty cell in the grid. If a player forms a row, a column or a diagonal with his/her letter, that player wins and the game ends. If the grid is full and there's no row, column or diagonal of the same letter, the game ends at draw. A player should try to win in the lowest possible number of moves.

Formal Definition

In working on an AI problem, one of the most fundamental tasks is to convert a verbal description of the problem into a formal description that can be used by a computer. This task is called **Formal Definition**. In formal definition we take something like the verbal description of Tic-Tac-Toe we wrote above, and transform it into something we can code. This step is extremely important because the way you formally define your problem will determine whether easy or difficult it will be to implement the AI that solves the problem, and we certainly want that to be easy.

Usually, there are some ready definitions we can use and tweak in formally defining a problem. These definitions have been presented and accepted by computer scientists and engineers to represent some classes of problems. One of these definitions is the game definition: If we know our problem represents a game, then we can define it with the following elements:

- **A finite set of states** of the game. In our game, each state would represent a certain configuration of the grid.

- **A finite set of players** which are the agents playing the game. In Tic-Tac-Toe there's only two players: the human player and the AI.
- **A finite set of actions** that the players can do. Here, there's only one action a player can do which is put his/her letter on an empty cell.
- **A transition function** that takes the current state and the played action and returns the next state in the game.
- **A terminal test function** that checks if a state is terminal (that is if the game ends at this state).
- **A score function** that calculates the score of the player at a terminal state.

Using this formal definition,we can start reasoning about how we're gonna code our description of the game into a working game:

The State

As we mentioned, one of the elements of our game is a set of states, so we now need to start thinking about how we're gonna represent a state computationally.

We said earlier that a state would represent a certain configuration of the grid, but this information need to be associated with some other bookkeeping. In a state, beside knowing the board configuration, we'll need to know who's turn is it, what the result of the game at this state is (whether it's still running, somebody won, or it's a draw), and we'll need to know how many moves the O player (AI player) have made till this state (you would think that we'll need to keep track of X player moves too, but no, and we'll see soon why).

The easiest way to represent the state is to make a class named State from which all the specific states will be instantiated. As we'll need to read and modify all the information associated with a state at some points, we'll make all the information public. We can see that modifying X moves count, the result of the state, and the board can be done with a simple assignment operation; while on the other hand, modifying the turn would require a comparision first to determine if it's X or O before updating it's value. So for modifying the turn, we'll use a little public function that would do this whole process for us.

There are still two more information we would need to know about a state. The 1st one is the empty cells in the board of this state. This can be easily done with a function that loops over the board, checks if a cell has the value "E" (Empty) and returns the indecies of these cells in an array. The second information we need to know is that if this state is terminal or not (**the terminal test function**). This also can be easily done with a function that checks if there are matching rows, columns, or diagonals. If there is such a configuration, then it updates the result of the state with the winning player and returns true. If there is no shuch thing, it checks if the borad is full. If it's full, then it's a draw and returns true, otherwise it returns false.

For the board, it would be simpler to represent it as a 9-elements one-dimensional array instead of a 3x3 two-dimensional one. The 1st three elements would represent the 1st row, the 2nd three for the 2nd row, and the 3rd three for the 3rd row. One last thing to worry about is how we can construct a state. Instead of at each time we construct a state form scratch and fill its information one by one (think about filling the board array

element by element), it would be convenient if we have a copy-constructor ability to construct a new state from an old one and have minimal information to modify.

You can read State of the Board.java file in the repo in >>

https://github.com/7osny/TicTakToe-Javafx-App .

The Human Player

We move on to the next element of our definition, which is the players. Our first player will be the human player, which we will represent and implement along with its actions through the UI and its controls , UI is very simple , it Designed By JavaFx ,u can Take a look at UI in the Class Main.java in github repo.

The Game

This is the structure that will control the flow of the game and glue everything together in one functioning unit. In such a structure, we'd like to keep and access three kinds of information : the AI player who plays the game with the human,the current state of the game, and the status of the game (wheter it's running or ended), we'll keep those as public attributes for easy access. We'd also need a way to move the game from one state to another as the playing continues, we can implement that with a public function that advances the game to a given state and checks if the game ends at this state and what is the result, if the game doesn't end at this state, the function notifies the player who plays next to continue. Finally, we'd need a public function to start the game.

Techniques Used

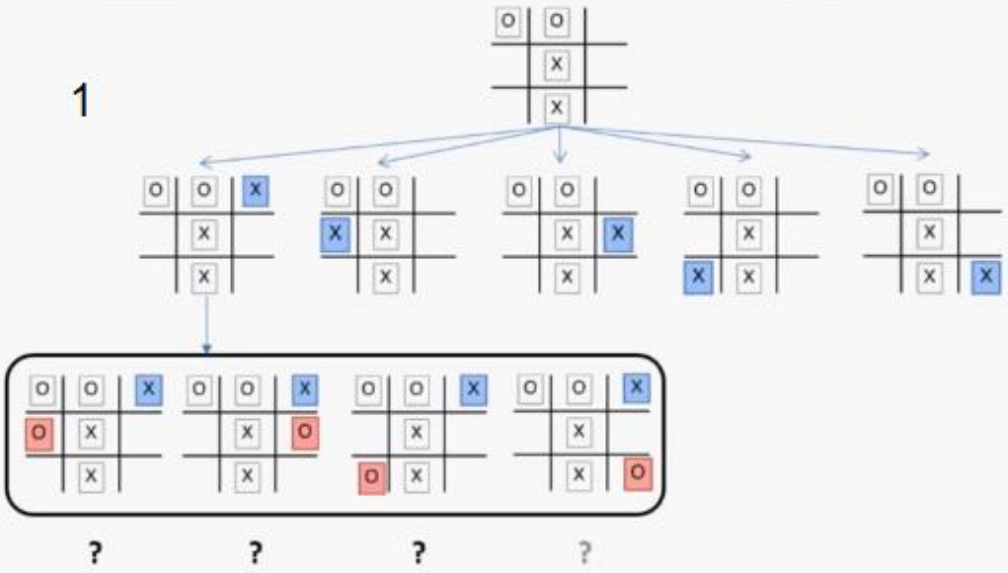
- JavaFX
- Minimax Algorithm

The Minimax Algorithm

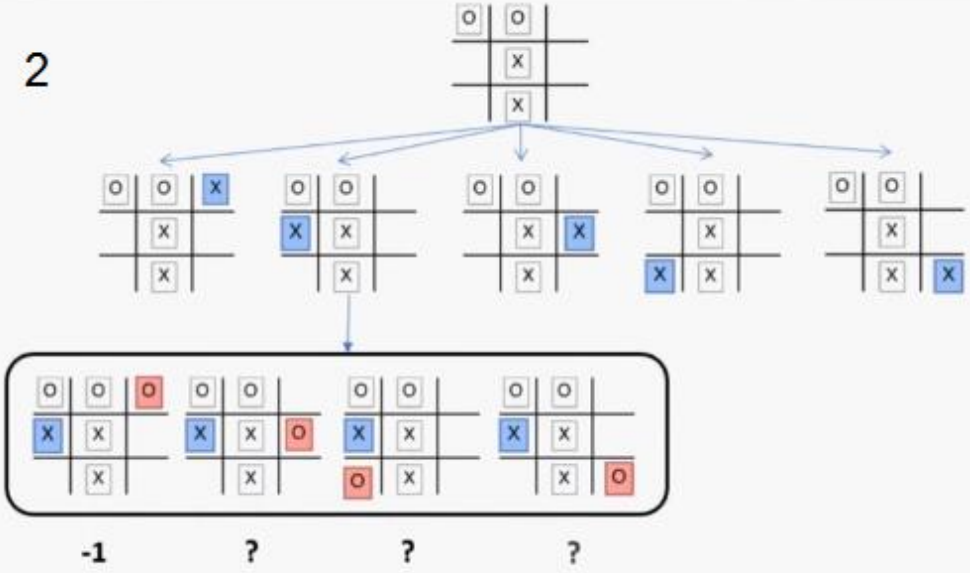
Now the AI needs a way to use the information provided by the score function in making an optimal decision, and this is where minimax comes to help. The Basic intuition behind the minimax decision algorithm is that at a given state, the AI will think about the possible moves it can make, and about the moves the opponent can make after that. This process continues until terminal states are reached (not in the actual game, but in the AI's thought process). The AI then chooses the action that leads to the best possible terminal state according to its score.

The algorithm is used to calculate the minimax value of a specific state (or the action that leads to that state), and it works by knowing that someone wants to **mini**mize the score function and the other wants to **max**imize the score function (and that's why it's called **minimax**). Suppose that O wants to calculate the minimax value of the action that leads it to the state at level 0 in the following figure:

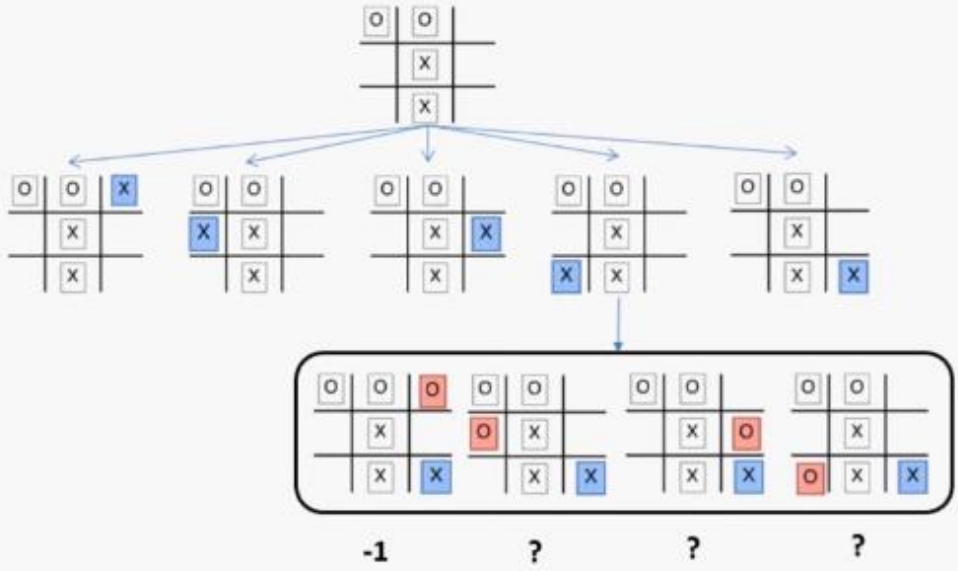
1



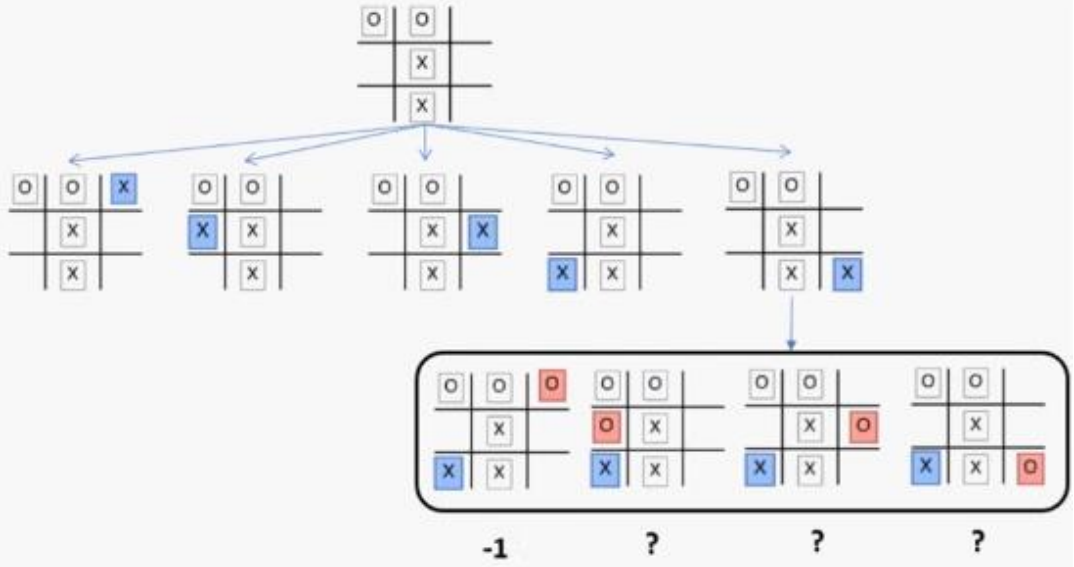
2

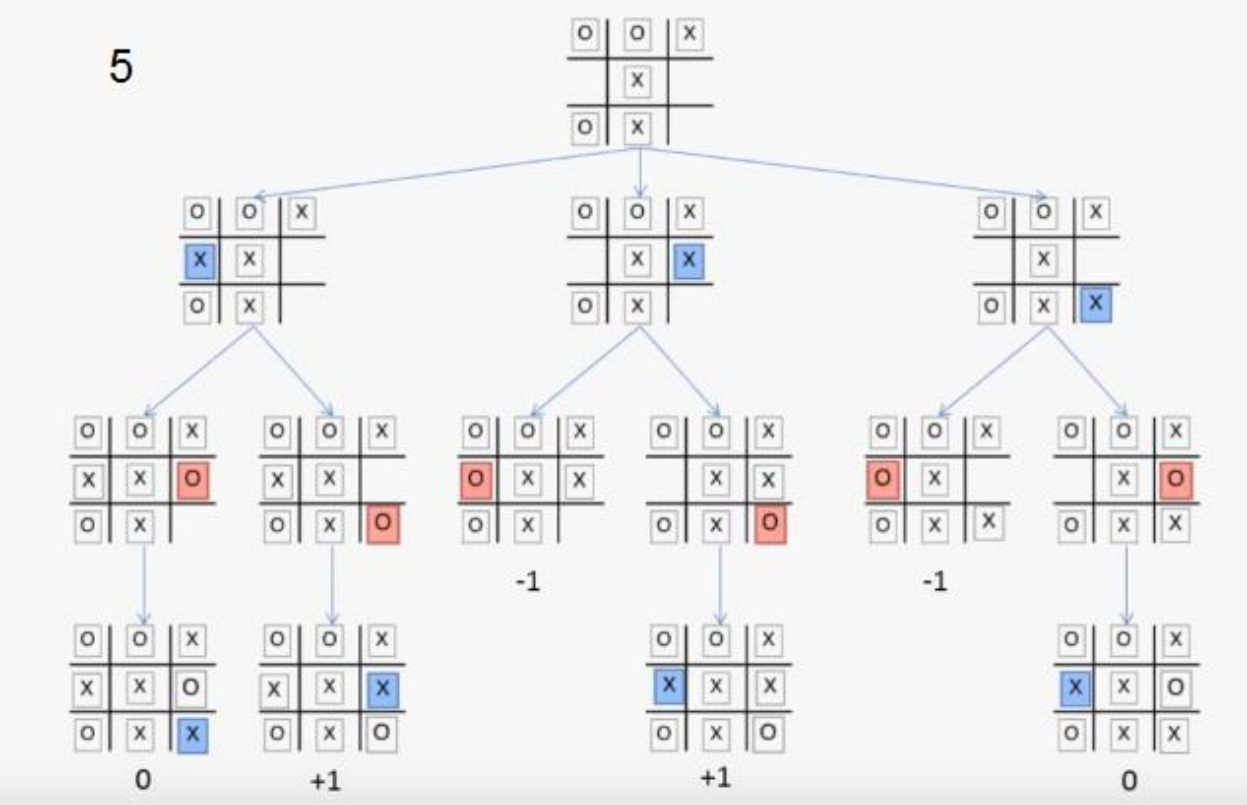


3



4




5
-1
-1
0
+1
+1
0