**Minimax algorithm**

An agent is anything that perceives its environment through **sensors** and is able to act upon those stimulations though its **actuators.** In a tic-tac-toe game, when the user plays against the “computer”, it is basically playing against an **agent** that is mimicking how a human would act: exploring all possibilities to try and make the best possible move.

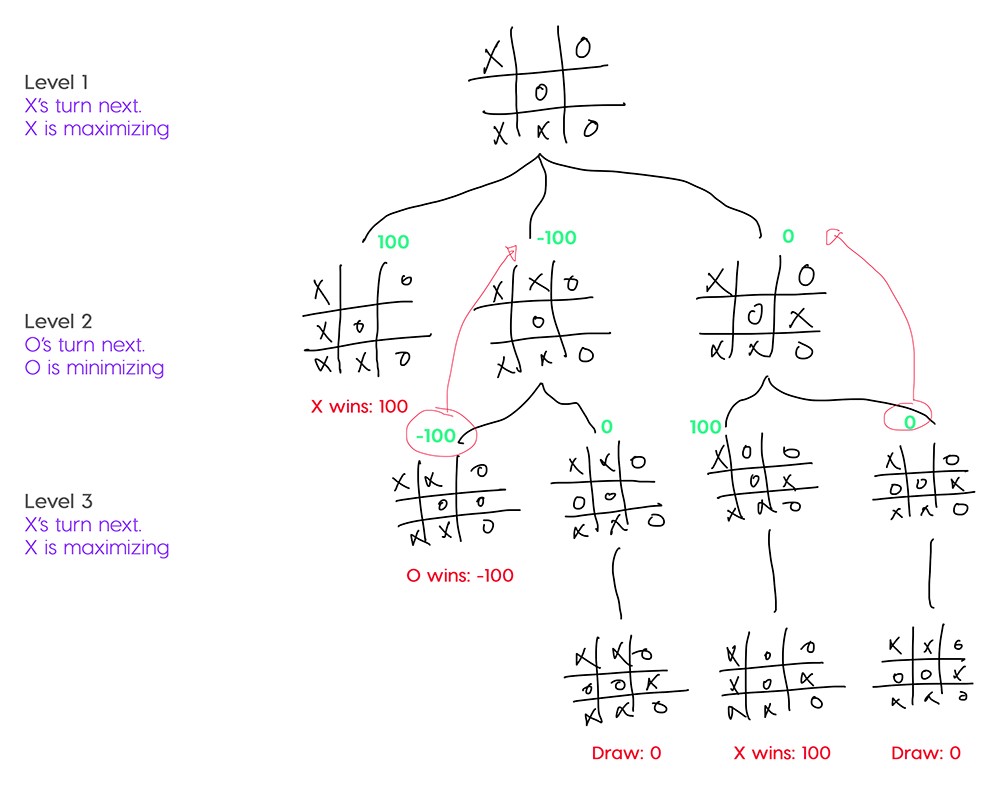
In this tic-tac-toe implementation, the agent uses the minimax algorithm. It is a **recursive** algorithm often used to decide the best possible move in turn-based strategy games where the next moves can be foreknown, such as in chess or, like in this case, tic-tac-toe.

One of the players is called the maximizer and the other is the minimizer. The maximizer tries to make the move with the highest value and the minimizer tries to minimize that score, therefore choosing the lowest. Minimax creates a search tree with all possible moves at a given time and uses depth first search to move through the tree. That means, it will explore the tree from left to right and always trying to go as deep as possible, till it finds a terminal state. Once it finds a terminal, it propagates the value up to the tree. In tic-tac-toe, there are 3 possible terminal states. If we take X as the maximizer and O as the minimizer, the terminal states and its values are:

1. X wins: Value of that board state is 100
2. O wins: Value of that board state is -100
3. Draw: Value of that board state is 0

For example, in the image below:

* Level 1: At the root of the **search tree**, we see the initial state of the board. The agent has three possible moves to make.
* Level 2: (X, maximizer turn)
  + Move 1: The first move leads to a terminal state where X wins so the value is 100.
  + Move 2: Leads to two more moves.
    - Level 3: (O, minimizer turn)
      * Move 1: O wins, board state is -100
      * Move 2: After going one level deeper results in a draw, value is 0
      * Since it’s simulating “O” turn, it would obviously choose the move with which it would win, so the **value for this node is -100**.
  + Move 3: Follows same pattern as move 2
* When depth first search has finished exploring all possible moves and propagating the scores from the bottom of the three up to the current state, X has three possible values to choose from at level 1: 100, -100 and 0. Therefore it would choose 100 to win the game.

[[1]](#footnote-1)

Furthermore, there is one more case covered in this implementation of tic tac toe. If two paths lead to a win, the algorithm would return the same value but there has to be a way of knowing which path is “better”. In this case, what was done was:

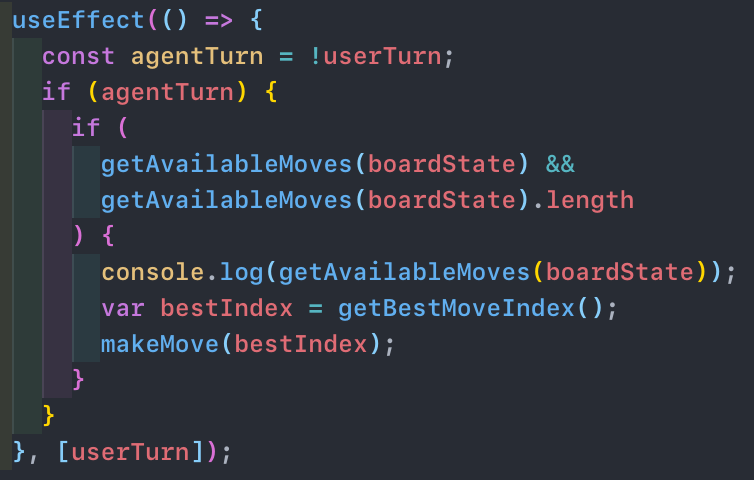
* If the maximizer wins: subtract the depth at which the terminal state is from 100
* If the minimizer wins: add the depth at which the terminal state is to -100

With this, the maximizer and the minimizer would choose the move that leads them faster to a win. For example:

* **Maximizer win at depth 3: 100-3 = 97**
* Maximizer win at depth 1: 100-1 = 99
* Minimizer win at depth 3: -100+3 = -97
* **Minimizer win at depth 1: -100+1 = -99**

This becomes useful because, if we can get faster to a win state, we decrease the number of turns our opponent will have to try to win the game instead.

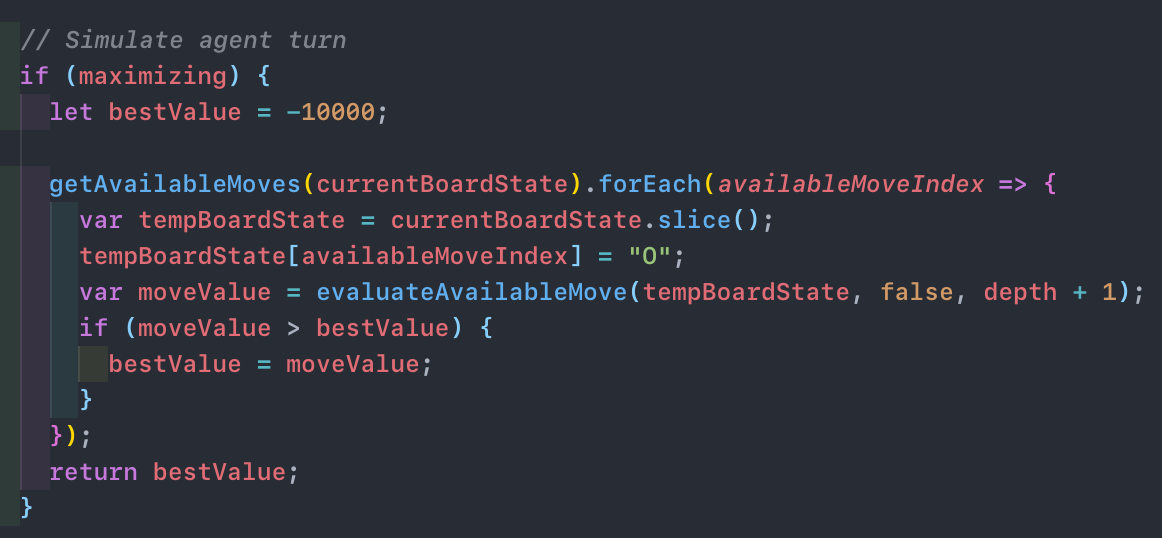
Now, as can be seen in the diagram at the end of this document, the minimax agent senses two things, the userTurn and the boardState. If it’s not the userTurn, the agent knows it is it’s turn so it has to get/sense the current boardState from the environment. Once it does, it has to verify that there are available moves in the board to make. If there are available moves, it begins constructing the search tree to get the index of the best possible move, as seen in the code below:



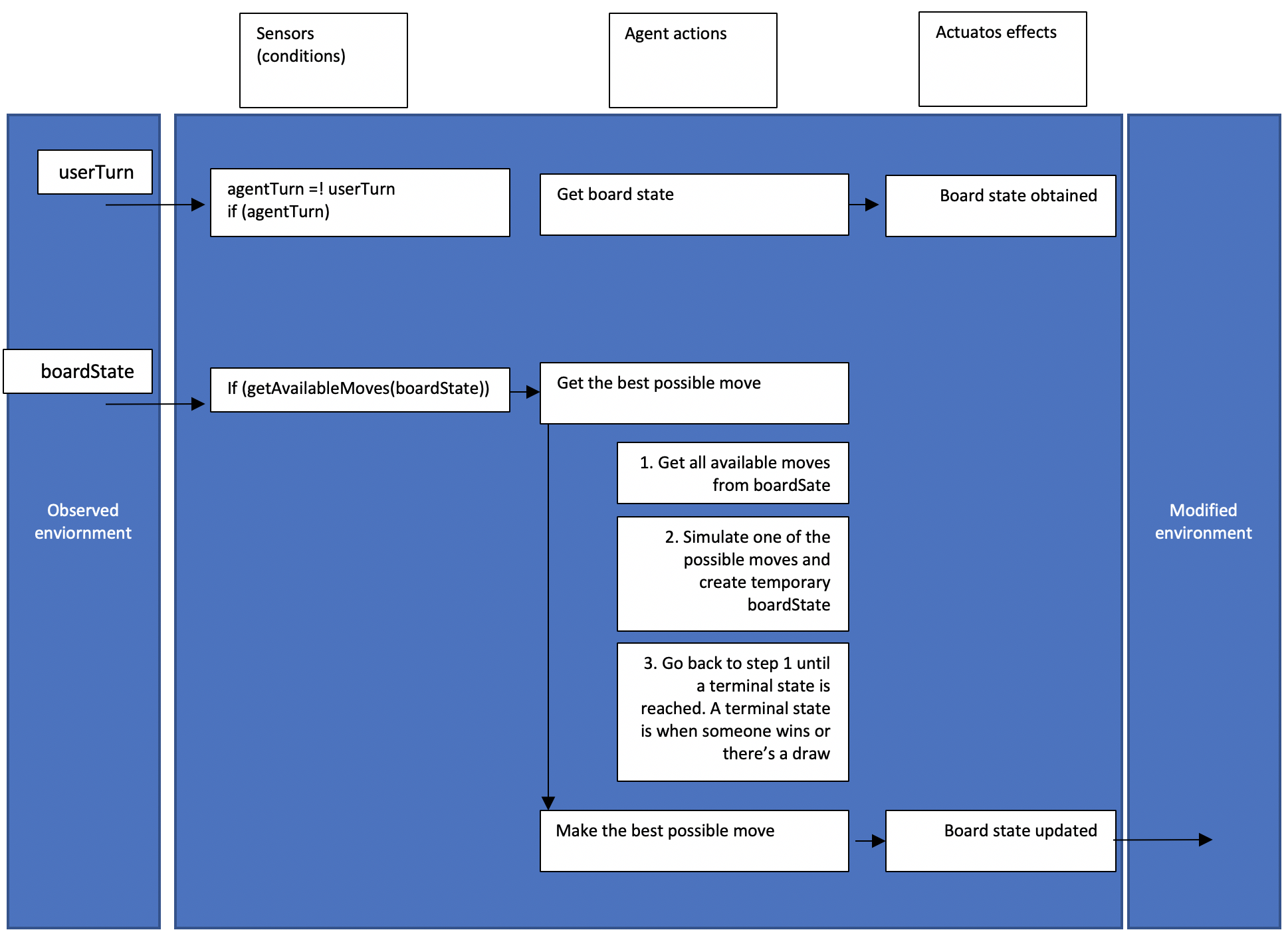
Basically, in the getBestMoveIndex function for each available move in the current boardState, it creates a temporary boardState supposing it made that move and calls the evaluateAvailableMove, which is the minimax function.



The evaluateAvailableMove recursively checks with the same process for available moves in that temporary boardState until it reaches a terminal state. At each recursive call, the new depth is passed and a parameter to know if it’s the maximizer or minimizer turn. So, in each level of the search tree created, it is simulating either the maximizer or minimizer turn and when it reaches a terminal node it knows whose turn it is at that level in order to propagate the correct value up the tree. Below is the logic to simulate the maximizing turn.



Once the bestIndex is found, the Agent updates the environment with an actuator, by updating and returning the state of the board with its new move.



**References**

<https://www.freecodecamp.org/news/playing-strategy-games-with-minimax-4ecb83b39b4b/>

<https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/>

<https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-2-evaluation-function/>

<https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-3-tic-tac-toe-ai-finding-optimal-move/>

<https://www.geeksforgeeks.org/implementation-of-tic-tac-toe-game/>

<https://medium.com/@alialaa/tic-tac-toe-with-javascript-es2015-ai-player-with-minimax-algorithm-59f069f46efa>

1. Source: <https://medium.com/@alialaa/tic-tac-toe-with-javascript-es2015-ai-player-with-minimax-algorithm-59f069f46efa> [↑](#footnote-ref-1)