The Game of Tic-Tac-Toe

Minimax Algorithm & Alpha-Beta Pruning Analysis

Problem 3

1 Problem Overview

Problem Statement

Given Initial Board State:

__	X	__
O	__	__
X	__	O

Objective: Find the optimal move for X (MAX player) using:

- Minimax algorithm with complete game tree expansion
- Alpha-Beta pruning for optimization
- Custom evaluation function for terminal states

1.1 Board Representation

In our implementation, the board is represented as a 3×3 matrix with:

- `'X'` for X player positions
- `'O'` for O player positions
- `None` for empty cells

```
INITIAL_BOARD = [
  [None, 'X', None],
  ['O', None, None],
  ['X', None, 'O']
]
```

2 Evaluation Function

Evaluation Function

Evaluation Formula:

$$\text{Evaluation}(s) = 8X_3(s) + 3X_2(s) + X_1(s) - (8O_3(s) + 3O_2(s) + O_1(s))$$

Where:

- $X_n(s)$ = Number of lines with exactly n X's and no O's
- $O_n(s)$ = Number of lines with exactly n O's and no X's
- Lines include: 3 rows + 3 columns + 2 diagonals = 8 total lines

Intuition: Higher weights for lines closer to completion favor aggressive play.

2.1 Implementation of Evaluation Function

```python
def evaluation(board):
    X3 = X2 = X1 = 0
    O3 = O2 = O1 = 0

    # Check all 8 lines (3 rows + 3 cols + 2 diagonals)
    for line in LINES:
        vals = [board[r][c] for r,c in line]
        xcount = vals.count('X')
        ocount = vals.count('O')

        # Count lines with only X's (no O's)
        if ocount == 0 and xcount > 0:
            if xcount == 3: X3 += 1
            elif xcount == 2: X2 += 1
            elif xcount == 1: X1 += 1

        # Count lines with only O's (no X's)
        if xcount == 0 and ocount > 0:
            if ocount == 3: O3 += 1
            elif ocount == 2: O2 += 1
            elif ocount == 1: O1 += 1

    return 8*X3 + 3*X2 + X1 - (8*O3 + 3*O2 + O1)
```

Listing 1: Evaluation Function Implementation

3 Minimax Algorithm

Algorithm Implementation

Minimax Principle:

- **MAX player (X):** Chooses move that maximizes evaluation
- **MIN player (O):** Chooses move that minimizes evaluation
- **Terminal states:** Return evaluation function value

Algorithm 1 Minimax Algorithm

Require: $board$, $player$, $stats$
  $stats[nodes] \leftarrow stats[nodes] + 1$
  if $is\_terminal(board)$ **then**
    return $evaluation(board)$
  end if
  if $player = X$ (MAX) **then**
    $best \leftarrow -\infty$
    for each $move \in legal\_moves(board)$ **do**
      $new\_board \leftarrow apply\_move(board, move, X)$
      $val \leftarrow minimax(new\_board, O, stats)$
      $best \leftarrow \max(best, val)$
    end for
    return $best$
  else
    $best \leftarrow +\infty$
    for each $move \in legal\_moves(board)$ **do**
      $new\_board \leftarrow apply\_move(board, move, O)$
      $val \leftarrow minimax(new\_board, X, stats)$
      $best \leftarrow \min(best, val)$
    end for
    return $best$
  end if

3.1 Minimax Implementation

```
def minimax(board, player, stats):
    stats['nodes'] += 1

    if is_terminal(board):
        return evaluation(board)

    if player == 'X':  # MAX player
        best = float('-inf')
        for move in get_legal_moves(board):
            new_board = apply_move(board, move, 'X')
            val = minimax(new_board, 'O', stats)
            best = max(best, val)
        return best
    else:  # MIN player (O)
```

```python
        best = float('inf')
        for move in get_legal_moves(board):
            new_board = apply_move(board, move, 'O')
            val = minimax(new_board, 'X', stats)
            best = min(best, val)
        return best
```

Listing 2: Minimax Implementation

4 Alpha-Beta Pruning

Alpha-Beta Pruning Analysis

Alpha-Beta Pruning Optimization:

- **Alpha (α):** Best value MAX can guarantee so far
- **Beta (β):** Best value MIN can guarantee so far
- **Pruning Condition:** If $\alpha \geq \beta$, prune remaining branches

Key Insight: Eliminates branches that cannot affect the final decision, significantly reducing search space.

```python
def alphabeta(board, player, alpha, beta, stats):
    stats['nodes'] += 1

    if is_terminal(board):
        return evaluation(board)

    if player == 'X':  # MAX player
        value = float('-inf')
        for move in get_legal_moves(board):
            new_board = apply_move(board, move, 'X')
            value = max(value, alphabeta(new_board, 'O', alpha, beta,
                stats))
            alpha = max(alpha, value)
            if alpha >= beta:
                stats['prunes'] += 1
                break  # Beta cut-off (prune)
        return value
    else:  # MIN player
        value = float('inf')
        for move in get_legal_moves(board):
            new_board = apply_move(board, move, 'O')
            value = min(value, alphabeta(new_board, 'X', alpha, beta,
                stats))
            beta = min(beta, value)
            if alpha >= beta:
                stats['prunes'] += 1
                break  # Alpha cut-off (prune)
        return value
```

Listing 3: Alpha-Beta Pruning Implementation

5 Experimental Results

5.1 Available Moves Analysis

From the initial board state, X has 5 possible moves:

Position	Coordinates	1-based
(0,0)	Top-left	(1,1)
(0,2)	Top-right	(1,3)
(1,1)	Center	(2,2)
(1,2)	Middle-right	(2,3)
(2,1)	Bottom-center	(3,2)

5.2 Minimax Results

Results & Analysis

Minimax Analysis (Complete Tree Expansion):

Move	Position	Minimax Value	Nodes Expanded
(0,0)	(1,1)	0	47
(0,2)	(1,3)	5	35
(1,1)	(2,2)	5	39
(1,2)	(2,3)	0	61
(2,1)	(3,2)	-8	43

Optimal Move: (1,3) - Top Right position with minimax value = 5
Total Nodes Visited: 225 nodes

5.3 Alpha-Beta Pruning Results

Results & Analysis

Alpha-Beta Pruning Analysis:

Move	Position	Value	Nodes	Prunes
(0,0)	(1,1)	0	20	6
(0,2)	(1,3)	5	17	6
(1,1)	(2,2)	5	31	6
(1,2)	(2,3)	0	42	14
(2,1)	(3,2)	-8	34	6

Optimal Move: (1,3) - Center position (same as minimax)
Total Nodes Visited: 144 nodes
Total Prunes: 38 pruning operations

6 Strategic Analysis

6.1 Why Center (1,3) is Optimal

Board after optimal move (1,3):

__	X	X
O	__	__
X	__	O

Strategic Advantages:

1. **Central Control:** Center position is part of 4 lines (row, column, both diagonals)
2. **Threat Creation:** Creates multiple potential winning lines
3. **Blocking Power:** Controls key intersections for opponent
4. **Evaluation Score:** Maximizes X_2 terms in evaluation function

6.2 Evaluation Breakdown for Optimal Move

After playing X at (1,1), the evaluation considers:

- **X lines:** Main diagonal (2 X's), middle column (2 X's), middle row (2 X's)
- **O lines:** Bottom row (2 O's)
- **Score:** $3 \cdot 3 \cdot X_2 - 3 \cdot 1 \cdot O_2 = 9 - 3 = 6$

7 Complexity Analysis

Algorithm Implementation

Time Complexity:

- **Minimax:** $O(b^d)$ where b is branching factor, d is depth
- **Alpha-Beta:** $O(b^{d/2})$ in best case (perfect ordering)
- **Tic-Tac-Toe:** $b \leq 9$, $d \leq 9$ (decreasing each level)

Space Complexity: $O(d)$ for recursion stack
Observed Performance:

- Minimax: 225 nodes explored
- Alpha-Beta: 144 nodes explored

8 Key Algorithmic Insights

1. **Evaluation Function Design:** The weighted scoring system effectively captures positional strength and winning potential.
2. **Pruning Efficiency:** Alpha-beta pruning achieved significant performance gains while maintaining optimality.
3. **Strategic Preferences:** The center position dominance demonstrates the importance of controlling multiple lines simultaneously.
4. **Move Ordering:** Better move ordering could further improve alpha-beta efficiency.

9 Conclusion

The analysis demonstrates that:

- The **Top Right position (1,3)** is the optimal move with minimax value 5
- **Alpha-beta pruning** significantly reduces computation while preserving optimality
- The **evaluation function** effectively guides strategic play toward winning positions
- Both algorithms agree on the optimal strategy, validating correctness