Course No: CSE 4110

Course Name: Artificial Intelligence Laboratory

Department of Computer Science and Engineering

Khulna University of Engineering & Technology

Khulna-9203

Project Name**: 16 Guti Game**

Submission Date: 4/9/2024

Submitted To-	Submitted By-
Md. Shahidul Salim **Lecturer** **Most. Kaniz Fatema Isha** **Lecturer** Department of Computer Science and Engineering Khulna University of Engineering & Technology	Chinmoy Modak Turjo Roll: 1907003 Hasibul Islam Roll: 1907006 Year: 4th Semester: 1st Department of Computer Science and Engineering Khulna University of Engineering & Technology

Table of Contents

1. Objectives 2

2. Introduction 2

3. Theory 4

4. Valid Move Generation 6

5. Minimax Algorithm 8

6. Alpha Beta Pruning 11

7. Genetic Algorithm 13

8. Mamdani Style Fuzzy Inference 14

9. Required Tools and Platforms 18

10. Game Flow 18

11. UI Mockup 19

12. Project Contribution 20

13. Discussion 20

14. Conclusion 20

15. References 21

Objectives:

- To learn about game-playing theory
- To gather knowledge about Pygame and its different functions
- To gather knowledge about the minimax algorithm
- To optimize the minimax algorithm using alpha-beta pruning
- To learn about various Genetic Algorithm techniques
- To get familiar with fuzzy logic and Mamdani-style fuzzy inference
- To develop an intelligent agent that can play competitively with human or rational being

Introduction:

Sholo Guti, also known as the 16 Guti game, is a traditional board game widely popular in South Asian countries like India, Bangladesh, and Nepal. The game is similar to Checkers and Alquerque, involving strategic movement of 16 pieces (Guti) for each player on a board to capture the opponent's pieces. The primary objective is to capture all the opponent's pieces or block them so they cannot make a move. The game features a square board with intersecting lines, forming a grid of points where the pieces move along the lines. Players move one piece at a time to an adjacent empty spot, and they can capture the opponent's piece by jumping over it, much like in Checkers. The game is known for its simplicity in rules but complexity in strategy, requiring players to think several moves ahead to secure a win. Incorporating artificial intelligence (AI) into Sholo Guti can enhance the game's complexity and provide a challenging experience for players. This report discusses how various AI techniques, including the Minimax algorithm with alpha-beta pruning, genetic algorithms for optimal strategy selection, and Mamdani-style fuzzy inference, can be applied to Sholo Guti to create an intelligent game-playing agent.

- **Minimax Algorithm:** This classical AI search technique explores all possible game states for a certain number of plies (moves) and chooses the move that leads to the most favourable outcome for the AI player.
- **Alpha-Beta Pruning:** This optimization technique significantly reduces the branching factor in the Minimax tree by eliminating unpromising moves early on. It defines alpha (best score for maximizing player) and beta (best score for minimizing player) values to prune branches that cannot improve upon already evaluated paths.
- **Genetic Algorithm:** This evolutionary approach can be used to train a population of AI models by simulating natural selection. Each model plays against others, and the "fittest" ones (those achieving better win rates) are selected for reproduction. This iterative process leads to progressively better AI performance.

- **Mamdani Style Fuzzy Inference:** Mamdani Style Fuzzy Inference deals with imprecise information. In Sholo Guti, it can analyze factors like the number of remaining pieces and moves required to win and then predict the likelihood of victory using fuzzy sets and rules.
- **Pygame:** Pygame is a set of Python modules designed explicitly for creating video games. It provides functionalities for computer graphics, sound, input handling, and more, all readily available in Python

Theory:

Game Features:

1. Initial Welcome Screen
2. 3 Levels of difficulty level by using three different algorithmic approaches
3. Total Moves Counter
4. Total Remaining pieces of human shown
5. Total Remaining Pieces of AI shown
6. Valid Moves shown by green highlighter
7. Can kill two pieces if needed
8. Based on how many pieces are remaining and the total moves taken into consideration, the game can inform your winning margin using fuzzy inference system
9. Portable exe file created so that the game can run on any x64-based system
10. Splash Screen Added

Board Generation:

Typically, the board is represented as a 2D array or a graph structure, where each node corresponds to an intersection on the physical board. Each node maintains information about its neighbors, allowing for quick determination of valid moves. The initialization process involves setting up this data structure and placing the initial 16 pieces for each player in their starting positions.

Below are the starting and ending co-ordinate demo points.

#point_list1 contains the starting of a line

point_list1 = [(0,0),(1,1),(2,0),(3,0),(4,0),(5,0),(6,0),(7,1),(8,0),(0,2),(2,4),(2,1),(2,3)]

#ith list of point_list2 contains the ending of a line starting from ith point of point_list1

point_list2=[[(0,4),(4,4)],[(1,3)],[(2,4),(6,4),(6,0)],[(3,4)],[(0,4),(4,4),(8,4)],[(5,4)],[(2,4),(6,4)],[(7,3)],[(4,4),(8,4)],[(8,2)],[(6,4)],[(6,1)],[(6,3)]]

#blank_list contains the coordinates of the grid which don't contain any piece

blank_list = [(0,1),(0,3),(1,0),(1,4),(7,0),(7,4),(8,1),(8,3)]

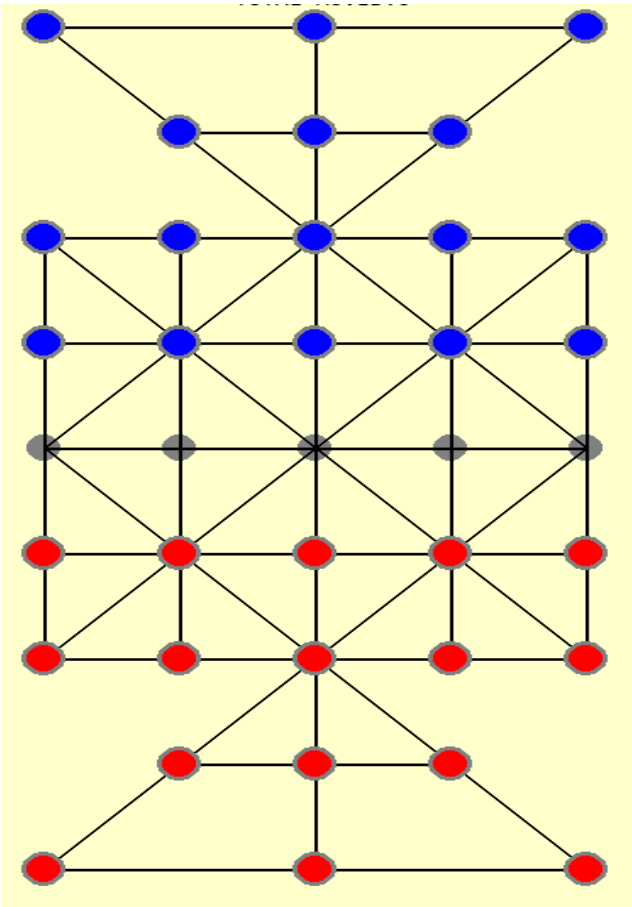


Fig-1: Sample board

First, we have taken a 9*5 2D grid. Then we created a virtual graph defining the starting and ending point of the line. Then, the valid moves from one node to another are determined explicitly. Dictionary structure is used to store the valid moves for a given point of node. We take the middle position of the grid and create nodes.

Valid Moves Dictionary is given below:

self.valids = {(0,0): [(0,2), (1,1)],

(0,2): [(0,0), (0,4), (1,2)],

(0,4): [(0,2), (1,3)],

……

]}

Then lines are drawn.

Here is the demo diagram of how the board is generated:

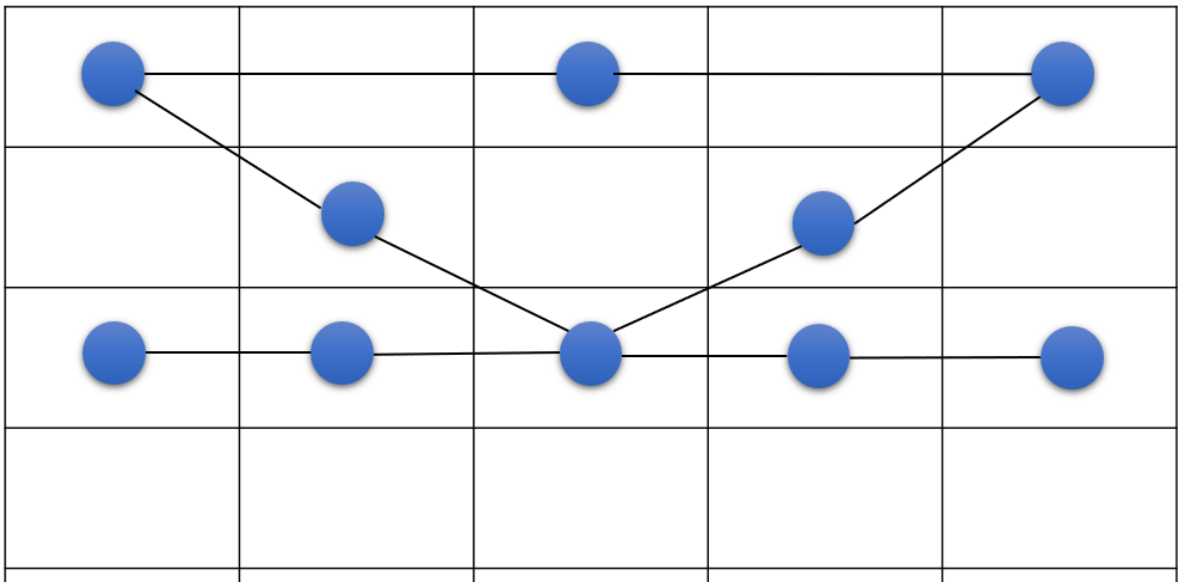


Fig-2: Position of the pieces

Valid Move Genration:

Valid Move generation is quite difficult in the context of the sholo guti game. Here the player or AI if they captured a piece then the captured piece has to be removed from the board. Next, if a player or an AI has a chance to capture any more pieces they can try to capture it or leave it according to their game strategy. Here is the pseudocode for that function which calculates the valid moves:

1. Initialization (Setting Up):

- Create an empty list called valid to store valid moves (both single-step and capture moves).
- Create an empty dictionary called skipped to keep track of "skipped" squares during capture moves (key: capture destination, value: square jumped over).
- Initialize a variable catch to 0, which will count the number of captured opponent pieces.

2. Looping Through Possible Moves:

- The function iterates through each move in the provided moves list.
    - move is likely a tuple containing the row and column coordinates of a potential destination square.
- Destructure the move tuple to separate row and column values into individual variables named row and col.

3. Checking for Empty Square (Single-Step Move):

- Access the current piece at the destination square (current) using the board (self.board[row][col]).
- If current is 0 (empty square), it's a valid single-step move:
    - Add the move (row, col) to the valids list.
    - In the skipped dictionary, store the move itself as the key with a value of 0 (indicating no piece skipped for single-step).

4. Checking for Opponent's Piece and Valid Capture:

- If current is not empty (-1) and has a different color than the current player (current.color != color):
    - This indicates a potential capture move.
    - Calculate the jump direction based on the difference between the current piece's position and the potential capture destination. Store this direction in a list named direction (e.g., [-1, 1] for diagonal jump up-right).

5. Checking Jump Destination and Capture Validity:

- Check if the jump destination is within the board boundaries:
    - Verify if row + direction[0] and col + direction[1] are within the valid range (0 to COLS-1 for columns and 0 to ROWS-1 for rows).
- If the jump destination is valid, access the piece at that square (next_square = self.board[row + direction[0]][col + direction[1]]).
- If next_square is empty (0), it's a valid capture move:
    - Increment catch by 1 to indicate a captured piece.
    - Create a variable called capture_move containing the capture destination's coordinates (row + direction[0], col + direction[1]).
    - Add the capture_move to the valid list to mark it as a valid capture.
    - In the skipped dictionary, store the capture_move as the key and the original move (where the jump originated) as the value. This tracks the square that was jumped over during the capture.

6. Returning the Results:

- The function returns a tuple containing three elements:
    - valid: The list of valid moves (including both single-step and capture moves).

- skipped: The dictionary containing information about jumped squares during capture moves.
- catch: The count of captured opponent pieces during the move exploration.

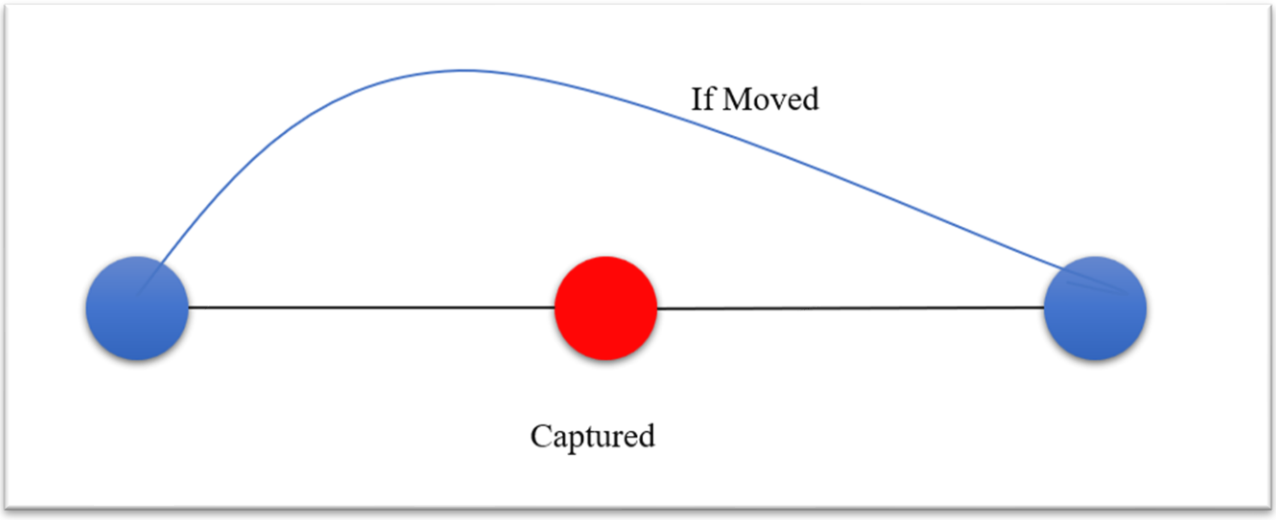


Fig-3: Capturing move

We check 2 steps forward and if a player can capture a piece and move to a new position then again on the new position we will calculate again the possible move for that player. If there is a chance to capture a new piece then consideration is to send it back to the desired player.

Minimax Algorithm:

Evaluation Function is defined as:

```
def evaluate(self):
    return self.blue_left - self.red_left
```

The minimax algorithm is a decision-making algorithm used in two-player games to determine the optimal move. It works by recursively evaluating all possible moves and their consequences, assuming that both players will play optimally. The algorithm constructs a game tree, where each node represents a game state, and edges represent possible moves.

In our Sholo Guti implementation, the minimax algorithm operates as follows:

1. Starting from the current game state, it generates all possible moves for the AI player (maximizing player).

2. For each of these moves, it then generates all possible responses by the human player (minimizing player).
3. This process continues, alternating between maximizing and minimizing players, until a terminal state is reached (game over) or a predefined depth limit is hit.
4. At terminal states or the maximum depth, an evaluation function assesses the desirability of the position for the AI player.
5. These values are then propagated back up the tree. The maximizing player selects the move with the highest value, while the minimizing player selects the move with the lowest value.
6. The process continues until the original game state is reached, at which point the AI selects the move that leads to the highest evaluated position.

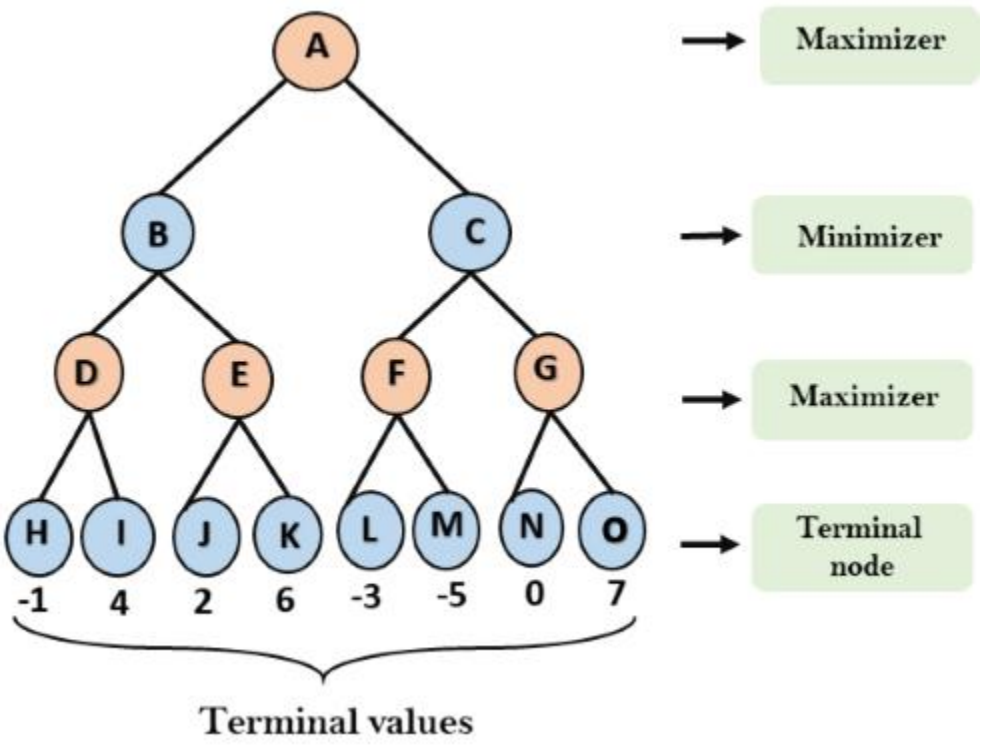


Fig-4: Minimax algorithm

The evaluation function is a critical component of the minimax algorithm. In Sholo Guti, this function might consider factors such as the number of pieces each player has, their positions on the board, and potential for future captures.

Minimax.py(pseudocode)

```
Import deepcopy and pygame
Define color constants RED and BLUE

Function minimax(position, depth, max_player, game):
    If depth is 0 or there is a winner in position:
        Return position's evaluation and position

    If max_player is True:
        Initialize maxEval to negative infinity
        Initialize best_move to None
        For each move in get_all_moves(position, BLUE, game):
            Recursively call minimax with reduced depth, False for max_player
            Update maxEval to the maximum of maxEval and the evaluation
            If maxEval is equal to evaluation, set best_move to move
        Return maxEval and best_move
Else (min_player):
        Initialize minEval to positive infinity
        Initialize best_move to None
        For each move in get_all_moves(position, RED, game):
            Recursively call minimax with reduced depth, True for max_player
            Update minEval to the minimum of minEval and the evaluation
            If minEval is equal to evaluation, set best_move to move
        Return minEval and best_move

Function simulate_move(piece, move, board, game, skipped):
    Move piece to new position on board
    If skipped position is not empty:
        Get piece at skipped position
        Remove skipped piece from board
    Return updated board

Function get_all_moves(board, color, game):
    Initialize moves as an empty list
    For each piece of the given color on the boar
        Get valid moves for piece
        For each move in valid_moves:
            Create a temporary copy of the board
            Get the piece from the temporary board
            Simulate the move on the temporary boa
            Append the new board state to moves
    Return moves
```

Alpha Beta Pruning:

The algorithm maintains two values, alpha, and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of, respectively. As the algorithm progresses:

1. If a node is found that causes beta to become less than or equal to alpha, the remainder of that node's siblings can be pruned.
2. For a maximizing player, alpha is updated to be the maximum of its current value and the value of the child node.
3. For a minimizing player, beta is updated to be the minimum of its current value and the value of the child node.

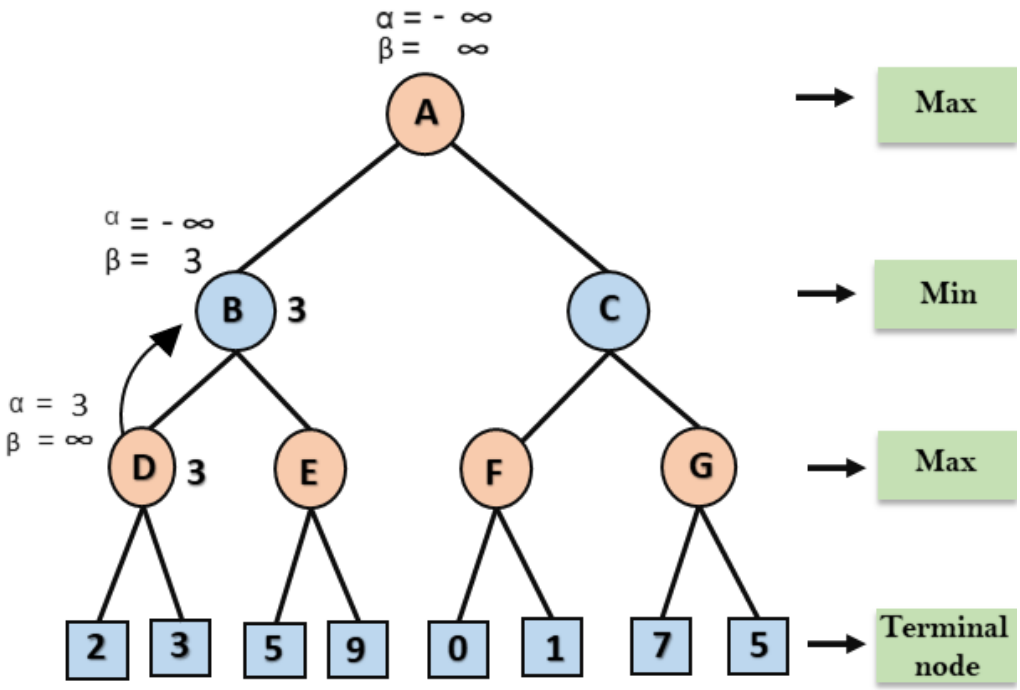


Fig-5: Alpha-beta pruning algorithm

In our Sholo Guti implementation, alpha-beta pruning significantly improves the efficiency of the AI, allowing it to consider more future moves and make better decisions within reasonable time constraints.

Alphabetapruning.py(pseudocode)

```
# Define constants for colors
RED = (255, 0, 0)
BLUE = (0, 0, 255)

# Alpha-Beta pruning function
FUNCTION alphabeta_pruning(position, depth
max_player, game, alpha, beta):
   IF depth == 0 OR position.winner() != None:
      RETURN position.evaluate(), position

   IF max_player:
      maxEval = -infinity
      best_move = None
      FOR each move IN get_all_moves(positio
BLUE, game):
         evaluation = alphabeta_pruning(move,
depth - 1, False, game, alpha, beta)[0]
         maxEval = MAX(maxEval, evaluation)
         IF maxEval == evaluation:
            best_move = move
         alpha = MAX(alpha, evaluation)
         IF beta <= alpha:
            BREAK
      RETURN maxEval, best_move

   ELSE:
      minEval = infinity
      best_move = None
      FOR each move IN get_all_moves(positio
RED, game):
         evaluation = alphabeta_pruning(move,
depth - 1, True, game, alpha, beta)[0]
         minEval = MIN(minEval, evaluation)
         IF minEval == evaluation:
            best_move = move
         beta = MIN(beta, evaluation)
         IF beta <= alpha:
            BREAK
      RETURN minEval, best_move
```

```
 # Simulate a move on the board
FUNCTION simulate_move(piece, move, board
game, skipped):
   board.move(piece, move[0], move[1])
   IF skipped[(move[0], move[1])] != 0:
      (r, c) = skipped[move[0], move[1]]
      piece = board.get_piece(r, c)
      board.remove(piece)
   RETURN board

# Get all possible moves for a given color on th
board
FUNCTION get_all_moves(board, color, game
   moves = []
   FOR each piece IN board.get_all_pieces(colo
      [valid_moves, skipped, catch] =
board.get_valid_moves(piece)
      FOR each move IN valid_moves:
         temp_board = deepcopy(board)
         temp_piece =
temp_board.get_piece(piece.row, piece.col)
         new_board = simulate_move(temp_piec
move, temp_board, game, skipped)
         moves.append(new_board)
   RETURN moves
```

Genetic Algorithm:

Genetic algorithms (GAs) are optimization techniques inspired by natural selection and genetics. In the context of Sholo Guti, GAs can be used to evolve optimal strategies over successive generations. By defining a fitness function that evaluates the performance of each strategy, we can apply selection, crossover, and mutation operations to create new generations of strategies. This approach helps in finding robust strategies that can adapt to different playing styles and scenarios.

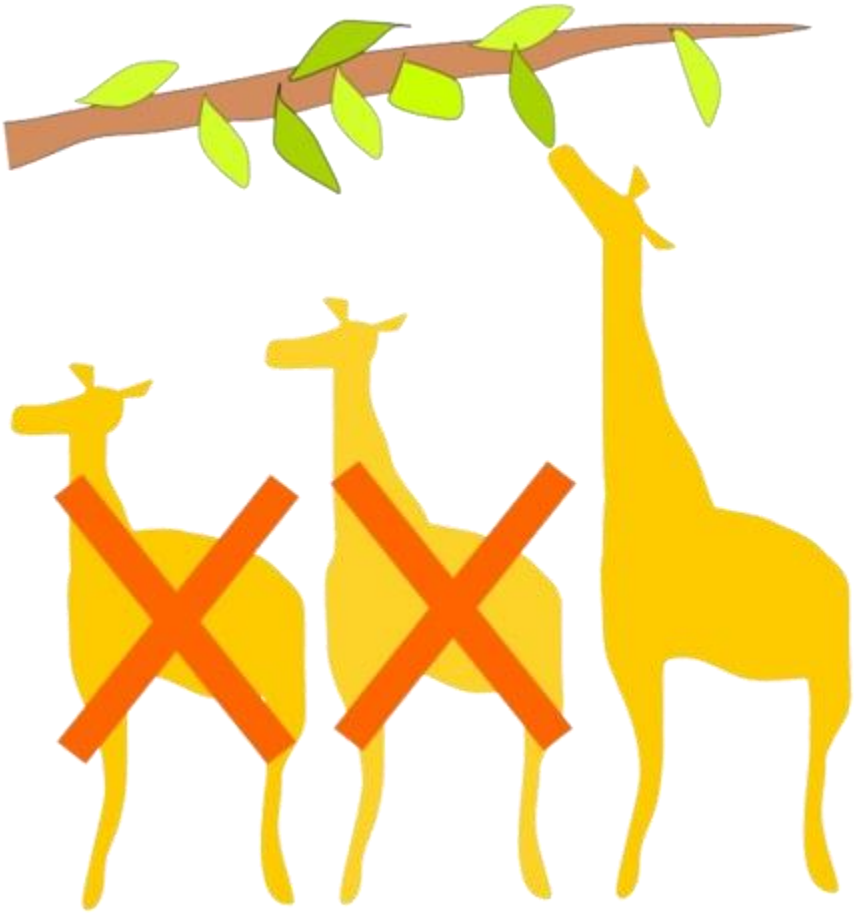


Fig-6: Selection process in Genetic algorithm

In our game we applied selection technique based on evaluation function. We generate for a given move all possible board moves then checks fitness value by using evaluation function and select the best possible move for next turn.

Genetic Algo.py(pseudocode**)**

```
# Define constants for colors
RED = (255, 0, 0)
BLUE = (0, 0, 255)

# Function to select the best move from a population
FUNCTION selection(population):
    maxEval = -infinity
    best_move = None
    FOR each position IN population:
        evaluation = position.evaluate()
        IF evaluation > maxEval:
            maxEval = evaluation
            best_move = position
    RETURN maxEval, best_move

# Genetic algorithm to find the best move for a given position
FUNCTION genetic_algorithm(position, generation, game):
    IF position.winner() != None:
        RETURN position.evaluate(), position

    population = get_all_moves(position, BLUE, game)
    value, best_move = selection(population)
    RETURN value, best_move
```

```
# Simulate a move on the board
FUNCTION simulate_move(piece, move, boar
game, skipped):
    board.move(piece, move[0], move[1])
    IF skipped[(move[0], move[1])] != 0:
        (r, c) = skipped[move[0], move[1]]
        piece = board.get_piece(r, c)
        board.remove(piece)
    RETURN board

# Get all possible moves for a given color on th
board
FUNCTION get_all_moves(board, color, game
    moves = []
    FOR each piece IN board.get_all_pieces(colo
        [valid_moves, skipped, catch] =
board.get_valid_moves(piece)
        FOR each move IN valid_moves:
            temp_board = deepcopy(board)
            temp_piece =
temp_board.get_piece(piece.row, piece.col)
            new_board = simulate_move(temp_piec
move, temp_board, game, skipped)
            moves.append(new_board)
    RETURN moves
```

Mamdani Style Fuzzy Inference:

In our Sholo Guti implementation, the Mamdani fuzzy inference system provides a nuanced evaluation of the game state, considering multiple factors to predict the likelihood of winning. This information can be used to inform the AI's decision-making process, complementing the minimax algorithm and genetic algorithm approaches.

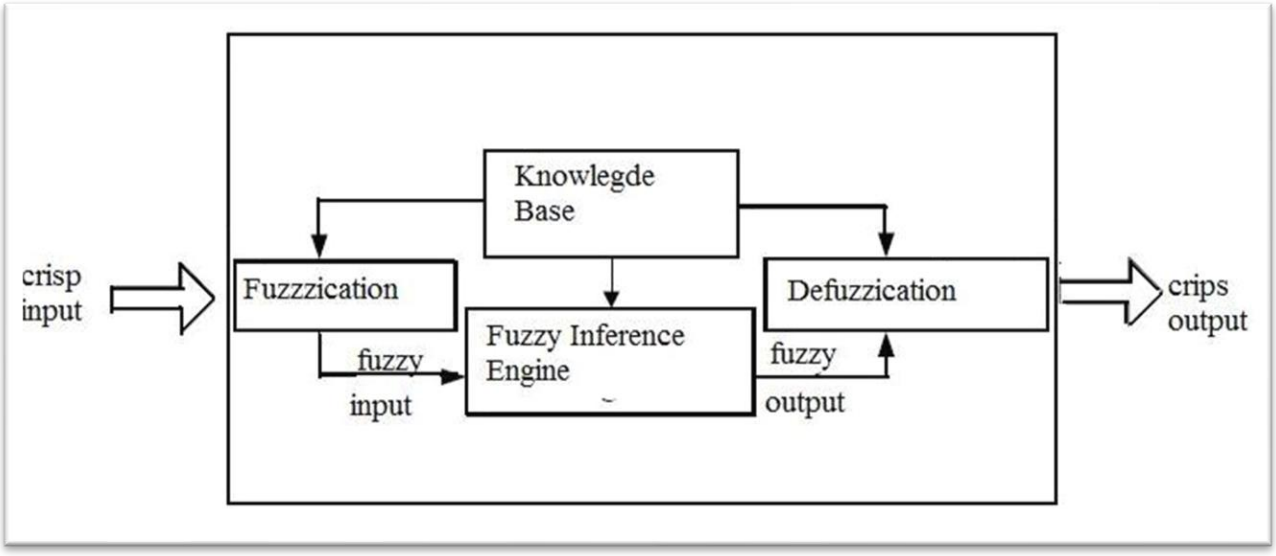


Fig-7: Mamdani Style Fuzzy Inference System

The Mamdani fuzzy inference system consists of several steps:

1. Fuzzification: Convert crisp input values into fuzzy values using membership functions. In our case, we fuzzify inputs such as "number of moves needed to win" and "number of pieces left."
2. Rule Evaluation: Apply fuzzy rules to the fuzzified inputs. These rules are typically in the form of "IF-THEN" statements. For example: "IF total moves_after_win is LOW AND pieces_left is HIGH THEN winning_probability is HIGH"
3. Aggregation: Combine the outputs of all rules into a single fuzzy set.
4. Defuzzification: Convert the aggregated fuzzy set back into a crisp output value, typically using methods like the centroid method.

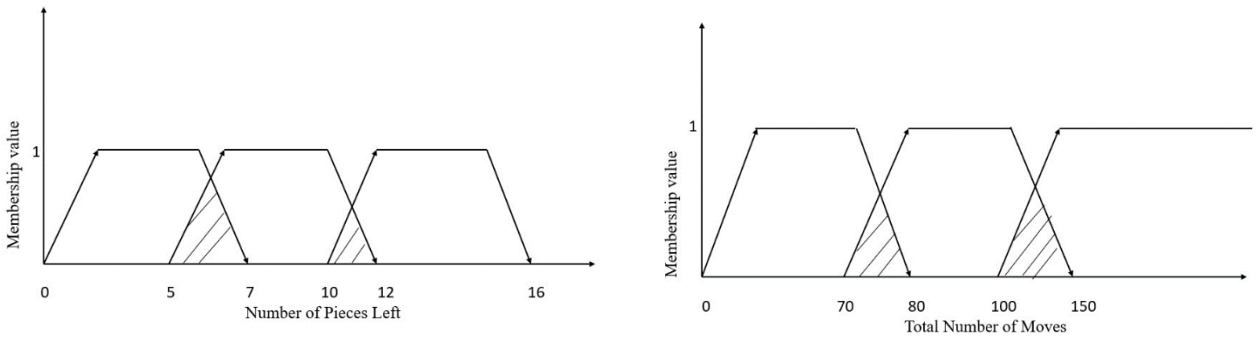


Fig-8: Input for Fuzzy system

Fuzzy_logic.py (pseudocode**)**

```
# Function to get piece membership degrees
FUNCTION
get_piece_membership(remaining_piece):
    degree = {
        "small_amount_piece_remaining": 0,
        "moderate_amount_piece_remaining": 0,
        "large_amount_piece_remaining": 0
    }

    IF remaining_piece is between 0 and 5
(inclusive):



    ELSE IF remaining_piece is between 5 and 7:
        degree["small_amount_piece_remaining"] =
- remaining_piece) / 2
        degree["moderate_amount_piece_remaining
= (remaining_piece - 5) / 2
        degree["large_amount_piece_remaining"] =

    ELSE IF remaining_piece is between 10 and 1
        degree["small_amount_piece_remaining"] =
        degree["moderate_amount_piece_remaining
= (12 - remaining_piece) / 2
        degree["large_amount_piece_remaining"] =
(remaining_piece - 10) / 2

    RETURN degree
# Function to get moves membership degrees
FUNCTION get_moves_membership(moves):
    degree = {
        "few": 0,
        "mid": 0,
        "huge": 0
    }
    IF moves is between 0 and 70 (inclusive):
        degree["few"] = 1
        degree["mid"] = 0
        degree["huge"] = 0
    ELSE IF moves is between 80 and 100
(inclusive):
        degree["few"] = 0
        degree["mid"] = 1
        degree["huge"] = 0
    ELSE IF moves is greater than 150:
        degree["few"] = 0
        degree["mid"] = 0
        degree["huge"] = 1
    ELSE IF moves is between 70 and 80:
        degree["few"] = (80 - moves) / 10
        degree["mid"] = (moves - 70) / 10
        degree["huge"] = 0
    ELSE IF moves is between 100 and 150:
        degree["few"] = 0
        degree["mid"] = (150 - moves) / 50
        degree["huge"] = (moves - 100) / 50

    RETURN degree

# Function to evaluate fuzzy rules
FUNCTION fuzzy_rules_evaluation(piece,
moves):
    degree = {
        "narrow": 0,
        "moderate": 0,
        "huge": 0
    }
    moderate1 =
MIN(piece["small_amount_piece_remaining"
moves["few"])
    narrow1 =
MIN(piece["small_amount_piece_remaining"
moves["mid"])
    narrow2 =
MIN(piece["small_amount_piece_remaining"
moves["huge"])
```

```
    huge1 =
MIN(piece["moderate_amount_piece_remaining'
moves["few"])
    moderate2 =
MIN(piece["moderate_amount_piece_remaining'
moves["mid"])
    narrow3 =
MIN(piece["moderate_amount_piece_remaining'
moves["huge"])

    huge2 =
MIN(piece["large_amount_piece_remaining"],
moves["few"])
    huge3 =
MIN(piece["large_amount_piece_remaining"],
moves["mid"])
    moderate3 =
MIN(piece["large_amount_piece_remaining"],
moves["huge"])

    degree["narrow"] = MAX(narrow1,
MAX(narrow2, narrow3))
    degree["moderate"] = MAX(moderate1,
MAX(moderate2, moderate3))
    degree["huge"] = MAX(huge1, MAX(huge2,
huge3))

    RETURN degree

# Function to perform defuzzification
FUNCTION defuzzification(b):
    sum = b["huge"] + b["narrow"] + b["moderate"
    up = (b["narrow"] * 10 + b["moderate"] * 20 +
b["huge"] * 30)
    IF sum == 0:
        RETURN 0
    cog = up / sum
    RETURN cog

# Function to perform fuzzy logic calculation
FUNCTION fuzzy(moves, remaining_piece):
    move_membership =
get_moves_membership(moves)
    piece_membership =
get_piece_membership(remaining_piece)
    fuzzy_rules =
fuzzy_rules_evaluation(piece_membership,
move_membership)
    cog_value = defuzzification(fuzzy_rules)
    RETURN cog_value
```

Tools and Platform Used:

1. **Pygame**: A set of Python modules designed for writing video games, which includes computer graphics and sound libraries.
2. **VSCode** (Visual Studio Code): A free, open-source code editor developed by Microsoft, known for its versatility and extensive extensions.
3. **Inno Setup**: A free installer for Windows programs, which provides a simple script-driven system to create setup applications.

Game Flow:

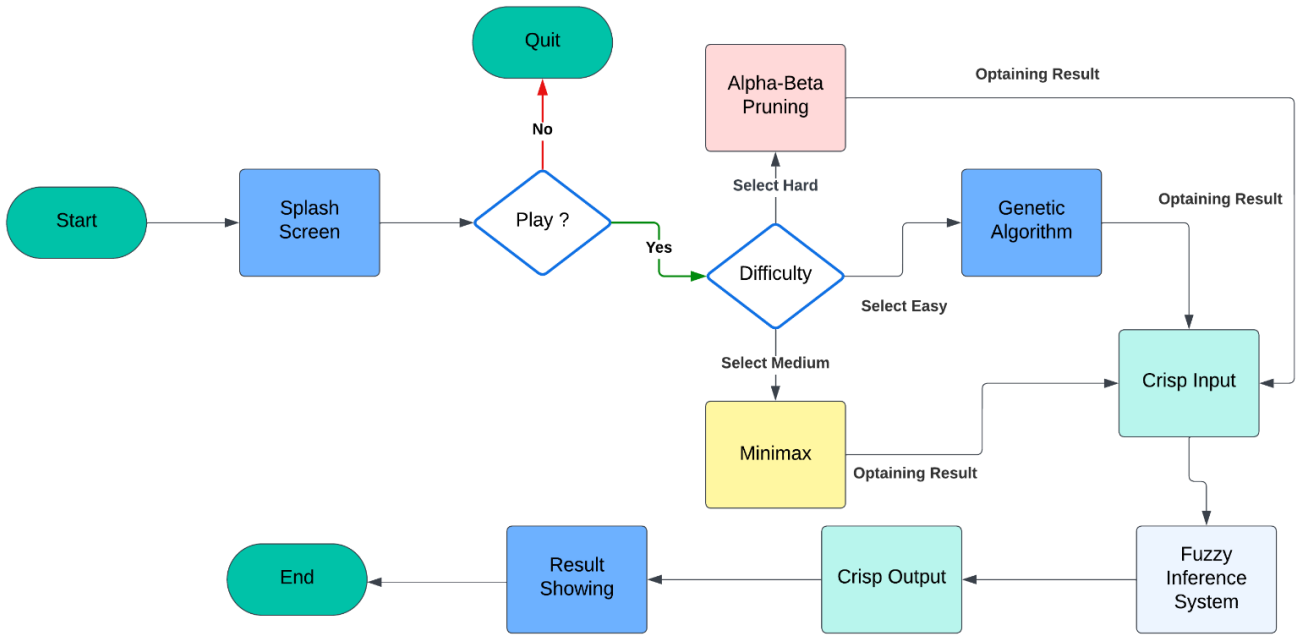
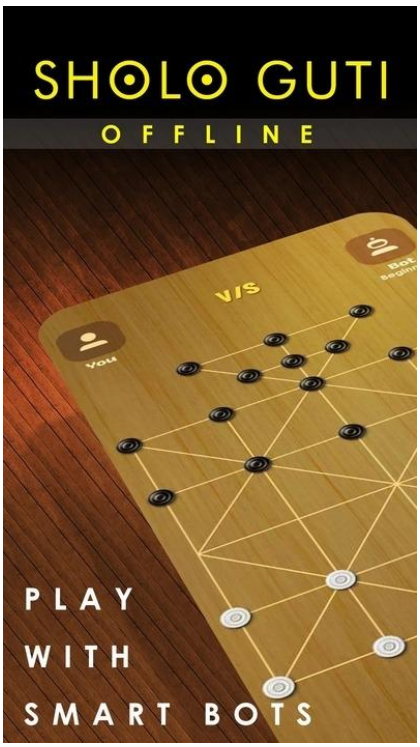


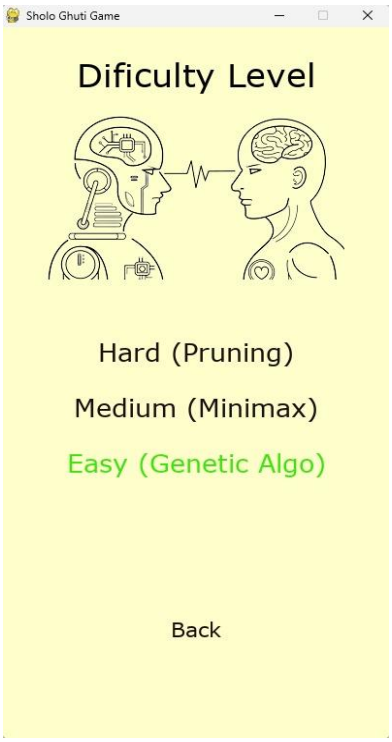
Fig-9: Game Flow Diagram

UI Mockup:

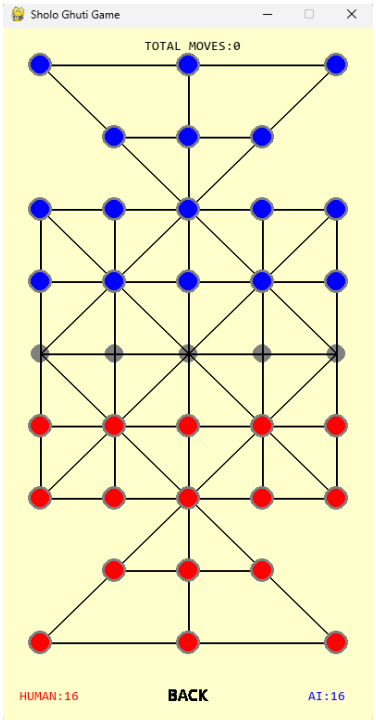


Splash Screen

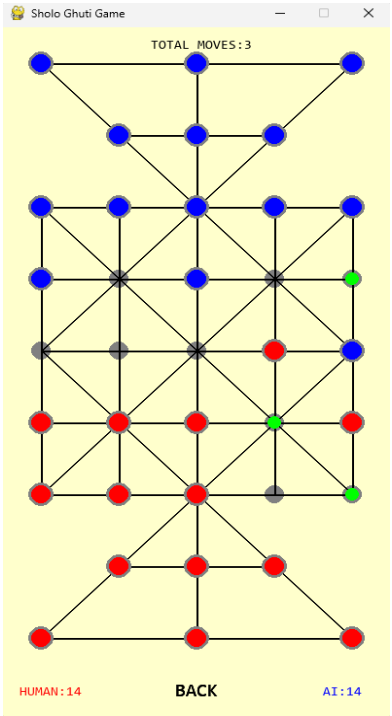
Home Page



Selection Page



Board (At start**)**



Board (Displaying Valid moves**)**

Result Display Page

Project Contribution:

1907003	1907006
▪ Board Creation	▪ Valid Moves Calculation
▪ Minimax	▪ Alpha-Beta Pruning
▪ Genetic Algorithm	▪ Fuzzy Inference System
▪ Exe File Generation	▪ Splash Screen
▪ Font End Design	▪ Font End Design
▪ Remaining Piece Counter	▪ Total Moves Counter
▪ Report Writing	▪ Report Writing

Discussion:

By integrating minimax with alpha-beta pruning, genetic algorithms, and Mamdani fuzzy logic, we have created a system that not only plays the game at a high level but also demonstrates the potential for AI in strategic decision-making. The minimax algorithm with alpha-beta pruning forms the backbone of the AI's decision-making process, allowing it to evaluate potential future game states and choose optimal moves. This approach ensures that the AI is a formidable opponent, capable of long-term planning and tactical play. The efficiency gained through alpha-beta pruning allows for deeper searches within reasonable time constraints, leading to stronger overall play. The incorporation of a genetic algorithm for move selection adds an element of adaptability and evolution to the AI's strategy. This approach allows the system to discover novel tactics and adapt its play style over time, potentially leading to strategies that might not be immediately apparent to human players or traditional game analysis. The genetic algorithm component also opens up possibilities for personalized gameplay, where the AI could evolve to match or challenge a specific player's skill level and style. The Mamdani fuzzy logic system brings a nuanced approach to game state evaluation. By considering factors such as the number of moves needed to win and the number of pieces remaining, the system can provide a more holistic assessment of the game's progress. This fuzzy evaluation can inform both the AI's decision-making process and provide valuable feedback to human players about the current state of the game.

However, it's important to acknowledge potential limitations and areas for future improvement. The computational complexity of the minimax algorithm, even with alpha-beta pruning, may limit the depth of search possible in real-time gameplay.

Conclusion:

In conclusion, our implementation of Sholo Guti using Pygame, enhanced with advanced AI techniques, represents a significant step forward in digital board game design and artificial intelligence application. By combining the minimax algorithm with alpha-beta pruning, genetic algorithms, and Mamdani fuzzy logic, we have created a system that not only plays Sholo Guti at a high level but also serves as a versatile platform for education, and research.

References:

1. https://www.researchgate.net/publication/268363154/figure/fig9/AS:347751591235591@1459921848469/Fuzzy-Inference-Engine-A-fuzzy-inference-system-FIS-consists-of-four-functional.png
2. https://i.pngimg.me/thumb/f/720/m2H7N4m2A0K9H7H7.jpg
3. www.javatpoint.org/alpha-beta _pruning
4. https://www.cs.princeton.edu/courses/archive/fall07/cos436/HIDDEN/Knapp/fuzzy004.htm
5. https://en.wikipedia.org/wiki/Selection_(genetic_algorithm)#:~:text=Selection%20is%20the%20stage%20of,individuals)%20for%20the%20next%20generation.
6. https://www.pygame.org/docs/