KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY

Department of Computer Science and Engineering

Course Code: CSE 4110

Course Title: Artificial Intelligence Laboratory

Project Title: "Hyper Backgammon" Game

Submitted to:

Md. Shahidul Salim
Lecturer
Department of Computer Science and Engineering
Khulna University of Engineering & Technology, Khulna

Most. Kaniz Fatema Isha
Lecturer
Department of Computer Science and Engineering
Khulna University of Engineering & Technology, Khulna

Submitted by:

Name: Swaraj Chandra Biswas
Roll:1907069
Department of Computer Science and Engineering
Khulna University of Engineering & Technology, Khulna

Objectives

1.Develop a fully functional Backgammon game using Python and the Tkinter library for the graphical user interface (GUI).

2.Implement game rules that ensure valid moves and movement restrictions according to traditional Backgammon rules.

3.Simulate the element of chance by incorporating dice rolling using Python's random module.

4.Create an AI opponent that utilizes minimax or fuzzy logic to make strategic decisions based on the current game state.

5.Provide a user-friendly and interactive gaming experience through a well-designed GUI.

6.Demonstrate the integration of game logic, GUI design, and AI to create a challenging and enjoyable board game experience.

Introduction:

Hyper Backgammon is a captivating two-player board game that blends strategic planning with elements of chance. The objective is to maneuver all checkers around the board and bear them off (remove them from the game) before opponent achieves the same feat. This implementation utilizes Python for core game logic and AI development, Tkinter to create a user-friendly graphical interface (GUI).

Features:

This implementation of the Hyper Backgammon game leverages the Python programming language along with the Tkinter library for the graphical user interface (GUI).

The game features include:

1.AI Opponent: Play against an AI opponent powered by the minimax algorithm and fuzzy logic for enhanced strategic decision-making.

2.Graphical User Interface (GUI): Developed using Tkinter, the GUI allows players to visually interact with the game board and checkers.

3.Checker Movement: Players can select and move checkers based on the dice rolls.

4.Dice Rolling: Dice rolls are simulated using Python's random module, providing the element of chance essential to Backgammon.

5.Win Detection: Automatically detects and announces the winner when a player bear all pieces off (remove them from the game).

Game UI:

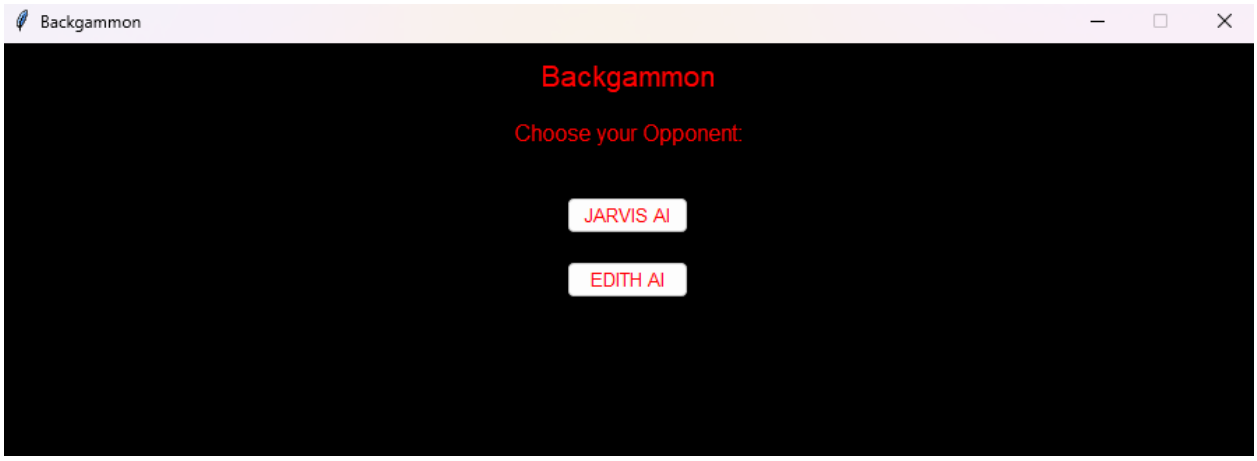

Fig1: Home screen GUI

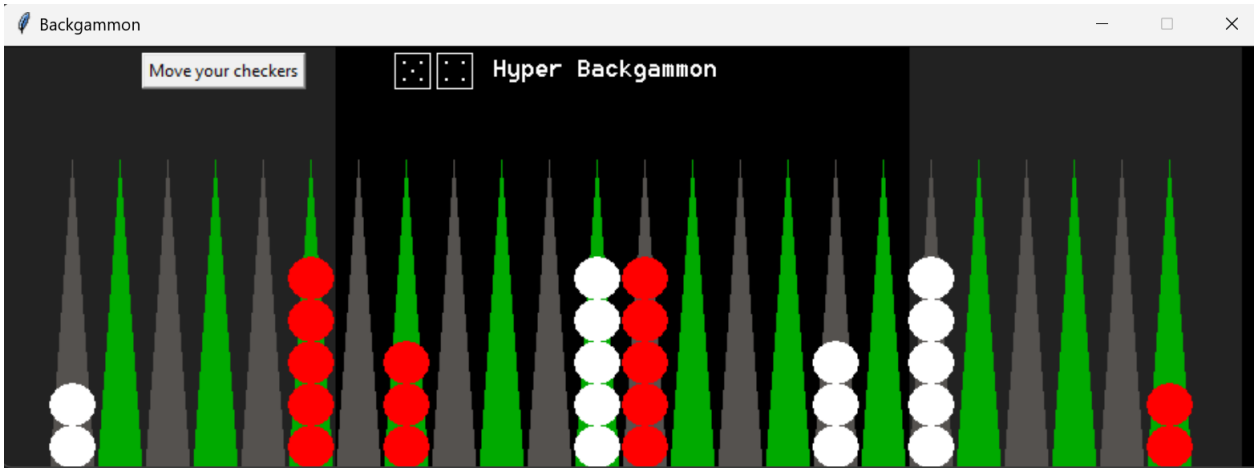

Fig2: Game Board GUI

Game Rules:

The objective is to maneuver all checkers around the board and bear them off (remove them from the game) before opponent achieves the same feat. The game follows Backgammon rules, which include:

1.Starting Position: The game board has 24 cones from 1 to 24.

- Each player has 15 pieces: Player has white pieces and the AI has red pieces.
- The initial setup places the white pieces on position 1->2 pieces, 12->5 pieces, 17->3 pieces, 19-> 5 pieces starting from left.
- The initial setup places the red pieces on same position starting from right.

2.Turns: Players take turns rolling two dice and moving their checkers based on the dice values.

3.Valid Moves:

- Players take turns rolling two dice to determine their possible moves.
- The numbers on the dice indicate how many points the checkers can be moved.
- A checker can only be moved to an open point, where an open point is either unoccupied, occupied by the player's own checkers, or occupied by exactly one opponent's checker (which can be hit).

4.Movement Restrictions:

- A player cannot move a checker to a point that is occupied by two or more of the opponent's checkers.
- If a player cannot make a valid move with the numbers rolled, they lose their turn.

5.Winning the Game:

- The first player to bear off all of their checkers wins the game.

6.AI Gameplay:

- When playing against the AI, the AI will make moves based on the minimax algorithm or fuzzy logic for strategic decision-making.
- The AI aims to maximize its chances of winning while minimizing the player's chances.

How to Play:

1. **Starting the Game**: Both players roll a single die to determine who starts. The player with the higher number goes first, using the numbers rolled on both dice.

2. **Rolling the Dice**: Each player rolls two dice on their turn to determine their moves.

3. **Moving Checkers**: Players move their checkers based on the dice roll, following the valid moves and movement restrictions.

4. **Hitting**: If a player moves a checker onto a point occupied by a single opponent's checker, the opponent's checker is hit and placed on the bar.

5. **Re-entering from the Bar**: A checker on the bar must be re-entered into the opponent's home board before any other move can be made.

6. **Bearing Off**: Once a player has moved all their checkers into their home board, they can begin bearing them off based on the dice rolls.

7. **Winning the Game**: The first player to bear off all of their checkers wins the game.

AI Development:

The AI opponent in this Backgammon game uses a simple form of fuzzy logic and minimax to make decisions. Fuzzy logic and Minimax allows the AI to evaluate the game state and make decisions that are not strictly binary (yes/no) but rather based on degrees of truth.

Fuzzy Logic Implementation:

The fuzzy rules and conditions can be designed to account for the uncertainties and complexities involved in the game.

Fuzzy Variables:

- **Checkers on Cone:** Number of checkers on a particular cone (few, moderate, many).
- **Enemy Presence:** Whether the cone is controlled by the enemy (yes, no).
- **Blot Risk:** Risk of leaving a single checker exposed (low, medium, high).
- **Bearing Off Potential:** Potential to bear off checkers (low, medium, high).
- **Move Quality:** Quality of the move (low, medium, high).

Fuzzy Rules:

A. Checker Density and Enemy Presence:

- If the number of checkers on present cone is double and the enemy is not present, then the move quality is low.
- If the number of checkers on a cone is many and the enemy is not present, then the move quality is medium.
- If the number of checkers on a cone is many and the enemy is present, then the move quality is high.

B. Combination of Conditions:

- If the number of checkers on a cone is moderate and the enemy is present then the move quality is high.
- If the number of checkers on a cone is two and the enemy is present the move quality is medium.
- If the number of checkers on a cone is two and the enemy is not present then the move quality is low.

C. Safety and Aggressiveness:

- If the move increases safety (reduces blots and consolidates checkers), then the move quality is high.
- If the move increases aggressiveness (hits enemy checkers and creates blots in the enemy's home board), then the move quality is high.

Pseudocode:

1.Risk Calculation: The AI evaluates the risk associated with each potential move. This involves calculating the likelihood of the AI's checkers being hit by the opponent based on their positions.

```
def risk_at_current_p(player_list, p):
    sum_risk = 0
    for i in player_list:
        risk_from_enemy = abs(i - p)
        if risk_from_enemy <= 6:
            sum_risk += 12* risk_from_enemy
        elif risk_from_enemy <12:
            sum_risk += 6* risk_from_enemy
    return  sum_risk
```

2.Move Evaluation: The AI evaluates potential moves and their associated risks to decide on the best move. This involves comparing the risks of different moves and selecting the one with the lowest risk.

```
def risk(player_list, p, move_list):
    min_risk = float('inf')
    for move in move_list:
        new_pos = p + move
        current_risk = risk_at_p(player_list, new_pos)
        if current_risk < min_risk:

       min_risk = current_risk
    return min_risk
```

3.Decision Making: Based on the evaluated risks, the AI decides which move to make. The AI aims to minimize the risk to its checkers while maximizing its chances of progressing toward bearing off its checkers.

```
def deside_move(position):
    move_list = possible_moves(position)
    best_move = None
    min_risk = float('inf')
    for move in move_list:
        current_risk = risk(player_list, position, move)
        if current_risk < min_risk:
            min_risk = current_risk
            best_move = move
    return best_move
```

Minimax Logic Implementation:

In backgammon, the minimax algorithm can be used to evaluate and make decisions by simulating potential future game states. The goal is to maximize the player's advantage while minimizing the opponent's possible gains.

Minimax Conditions:

A. **Game State Evaluation**:

- **Board Control**: Evaluate the control of key points (opponent's bar point or own 5-point).
- **Checker Positioning**: Consider the distribution of checkers across the board—whether they are well-distributed or clustered in a way that makes movement difficult.
- **Blots (Exposed Checkers)**: Evaluate the number of blots on the board. Leaving a blot is risky because it can be hit by the opponent.
- **Opponent's Checkers**: Consider the opponent's checker positions and potential moves. The opponent's bar and home boards are crucial in determining the best move.
- **Doubling Cube**: The current state of the doubling cube and the potential impact of doubling or accepting a double should be evaluated.

B. **Move Evaluation**:

- **Safety vs. Aggression**: Determine whether to prioritize safe moves (minimizing the number of blots) or aggressive moves (hitting the opponent's blots and taking risks).
- **Pip Count**: Evaluate the number of pips (the total number of points your checkers need to move to get off the board). A lower pip count is advantageous.
- **Home Board Strength**: Assess how strong your home board is (how many points are occupied and blocked). A strong home board can trap the opponent's checkers.
- **Bearing Off**: In the endgame, focus on bearing off checkers efficiently. Moves that bring checkers closer to bearing off should be prioritized.

C. **Opponent's Move Simulation**:

- **Worst-Case Scenario**: Consider the worst possible moves your opponent can make in response to your move. The minimax algorithm simulates these moves to find the optimal strategy.
- **Opponent's Strength**: Evaluate the strength of the opponent's position after each simulated move. A move that leaves the opponent in a weak position is preferred.

D. **Heuristic Evaluation Function**:

- **Positional Score**: Assign a score to the board position based on the above factors. The score reflects the strength of the position, with higher scores representing better positions.
- **Depth of Search**: Determine how many moves ahead to simulate. The deeper the search, the more accurate the evaluation, but it also requires more computational resources.
- **Pruning**: Use alpha-beta pruning to eliminate branches of the game tree that won't be considered by the opponent, reducing the computational load.

Minimax Rules:

A. Maximization of Player's Position:

- **Best Move Selection**: Choose the move that maximizes the player's positional score after considering the opponent's best possible responses.
- **Doubling**: Offer a double if the evaluation suggests a strong position that is likely to improve further.

B. Minimization of Opponent's Opportunities:

- **Blocking Opponent's Checkers**: Prioritize moves that limit the opponent's ability to advance their checkers, especially on their next roll.
- **Minimizing Blots**: Avoid leaving blots whenever possible, especially in the opponent's home board or other dangerous areas.

C. Balance Between Risk and Reward:

- **Risk Assessment**: Weigh the potential rewards of an aggressive move against the risk of being hit or leaving a weak position.
- **Defensive Moves**: When in a weak position, focus on defensive moves that secure key points and reduce the opponent's chances to hit your checkers.

D.Endgame Strategy:

- **Bearing Off**: Prioritize moves that get checkers off the board as quickly as possible, while still considering the opponent's ability to hit.
- **Avoiding Hits**: In the final stages, avoid leaving blots that could be hit, as this would significantly delay the bearing off process.

Pseudocode:

```
function Minimax(board, depth, maximizingPlayer):
    if depth == 0 or game_is_over(board):
        return evaluate_board(board)
    if maximizingPlayer:
        maxEval = -Infinity
        for each move in generate_moves(board, maximizingPlayer):
            eval = Minimax(apply_move(board, move), depth - 1, False)
            maxEval = max(maxEval, eval)
        return maxEval
    else:
        minEval = Infinity
        for each move in generate_moves(board, maximizingPlayer):
            eval = Minimax(apply_move(board, move), depth - 1, True)
            minEval = min(minEval, eval)
        return minEval
function Minimax decision (board, depth):
    bestMove = None
    maxEval = -Infinity
    for each move in generate_moves(board, True):
        eval = Minimax(apply_move(board, move), depth - 1, False)
        if eval > maxEval:
            maxEval = eval
            bestMove = move
    return bestMove
```

Discussion:

The Backgammon game We've developed using Python and Tkinter offers a pretty engaging blend of strategy and luck, thanks to a solid implementation of the game rules. The Python and Tkinter implementation of Backgammon effectively combines strategy, chance, and AI. The GUI provides a user-friendly interface, while the adherence to game rules ensures authenticity. Fuzzy logic and Minimax for the AI add complexity and challenge. However, to further enhance the game, exploring advanced AI techniques like reinforcement learning could be beneficial. Additionally, incorporating multiplayer functionality and customizable options would broaden the game's appeal.

Conclusion:

This Backgammon game implementation demonstrates the integration of graphical interfaces, game logic, and artificial intelligence using Python. The AI opponent uses fuzzy logic, minimax to make strategic decisions, providing a challenging experience for players. The use of Tkinter for the GUI ensures that the game is accessible and easy to interact with, making it a great project for both learning and entertainment. The project successfully combines traditional gameplay mechanics with modern technology, resulting in a well-rounded and entertaining experience. While the current implementation showcases the core elements of Backgammon, there is ample scope for further development, including advanced AI, multiplayer features, and enhanced customization. This project serves as a strong foundation for future explorations in game development and AI integration.