Motivation :

I have been looking forward to getting to know a new algorithm that has been intriguing me for quite some time . I am talking about the Minimax algorithm and my main motivation to learn it was the possibility to create a program that was capable of making a smart decision in a turn-based game like tic-tac-toe , to see what kind of strategies would the program develop to maximize the chances of winning . The use of AI algorithms to play such games dates back to the 1950s, when Alan Turing alongside David Champernowne developed the Turochamp, a program designed to play an entire game of chess against a human player. Unfortunately, due to the complexity of the algorithm, Alan Turing was never able to run his algorithm on a computer before his death, leaving an unfinished project that was not continued. Later, several new proposals were made as alternatives to Turing's original idea, amongst them Alex Bernstein's program , developed to work on the IBM704 computer, which was very powerful for the time being .

What is Minimax :

Minimax is a recursive backtracking algorithm normally used in the development of AI agents capable of playing turn-based zero-sum games (games in which one player's win is offset by the loss of another player , for instance the victory of a player in tic-tac-toe implies the loss of the other ) consisting of at least two players / agents , such as tic-tac-toe, chess, checkers , connect four or even pacman and others.

The Minimax algorithm works in the following way . We have two types of players, the maximizer which represents the AI and aims to maximize the chances of winning, and the minimizer which represents the opponent and will do everything possible to minimize the AI's chances of winning , assuming that both players play optimally so that the AI doesn’t miss any clever corner case that would lead to defeat . It is important to note that these two types of players are hypothetical and the algorithm mimics them to simulate what the next best move will be . The algorithm does this by foreseeing all possible game states some number of moves away , picking the path which produces most favourable results . The following pseudocode shows the inner workings of the algorithm .

Pseudocode :

The function receives three arguments . The node that could represent a game state, the depth which is used to track the number of turns played so far to reach the current game state (alternatively, it could represent the depth of the game tree which we will see later down the road) and the boolean variable “maximizing” which tells us whose turn it is , if it is true , then it is the maximizer’s turn . If it is the maximizer's turn, we keep the maximum possible score from the bottom layer of the game states, otherwise we keep the minimum possible score with the goal of simulating optimal decisions of both players. We repeat this process until we reach a final state (the node is a leaf) .

Game Tree :

The game tree, as it is commonly called in game theory, is a directed graph whose nodes represent game states. Game trees are of great importance in AI, as they allow us to search through all possible paths a game can take , granting us the possibility to hand-pick the most advantageous scenarios .

There are different types of game trees, among them we can find the complete game tree, which represents all possible ways of playing a game, and the partial game tree, which creates the game tree only up to a certain specified depth . With a complete game tree, it is possible for the minimax algorithm to find the path that increases the chances of winning. Even though minimax looks like a neat algorithm, there is a colossal underlying issue attached to it. The following image shows a section of a tic-tac-toe game tree in which the maximizer has two options . if it plays in the upper right corner, the game ends in a tie, if it plays in the lower right corner, the outcome will be a victory for the maximizer, therefore the optimal move is the latter .

Time Complexity :

The complete game tree grants us the path to success by generating all possible games that can be played, however this can be extremely costly as the game becomes more convoluted. This problem is caused by the recursive nature of minimax, which produces an exponential time complexity, more precisely O(b^m), where "b" is the average branching factor and "m" is the average game length . As we can see from the image above, the calculation of the complete game tree might be feasible for games like tic-tic-toe (my implementation of minimax to solve tic-tac-toe can be found here) since it has about 255168 possible games, which can be calculated quite fast in modern computers, however it becomes impossible in other games like chess, which have about 10^120 possible games . A possible solution is to use a partial game tree instead of the complete game tree and compute a fixed number of moves in advance instead of computing all possible game states . Another alternative is to use some tree pruning algorithm such as alfa-beta pruning ( which I might write later in a separate blog post ) to cut unecessary recursion , which saves a lot of extra computation .

The Heuristics :

So far we have seen practically everything that is needed to implement a functional AI agent that is capable of making smart decisions using the Minimax algorithm. The final necessary step is to figure out how we can tell the algorithm which game states seem more favorable. If the current node in the game tree is a leaf, then we know that this game has reached a final state, so we return the corresponding heuristic. The question one might ask is what these heuristics are and how can they be defined. The heuristics are the ratings given to the leaf nodes in the game tree, these are the ratings that define how good a particular game state really is. Let's see how this works with a fairly simple example, the tic-tac-toe has three possible game results, a win, a loss or a tie. Since we want the AI to win, and the AI is looking for the maximum possible score, we need to associate the winning game states with a positive score for the algorithm to prioritize them, while the losing game states need to be associated with a negative score for the algorithm to avoid them, the tie game states can have a neutral value between the winning and the losing value . The following image shows a tic-tac-toe game tree , where the winning , losing , tie game states have the ratings of 1, -1, 0 respectively . As we can see from the labels, the orange layer represents the maximizer and the blue layer the minimizer. It is also important to note that the orange layer selects the maximum score from its children, while the blue layer selects the minimum score in an attempt to simulate optimality from both sides . Since the root of the tree was assigned a one we can prove that there is a sequence of moves that will grant the maximizer a certain victory .

We have seen how to define heuristics for tic-tac-toe , but how would heuristics work in a more complex game like chess ? One might say that chess also has three possible results , therefore implementing a similiar idea to the previous tic-tac-toe example would be a good idea , but that would throw us down the rabbit hole since this would imply being able to compute the game tree down to the leaf nodes to get precise results , which would be hopeless given the mind-boggling number of leaf nodes in the complete chess game tree . A much more reasonable approach is to assign values to each piece so that the algorithm prioritizes the most valuable pieces. For example, if the king was threatened, the algorithm should do something immediately to avoid a possible loss, on the other hand if a pawn was threatened, it should not be of much importance .

The End :

So there we go! This should be all we need to know to implement a functional AI agent using Minimax, but now it's your turn, so I'll leave the reader with a challenge. Try to implement a Minimax approach for the CodinGame ultimate tic-tac-toe challenge and optimize it to get a good score on the leaderboard.