KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY

Department of Computer Science and Technology

Course Code: CSE 4110

Course Title: Artificial Intelligence Laboratory

Project Title: Chain Reaction 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: Hasibul Hasan Hasib

Roll: **1907089**

Name: Md Robiul Islam

Roll: **1907101**

Department of Computer Science and Engineering

Khulna University of Engineering & Technology, Khulna

1 Introduction

1.1 Background

Chain Reaction is a multiplayer strategy game developed using Pygame. The game involves players taking turns to place and infect cells on a grid-based board. Each cell can "explode" under certain conditions, infecting adjacent cells and leading to a chain reaction. The objective is to eliminate opponents by strategically managing these chain reactions. This game not only challenges players' strategic thinking but also offers an engaging and competitive gameplay experience.

1.2 Objectives

The main objectives of developing Chain Infect are:

- To develop an interactive and engaging multiplayer game.
- To implement game mechanics that involve strategic placement and infection of cells.
- To provide a visually appealing user interface using Pygame.
- To ensure smooth gameplay with responsive controls and animations.

2 Methodology

2.1 Description of processes

The Chain Reaction game in AI refers to a strategic and tactical game often used to study and develop artificial intelligence algorithms. Here’s a breakdown of how the game works:

a) Game Objective:

- The objective of Chain Reaction is to take control of the entire grid by strategically placing orbs in cells, causing chain reactions that convert opponent's orbs into the player's orbs.

.

b) Game Setup:

- The game is played on a grid (5x5).
- Players take turns placing orbs in empty cells or cells containing their own orbs.
- Each cell can hold a certain number of orbs before it reaches a critical mass and explodes.

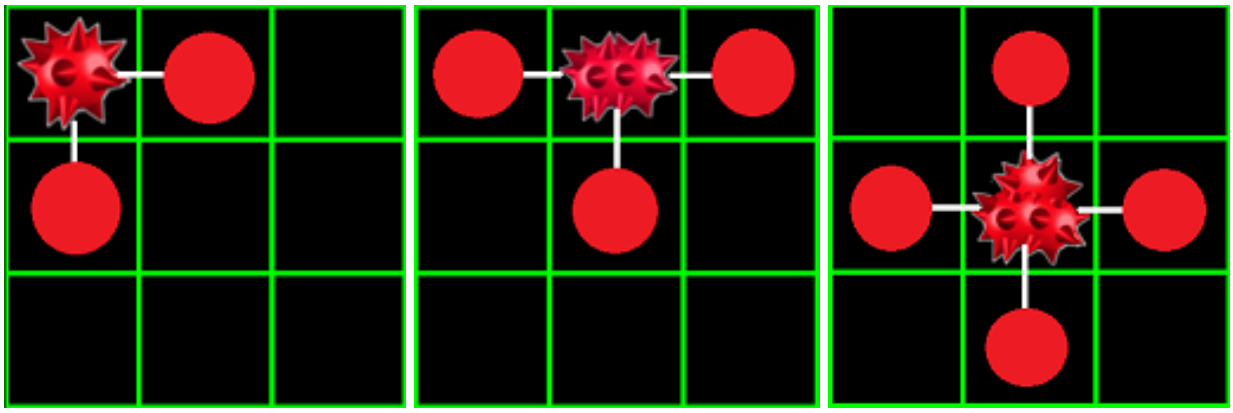


figure 1. Criteria for exploding orbs

c) Gameplay Mechanics:

- When a cell reaches its critical mass (number of orbs equal to the number of its neighboring cells), it explodes, sending one orb to each neighboring cell.
- The explosion can cause neighboring cells to also reach critical mass and explode, creating a chain reaction.

- Orbs from an explosion convert any orbs in neighboring cells to the player's color.
- The game continues until one player dominates the grid or the other player cannot make a move.

2.2 Methods Used in The Game

The development of the Chain Reaction game involved implementing various classes and methods to handle different aspects of the gameplay. Below is a detailed description of the key methods used in the project, along with advanced AI techniques like Minimax, Alpha-Beta Pruning , Genetic Algorithm, and Fuzzy Logic.

2.2.1 **Add(x, y, playerTurn1)**:

1. Updates the board state by adding a player's move to the specified position (x, y).
2. If the position is empty, it places the player's marker.
3. If the position is occupied by the same player, it increases the marker count.
4. If the position reaches its maximum capacity, it triggers an explosion based on the position type (corner, edge, or middle).

2.2.2 **Genetic Algorithm Methods**

1. **generate_population(**size, player**):**
    - Generates a list of possible moves (population) for the given player based on the current board state.

2. **evaluate_fitness(**player, candidate**)**:
    - Evaluates the fitness of a candidate move by simulating the move and calculating the board's evaluation score for the given player.
3. **rank_based_selection(**population, fitness_scores, num_parents**)**:
    - Selects the top moves (parents) from the population based on their fitness scores.
4. **crossover(**parent1, parent2**)**:
    - Generates two new moves (children) by randomly selecting positions on the board.
5. **mutate(**candidate, mutation_rate**)**:
    - Introduces random variations to a move (candidate) with a certain probability (mutation rate) to maintain genetic diversity.
6. **genetic_algorithm(**board, player, population_size, num_parents, num_generations, mutation_rate**)**:
    - Executes the genetic algorithm to evolve the best move over several generations using selection, crossover, and mutation.

2.2.3 **Minimax and Alpha-Beta Pruning Methods**

1. **minimax(**depth, alpha, beta, maximizing_player, player**)**:
    - Implements the Minimax algorithm with alpha-beta pruning to evaluate and choose the best move by exploring possible future board states up to a certain depth.

2.2.4 **Move Selection Methods**

1. **best_move(**player**)**:
    - Determines the best move for the player using a combination of the genetic algorithm and the Minimax algorithm with alpha-beta pruning. It prioritizes the move that maximizes the player's advantage.

2.2.5 **Fuzzy Logic Methods**

1. **fuzzy_membership_threat(x)**:
    - Determines the fuzzy membership levels for a given threat value, categorizing it into Very Low, Low, Medium, High, and Very High.
2. **fuzzy_membership_opportunity(x)**:
    - Determines the fuzzy membership levels for a given opportunity value, categorizing it into Very Low, Low, Medium, High, and Very High.
3. **fuzzy_evaluate(threat_levels, opportunity_levels)**:
    - Evaluates and returns descriptive comments on the threat and opportunity levels based on their fuzzy membership values.
4. **fuzzy_evaluate_player_move(board, player, x, y)**:
    - Evaluates a player's move based on the current board state using fuzzy logic, providing comments on the threat and opportunity levels for the move.

2.3 Pseudo Code

2.3.1 Main Game loop

```
WHILE game is not over:
    Print current board state
    best_move = best_move(playerTurn)
    add(best_move.x, best_move.y, playerTurn)
    Check for explosions and update the board accordingly
    Switch to the next player
    Check if there is a winner or if the game is over
```

2.3.2 Add Orbs Function

```
FUNCTION add(x, y, playerTurn):
   IF position (x, y) is empty:
      Place playerTurn's marker at (x, y)
   ELSE IF position (x, y) is occupied by playerTurn:
      IF position is a corner:
         explodeCorner(x, y, playerTurn)
      ELSE IF position is an edge:
         IF marker count < 2:
            Increment marker count at (x, y)
         ELSE:
            explodeEdge(x, y, playerTurn)
      ELSE:
         IF marker count < 3:
            Increment marker count at (x, y)
         ELSE:
            explodeMiddle(x, y, playerTurn)
```

2.3.3 Min-Max Alpha Beta Function

```
FUNCTION minimax(depth, alpha, beta, maximizing_player, player):
 IF depth is 0 OR game is over:
      RETURN evaluate_board(board, player), None

   IF maximizing_player:
      max_eval = negative infinity
      best_move = None
      candidate_moves = genetic_algorithm(board, player)
      FOR each move in candidate_moves:
         IF move is valid:
            Copy current board state
            dummy_add(move.x, move.y, player)
            eval, _ = minimax(depth - 1, alpha, beta, False, player)
            Restore board state
            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 = positive infinity
    best_move = None
    candidate_moves = genetic_algorithm(board, opponent)
    FOR each move in candidate_moves:
        IF move is valid:
            Copy current board state
            dummy_add(move.x, move.y, opponent)
            eval, _ = minimax(depth - 1, alpha, beta, True, player)
            Restore board state
            IF eval < min_eval:
                min_eval = eval
                best_move = move
            beta = min(beta, eval)
            IF beta <= alpha:
                BREAK
    RETURN min_eval, best_move
```

2.3.4 Genetic Algorithm

```
FUNCTION genetic_algorithm(board, player):
    population = generate_population(player)
    FOR generation in range(num_generations):
        fitness_scores = []
        FOR candidate in population:
            fitness = evaluate_fitness(player, candidate)
            fitness_scores.append(fitness)
        parents = rank_based_selection(population, fitness_scores)
        new_population = []

  FOR i in range(population_size // 2):
            parent1, parent2 = random sample from parents
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])
   population = new_populatio
  RETURN population
```

2.3.5 Player Evaluation Using Fuzzy

```
FUNCTION fuzzy_evaluate_player_move(board, player, x, y):
    threat = 0
    opportunity = 0
    FOR each position on board:
        IF position is occupied by opponent:
            threat += 1
        IF position is occupied by player:
            opportunity += 1
    threat_levels = fuzzy_membership_threat(threat)
    opportunity_levels = fuzzy_membership_opportunity(opportunity)
    threat_comment, opportunity_comment = fuzzy_evaluate(threat_levels,  opportunity_levels)
    RETURN threat_comment, opportunity_comment
```

2.3.6 Best Move

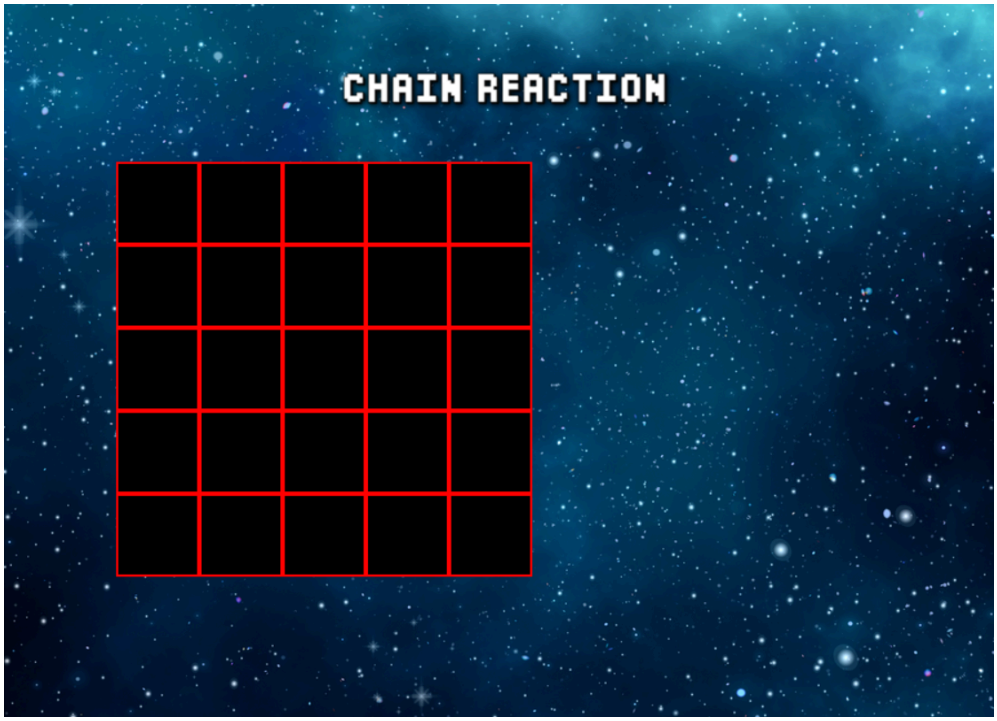
```
FUNCTION best_move(player):
    best_val = negative infinity
    best_move = best_check()
  IF  best_move is None:
        candidate_moves = genetic_algorithm(board, player)
        FOR each move in candidate_moves:
            IF move is valid:
                Copy current board state
                dummy_add(move.x, move.y, player)
                move_val, _ = minimax(depth=8, alpha=negative infinity, beta=positive
        infinity, maximizing=False, player)
                Restore board state
                IF move_val > best_val:
                    best_val = move_val
                    best_move = move
        RETURN best_move
```

3. Result

3.1 User Experience

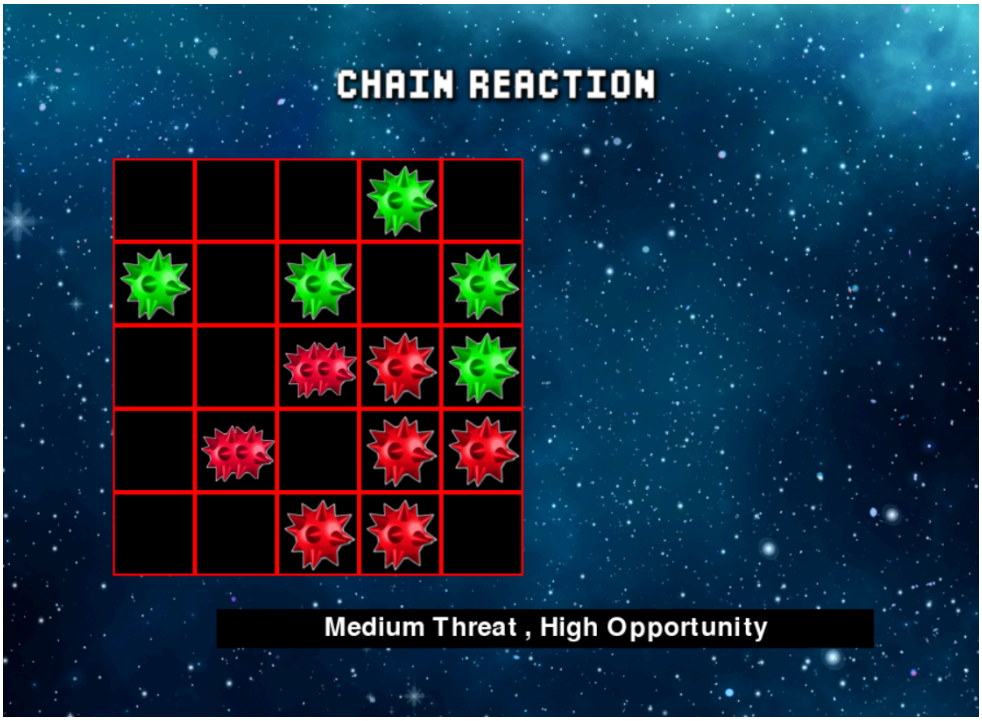
Starting Interface: After opening the game this interface will occur. Then clicking the play button can go to the settings interface.

Player Selecting Interface: User can select the number of the total players. Opponent 2 will be AI.



Initial Board: The board has 5x5 cells. The color of cells are the color of the player ,who will give next move

Comment on Current Move: Using fuzzy logic the comment gives the information of goodness of the current move and risk of losing the game.



Winner Declaration: The player who removes all the orbs of the all opponents wins the game

3.2 Performance Metrics

3.2.1 Execution Time:

- The use of the Genetic algorithm reduces the number of nodes at each level.
- Alpha-Beta Pruning in the Minimax algorithm reduced the execution time for the AI's decision-making process. This allowed for real-time gameplay without noticeable delays.

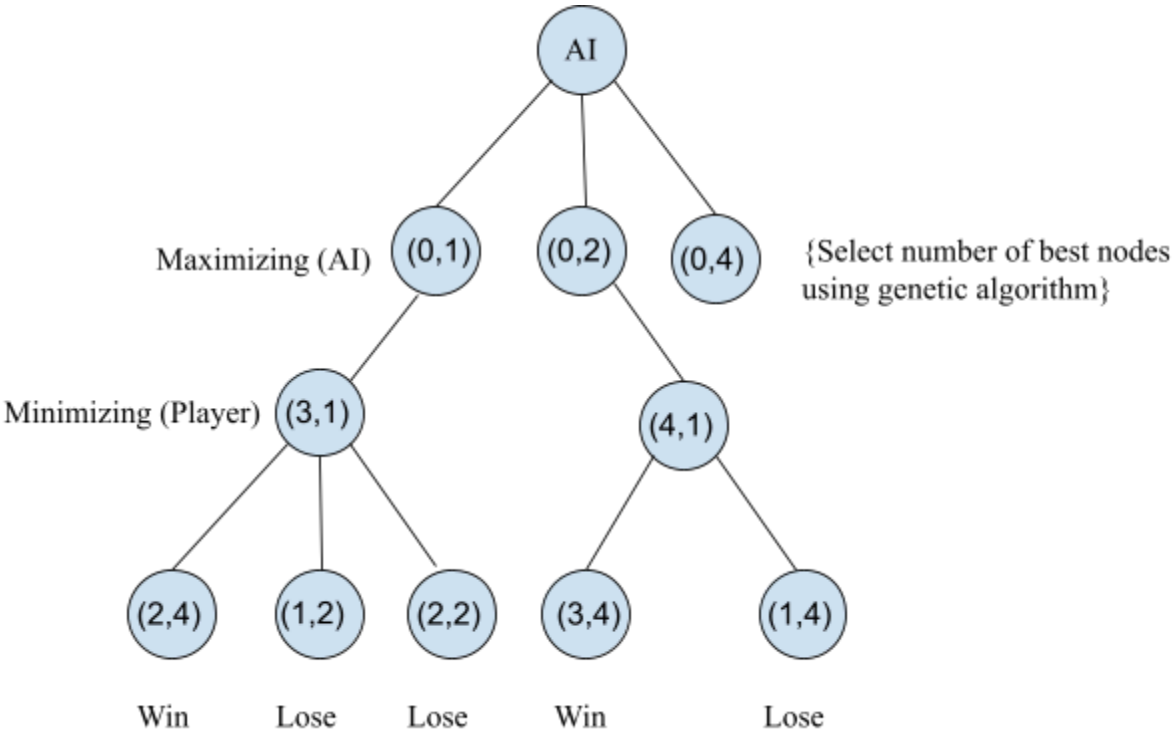


Figure 2. Alpha Beta pruning with genetic Algorithm

3.2.2 Accuracy:

- The Genetic Algorithm's optimization process resulted in a highly accurate evaluation function, enabling the AI to make strategic decisions that closely mimic human expertise.
- The fuzzy evaluation method provided a reliable assessment of ambiguous game states, leading to improved player performance.

3.2.3 Board Evaluation :

- The Evaluation is based on atom counts and additional scores for controlling strategic positions like corners and edges.
- It adjusts the score positively for the player and negatively for opponents.

4. Discussion

Our game project integrates strategic board mechanics with advanced AI techniques to create an engaging player experience. The game features strategic marker placement and explosive mechanics governed by complex rules, complemented by AI decision-making using genetic algorithms, minimax with alpha-beta pruning, and fuzzy logic. Evaluation metrics such as win rates, player engagement, and bug tracking provide insights into game performance and player satisfaction.

5. Future Plans

- **Enhanced AI Capabilities**: Expand the AI's strategic depth by incorporating machine learning algorithms to improve decision-making.
- **Advanced Game Mechanics**: Introduce new game mechanics and challenges to keep gameplay fresh and engaging.
- **Improved Visuals and Sound**: Enhance the game's aesthetic appeal with improved graphics, animations, and sound design.
- **Cross-Platform Compatibility**: Enable cross-platform compatibility to allow players to enjoy the game on various devices and platforms seamlessly. This could involve optimizing the game for mobile devices, consoles, and PCs to reach a broader audience.

6. Conclusion

In conclusion, the completion of this game project represents a significant achievement in integrating strategic board mechanics with advanced AI algorithms. The implementation of genetic algorithms, minimax with alpha-beta pruning, and fuzzy logic has resulted in dynamic gameplay that challenges players while providing opportunities for strategic decision-making.

7. Contributions

1907089 - Game logic, Minimax , genetic algorithm

1907101 - Fuzzy logic, UI

8. References:

- https://realpython.com/python-minimax-nim/
- https://www.gamedeveloper.com/design/genetic-algorithms-in-games-part-1-