# ARTIFICIAL INTELLIGENCE PROJECT

**TIC-TAC-TOE**

| Name | Id | Level | القسم |
|---|---|---|---|
| مريم هاني رفعت ابراهيم | 20220473 | 3 | AI |
| جودي محمد اسامة عبيدو | 20220127 | 3 | CS |
| اية محمد عبدالحليم محمود الشيخ | 20220096 | 3 | CS |
| مريم سيد محمود محمد | 20220465 | 3 | CS |
| منة الله محمد مصطفي طه خليفة | 20220507 | 3 | CS |
| منة الله هيثم صلاح حسن | 20220508 | 3 | CS |

## Code and documentation link

https://drive.google.com/file/d/1-4zA2bUUWWIgGoT5DGQ_ncmKDo2o3ZAR/view?usp=drivesdk

https://github.com/Judyzox/AI-tic-tac-toe-project.git

# Introduction and overview

## -Project Idea and overview:

**An Intelligent Tic-Tac-Toe Player**

- Tic-Tac-Toe is a two-player game where each player takes turns marking spaces on a 3x3 grid. The objective is to get three of their marks in a horizontal, vertical, or diagonal row.
- This project aims to develop an intelligent Tic-Tac-Toe player that can compete against human players and other AI opponents.
- The AI will use advanced algorithms to ensure optimal gameplay.

## -Similar Applications:

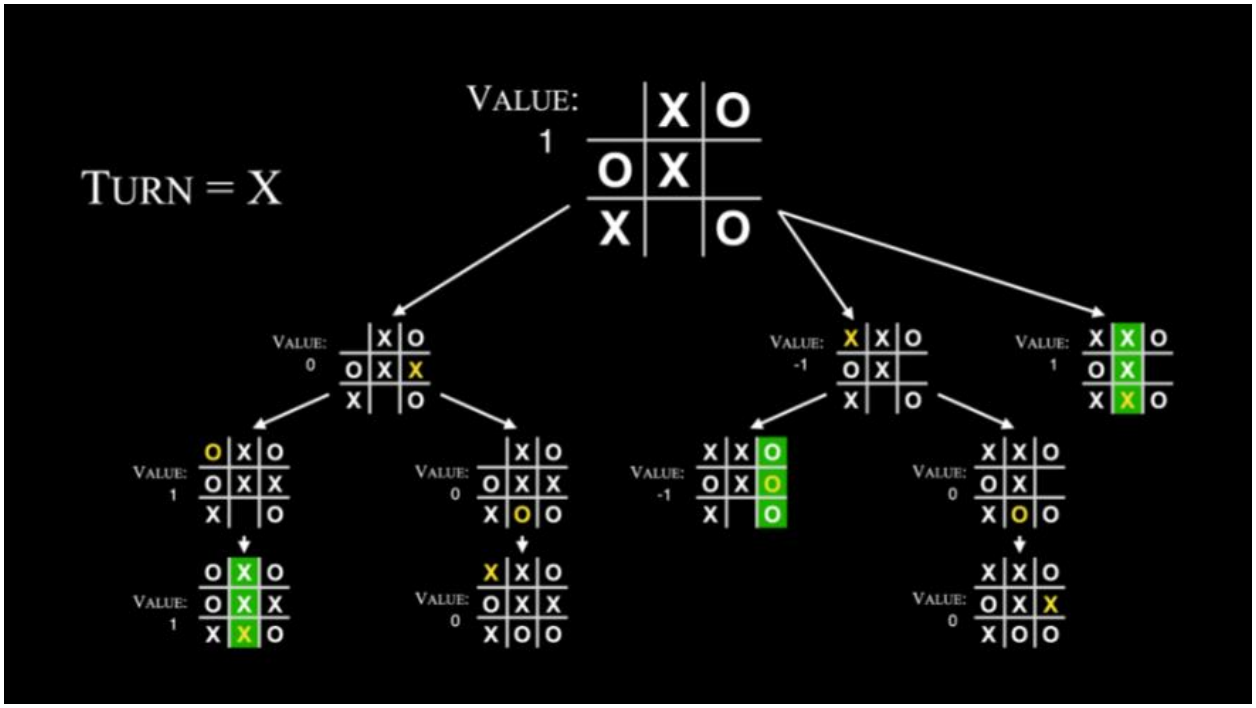
- **AI for Strategic Decision Making (Minimax and Alpha-Beta Pruning):**
  - **Poker AI**: AI players use Minimax with Alpha-Beta Pruning to analyze potential outcomes and make optimal betting decisions based on the current state of the game.
  - **Go**: While more advanced algorithms like Monte Carlo Tree Search are more common, simpler versions of Go AI use Minimax for strategic decision-making.
- **Chatbots.**
  - **Chatbots:** interact with customers to solve problems, answer questions, or guide them through processes, like how AI in Tic-Tac-Toe game responds to player moves.
- **Robotics and Planning (Hill Climbing and Greedy Best-First)**:
  - **Robot Path Planning**: Autonomous robots use these algorithms for obstacle avoidance and efficient route planning, using Greedy Best-First Search to focus on the shortest path or Hill Climbing for local optimization.

## - A Literature Review of Academic publications.

1. https://www.geeksforgeeks.org/finding-optimal-move-in-tic-tac-toe-using-minimax-algorithm-in-game-theory/

2. https://www.neverstopbuilding.com/blog/minimax

3. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

4. 

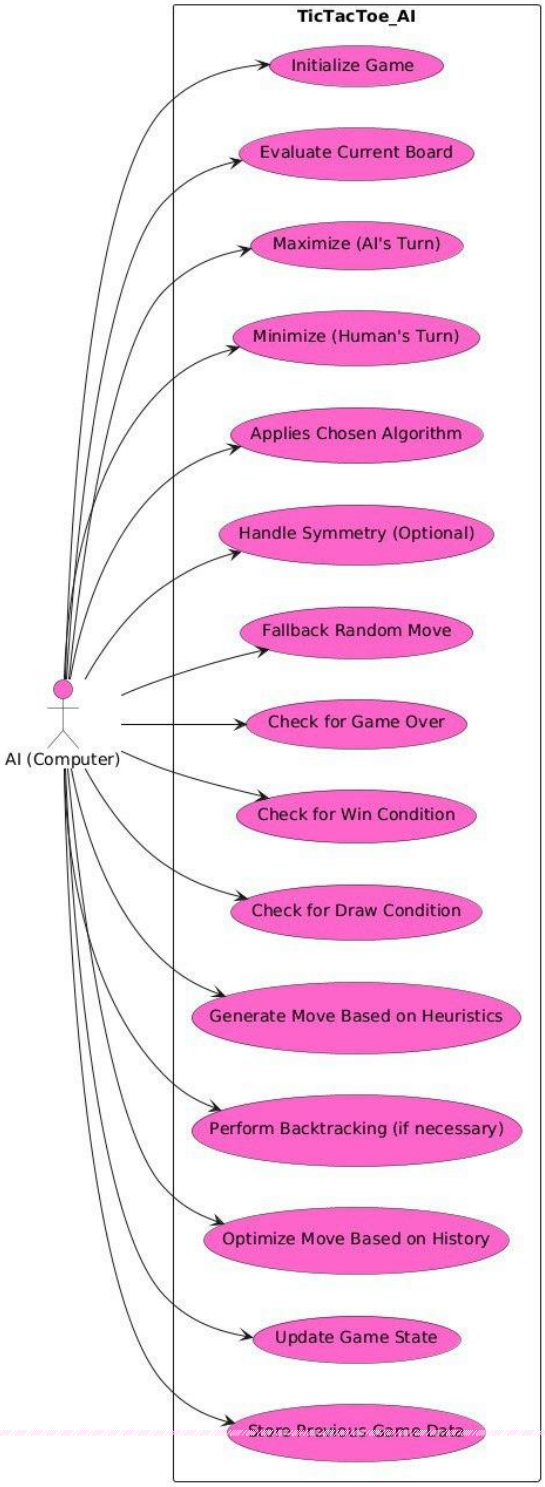

5. https://uyennguyen16900.medium.com/minimax-with-alpha-beta-pruning-7e2091ae7d95

# Proposed Solution:

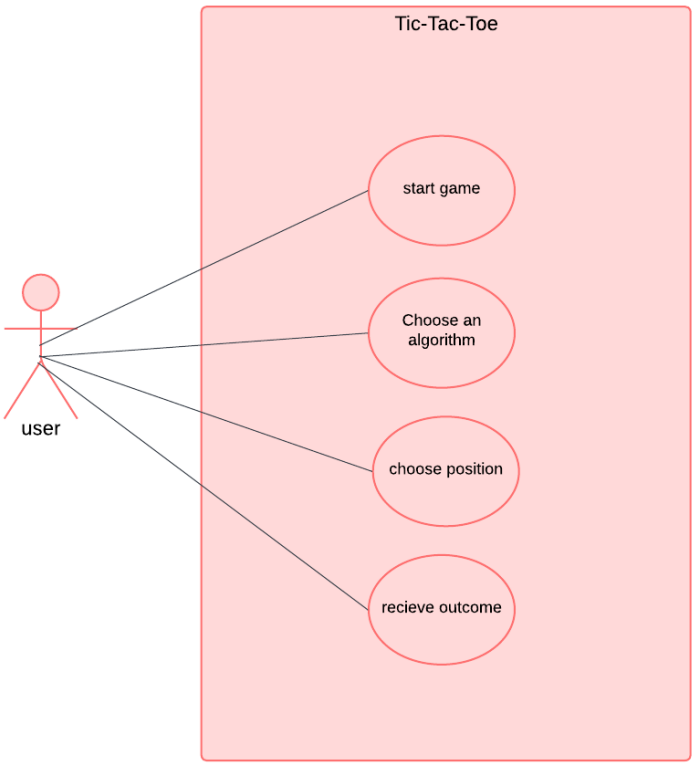
## -Main functionalities (from the users' perspective):

### Use-Case Diagram:

### AI Agent point of view:

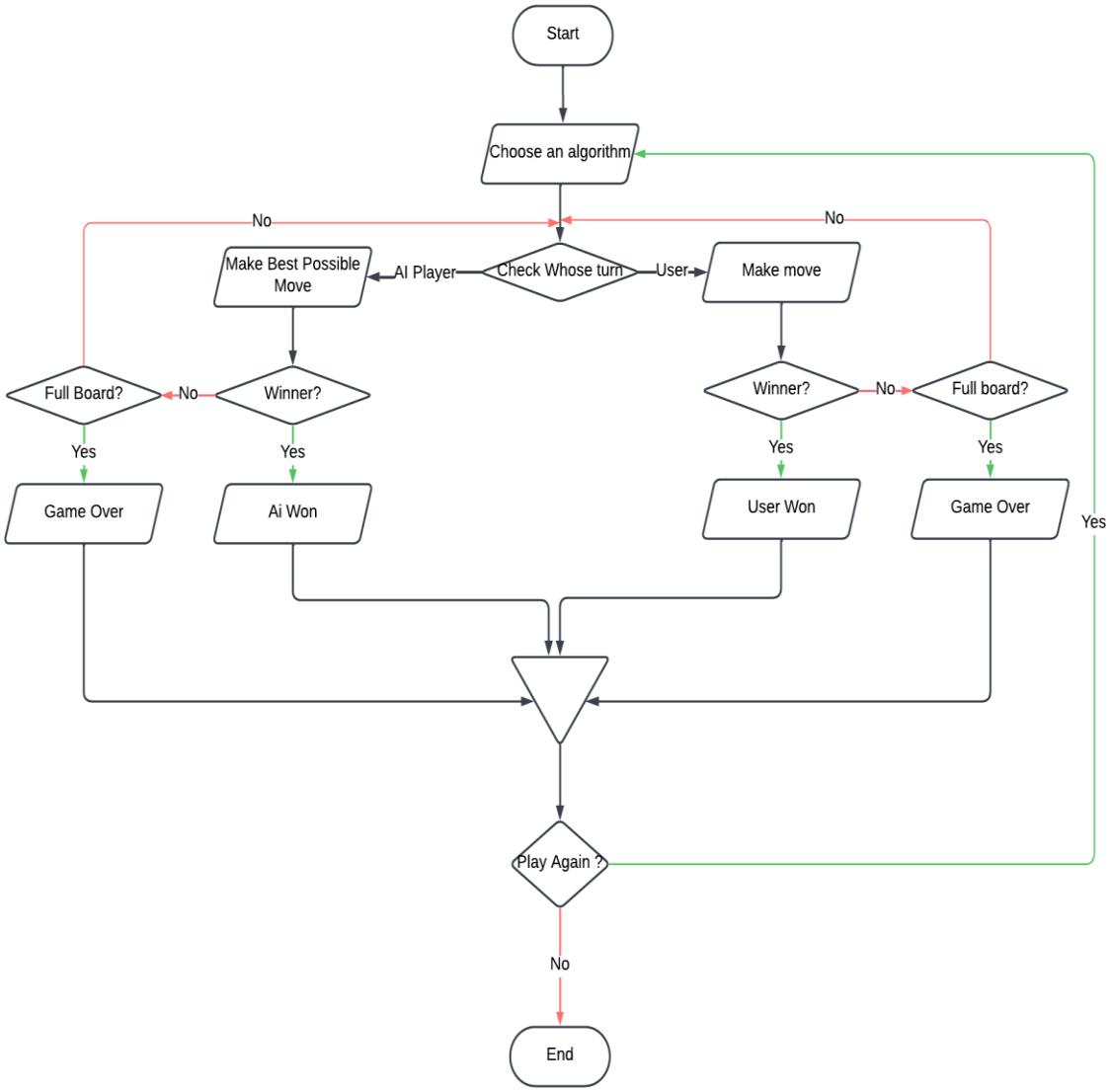



### User point of view:



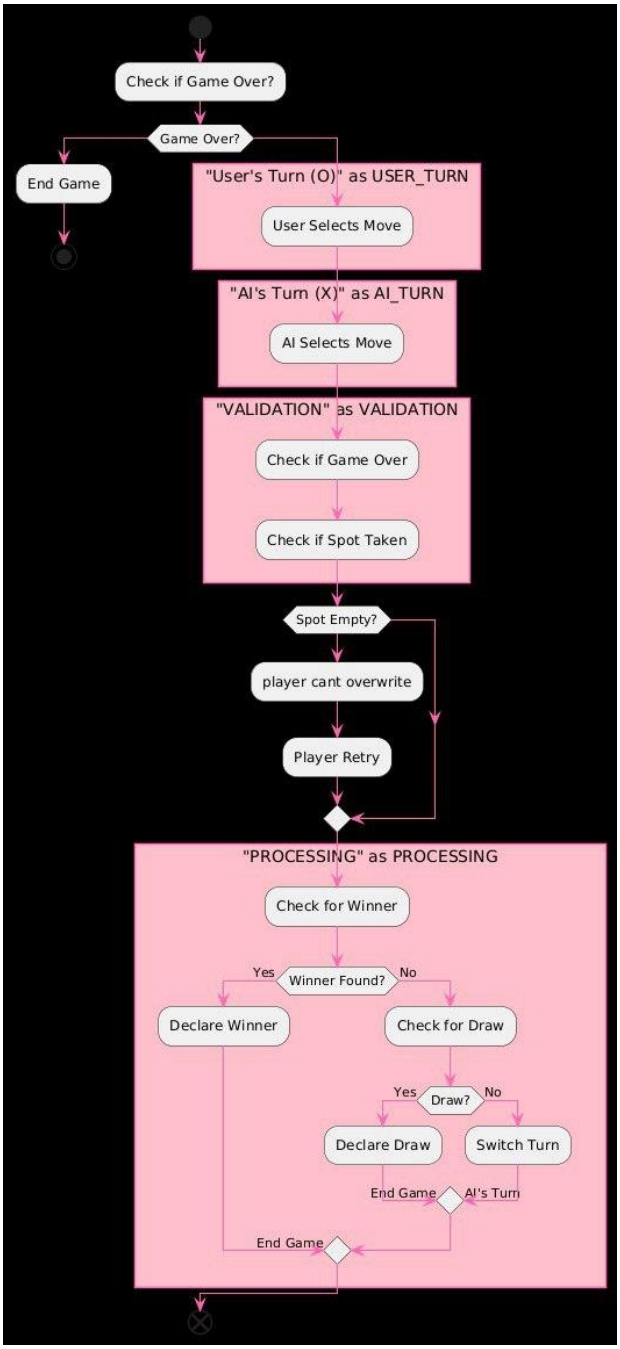
# System overview:

## Flowchart:



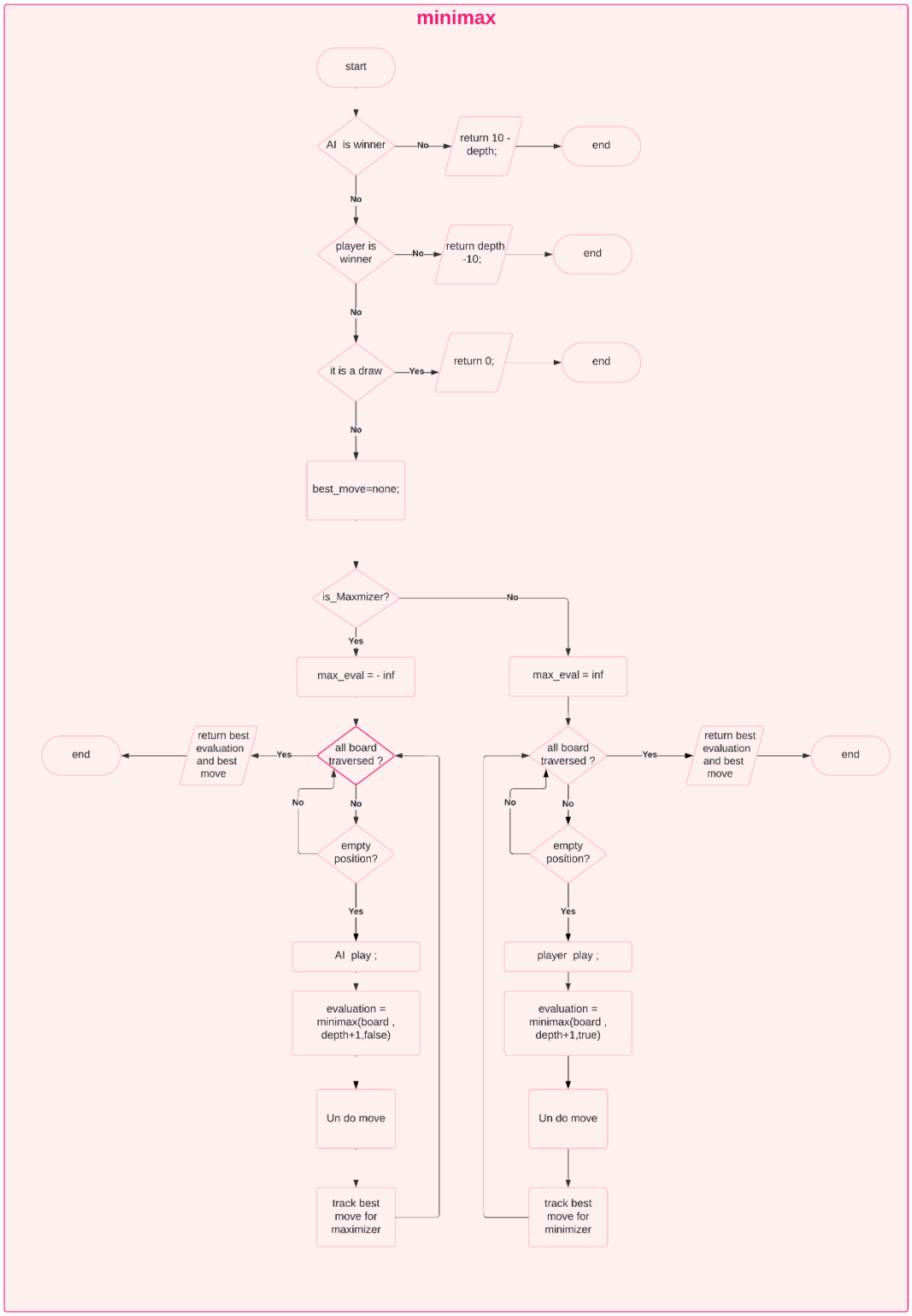
## Block Diagram:

- It's a decision-making algorithm used in two-player games to minimize the possible loss in a worst-case scenario.

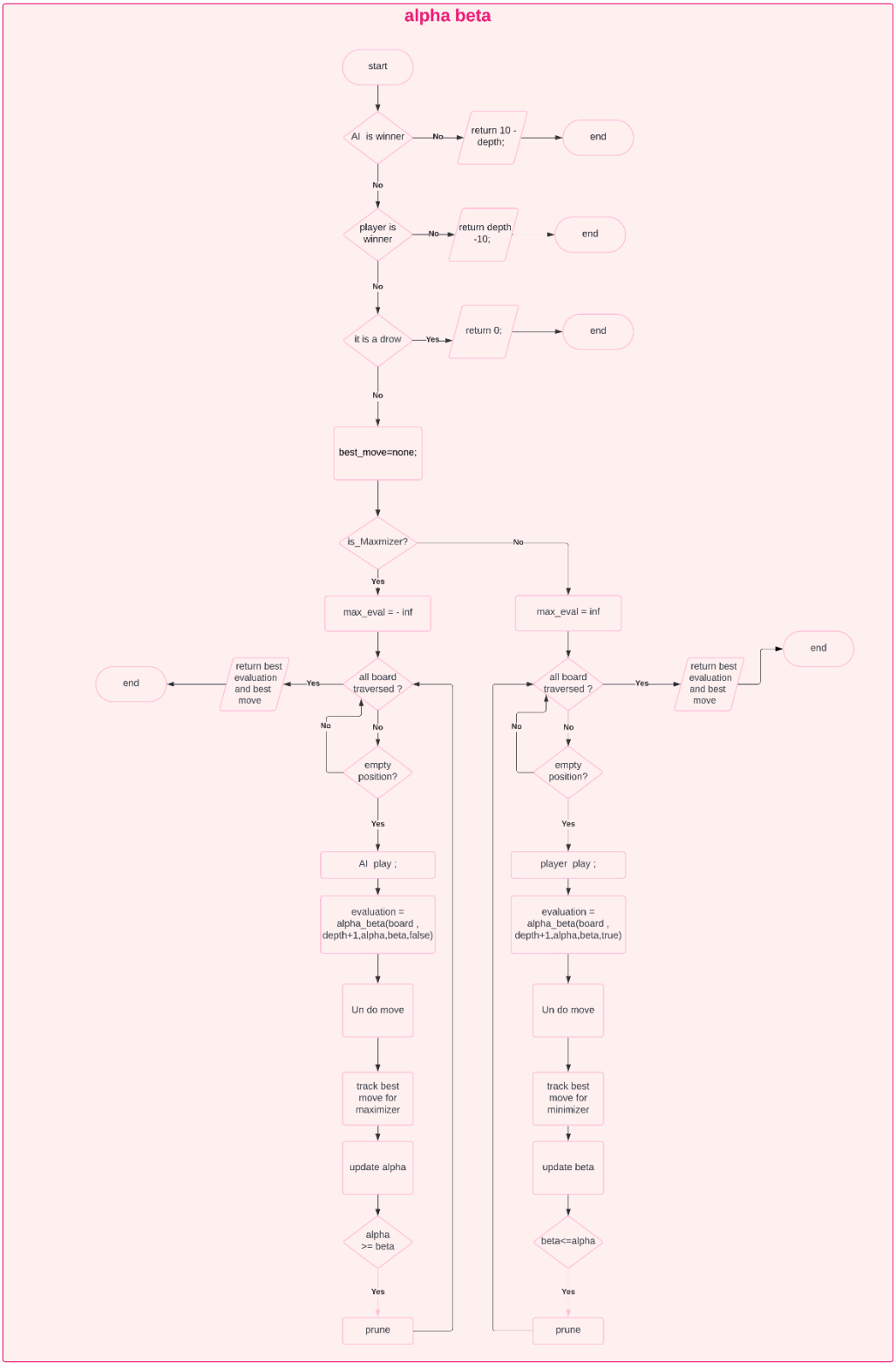


# Applied Algorithms Overview:

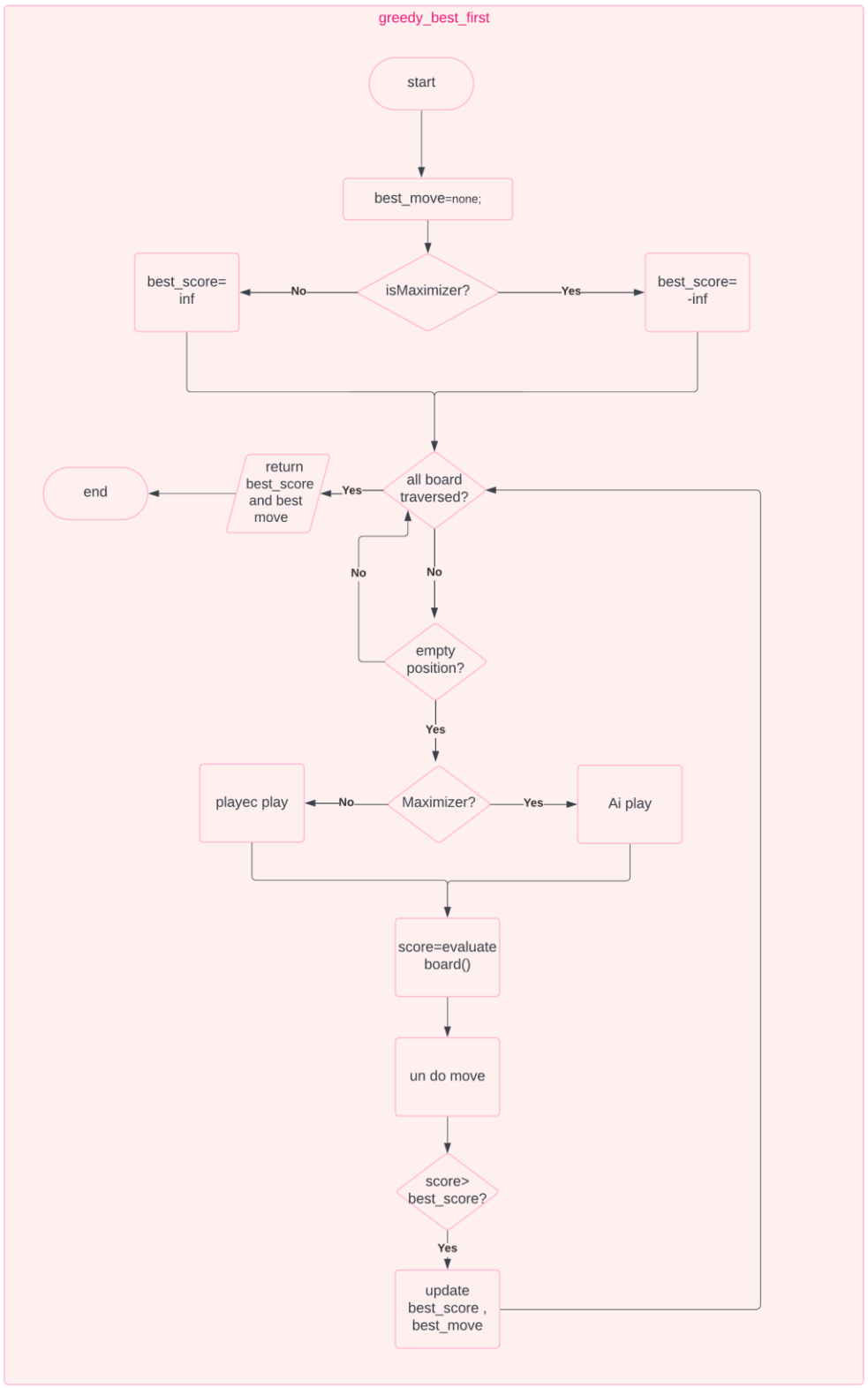
## *minimax:*



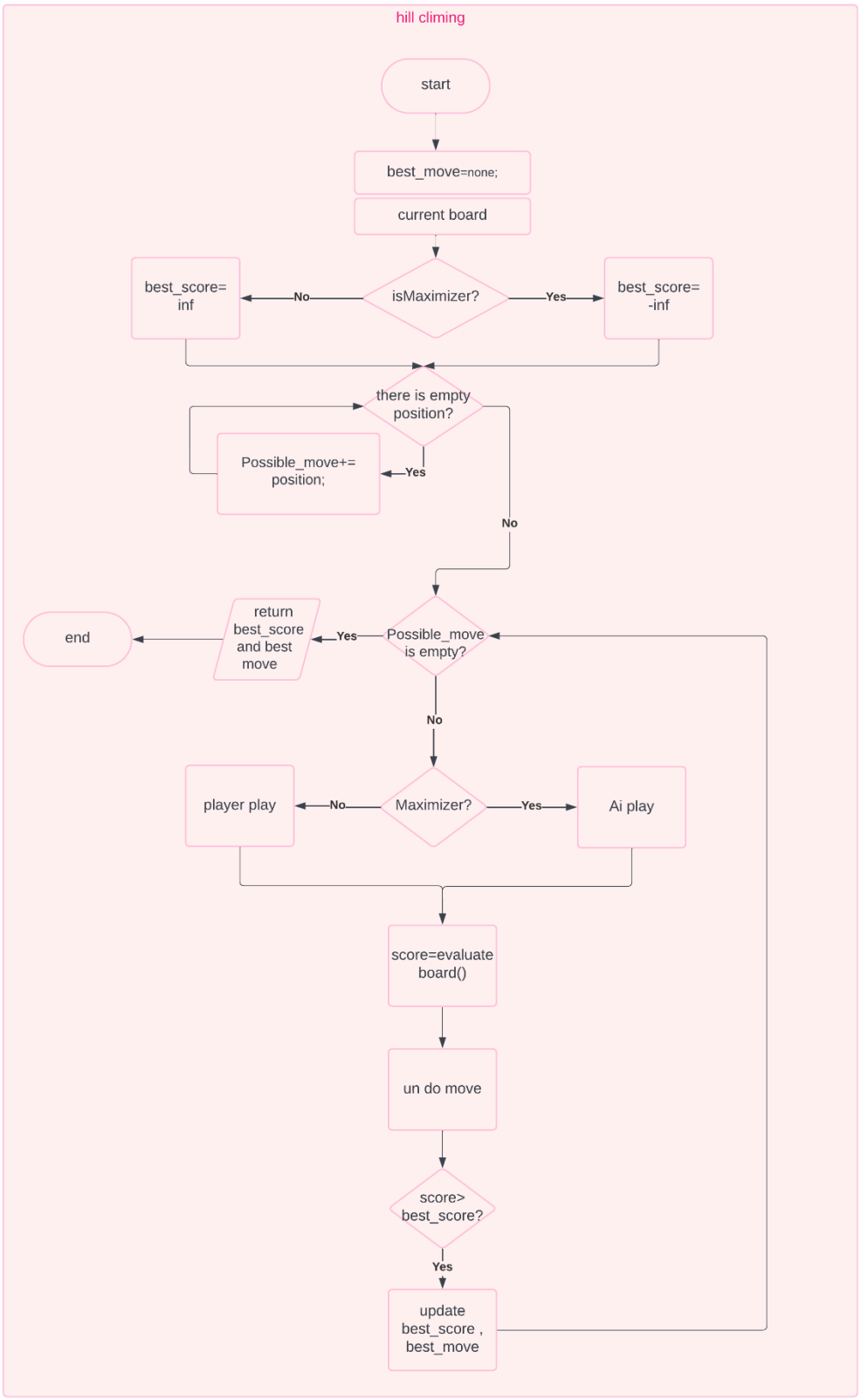
# *Alpha-Beta:*


alpha beta
start
AI is winner
No
return 10 - depth;
end
No
player is winner
No
return depth -10;
end
No
it is a drow
Yes
return 0;
end
No
best_move=none;
is_Maxmizer?
No
Yes
max_eval = - inf
max_eval = inf
all board traversed ?
Yes
return best evaluation and best move
end
No
No
empty position?
Yes
AI play ;
evaluation = alpha_beta(board , depth+1,alpha,beta,false)
Un do move
track best move for maximizer
update alpha
alpha >= beta
Yes
prune
all board traversed ?
Yes
return best evaluation and best move
end
No
No
empty position?
Yes
player play ;
evaluation = alpha_beta(board , depth+1,alpha,beta,true)
Un do move
track best move for minimizer
update beta
beta<=alpha
Yes
prune

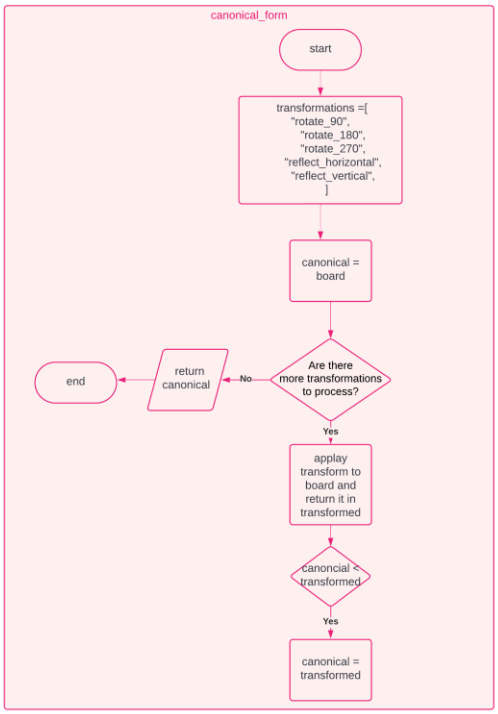
# *Minimax-first-Heuristic:*

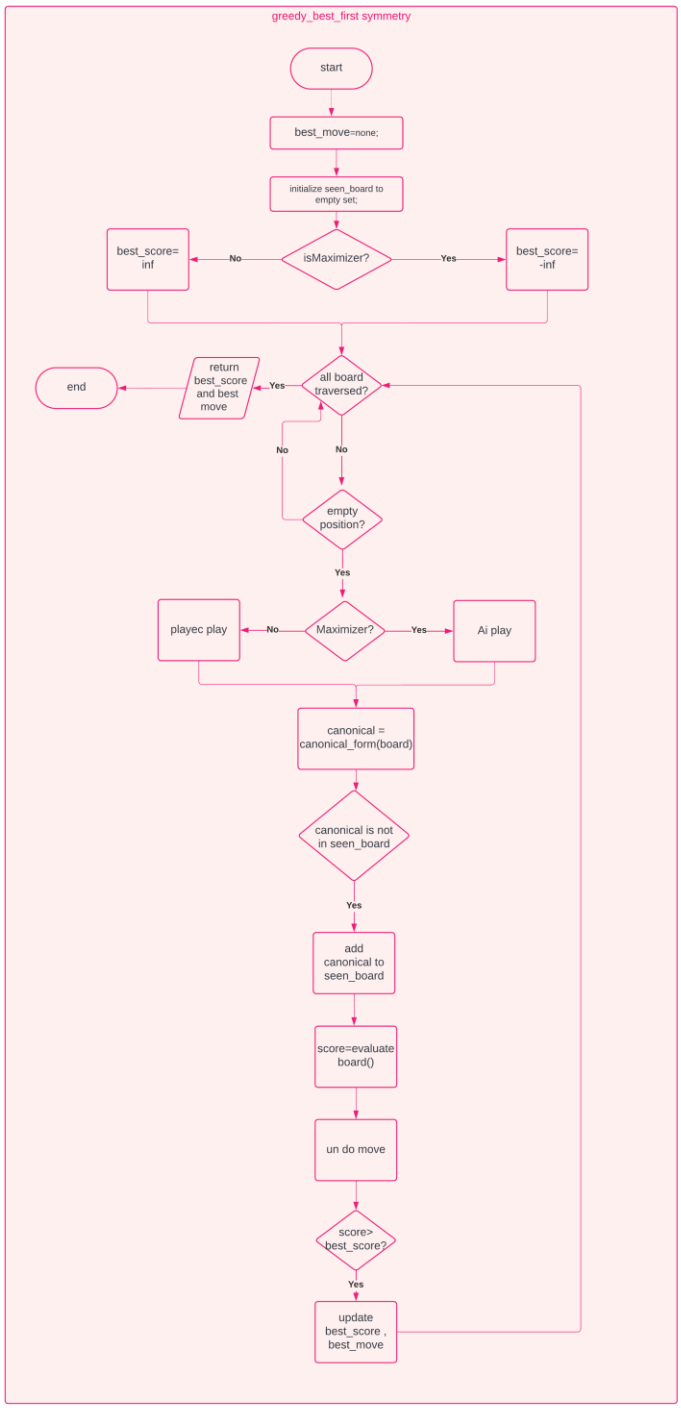


# *Minimax-Second-Heuristic:*

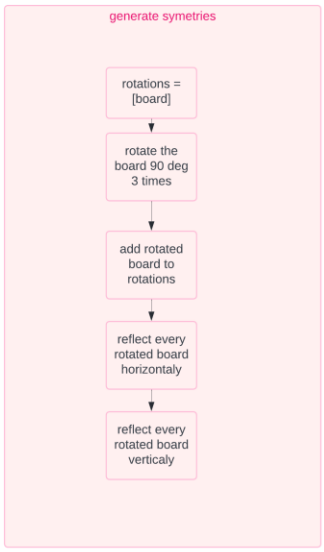


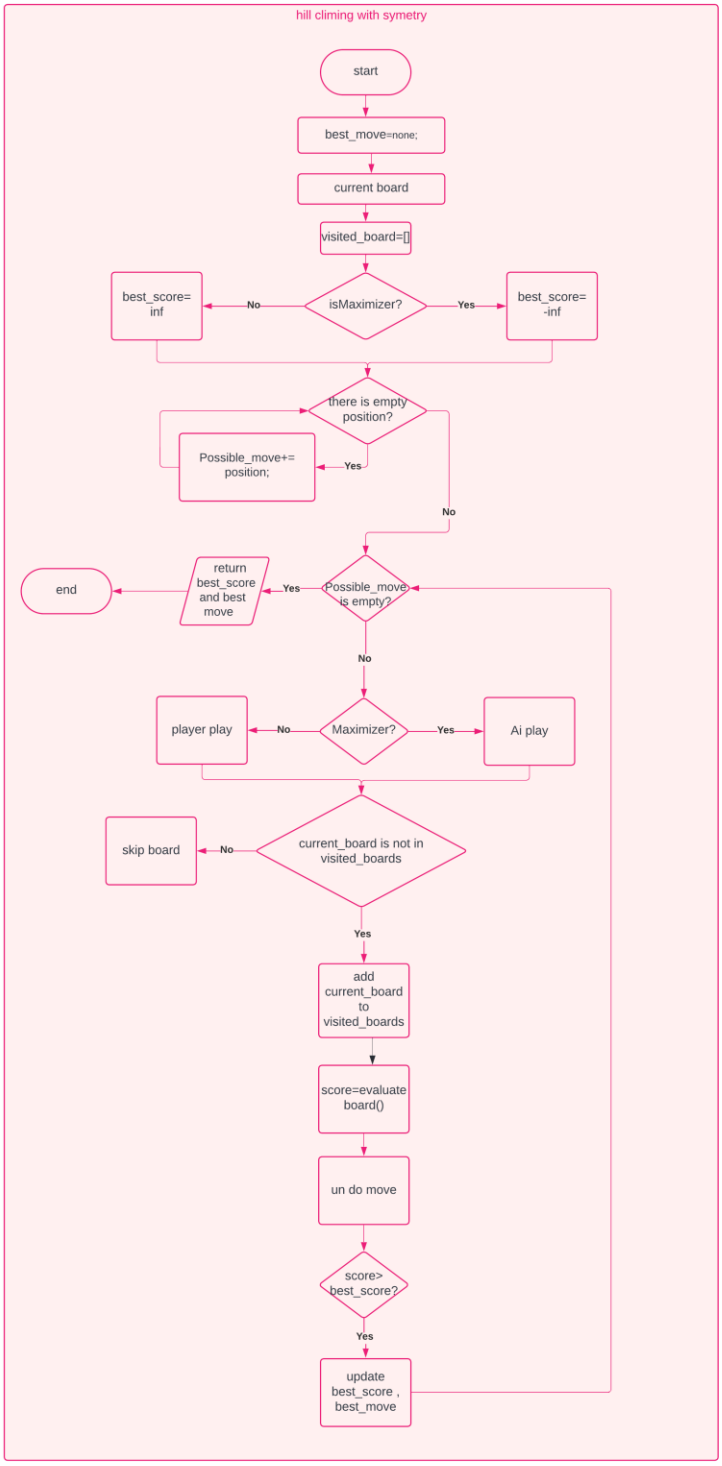
# Heuristic 1- symmetry:



# Heuristic2-symmetry:



## Minimax Algorithm:

The minimax function implements the Minimax algorithm to determine the optimal move in a Tic-Tac-Toe game. Here's a summary:

**Purpose**

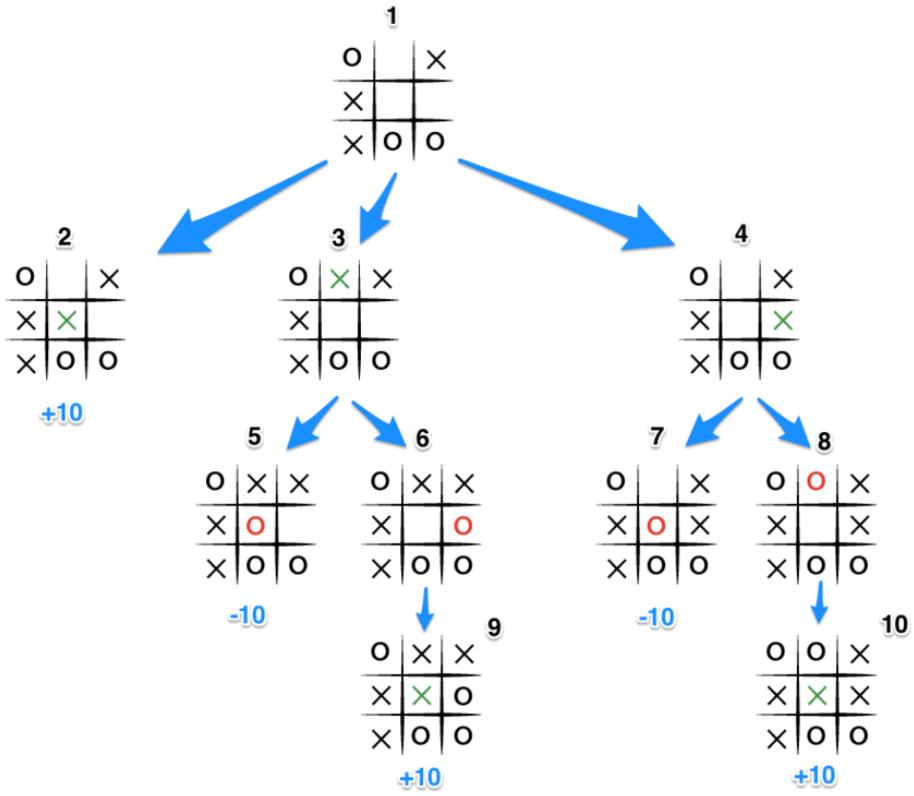
- **Goal:** To recursively simulate all possible game outcomes and decide the best move for the AI ("O") player while considering the opponent's moves ("X").

**Inputs**

1. board: A 3x3 list representing the Tic-Tac-Toe grid.
2. depth: The current recursion depth, used to prioritize faster victories or longer delays in losing.
3. is_maximizing: A boolean indicating if the current turn is for the maximizing player (AI) or the minimizing player (PLAYER).

**Key Features**

- **Evaluation Function:** Scores favor quicker victories or delayed defeats.
- **Complete Search:** Explores all possible moves and outcomes.
- **No Symmetry Reduction:** Does not account for symmetrical boards, leading to redundant calculations.



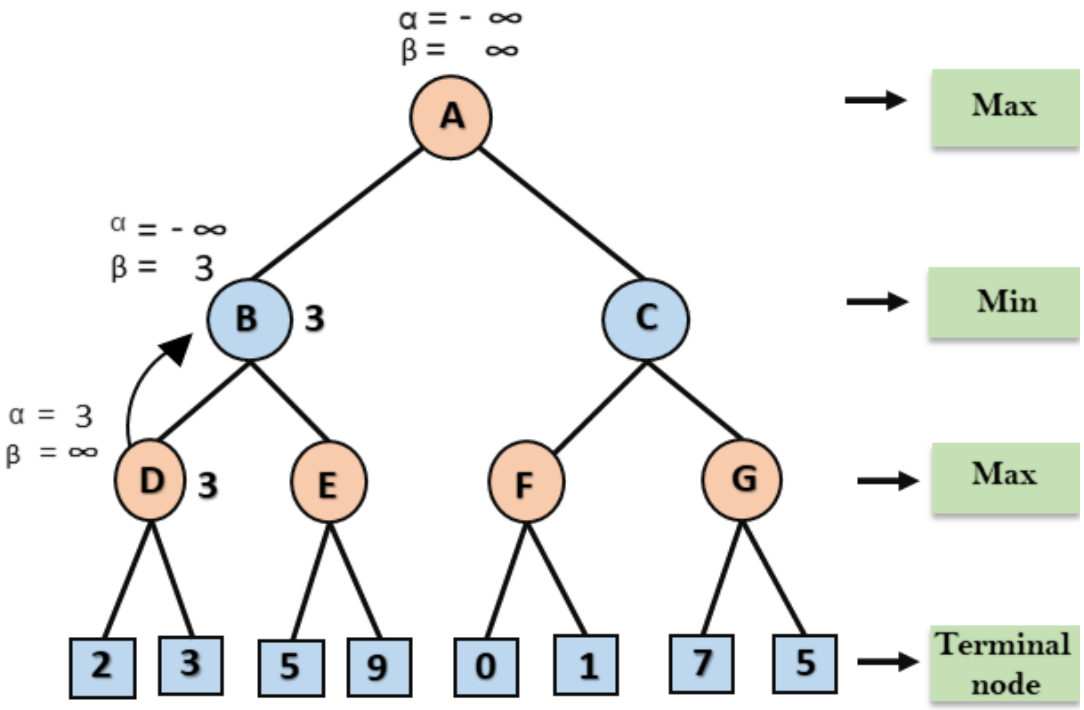
## Alpha-Beta Pruning in Minimax:

- used to eliminate branches in the game tree that don't need to be explored. This is achieved by keeping track of two values, alpha and beta, which represent the minimum score that the maximizing player is assured and the maximum score that the minimizing player is assured, respectively.

  **Alpha (α):** The best value that the maximizer currently can guarantee.

  **Beta (β):** The best value that the minimizer currently can guarantee.

  **Pruning:** If during the tree exploration it is found that a certain branch will not provide a better outcome than previously evaluated branches it is pruned.



## Heuristic Functions:

Minimax First heuristic function

***Greedy best first heuristic:***

- **Purpose**: This heuristic evaluates the current board state to determine the best possible move for a player (either maximizing for AI or minimizing for the opponent).

**greedy_best_first Function:**

- **Inputs**: The board state and the current player (AI or Player).
- **Logic**:
    - Iterates through all empty cells on the board.
    - Simulates placing the current player's piece on an empty cell.
    - Evaluates the resulting board state using evaluate_board.
    - Restores the board to its original state after evaluation.
    - Updates the best move based on the score:
        - Maximizing player (AI) selects the move with the highest score.
        - Minimizing player (opponent) selects the move with the lowest score.
- **Outputs**: Returns the best score and the corresponding move.

**Heuristic Strategy:**

- **Maximize AI's Chances**: Prioritizes moves that lead to immediate or potential future wins for the AI.
- **Block Opponent's Wins**: Deduces and prevents the Player from achieving winning conditions.
- **Greedy Approach**: Focuses on the immediate best move based on current board evaluation without exploring deeper game states.

## Minimax Second Heuristic Function

### *Hill Climbing heuristic:*

**Purpose**:

- The heuristic evaluates the board state to guide the AI's decisions in making the next move.
- It assigns scores to potential moves based on the likelihood of achieving a win or blocking the opponent.
- ensures a balanced approach to both offensive and defensive play, leveraging immediate opportunities while mitigating threats.

### Hill Climbing Strategy:

- **Initial Setup**:
  - Generates all possible moves by identifying empty spaces on the board.
- **Simulation**:
  - For each possible move, simulates the move by placing the AI's or Player's symbol temporarily.
- **Evaluation**:
  - The heuristic score of the resulting board state is calculated using evaluate_board.
- **Key Features**:
  - **Blocking Moves**: Identifies critical positions to block the opponent's potential winning opportunities.
  - **Maximizing Opportunities**: Prioritizes moves that improve AI's chances of winning.
  - **Efficiency**: Focuses on heuristic scoring without exploring deeper game tree levels, making it suitable for quick decision-making.

## *MINIMAX Symmetry*

The minimax_sym function is an optimized implementation of the minimax algorithm for games like Tic-Tac-Toe. It evaluates possible moves and selects the best one to maximize the AI's chances of winning or minimizing the player's chances, considering the following optimizations:

## Key Features and Steps:

1. **Symmetry Optimization**:
    - The board is transformed into its **canonical form**, the lexicographically smallest representation among all symmetries (rotations and reflections). This avoids redundant computations for symmetric board states.
2. **Caching**:
    - A cache (cached_results) is used to store evaluations of previously analyzed board states in their canonical forms, reducing redundant calculations.
3. **Recursive Minimax Logic**:
    - The function is recursive and alternates between maximizing (AI's turn) and minimizing (Player's turn).
    - Terminal states (win, loss, draw) are evaluated with predefined scores:
        - 10 - depth for an AI win (favoring faster wins),
        - depth - 10 for a Player win (penalizing longer losses),
        - 0 for a draw.
4. **Move Selection**:
    - Iterates over all empty cells on the board, makes a move, recursively evaluates its outcome, and then undoes the move (backtracking).
    - Keeps track of the move leading to the best evaluation.
5. **Return Value**:
    - Returns a tuple (evaluation, best_move):
        - evaluation: The numerical score of the board.
        - best_move: The coordinates (row, col) of the optimal move.

## Auxiliary Functions:

- **rotate_90, reflect_horizontal, reflect_vertical**: Transformations to generate board symmetries.
- **generate_symmetries**: Produces all possible symmetries of the board.
- **canonical_form**: Finds the canonical form for a board by comparing its symmetries.

## *Alpha-Beta Symmetry:*

The alpha_beta_sym function is an implementation of the Minimax algorithm with Alpha-Beta pruning and symmetry reduction. It is designed to determine the best possible move in a Tic-Tac-Toe game for either the AI ("O") or the player ("X") by evaluating all potential game states. Here's a summary:

## Key Features:

1. **Alpha-Beta Pruning**:
    - Optimizes the Minimax algorithm by pruning branches of the game tree that don't need evaluation, based on the current best-known values for the maximizing (alpha) and minimizing (beta) players.
2. **Symmetry Reduction**:
    - Leverages board symmetry to avoid redundant evaluations by caching results for all symmetric forms of a board state, significantly improving efficiency.
3. **Recursive Evaluation**:
    - Evaluates game states recursively for both maximizing (AI's turn) and minimizing (Player's turn) scenarios until a terminal state (win, loss, or draw) is reached.
4. **Caching**:
    - Stores evaluations of board states in a dictionary (cached_results) to avoid redundant computations for previously seen configurations.

**Function Workflow:**

1. **Terminal State Check**:
    - Checks if the current board is in a winning state for AI or Player, or if the game is a draw. Returns a score (10 - depth, depth - 10, or 0 respectively).
2. **Maximizing Player (AI)**:
    - Simulates AI moves by placing "O" in empty spots and recursively evaluates the resulting board states.
    - Keeps track of the maximum evaluation and the best move.
3. **Minimizing Player (Player)**:
    - Simulates Player moves by placing "X" in empty spots and recursively evaluates the resulting board states.
    - Keeps track of the minimum evaluation and the best move.
4. **Pruning**:
    - Stops exploring further moves when alpha >= beta.
5. **Caching and Symmetry**:
    - Stores the computed result for the current board state and all its symmetric forms in the cached_results dictionary.
6. **Return**:
    - Returns a tuple containing the evaluation score and the best move for the given board state.

## ***Heuritic1-symmetry:***

This script implements a strategy for making optimal moves in a 3x3 board game (like Tic-Tac-Toe) using heuristic evaluation and board transformations. Here's a summary of the key functions:

1. **evaluate_board(board)**:
    - Calculates a score for the given board configuration.
    - Considers rows, columns, and diagonals to determine potential winning, blocking, or neutral states.
    - Uses evaluate_line to score individual lines based on the number of AI, Player, and empty spots.
2. **evaluate_line(line)**:
    - Scores a single line (row, column, or diagonal) based on its composition:
        - +10: AI is close to winning (2 AI pieces, 1 empty).
        - -10: Player is close to winning (2 Player pieces, 1 empty).
        - +1: AI has 1 piece, 2 empty spaces.
        - -1: Player has 1 piece, 2 empty spaces.
3. **transform_board(board, transform)**:
    - Applies transformations (rotations and reflections) to the board.
    - Enables exploration of symmetric equivalents of the board.
4. **canonical_form(board)**:
    - Determines the "canonical" version of the board by comparing all symmetrical transformations.
    - Ensures that equivalent board states are treated identically during evaluation.
5. **greedy_best_first(board, maximizing_player)**:
    - Implements a heuristic-based strategy to determine the best move.
    - Considers all possible moves and evaluates their outcomes using evaluate_board.
    - Avoids revisiting symmetric board states by using the canonical form and a seen_boards set.
    - Returns the best score and the move associated with it for the maximizing player (AI) or minimizing player (Player).

## ***Heuristic2-symmetry:***

This code is part of a Tic-Tac-Toe AI system that evaluates board states and determines the best move using heuristic evaluation, symmetries, and a hill-climbing strategy. Here's a breakdown:

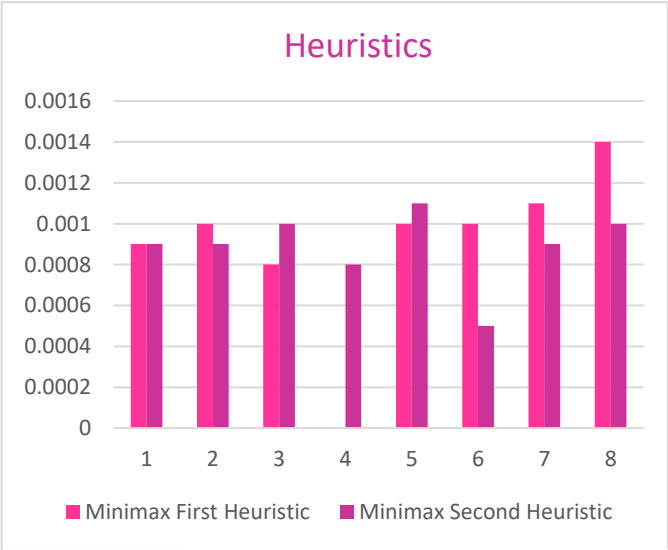
**Key Components:**

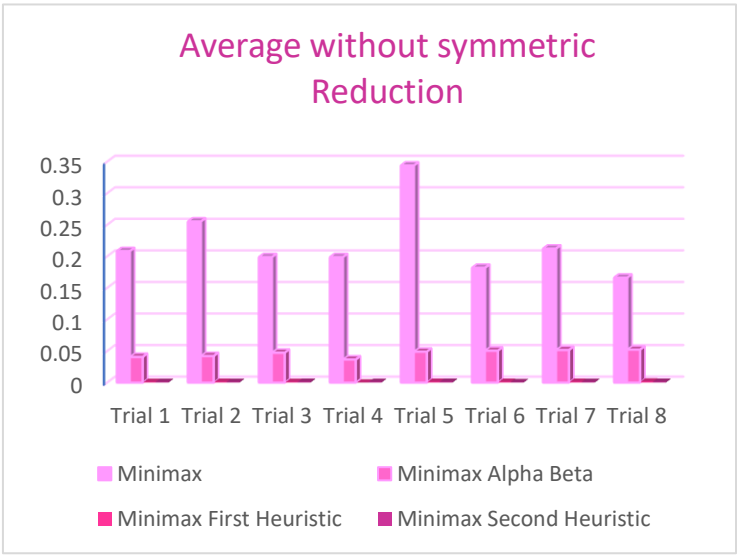
1. **Constants:**
    - EMPTY: Represents an empty cell on the board.
    - PLAYER ("X"): Human player's symbol.
    - AI ("O"): AI player's symbol.
2. **Functions:**
    - **evaluate_board(board)**:
        - Computes the board's heuristic score.
        - Positive values favor the AI; negative values favor the player.
        - Evaluates rows, columns, and diagonals using evaluate_line.
    - **evaluate_line(line, ai_symbol, player_symbol)**:
        - Assigns scores based on potential winning opportunities in a line:
            - +10/-10 for two AI/player symbols and one empty cell.
            - +1/-1 for one AI/player symbol and two empty cells.
    - **generate_symmetries(board)**:
        - Produces symmetrical transformations (rotations and reflections) of the board.
        - Helps identify equivalent board states to avoid redundant evaluations.
    - **is_unique_board(board, visited_boards)**:
        - Checks if a board is unique by comparing it with stored board symmetries.
    - **hill_climbing(board, maximizing_player)**:
        - Searches for the best move for either AI (maximizing) or Player (minimizing).
        - Explores all valid moves and evaluates the board.
        - Ensures unique boards are considered, avoiding redundant evaluations.
        - Returns the best score and move.

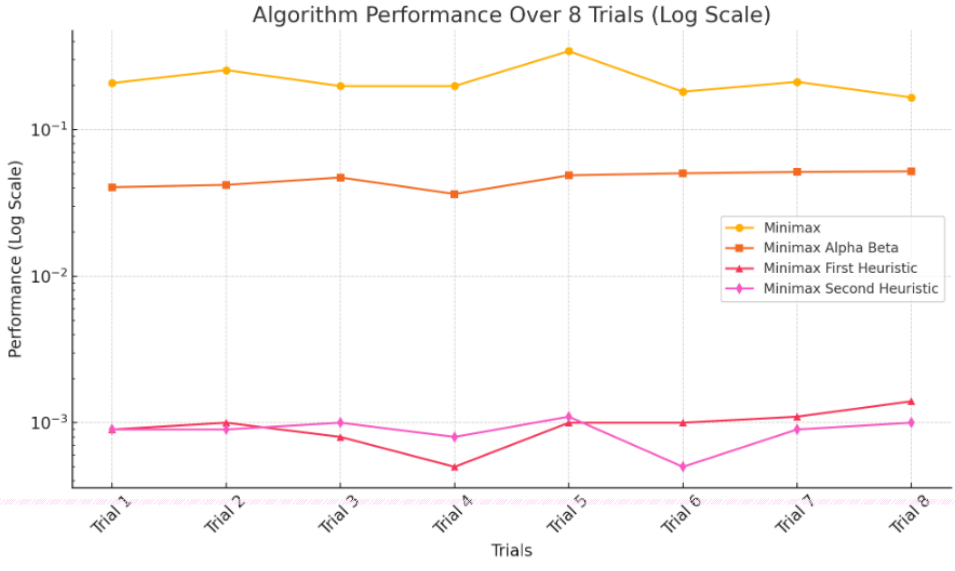
# Experiments & Results - Experiments, testing, and the results:

## 1)Without Symmetry Reduction:

| | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Trial 5 | Trial 6 | Trial 7 | Trial 8 | Average without symmetric Reduction |
|---|---|---|---|---|---|---|---|---|---|
| **Minimax** | 0.2081 | 0.2551 | 0.1986 | 0.1986 | 0.3438 | 0.1817 | 0.2122 | 0.1661 | 0.2205 s |
| **Minimax Alpha Beta** | 0.0405 | 0.042 | 0.0472 | 0.0364 | 0.0488 | 0.0504 | 0.0515 | 0.052 | 0.0461 s |
| **Minimax First Heuristic** | 0.0009 | 0.001 | 0.0008 | 0.0005 | 0.001 | 0.001 | 0.0011 | 0.0014 | 0.0009 s |
| **Minimax Second Heuristic** | 0.0009 | 0.0009 | 0.001 | 0.0008 | 0.0011 | 0.0005 | 0.0009 | 0.001 | 0.0008 s |

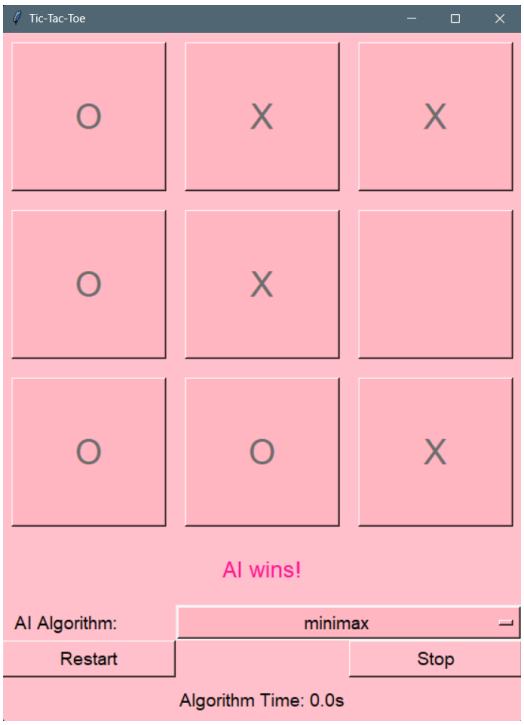


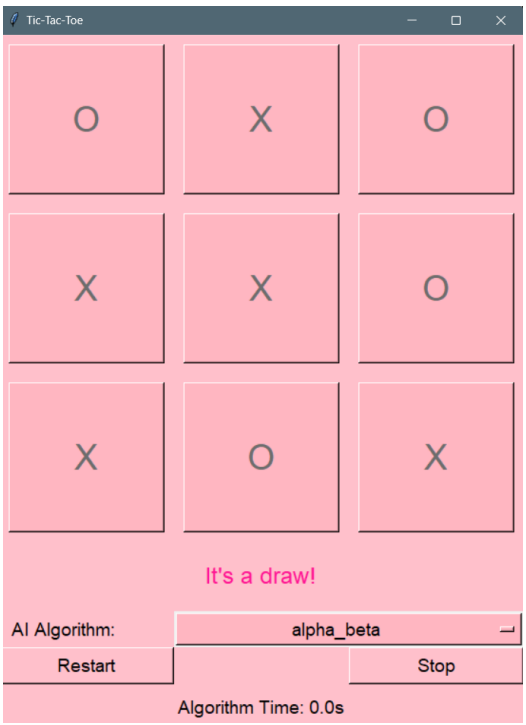



## Example output for :-

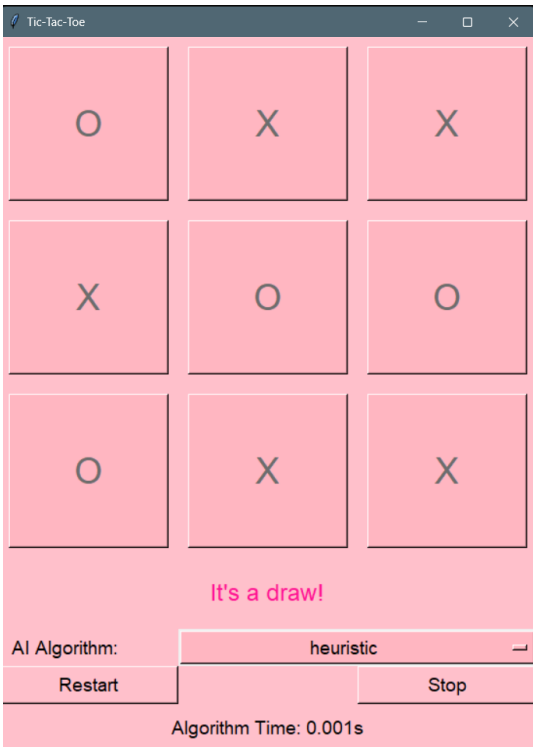
### minimax:



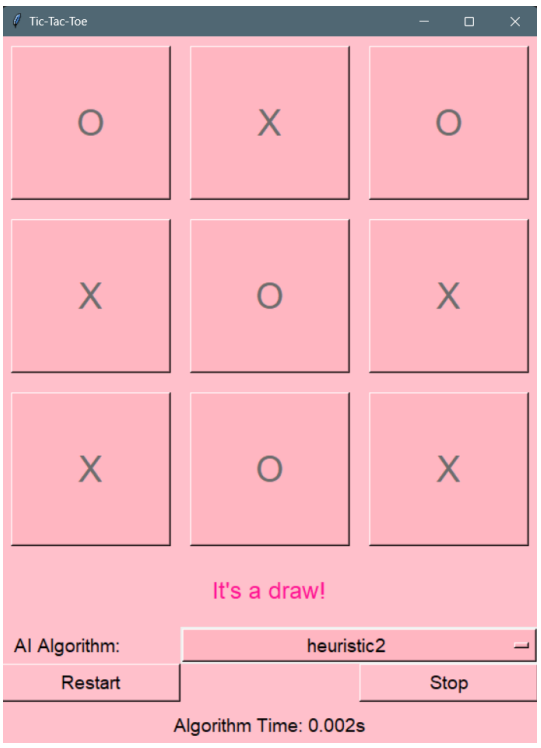

### Alpha-Beta pruning:




### Heuristic1:

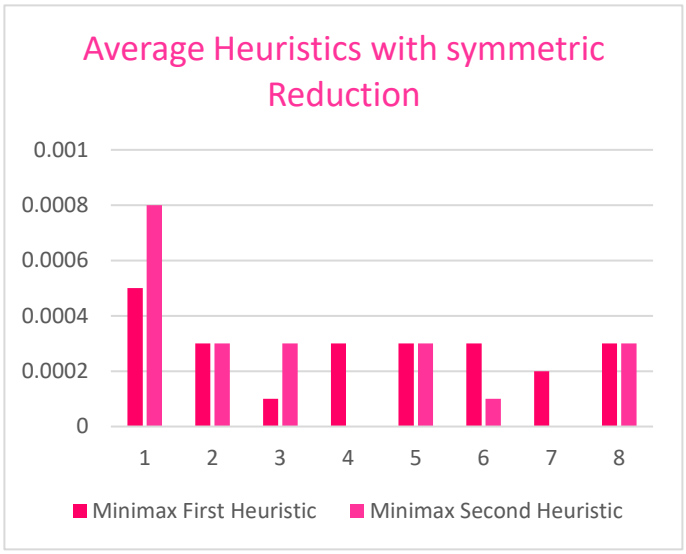


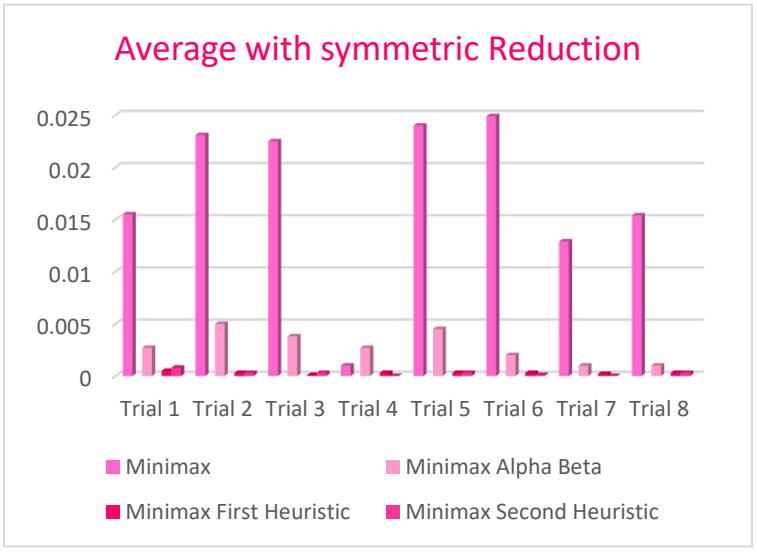

### Heuristic2:

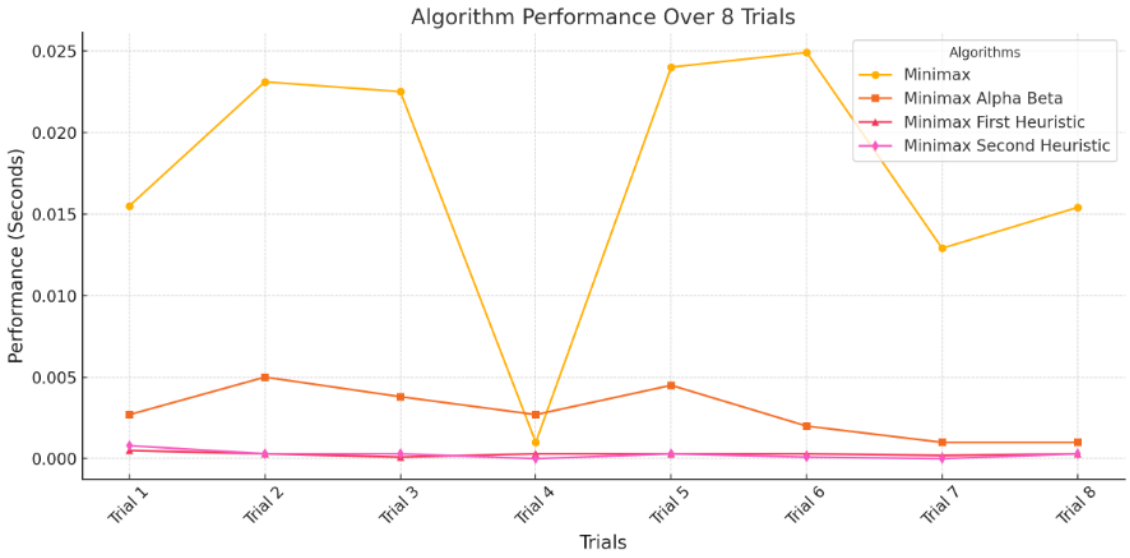


## With Symmetry Reduction:

| | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Trial 5 | Trial 6 | Trial 7 | Trial 8 | Average with symmetric Reduction |
|---|---|---|---|---|---|---|---|---|---|
| Minimax | 0.0155 | 0.0231 | 0.0225 | 0.0010 | 0.0240 | 0.0249 | 0.0129 | 0.0154 | 0.0174125 seconds |
| Minimax Alpha Beta | 0.0027 | 0.0050 | 0.0038 | 0.0027 | 0.0045 | 0.0020 | 0.0010 | 0.0010 | 0.0028375 seconds |
| Minimax First Heuristic | 0.0005 | 0.0003 | 0.0001 | 0.0003 | 0.0003 | 0.0003 | 0.0002 | 0.0003 | 0.000288 seconds |
| Minimax Second Heuristic | 0.0008 | 0.0003 | 0.0003 | 0.0000 | 0.0003 | 0.0001 | 0.0000 | 0.0003 | 0.000263 seconds |



## Conclusion

- Minimax vs. Minimax Alpha Beta: Minimax Alpha Beta Pruning performs better with both symmetric reduction and without it. The addition of alpha-beta pruning significantly cuts down on computation time by pruning unnecessary branches in the search tree.

- Heuristics vs. Minimax Algorithms: The heuristic-based approaches are much faster compared to Minimax and Minimax Alpha Beta. This is expected since heuristics are designed to evaluate board states more efficiently rather than exhaustively exploring all possibilities. Both heuristics show significant improvements with symmetric reduction.

- Effectiveness of Symmetric Reduction: Symmetric reduction generally improves performance for all algorithms, but the degree of improvement varies. It is especially effective for Minimax Alpha Beta and the heuristic-based algorithms. However, the overall speed-up with Minimax is relatively modest, suggesting that while symmetric reduction helps, the reduction in unique board states might be limited by the inherent structure of the Minimax algorithm.

# Analysis, Discussion, and Future Work

## -Analysis of the results, what are the insights:

| | Time Complexity | Space Complexity | Optimality | Performance |
|---|---|---|---|---|
| **Minimax** | $O(b^d)$, b is the branching factor. d is the depth of the recursion. | $O(d)$, d is the depth of the recursion. | The algorithm is optimal for fully explored game trees. | 0.2205 seconds |
| **Minimax Alpha Beta** | $O(b^{d/2})$, b is the branching factor. d is the depth of the recursion. | $O(d)$, d is the depth of the recursion. | Guaranteed optimal play for Tic-Tac-Toe, though could be improved by exploiting board symmetries and move ordering. | 0.0461 seconds |
| **Minimax First Heuristic** | $O(n^2)$, n is the board dimension | $O(1)$ | Suboptimal for strategic play; greedy approach may not lead to the best long-term outcome. | 0.0009 seconds |
| **Minimax Second Heuristic** | $O(n)$, n is the dimension of the board | $O(1)$ | Suboptimal due to the limitations of the heuristic and the hill climbing approach. | 0.0008 Seconds |

## -Advantages and Disadvantages:

- **Minimax**

  Advantages:

  - Guarantees optimal play
  - Simple to implement
  - Exhaustively explores all possible moves
  - Provides deterministic outcomes

  Disadvantages:

  - Computationally expensive for larger games
  - Requires significant memory
  - Assumes optimal play by the opponent: In real-world scenarios, players may not always act rationally or optimally, which Minimax does not account for.

- **Alpha-Beta Pruning in minimax**

  Advantages:

  - Reduces computational load by pruning unnecessary branches
  - Speeds up decision making
  - Makes deeper searches feasible
  - Retains the optimal play of the Minimax algorithm.

  Disadvantages:

  - More complex to implement than basic Minimax
  - Efficiency depends on move ordering
  - Requires additional memory
  - Alpha-beta pruning does not change the fundamental complexity of the game itself. It only improves the search process

## First Heuristic function

Advantages:

- Prioritizes critical moves (blocking and winning)
- Simple and intuitive
- Provides immediate feedback
- Effective for the small state space of Tic-Tac-Toe

Disadvantages:

- Narrow focus on blocking and winning, missing other strategic factors
- Short-term evaluation without long-term planning
- Assumes opponent's play is not considered
- Potential imbalance in scoring between blocking and winning
- Dependent on the accuracy of utility functions

## Second Heuristic function

Advantages:

- Evaluates strategic value based on line configurations
- Provides high reward for immediate wins
- Adapts to various board states with line weighting

Disadvantages:

- Single line focus rather than multiple line aggregation
- Assumes specific player symbols which may need adaptation

# - Future Modifications:

### Why does The Algorithm Behave This way?

| Minimax | Alpha-Beta pruning |
|---|---|
| The Minimax algorithm in this Tic-Tac-Toe implementation recursively explores all possible moves for both the AI and the PLAYER. It maximizes the AI's chances of winning and minimizes the PLAYER's advantage by evaluating each potential game state. The algorithm checks for terminal states (win, loss, or draw) and assigns scores based on the outcome, rewarding quick wins and penalizing deeper wins. It simulates all moves and chooses the best one, but without pruning, which can make it slow for larger games. The approach ensures optimal play for both players. | The Alpha-Beta Pruning algorithm is an optimized version of Minimax for Tic-Tac-Toe. It works by recursively exploring all possible moves and using two parameters, alpha and beta, to prune branches of the search tree that cannot affect the final decision. If the AI is maximizing, it seeks to maximize its score, while the PLAYER minimizes the AI's advantage. The algorithm stops evaluating a branch if it finds that it can't lead to a better outcome (alpha >= beta), making it more efficient than regular Minimax. It checks for terminal states (win, loss, or draw) and assigns scores accordingly, ensuring optimal play for both players. |
| **Greedy-best-first heuristic** | **Hill Climbing heuristic** |
| The Greedy Best-First algorithm evaluates each possible move in Tic-Tac-Toe by assigning a score based on the current board configuration. It prioritizes moves that appear to offer the best immediate outcome using a simple evaluation function, which rewards the AI for having two pieces in a row with an empty space and penalizes the PLAYER similarly. The algorithm then selects the move with the highest (for AI) or lowest (for PLAYER) score, without considering future moves or deeper game states. It chooses the best immediate move based on this heuristic, making it faster but less strategic compared to Minimax or Alpha-Beta Pruning. | The Hill Climbing algorithm for Tic-Tac-Toe uses a heuristic to evaluate board states based on the potential for winning moves. It assigns positive scores for moves that help the AI and negative scores for moves that favor the PLAYER. The algorithm evaluates each possible move, selects the one with the best immediate score, and doesn't consider future moves. It's a local search approach that may get stuck in suboptimal solutions if there are no better moves immediately available. |

### Enhanced Pruning Techniques:

- **Adaptive Pruning:** Investigating adaptive pruning strategies where pruning decisions are based on the game state or opponent's strategy could improve performance for larger game trees.

### Learning-Based Approaches:

- **Machine Learning Integration:** Implementing machine learning techniques to adapt and learn from game outcomes could potentially enhance decision-making strategies.