Project 1: Tic Tac Toe - Alejandro Vargas (avarg116)

Introduction

Usually played on a 3x3 grid, Tic-Tac-Toe is a game where a player marks X's and the other O's until one gets either a complete horizontal, vertical, or diagonal row. The game provides perfect information to each player as they can see the piece on the board. Implementing an AI algorithm is important for this project because games are a competitive exercise of intelligence that give us a structure for planning and decision-making. Moreover, Tic-Tac-Toe's limited search space allows us to more easily visualize/understand the algorithms we are testing.

Algorithm Description

Minimax:

Since Tic-Tac-Toe is a 2 player game, we apply adversarial search. It is used in situations where our goal is to defeat an opponent with the assumption they make the optimal decision. Therefore, max always moves first and min is the opponent. The AI algorithm I choose for the player 2 class is a type of adversarial search called minimax. It chooses the best move by iterating over each cell on the board and using a recursive approach (DFS) to explore all possible moves. The AI's strategy is to maximize their score while minimizing the human player's chances of winning (highest utility). The main disadvantage of minimax is its time complexity O(bm). It is dependent on having a good evaluation function.

Cutoff:

Despite its simplicity, Tic-Tac-Toe can be computationally expensive and time consuming when we explore a large part of a tree. A potential design choice is to use a cutoff depth which specifies how many moves ahead the AI should explore. In my minimax function, it would limit how many levels deep the recursion goes. For example, if set to 3, it explores moves that look 3 moves ahead. A problem with cutoff is the horizon effect, where minimax can only evaluate up to a certain depth, resulting in less ideal moves.

Alpha-Beta Pruning:

To deal with the huge time complexity, I implemented alpha-beta pruning. First, we initialize the worst possible scores for maximizing and minimizing (-inf, inf). If the score is worse than the current best move (>= beta or <= alpha), the branch is pruned and exits the loop early. This allows us to effectively search about twice as deep (best case) by reducing the number of nodes explored. It is guaranteed to compute the same root node value as minimax.

Example Scenario

EX: [X, O, X], [O, X, -], [-, -, O]

my_play: calls minimax with the current board state and boolean since it is the AI's turn (maximizing is true).

minimax winner: checks if there's a winner or the board is full. Nobody has won so continue.

```
for row in range(len(board)):
            for col in range(len(board[0])):
                if board[row][col] == '-':
                    board[row][col] = 'O' if maximizing else 'X'
                    score, _ = self.minimax(board, not maximizing, alpha,
beta)
                    board[row][col] = '-'
```

for loop: iterates over each cell on the board until empty ('-') is found. The cell is temporarily set to 'O' for maximizing or 'X' for minimizing. This simulates a move. The first empty cell [1, 2] is set to 'O'.

Recursive call: calls minimax to explore the opponent's move, switching between maximizing and minimizing. The cell is changed back to empty to avoid affecting other simulations. The next empty cell [2, 0] returns a higher score because it is a winning state.

If the score of the current move > high score then the move for 'O' (maximizing) is better and updated. The best move is [2, 0] as 'X' cannot possibly win ('O' can win or tie).

```
if maximizing and high_score >= beta:
                        break
                    elif not maximizing and high_score <= alpha:
                        break
```

alpha-beta pruning: beta is the best value that the minimizing player can guarantee at the current level. If the high score >= beta then we don't need to explore a branch further because there is a better move elsewhere. The opposite is true for alpha.

Conclusion

Studying adversarial search in the context of games gives us accessible insight into how an AI can play ahead. Overall, a combination of minimax and alpha-beta-pruning is the best algorithm for Tic-Tac-Toe. Evidently, we need to find a good trade-off between time and space complexity when designing search algorithms. To improve my algorithm, I would need better evaluation functions. This is fine for Tic-Tac-Toe as it is a simple game that is solved, meaning a perfect run will always end in a tie. However, for a complex game like chess heuristic functions need to be much more strategic and adaptable.