Tic-Tac-Toe AI

Arafat Anam Chowdhury - 203 1514 042 | Rajib Mohammed Nasim - 203 1050 042 | Khandker Shakila Rowshan Puspita - 211 1886 042

This project creates an AI Tic-Tac-Toe agent using Minimax and alpha-beta pruning algorithms to play optimally. It includes a user-friendly interface for interactive gameplay.

About Our Project:

- Developing an AI agent for Tic-Tac-Toe.
- Utilizing advanced game-tree search techniques: Minimax and Alpha-Beta Pruning.
- Improving decision-making with heuristics.

Code Modules:

- **ai.py:** Implements Minimax, Alpha-Beta Pruning, and Heuristic evaluation.
- **game.py:** Handles rules and game mechanics.
- **main.py:** GUI for the project

Core AI Techniques

Minimax Algorithm:

- Ensures optimal decisions by evaluating all possible moves.
- Recursively computes scores for terminal states (win/loss/draw).

Alpha-Beta Pruning:

- Skips unnecessary game tree branches to improve efficiency.
- Maintains optimal outcomes while reducing search depth.

Heuristic Values:

- Assign scores to intermediate game states.
- Guides AI decision-making without full tree exploration.

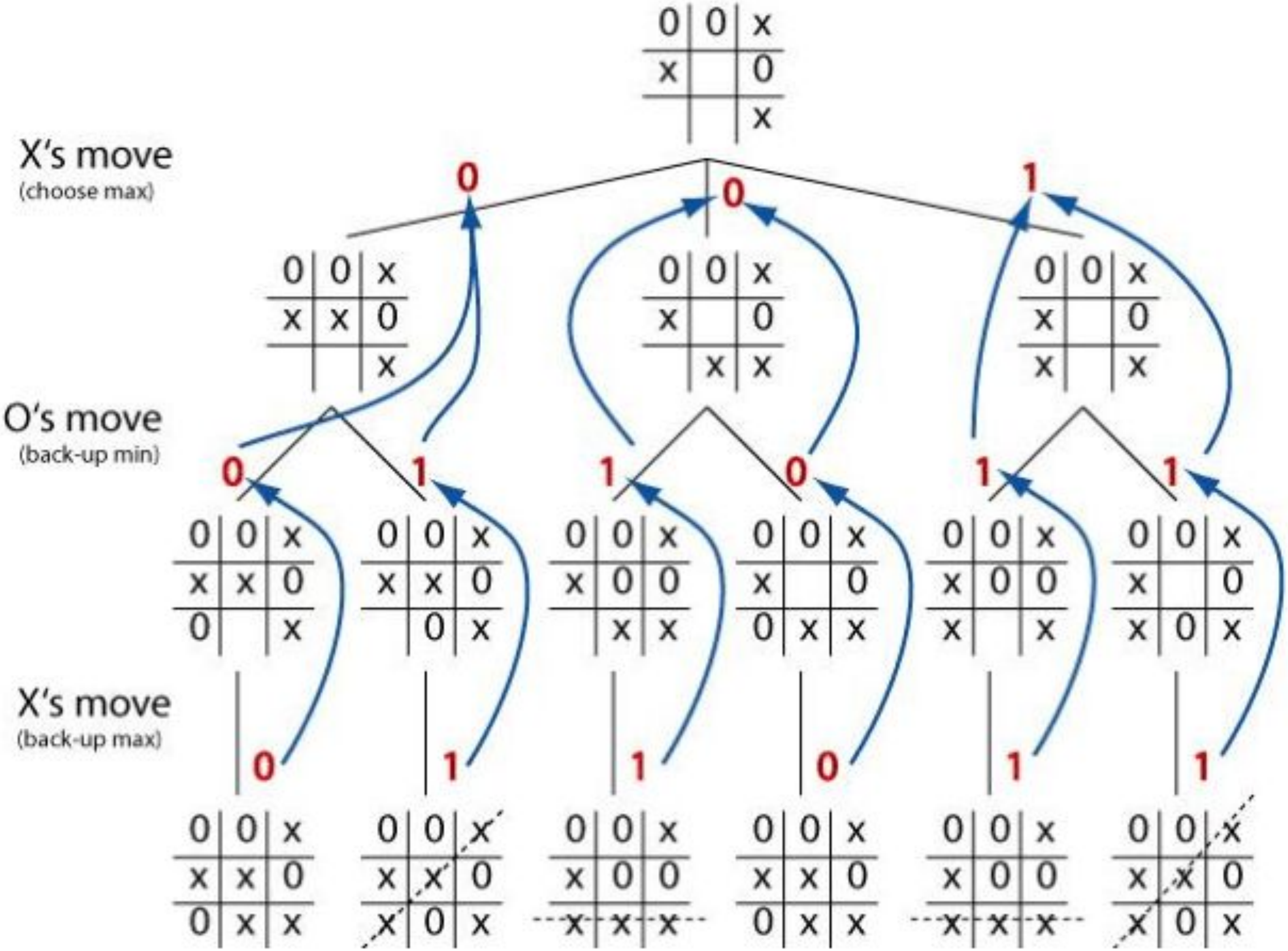
How Minimax Works

Steps:

1. Generate the entire game tree.
2. Evaluate scores for terminal states.
3. Propagate scores back to determine the optimal move.

Example:

- *Terminal States:*
  - Win = +10, Loss = -10, Draw = 0.
- *Intermediate State Propagation:* Show moves with back-propagated scores.



X's move
(choose max)
O's move
(back-up min)
X's move
(back-up max)

How Alpha Beta Pruning Works

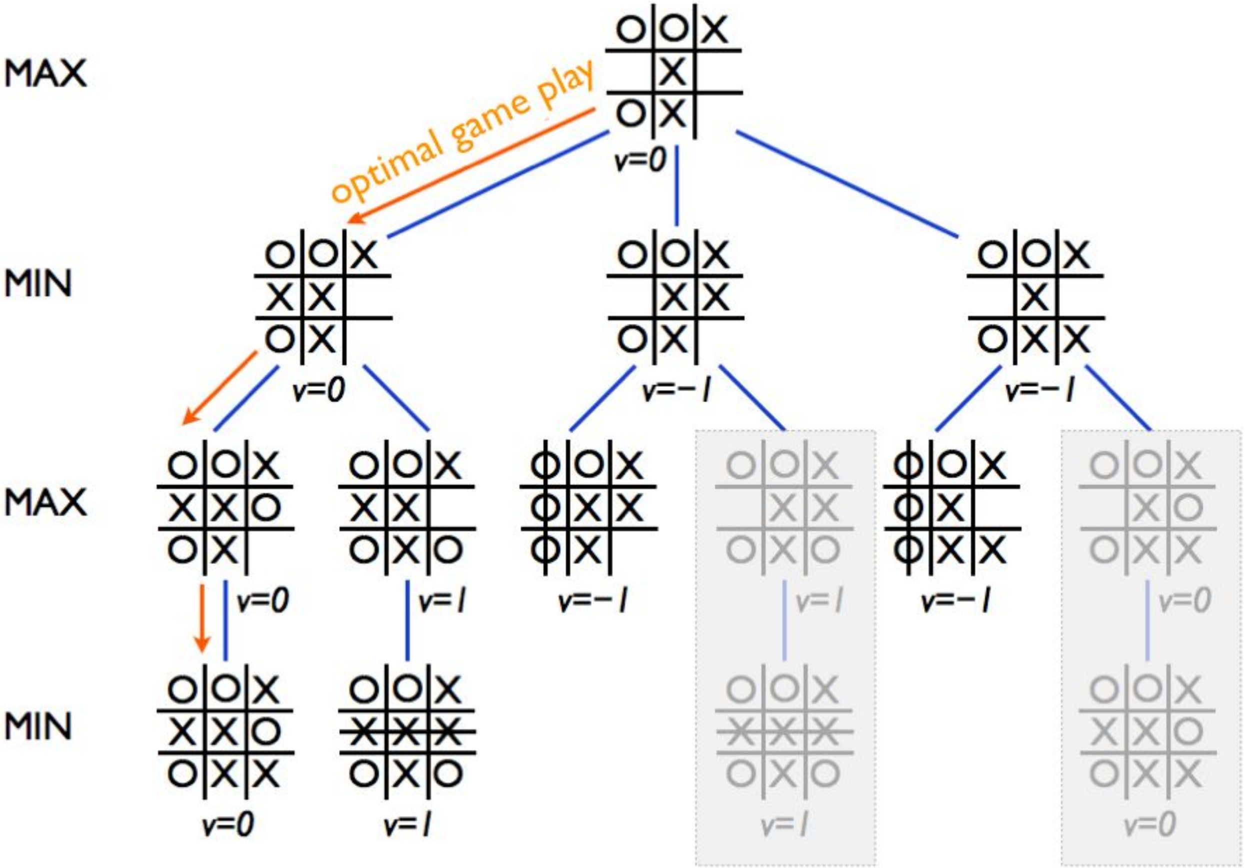
Keeps track of two bounds:

- Alpha = Best score Maximizer can guarantee.
- Beta = Best score Minimizer can guarantee.

- Prunes branches where the outcome won't influence the final decision.

Advantages:

- Reduces computational load by skipping irrelevant paths.
- Speeds up decision-making.



MAX
MIN
MAX
MIN
optimal game play
v=0
v=0
v=−1
v=−1
v=0
v=1
v=−1
v=1
v=−1
v=0
v=0
v=1
v=1
v=0

```
def minimax(self, state, depth, alpha, beta, is_maximizing):
  if state.current_winner == self.letter:
      return 100 - depth  # Prioritize faster wins
  elif state.current_winner == ('O' if self.letter == 'X' else 'X'):
      return -100 + depth  # Penalize slower losses
  elif not state.empty_squares():
      return 0  # Tie game

  if depth > 3:
      return self.heuristic(state)

   if is_maximizing:
      max_eval = -math.inf  # Maximizing player (AI)
      for move in state.available_moves():
          state.make_move(move, self.letter)
          sim_score = self.minimax(state, depth + 1, alpha, beta, False)
          state.board[move] = ' '  # Undo move
          state.current_winner = None  # Reset winner
          max_eval = max(max_eval, sim_score)
          alpha = max(alpha, sim_score)
          if beta <= alpha:  # Alpha-beta pruning
              break
      return max_eval
```

```python
    else:
        min_eval = math.inf  # Minimizing player (opponent)
        opponent = 'O' if self.letter == 'X' else 'X'
        for move in state.available_moves():
            state.make_move(move, opponent)
            sim_score = self.minimax(state, depth + 1, alpha, beta, True)
            state.board[move] = ' '  # Undo move
            state.current_winner = None  # Reset winner
            min_eval = min(min_eval, sim_score)
            beta = min(beta, sim_score)
            if beta <= alpha:  # Alpha-beta pruning
                break
        return min_eval

  def get_move(self, game):

    best_move = None
    best_score = -math.inf
    for move in game.available_moves():
        game.make_move(move, self.letter)
        score = self.minimax(game, 0, -math.inf, math.inf, False)
        game.board[move] = ' '  # Undo move
        game.current_winner = None  # Reset winner
        if score > best_score:  # Update best move if a better score is found
            best_score = score
            best_move = move
    return best_move
```

Use of Heuristics

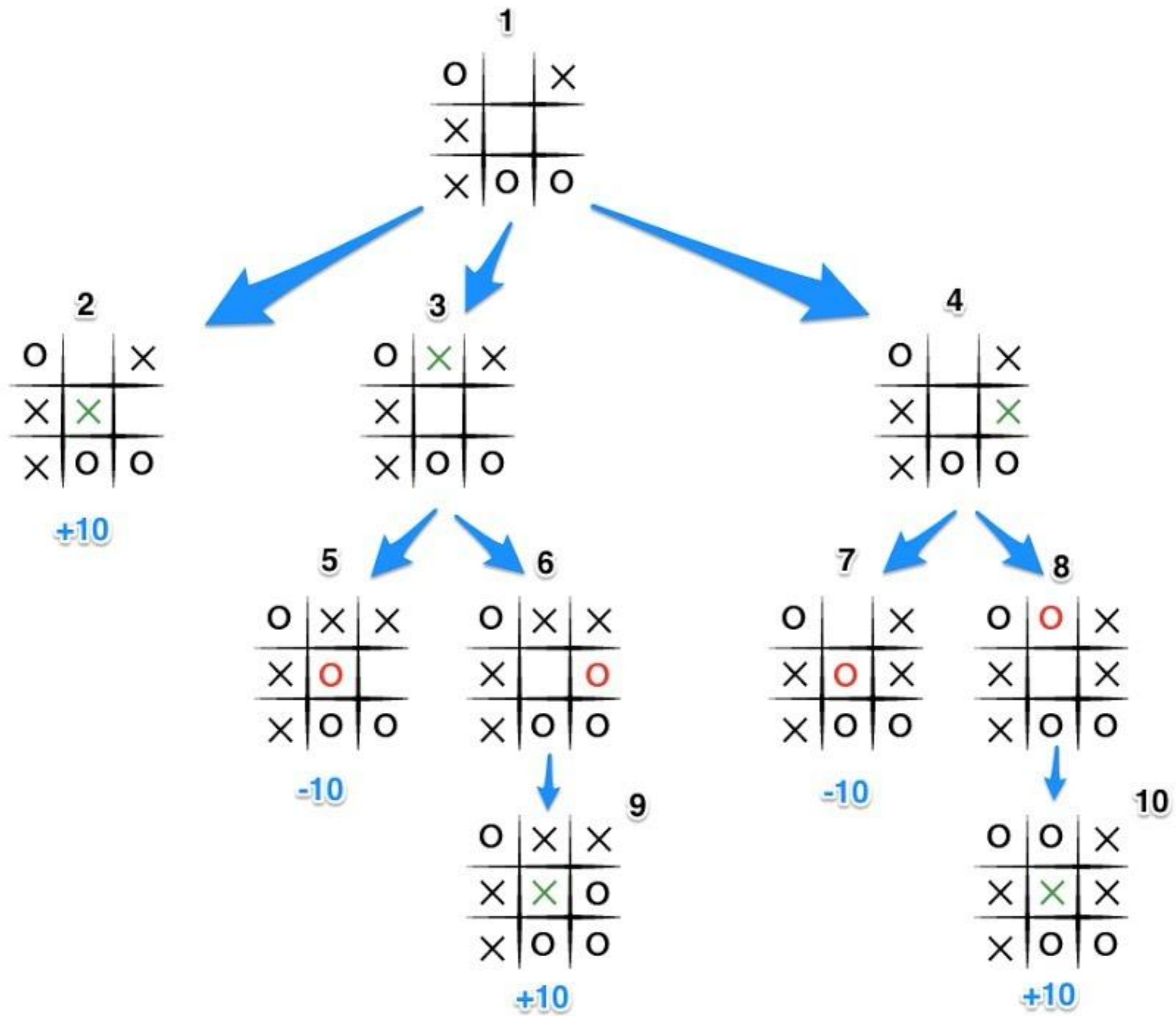
Heuristic Evaluation

- Assign heuristic scores to non-terminal states.
- Combine heuristics with depth-limited Minimax for efficiency.

Tic-Tac-Toe Heuristic Strategy:

- Win = +10, Loss = -10, Neutral = 0.
- Example board with highlighted heuristic scores for potential moves.

```python
def heuristic(self, state):
  opponent = 'O' if self.letter == 'X' else 'X'
  score = 0
  winning_lines = [
      [0, 1, 2], [3, 4, 5], [6, 7, 8],  # Rows
      [0, 3, 6], [1, 4, 7], [2, 5, 8],  # Columns
      [0, 4, 8], [2, 4, 6]]             # Diagonals
  for line in winning_lines:

      ai_count = sum([1 for i in line if state.board[i] == self.letter])

      opponent_count = sum([1 for i in line if state.board[i] == opponent])

      empty_count = sum([1 for i in line if state.board[i] == ' '])

      # AI advantage
      if ai_count == 2 and empty_count == 1:
          score += 10  # AI is winning
      elif ai_count == 1 and empty_count == 2:
          score += 1  # AI might win

      # Opponent threat
      if opponent_count == 2 and empty_count == 1:
          score -= 10  # Opponent is winning
      elif opponent_count == 1 and empty_count == 2:
          score -= 1  # Opponent might win
  return score
```


1
2
3
4
+10
5
6
7
8
-10
-10
9
10
+10
+10

Conclusion

Key Achievements:

- Minimax guarantees optimal moves.
- Alpha-Beta Pruning optimizes search efficiency.
- Heuristics balance speed and decision quality.

This project successfully demonstrates the application of game tree search algorithms in AI through a simple yet effective Tic-Tac-Toe game. The AI achieves optimal gameplay by implementing Minimax with alpha-beta pruning and incorporating heuristics while providing users with an interactive platform.