CSE 4110: Artificial Intelligence Laboratory

Last Mouse Lost

By

Soummo Bhattacharya

Roll: 1907105

Saugata Roy Arghya

Roll: 1907116

Submitted To:

Md. Shahidul Salim

Lecturer

Department of Computer Science and Engineering

Khulna University of Engineering & Technology

Most. Kaniz Fatema Isha

Lecturer

Department of Computer Science and Engineering

Khulna University of Engineering & Technology

Department of Computer Science and Engineering

Khulna University of Engineering & Technology

Khulna 9203, Bangladesh

September, 2024

Contents

List of Figures

CHAPTER I

Introduction

1.1 Objectives

The objectives of this project are:

- Develop a fully functional *Last Mouse Lost* game using Python and the Tkinter library for the graphical user interface (GUI).
- Implement game rules that ensure valid moves and enforce movement restrictions according to the game's mechanics.
- Integrate different AI strategies such as Random, Smart, Fuzzy Logic, Minimax, Genetic Algorithm, and A* to challenge the player.
- Provide a user-friendly and interactive gaming experience through a well-designed GUI.
- Demonstrate the integration of game logic, GUI design, and AI to create an engaging board game experience.

1.2 Introduction

Last Mouse Lost is a strategic two-player game where each player takes turns removing one or more 'mice' (game pieces) from a row. The player forced to remove the last mouse loses the game. The game can be played against another human or an AI opponent that utilizes different strategies to simulate varied difficulty levels.

Features

This implementation of the *Last Mouse Lost* game leverages the Python programming language along with the Tkinter library for the graphical user interface (GUI).

Game Features Include:

1.2.1 AI Opponents:

Play against various AI opponents powered by algorithms such as Random, Smart, Fuzzy Logic, Minimax, Genetic Algorithm, and A* for strategic decision-making.

1.2.2 Graphical User Interface (GUI):

Developed using Tkinter, the GUI allows players to interact with the game board visually, enhancing the user experience.

1.2.3 Player and AI Turns:

Players can select rows and specify the number of mice to remove based on their strategy. The AI computes its moves based on the selected strategy and updates the board accordingly.

1.2.4 Game Rules:

- Players take turns removing one or more mice from a single row.
- The player forced to remove the last mouse loses the game.

CHAPTER II

Theory

2.1 Overview

This game has Minimax, Genetic, Fuzzy logic, Game logic and A* algorithm implemented for the AI. The human player will able to choose any of these AI to play against.

2.2 Game Basic

The game interface is designed to be intuitive and user-friendly, allowing easy interaction for players. It includes options for selecting an AI opponent, displaying the current game state, and updating moves in real-time.

Game Rules

These are the rules of the game -

Objective:

The objective is to avoid removing the last mouse from the board.

Turn-Based Play:

Players alternate turns, removing one or more mice from a single row during their turn.

Valid Moves:

A move is valid if it involves removing at least one mouse from a row that still has mice remaining.

Winning the Game:

The player forced to remove the last mouse loses the game.

2.3 AI Development

In the *Last Mouse Lost* game, various AI strategies have been implemented to challenge the player. Each AI player uses a different algorithm to decide the moves based on the current state of the game board. Below is an extensive description of each AI strategy, including the logic behind them, their decision-making processes, and pseudocode to illustrate their functionality.

2.3.1 Random Player

Overview: The Random Player AI makes moves without considering the current state of the game board. It selects a random row and a random number of mice to remove from that row. This player provides a baseline challenge and is the easiest AI opponent.

Logic:

- Randomly selects a row that is not empty.
- Randomly selects a number of mice to remove from the chosen row, ensuring that it does not exceed the number of remaining mice in that row.

Pseudocode:

```
function random_move(board):
    # Find a non-empty row
    row = random_integer(0, 5)
    while board.row_empty(row):
        row = random_integer(0, 5)

    # Select a random number of mice to remove from the selected row
    amount = random_integer(1, board.spot_avail(row))

    return (row, amount)
```

2.3.2 Smart Player

Overview: The Smart Player uses a deterministic strategy based on game heuristics. It tries to create a board state that is difficult for the opponent to play against, often aiming for symmetry or specific configurations that limit the opponent's choices.

Logic:

- Evaluates the current board configuration.
- Attempts to reduce the board to a configuration known to be winning or difficult for the opponent.
- Selects moves that either create symmetrical patterns or leave an odd number of mice in each row.

Pseudocode:

```
function smart_move(board):
    current_configuration = calculate_board_configuration(board)

    if current_configuration is a known winning configuration:
        return make_move_to_reduce_to_smaller_winning_configuration(board)

    return make_random_valid_move(board)
```

2.3.3 Fuzzy Logic Player

Overview: The Fuzzy Logic Player uses fuzzy logic principles to evaluate potential moves. This AI evaluates the game state using fuzzy rules to make decisions that are not strictly binary but based on degrees of truth, such as "more" or "less risky."

Logic:

- Generates all possible moves.
- Evaluates each move based on various fuzzy factors such as risk, potential gain, and opponent's difficulty.
- Chooses the move with the highest fuzzy score.

Pseudocode:

```
function fuzzy_logic_move(board):
    possible_moves = generate_all_possible_moves(board)
    best_move = None
    best_score = -infinity

    for move in possible_moves:
        score = evaluate_fuzzy_move(board, move)
        if score > best_score:
            best_score = score
            best_move = move

    return best_move

function evaluate_fuzzy_move(board, move):
    # Evaluate move based on fuzzy logic criteria
    risk = calculate_risk(board, move)
    potential_gain = calculate_potential_gain(board, move)
    opponent_difficulty = calculate_opponent_difficulty(board, move)

    return risk + potential_gain + opponent_difficulty
```

2.3.4 Minimax Player

Overview: The Minimax Player uses the minimax algorithm with alpha-beta pruning to evaluate potential future game states. It simulates both its own and the opponent's moves to find the optimal strategy that maximizes its advantage while minimizing the opponent's potential gains. The Minimax algorithm is a strategic decision-making tool used in two-player games to identify the best possible move. By simulating each player's potential actions and responses, the algorithm evaluates all possible outcomes. It operates under the assumption that the opponent will also play optimally, aiming to maximize their advantage. The goal of the Minimax algorithm is to find a move that minimizes the potential loss in a

worst-case scenario while maximizing the possible gain. It systematically explores the game tree to determine the optimal move, taking into account both the player's and the opponent's best strategies.

Logic:

- Recursively explores all possible moves to a certain depth.
- Evaluates each potential move based on the minimax algorithm.
- Uses alpha-beta pruning to eliminate branches that do not need to be explored, optimizing the search process.

Pseudocode:

```
function minimax(board, depth, maximizing_player, alpha, beta):
    if depth == 0 or board.is_game_over():
        return evaluate_board(board)

    if maximizing_player:
        max_eval = -infinity
        best_move = None
        for move in generate_all_moves(board):
            new_board = apply_move(board, move)
            eval = minimax(new_board, depth - 1, False, alpha, beta)
            if eval > max_eval:
                max_eval = eval
                best_move = move
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval, best_move
    else:
        min_eval = infinity
        best_move = None
        for move in generate_all_moves(board):
```

```
            new_board = apply_move(board, move)
            eval = minimax(new_board, depth - 1, True, alpha, beta)
            if eval < min_eval:
                min_eval = eval
                best_move = move
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval, best_move
```

2.3.5 Genetic Algorithm Player

Overview: The Genetic Algorithm Player evolves a population of move sequences over multiple generations. It uses concepts from natural selection, crossover, and mutation to optimize the strategy over time, selecting for the most effective moves.

Logic:

- Initializes a population of random move sequences.
- Evaluates each individual in the population based on a fitness function.
- Selects the top-performing individuals to form the basis of the next generation.
- Applies crossover and mutation to produce a new population.
- Repeats this process for a predefined number of generations.

Pseudocode:

```
function genetic_algorithm_move(board):
    population = initialize_population()

    for generation in range(num_generations):
        fitness_scores = evaluate_population(population, board)
        population = select_top_individuals(population, fitness_scores)
        new_population = []
        while len(new_population) < population_size:
```

```
            parent1, parent2 = select_parents(population)
            child1, child2 = crossover(parent1, parent2)
            new_population.extend([mutate(child1), mutate(child2)])
        population = new_population

    best_individual = max(population, key=lambda ind: evaluate_individual(ind, boa
    return best_individual[0]

function evaluate_individual(individual, board):
    # Simulate the moves of the individual
    simulated_board = board.clone()
    score = 0
    for move in individual:
        apply_move(simulated_board, move)
        if simulated_board.is_game_over():
            score -= 100  # Heavy penalty for a losing move
            break
        else:
            score += evaluate_move(simulated_board)
    return score
```

2.3.6 A* Player

Overview: The A* Player uses the A* search algorithm to find the optimal path to victory. It evaluates moves based on a heuristic function that estimates the cost to reach the end of the game, favoring moves that appear to lead to a win in the shortest time.

Logic:

- Maintains an open set of potential moves, prioritized by their estimated total cost (current cost + heuristic estimate).
- Iteratively selects the move with the lowest estimated total cost, exploring its consequences.
- Continues until a winning move is found or all possibilities are exhausted.

Pseudocode:

```
function a_star_move(board):
    open_set = set()
    open_set.add((0, None, board))  # (heuristic cost, move, board state)
    closed_set = set()

    while open_set:
        current_node = select_node_with_lowest_cost(open_set)
        open_set.remove(current_node)
        _, move, current_board = current_node

        if current_board.is_game_over():
            return move

        closed_set.add(current_board)

        for move in generate_all_possible_moves(current_board):
            new_board = apply_move(current_board, move)
            if new_board in closed_set:
                continue
            heuristic_cost = calculate_heuristic(new_board)
            open_set.add((heuristic_cost, move, new_board))

    return select_best_move_based_on_heuristic(open_set)
```

Detailed AI Player Behavior

Random Player:

Provides a simple challenge by making completely random moves. This AI is ideal for beginners learning the game mechanics without being overwhelmed by strategy.

Smart Player:

Employs a simple deterministic strategy, often reducing the game to patterns that it knows to be favorable. This AI is more predictable and provides intermediate difficulty.

Fuzzy Logic Player:

Uses fuzzy logic to assess moves based on multiple heuristics. This AI offers varied difficulty depending on how well the heuristics match the current game state.

Minimax Player:

Leverages the depth and complexity of the minimax algorithm to explore potential moves ahead of time. It uses alpha-beta pruning to optimize the decision process, providing a challenging opponent.

Genetic Algorithm Player:

Simulates evolutionary strategies to improve over time, providing a dynamic opponent that can potentially evolve better strategies as the game progresses. This AI is complex and can offer a unique challenge depending on the evolved strategies.

A* Player:

Utilizes a heuristic-driven search algorithm to find the shortest path to victory. It's particularly challenging when the heuristic is well-tuned to the game's dynamics, making it a formidable opponent in strategic scenarios.

CHAPTER III

Implementation and Results

4.1 Main Menu

This is the main menu. Player first encounters this game when running the code.

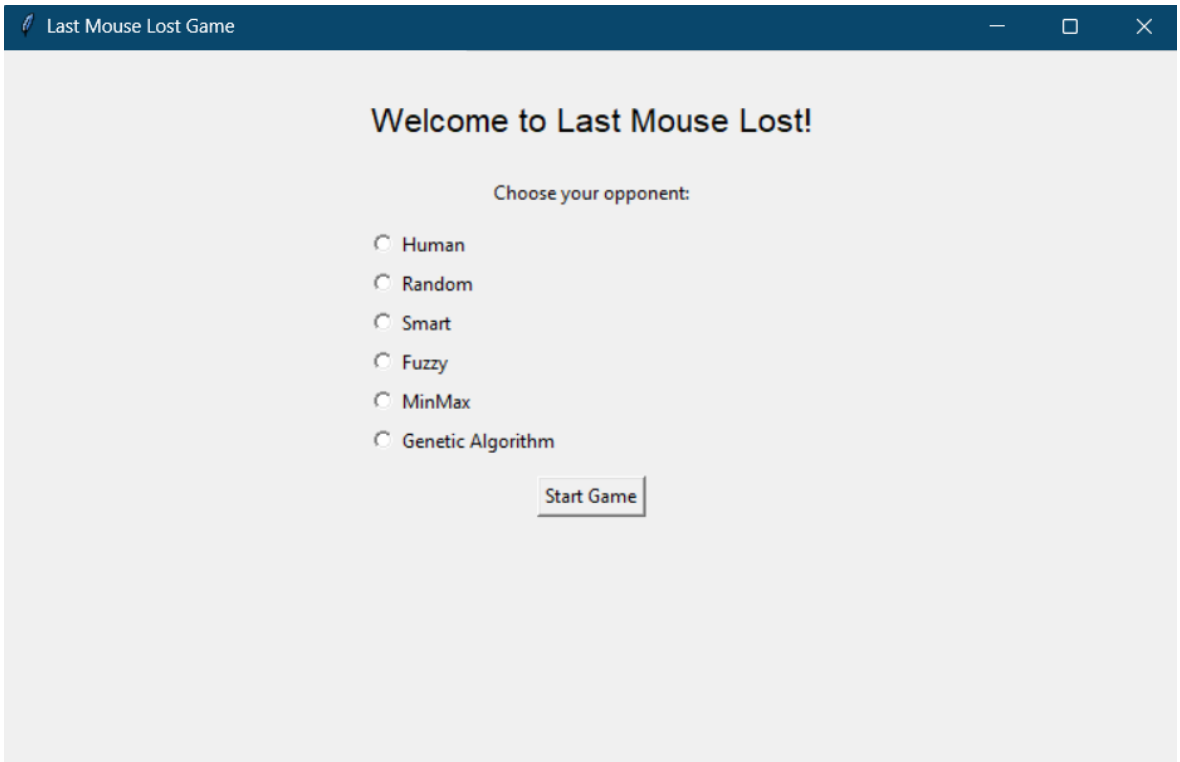

Figure 4.1: Main Menu.

4.2 Game Board

This is the game board.

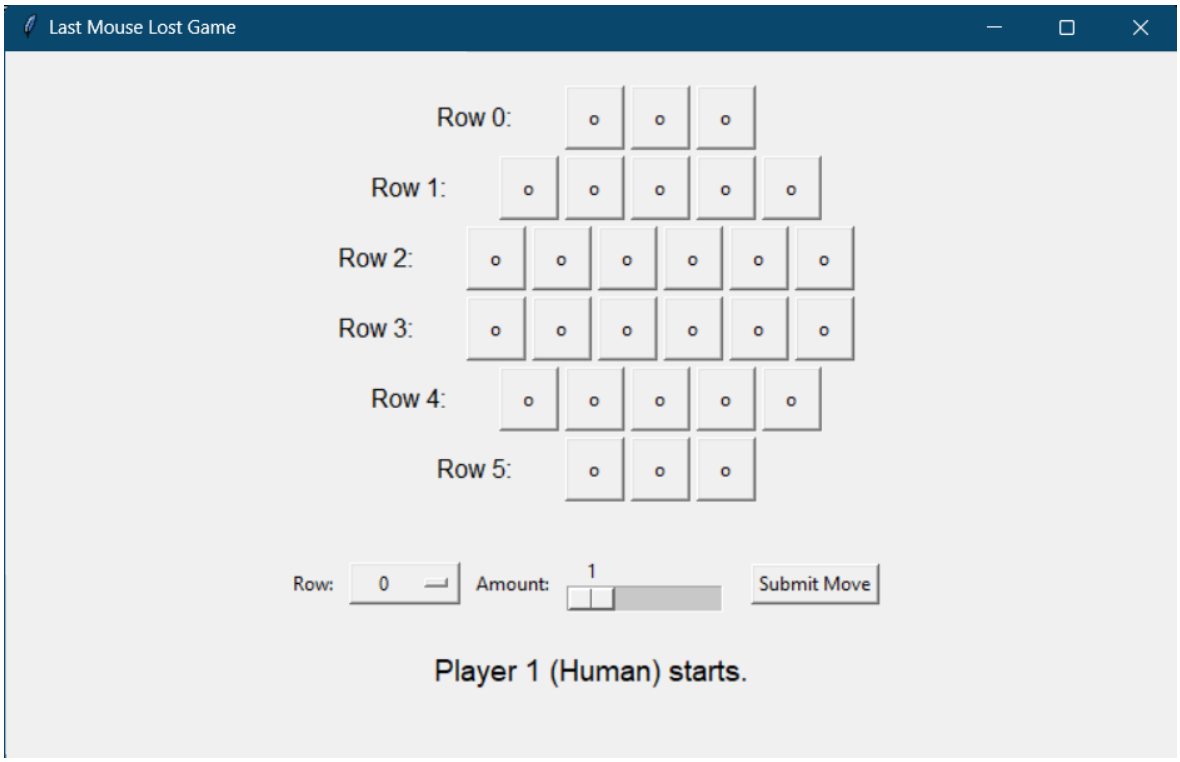

Figure 4.2: Game Board.

4.3 Row Selector

This allows player to select rows (0-5).

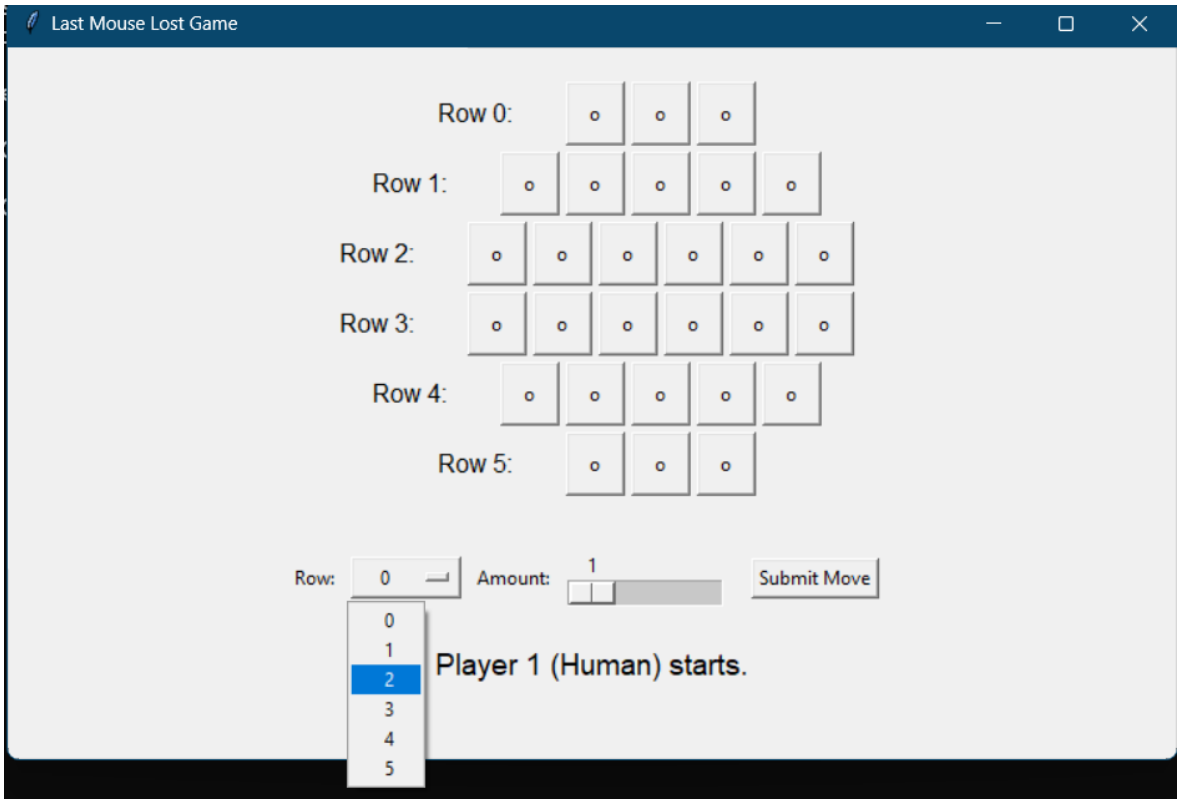

Figure 4.3: Row Selector.

4.4 Amount Slider

This selects the amount of blocks selected.

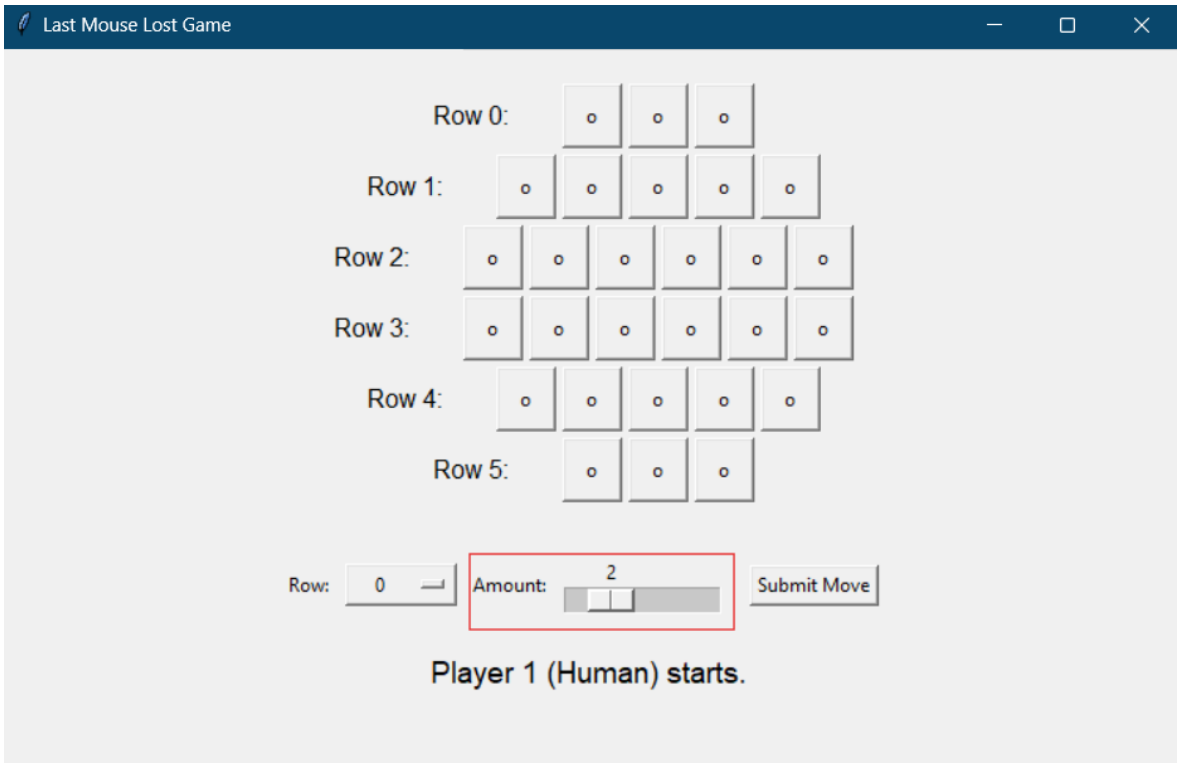

Figure 4.4: Amount Slider.

Minimum amount of 1 and max amount of 6 will have to be selected.

4.5 Invalid Move Warning

This shows up when invalid move is made. Here invalid move is made when a player selects row or amount out of range or selects a row where no available spots are left.

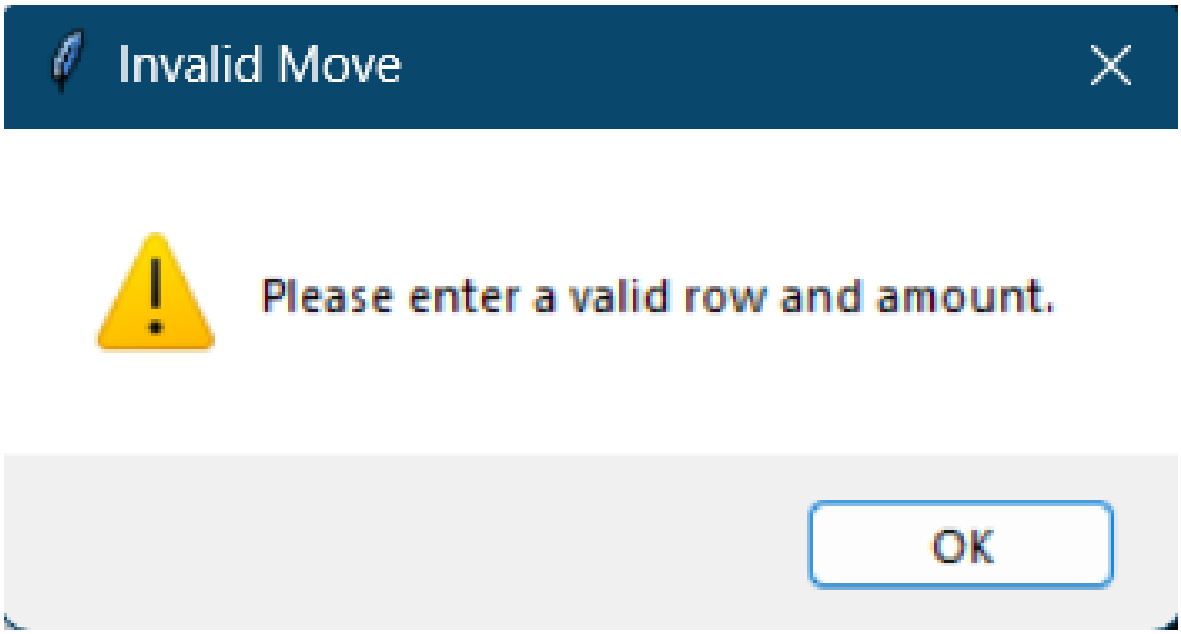

Figure 4.5: Invalid Move Warning.

4.6 Final Result

This will be shown when a player the game.

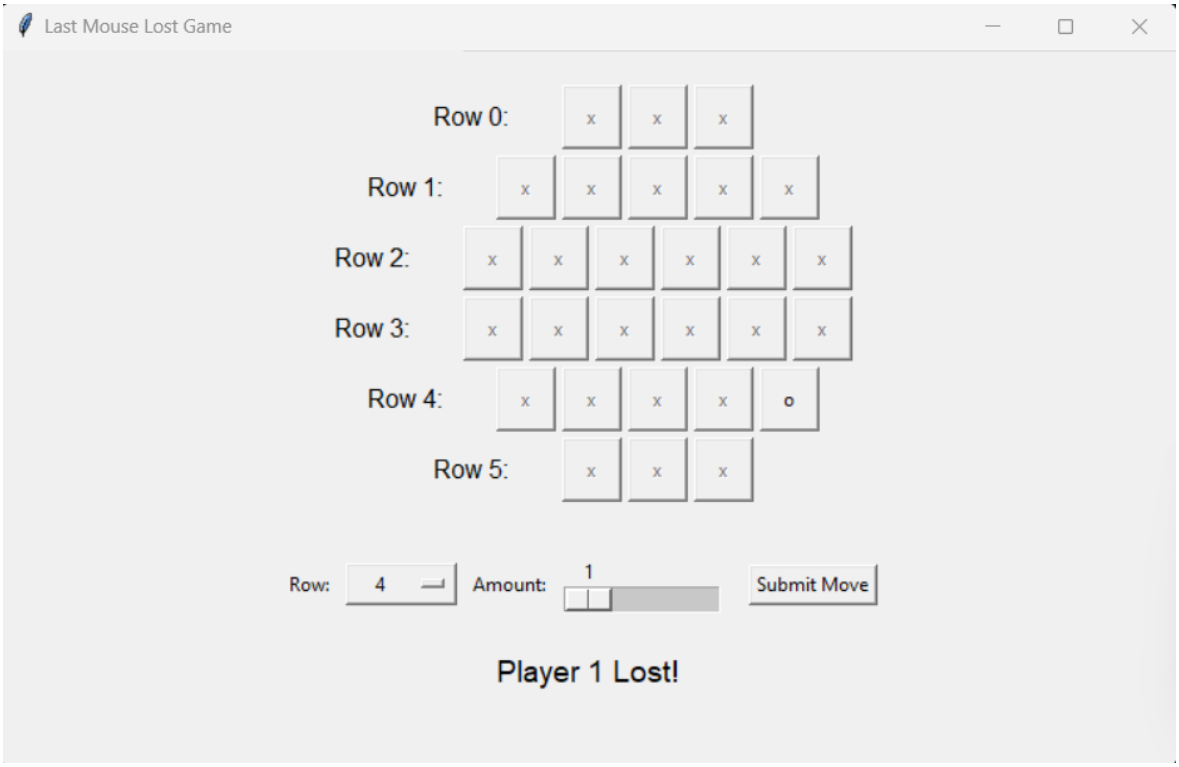

Figure 4.6: Final Result.

4.7 New Game/ Quitting Game

This is will be shown after a game to give the player to restart the game with a new or same opponent or totally quit the game

Figure 4.7: Game Options.

CHAPTER IV
Discussion and Conclusion

5.1 Discussion

The Python and Tkinter implementation of "Last Mouse Lost" effectively combines strategy, user interaction, and AI. The GUI provides an accessible and engaging interface, while the adherence to game rules ensures a consistent experience. Each AI opponent adds a different level of complexity and challenge, enhancing replayability.
However, there is potential for further development, such as incorporating more advanced AI techniques like reinforcement learning or adding multiplayer functionality and customizable options to broaden the game's appeal.

5.2 Conclusion

This "Last Mouse Lost" game implementation demonstrates the integration of graphical interfaces, game logic, and artificial intelligence using Python. The variety of AI strategies provides a challenging experience for players, and the use of Tkinter for the GUI ensures that the game is accessible and easy to interact with. The project serves as a foundation for future explorations in game development and AI integration, offering a well-rounded and entertaining experience.

5.3 Future Works

- **Optimization**: Improve the efficiency of the "Last Mouse Lost" game by optimizing move validation and board updates to ensure smoother gameplay with minimal delays.
- **Mechanics Refinement**: Refine the rules and interaction mechanics, including move restrictions and turn-based play, to enhance game balance and provide a more enjoyable experience.

- **Enhancement of AI**: Develop more advanced AI strategies, such as Minimax and Fuzzy Logic, to create more challenging and dynamic opponents.
- **GUI Improvements**: Upgrade the game's user interface for better navigation, clearer visual feedback, and a more engaging player experience.
- **Features**: Introduce new game modes, customizable board settings, and varying difficulty levels to increase replay value and player engagement.
- **Cross-Platform Support**: Explore options for cross-platform compatibility to make "Last Mouse Lost" accessible to a broader audience across different devices.

CHAPTER VI

References

References

[1] Baker, C. (2014). *Physics for Game Developers*. O'Reilly Media.

[2] Pygame Documentation. (2024). *Pygame*.

[3] Sweigart, A. (2015). *Making Games with Python Pygame*. No Starch Press.