Nassim Sbai

INFO 550

Final Project

Implementation of the Minimax algorithm in the Tic-Tac-Toe game using Alpha-Beta pruning

I implemented a traditional Tic-Tac-Toe game using an AI as the opponent. The AI made all its decisions following the logic of the Minimax algorithm. The game was set using a board played on a 3 by 3 grid, with two players, one represented by "X" and the other one represented by "O". The Minimax algorithm is an algorithm often used in game theory to determine, among multiple legal moves, the best move to maximize the score. In this case our AI player used the minimax algorithm to explore all possible moves and select the move that will maximize its score.

The reason why I picked this particular project is because I was totally fascinated by how a simple algorithm such as Minimax can produce incredible results in games. Before we dive further into the project, let’s get the intuition behind the minimax algorithm and understand the mechanisms behind it. The Minimax algorithm works by representing all the possible legal moves of the game by building a game tree that includes nodes representing game states and edges representing the possible legal moves.

The algorithm performs depth-first search (DFS) on the game tree and examines the leaf nodes also known as the terminal states and assigns a score to each terminal state. The scores assigned are then propagated to the root of the tree which enables the algorithm to make the best possible decision at each level.

The algorithm takes the current game state as input and determines which player's turn it is, either the maximizing player or minimizing player. In the case of Tic-Tac-Toe, the maximizing player would be the AI that strives to maximize its score while minimizing the score of the opponent which would be the minimizing player (the human player). The algorithm is said to be recursive because it recursively explores all the possible moves at each level also known as game state. After examining all the possible moves, it selects the move that will lead to the highest possible score. Once the algorithm reaches a terminal state, which is the leaf node in a game tree, it assigns a score to each state. In Tic-Tac-Toe, the algorithm assigns a score of 1 if the AI wins (maximizing player), -1 if the human player wins (minimizing player), and 0 for a draw.

The Minimax algorithm assumes that both players play optimally. While the algorithm performs very well on simple games such as checkers and Tic-Tac-Toe, its time complexity grows rapidly when the number of possible moves and the depth of the game tree grows which makes it not optimal for complex games. However, there are some strategies to deal with this issue, a very effective option would be alpha-beta pruning that prunes unnecessary nodes.

Now that we have an idea of the intuition behind the minimax algorithm, let's see how I implemented it in my Tic-Tac-Toe project. I implemented a function called minimax which takes the current board state, the depth, and another element depending on whether the goal is to maximize or minimize the score. The algorithm considers all possible moves and assigns a score to each move based on a thorough evaluation.

To make the algorithm run even faster, I decided to include the Alpha and Beta pruning to prunes the unnecessary nodes that increased the time complexity.
The evaluate function evaluates the score of the board for the AI player using the check_winner function. If the AI player wins, it returns a score of 1. If the human player wins, it returns a score of -1. If the game is a draw or still in progress, it returns a score of 0.

The minimax function implements the Minimax algorithm. It takes the current board state, the depth of the search, and a flag indicating whether it is maximizing or minimizing the score. The algorithm is implemented in the form of a function that evaluates all possible moves and assigns a score to each move based on the evaluation.

In maximizing, the algorithm goes over the empty cells on the board, simulates placing the AI player's symbol in each empty cell, and calls the minimax function to evaluate the state. It selects the move that comes with the highest score and updates the best_score variable.
In the minimizing phase, the algorithm performs the same process, however, it selects the move with the lowest score.

To optimize the algorithm's performance and improve its time complexity, alpha-beta pruning is implemented. Alpha-beta pruning allows the algorithm to disregard the branches of the game tree that are known to be worse than the previously evaluated branches.

The ai function uses the Minimax algorithm to determine the best move for the AI player. It initializes the best_score variable to negative infinity and iterates over the empty cells on the board, using the minimax function for each possible move. It selects the move with the highest score and places the AI's symbol in that spot.

The play_game function is the main game loop. It initializes the board and starts the game. It also checks to see what is the current state to see if there is a win, a loss or a draw. If the game is still in progress, it calls the AI function to make the AI 's move.