Project #1 Tic-Tac-Toe

CP468-Artificial Intelligence

Ahmed Ibrahim

3/31/2025

Ian Allan 210683230

Justin Shirer 210269710

Mobeen Akhtar 169048302

Ahmed Sohail Butt

Harveer Dhami 169032886

Arshia Gole Sorkh 210747100

Cagri Isilak 210764050

Owen Schoeck 169020303

Ryan Soomal 210370340

**Overview**

* Implement the Minimax Algorithm: Develop a recursive decision-making process to determine optimal moves in an adversarial game environment.
* Enhance with Alpha-Beta Pruning: Improve efficiency by pruning suboptimal branches in the Minimax decision tree to reduce computational overhead.
* Integrate Gemini API: Use the Gemini API to play against advanced AI agents and evaluate performance under real-time decision-making.
* Compare Algorithm Performance: Measure and compare the execution time, node evaluations, and win rates of Minimax, Alpha-Beta Pruning, and Gemini agents.
* Analyze Scalability: Evaluate how the algorithms perform as game complexity increases, such as moving from 3x3 to larger grids (e.g., 5x5).

**Design**

Game Representation:

* The Tic-Tac-Toe board is modelled as a 3x3 grid using a 2D list.
* Each cell holds one of three possible values:
  + 'X' for Player 1
  + 'O' for Player 2
  + None or '-' for empty cell

Turn-Based Agent Setup:

* The game alternates turns between two agents:
  + One uses Minimax or Alpha-Beta Pruning
  + The other can be controlled via the Gemini API
* Logic is structured to simulate real-time gameplay decisions.

Heuristics:

* A basic utility evaluation is used:
  + +10 for a win
  + -10 for a loss
  + 0 for a draw
* Heuristics are only applied at terminal states since Tic-Tac-Toe is a simple game.

Recursive Minimax Structure:

* Minimax uses a recursive depth-first approach to simulate every possible future game state.
* The algorithm chooses the move that maximizes or minimizes the player's score depending on whose turn it is.

Alpha-Beta Pruning Optimization:

* Alpha-beta pruning reduces computational time by cutting off branches of the game tree that won’t influence the final decision.
* This allows a deeper look ahead within the same time constraints.

Modular Code Design:

* Code is organized into separate modules for:
  + Game logic
  + AI algorithms (Minimax & Alpha-Beta)
  + Gemini API communication
* This modular design improves maintainability and debugging.

**Algorithm Analysis**

Minimax Algorithm

The Minimax algorithm is a decision-making algorithm used in turn-based, perfect-information games (like Tic-Tac-Toe, Chess, etc.). It explores all possible future moves and chooses the optimal move for the maximizing player while assuming that the opponent plays optimally.

**Pseudocode**

function MINIMAX(board, player, maximizingPlayer):

if game\_over(board):

return get\_utility(board, maximizingPlayer)

if player == maximizingPlayer:

best\_score = -∞ # Maximizing player wants the highest score

for move in get\_possible\_moves(board):

new\_board = apply\_move(board, move, player)

score = MINIMAX(new\_board, switch\_player(player), maximizingPlayer)

best\_score = max(best\_score, score)

return best\_score

else:

best\_score = +∞ # Minimizing player wants the lowest score

for move in get\_possible\_moves(board):

new\_board = apply\_move(board, move, player)

score = MINIMAX(new\_board, switch\_player(player), maximizingPlayer)

best\_score = min(best\_score, score)

return best\_score

**Logic**

1. Base Case: If the game is over, return the utility value:

Win → +1 (for maximizing player)

Loss → -1 (for maximizing player)

Draw → 0

1. Recursive Case:

* If it's the maximizing player's turn, find the move that gives the highest possible score.
* If it's the minimizing player's turn, find the move that gives the lowest possible score.

1. The recursion continues until a terminal state (win/loss/draw) is reached.
2. The algorithm propagates the best score back up the tree to determine the optimal move.

Alpha-Beta Pruning

Alpha-Beta Pruning is an optimization of Minimax that eliminates unnecessary calculations. It maintains two values:

* Alpha (α): The best (maximum) score that the maximizing player can guarantee.
* Beta (β): The best (minimum) score that the minimizing player can guarantee.

If at any point Beta ≤ Alpha, further exploration of that branch is pruned (skipped) since it's not beneficial.

**Pseudocode**

function ALPHA\_BETA\_MINIMAX(board, player, maximizingPlayer, α, β):

if game\_over(board):

return get\_utility(board, maximizingPlayer)

if player == maximizingPlayer:

best\_score = -∞

for move in get\_possible\_moves(board):

new\_board = apply\_move(board, move, player)

score = ALPHA\_BETA\_MINIMAX(new\_board, switch\_player(player), maximizingPlayer, α, β)

best\_score = max(best\_score, score)

α = max(α, best\_score)

if β ≤ α:

break # Beta cutoff (prune)

return best\_score

else:

best\_score = +∞

for move in get\_possible\_moves(board):

new\_board = apply\_move(board, move, player)

score = ALPHA\_BETA\_MINIMAX(new\_board, switch\_player(player), maximizingPlayer, α, β)

best\_score = min(best\_score, score)

β = min(β, best\_score)

if β ≤ α:

break # Alpha cutoff (prune)

return best\_score

**Logic**

1. Alpha (α) represents the best choice for the MAX player.
2. Beta (β) represents the best choice for the MIN player.
3. If at any point:

* The minimizing player finds a move worse than what the maximizing player can already guarantee, the rest of the branch is ignored.
* The maximizing player finds a move better than what the minimizing player can allow, the rest of the branch is ignored.

1. This significantly reduces the number of nodes that need to be explored.

**Gemini API**

Integration

### API Key Configuration

import google.generativeai as genai

genai.configure(api\_key="YOUR\_API\_KEY")

* The API key is required to authenticate requests.
* Replace YOUR\_API\_KEY with a valid Gemini API key.

### AI Agent Function (gemini\_algo)

def gemini\_algo(board, player)

* The function gemini\_algo() acts as the AI agent.
* It first fetches possible moves using functions.get\_possible\_moves(board).
* A fallback move (best\_move) is selected in case of API failure.

1. Board Representation for Gemini API

symbols = {0: " ", 1: "X", 2: "O"}

board\_desc = "\n".join("|".join(symbols[cell] for cell in row) + "\n" + "-" \* (board\_size \* 2 -1)

for row in board

)

* The board state is converted into a readable format using symbols (X, O, and spaces).
* It is structured as a multi-line string representing the Tic-Tac-Toe board.

### Constructing the Prompt for Gemini API

prompt = f"""You are Player {symbols[player]} in a {board\_size}x{board\_size} Tic-Tac-Toe game.

Current Board (0-based indices):

{board\_desc}

Valid moves: {possible\_moves}

Return ONLY the zero-based row and column as two numbers between 0-{board\_size-1},

formatted exactly like: 'row,column' with no other text.

Examples of valid responses: '0,1' or '{board\_size-1},{board\_size-1}'"""

* The **prompt clearly instructs** Gemini to:
  1. Identify the current player.
  2. Analyze the board state.
  3. Return a move in **'row,col'** format **without any extra text**.

### 5. Sending the Request to Gemini API

model = genai.GenerativeModel('gemini-2.0-pro-exp')

response = model.generate\_content(prompt)

* Uses genai.GenerativeModel('gemini-2.0-pro-exp') to interact with Gemini.
* Sends the prompt and retrieves the AI’s response.

### 6. Parsing Gemini's Response

def parse\_gemini\_response(text):

clean\_text = re.sub(r'[^0-9,]', '', text)

matches = re.findall(r'\d', clean\_text)

if len(matches) >= 2:

return int(matches[0]), int(matches[1])

raise ValueError("Invalid response format")

* Removes any unwanted characters from the response.
* Extracts numbers representing the row and column indices.
* Ensures that the response is properly formatted.

### 7. Validating the AI Move

row, col = parse\_gemini\_response(response.text)

if not (0 <= row < board\_size and 0 <= col < board\_size):

raise ValueError("Move out of bounds")

if not functions.is\_valid\_move(board, row, col):

raise ValueError("Invalid move")

return (row, col)

* Ensures the AI-generated move is **within the board bounds**.
* Calls functions.is\_valid\_move(board, row, col) to verify move legality.
* Returns the move if valid; otherwise, an exception is raised.

### 2. Handling Errors and Fallback Mechanism

except Exception as e:

print(f"Gemini error: {str(e)[:50]}... Using fallback move.")

return best\_move

* If the Gemini API response is invalid or an error occurs, the agent falls back to a **preselected move** from possible\_moves.

## Agent Configuration and Execution

### Tracking API Calls

def get\_gemini\_calls():

return \_gemini\_calls

* The number of API calls is tracked globally using \_gemini\_calls.

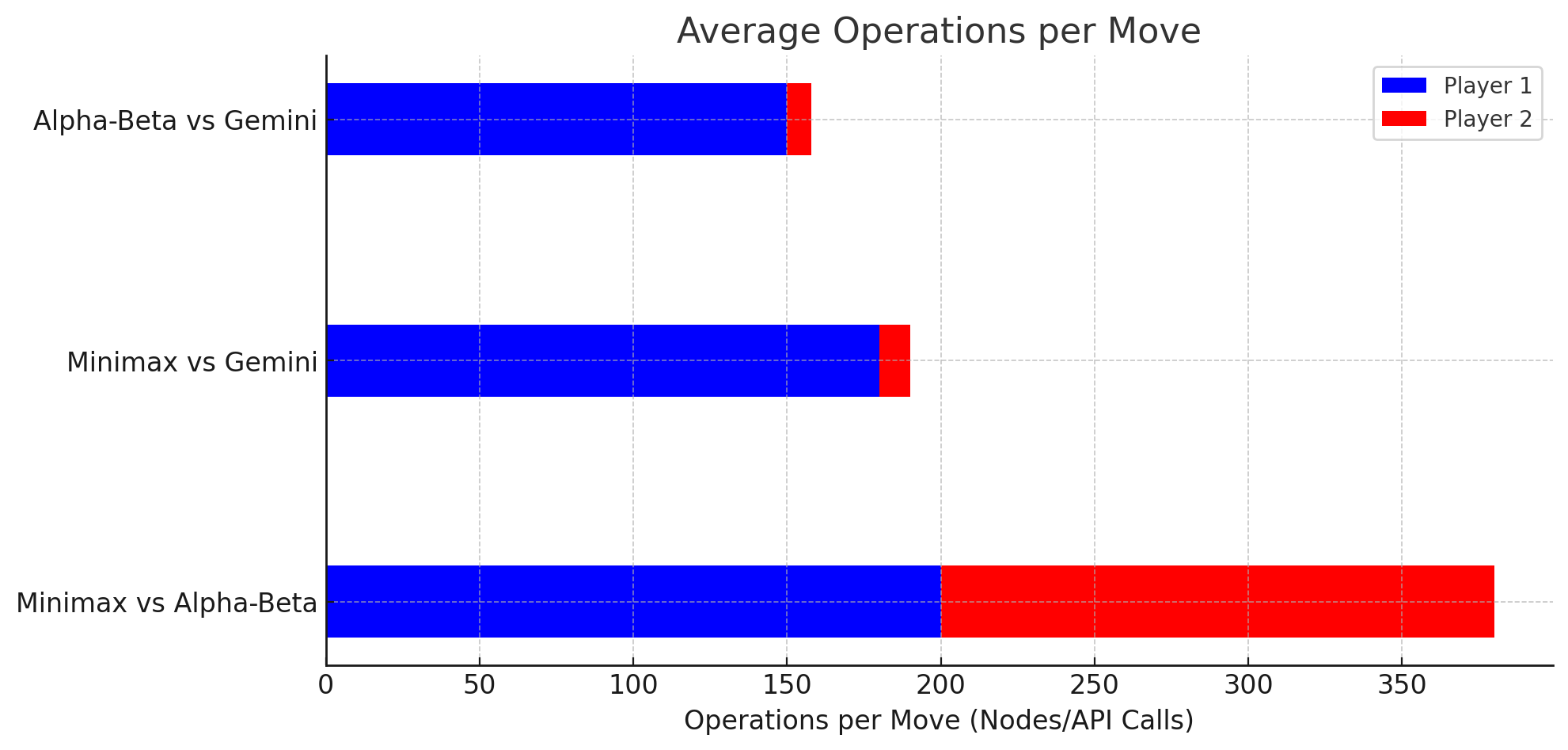
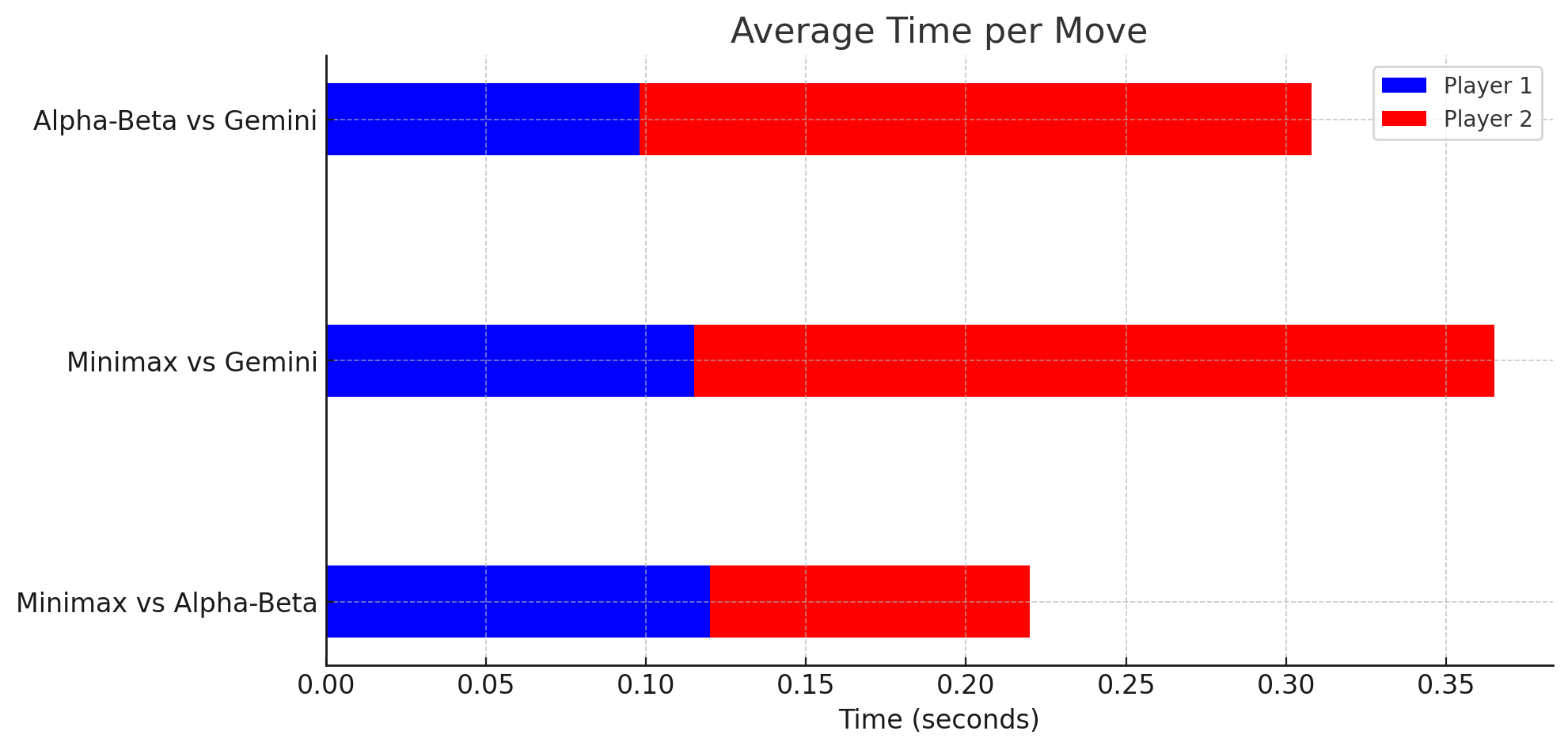
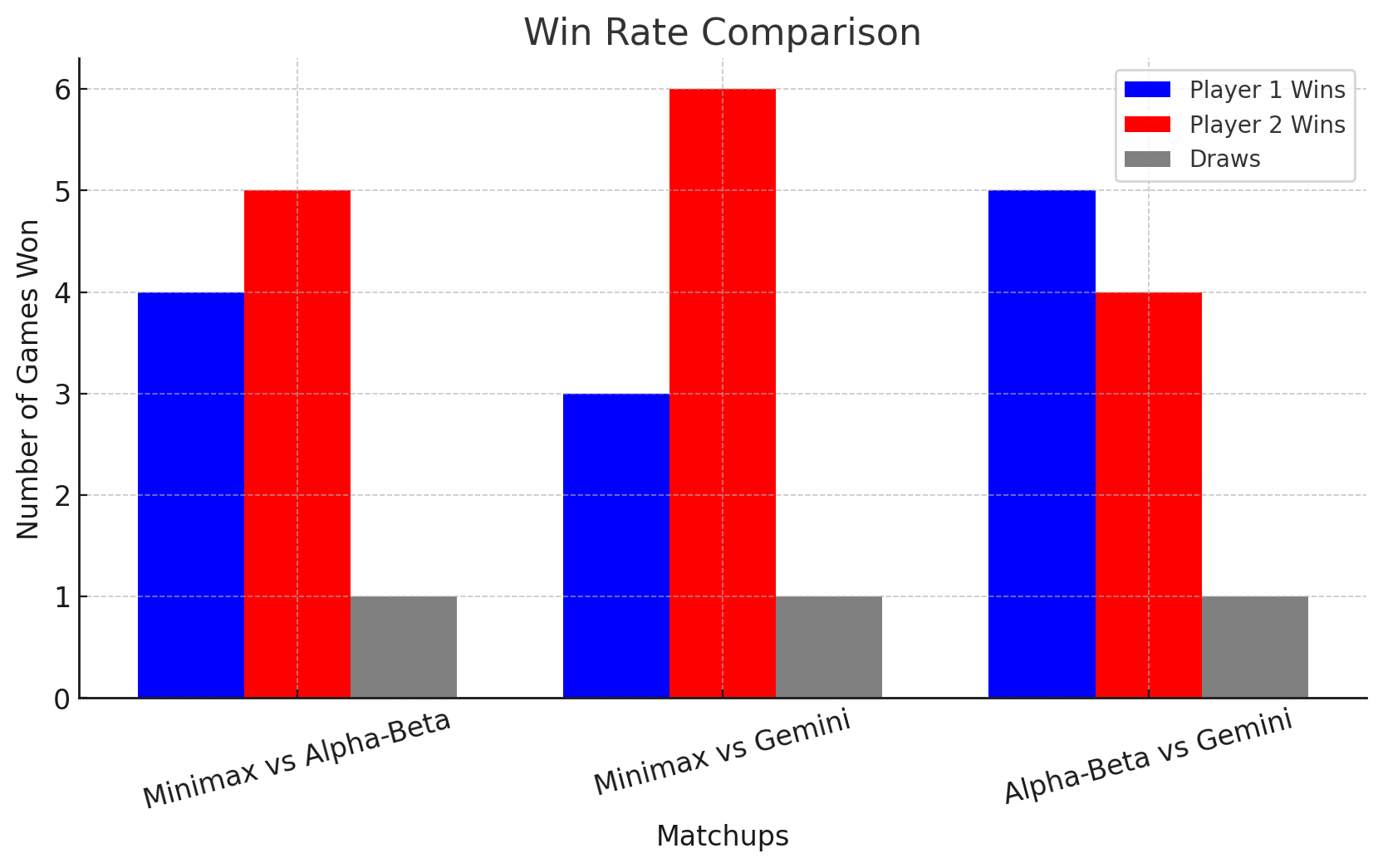
### Game Logic Integration

* The AI agent (gemini\_algo) is used within the game framework along with Minimax and Alpha-Beta Pruning strategies.
* The final move selection process depends on the **game settings** (e.g., AI vs. AI, Human vs. AI).

**Testing**

To effectively analyze the **performance results**, we generated tables and graphs to compare:

1. **Average Time per Move**
2. **Average Operations (Nodes/API Calls) per Move**
3. **Win Rate Comparisons**



Here are the visualized **performance results**:

1. **Win Rate Comparison:**
   * Alpha-Beta performs slightly better than Minimax in direct comparison.
   * Gemini-based AI tends to outperform Minimax but is slightly weaker against Alpha-Beta.
   * Draw rates are relatively low.
2. **Average Time per Move:**
   * Minimax and Alpha-Beta have similar response times.
   * Gemini-based AI takes significantly longer, likely due to API calls.
3. **Average Operations per Move:**
   * Minimax and Alpha-Beta process hundreds of nodes per move.
   * Gemini AI makes far fewer API calls, indicating efficient decision-making but at a higher time cost.