Minimax: given a state *s*, the MaxP takes the action *a* from Actions(*s*) that, after passing the function Min-Value(Result(*s,a*), returns the highest value. Result(*s,a*) return how will be the state *s* after performing the action *a*. Min-Value() returns what happens after the MinP tries to minimize the state value. The algorithm runs this function for all the available possible action and takes the action that returns the highest value after passing the function Min-Value(). The MinP makes the same thing but the function is Max-Value() and the choose will be on the actions that returns the lowest value.

How can we calculate a state value?

Max-Value(*state*): Where *state* stands for the state we want to calculate the value.

If Terminal(*state*): This is still the same *state.* (If the state reached a goal state) Return Utility(*state*) Still same *state.* (Assign a value to the board defining who won)

If the game’s not finished, we want that the value of the state is the highest as possible. We are  
 going to keep track of that value in a variable called *v.* We start by assigning to *v* the lowest possible  
 value.

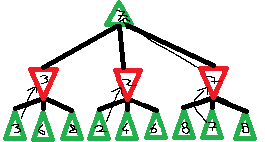
V = -∞ seems strange, but by doing so we are certain that for each action, we will do better than -infinite

For action in Actions(*state*):

V = Max(v,Min-Value(Result(*state,action)))* We are trying to pick the maximum value  
 between v (-infinuty) and the function Min-Value() that return the value of a state after that my  
 opponent tries to lower it. Now v is the maximum value between all the results

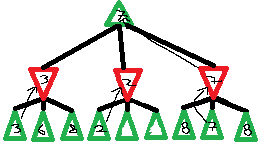
Return v and we get the state value

For the Min-Value it is the same thing but reversed where v = infinity and not -infinity, and we are getting the Minimum value between the function Max-Value(), the function that returns the value of a state after that MaxP tries to maximize the value.

Since this is a lot of work to do, we have to find a way to optimize the algorithm.  
This is how an algorithm would work

MaxP chooses the highest value between the 3 futureaction by the MinP

MinP chooses the lowest value between the 3 future actions

So how can we optimize it? By removing the useless calculation when we know that that action will make it a low value state.

Since this state will have a value lower than the previous state, it’s useless to continue.

If I already know that 2 is lower than 3, why should I continue? This reasoning is possible is by some keeping tracking of values while executing the algorithm. This optimization technique is called Alpha-Beta Pruning. Alpha and Beta are the two values that we are comparing, and Pruning is the idea of remove useless node in long and deep trees.

But, even if we optimize the Minimax algorithm, it won’t be enough for complex games. Since tic tac toe is a simple game, the PC doesn’t take so much time to calculate al the possible actions and sates after that action till the possible end state (there are 255,168 possible states).   
In more complex games, things get bigger, harder, and longer. For example, in chess game there are 1029000 (29thousand). A number that PCs are unable to manage.

To avoid calculating all the possible actions, we can use the Depth-Limited Minimax. Minimax is Depth-Unlimited since it goes moves after moves till the end. The Depth-Limited Minimax calculates a limited number of moves. So, after having a limited number of moves, how can the AI estimate the value of that state if the game’s not ended? Here it come the Evaluation Function, a function that returns the expected utility() of a given state. As the GBFS works better as better works the heuristic function, here the Depth-Limited Minimax works better as the Evaluation Function works better so the AI can understand the current value in a better way.