Celine SALAME (200070)
Thea Marie Elise SAAB (200750)

TIC TAC TOE GAME

Youtube video link: https://youtu.be/YC3UpbxJJO0

Github link: https://github.com/CelineSalame/TicTacToeAi

How can we describe a perfect game of Tic Tac Toe?

In this assignment, the goal is to create a Tic-Tac-Toe bot that cannot be defeated. To achieve this, we need to establish what constitutes a perfect game of Tic-Tac-Toe. To quantify these scenarios, we assign scores to the three end-game conditions:

Winning: The utility is calculated as 1 multiplied by the number of remaining free spaces on the board.

Losing: The utility is determined as -1 multiplied by the number of remaining free spaces on the board.

Drawing: The utility is assigned a score of zero.

Before deciding on the algorithm to implement, it is crucial to express these conditions in a quantitative manner.

The Minimax algorithm operates as a sequential exchange between the two players, where the current player aims to select the move that maximizes utility. This process involves determining scores for each available move, with the opposing player subsequently choosing the move with the minimum score. The scores for the opposing player's moves are then influenced by the turn-taking player seeking to maximize their own score, continuing this recursive process until reaching an end state.

In our class, we covered two algorithms—Minimax and Expectimax—that are particularly suitable for Tic-Tac-Toe. Here, we will outline the advantages and disadvantages of each algorithm.

MINIMAX	EXPECTIMAX
Impossible to lose	Possible to lose if opponent is optimal
Availability for Alfa Beta pruning	No pruning can be done

That's why we chose the Minimax algorithm over Expectimax.

How did we implement the Minimax algorithm?

The minimax algorithm is implemented through three functions: `minimax`, `minimax_alphabeta`, and `minimax_dfs`, each representing different variations of the algorithm.

1. minimax Function:

```
def minimax (self, isMaximizing):
    # Implementation details
```

2. minimax_alphabeta Function:

```
def minimax_alphabeta (self, isMaximizing, alpha, beta):
    # Implementation details
```

3. minimax_dfs Function:

```
def minimax_dfs (self, isMaximizing):
    # Implementation details
```

All three functions follow the basic structure of the minimax algorithm. They evaluate the game state recursively, assigning scores to different outcomes (win, lose, draw). The algorithm explores possible moves and selects the one that maximizes or minimizes the score, depending on whether it's the AI's turn or the opponents. The `isMaximizing` parameter determines whether the function is maximizing or minimizing the score.

The `minimax_alphabeta` function introduces alpha-beta pruning to optimize the minimax algorithm, reducing the number of nodes explored and improving performance. This enhancement is particularly valuable in scenarios with a large search space.

The `minimax_dfs` function is a variation that utilizes depth-first search (DFS) as an alternative approach to the minimax algorithm. It explores possible moves deeply before backtracking, potentially offering different results compared to the standard minimax strategy.

The compMove function is activated during the bot's turn, and to streamline computations, we've predefined the first move if the bot is the first to play. In this case, the bot strategically places an 'X' in the top-left corner, recognized as the optimal starting move for 'X'. Alternatively, when the bot is not making the first move, the function invokes the minimax function with initial scores set to -1000 for the bot and 1000 for the player. The objective is to minimize the player's score while maximizing the bot's score.

The minimax function employs a maximizer and minimizer alternation, evaluating all potential game outcomes on a duplicate of the game board represented as strings. This approach is adopted for expedited computation.