Battleship: A Game with Search Techniques, MiniMax Algorithm,Genetic Algorithm and Fuzzy Logic Implementation

Khulna University of Engineering & Technology
Department of Computer Science and Engineering

CSE 4110: Artificial Intelligence Laboratory

Submitted To:

Md. Shahidul Salim
Lecturer
Department of Computer Science and Engineering

Most. Kaniz Fatema Isha
Lecturer
Department of Computer Science and Engineering

Submitted By:

Farhatun Shama (Roll: 1907033)
Lamisa Bintee Mizan Deya (Roll: 1907049)
Year: Fourth
Semester: First

Date: 03 September 2024

Objectives

The objectives of this project are:

- To develop a Battleship game where the user can play against an AI opponent.
- To implement Genetic Algorithms (GA) for the strategic placement of AI's ships.
- To utilize basic search techniques, fuzzy logic, and minimax algorithms to enhance the AI's gameplay strategy.
- To provide a drag-and-drop interface for the human player to place their ships.
- To know the practical implementation of the topics that we have covered in the laboratory.

Introduction

Battleship is a classic two-player game where the objective is to sink the opponent's fleet of ships before they sink yours. Each player arranges their ships on a grid and then takes turns guessing the locations of the opponent's ships. The game continues until one player has sunk all of the opponent's ships.

In this project, we have developed a digital version of the Battleship game where a human player can play against an AI opponent. The AI uses various techniques to improve its gameplay:

- **Genetic Algorithms (GA)**: These are used for the strategic placement of the AI's ships, making it more challenging for the human player to detect and hit the ships. The GA optimizes the ship placements through a process of selection, crossover, and mutation, ensuring that the placements are both strategic and difficult for the opponent to guess.
- **Basic Search Techniques**: Implemented in the basic_ai function, these techniques help the AI explore the grid and make informed guesses based on previous hits. This includes prioritizing cells adjacent to known hits to find and sink ships efficiently.
- **Fuzzy Logic**: Used to enhance the AI's decision-making process, particularly in generating random moves that are more likely to hit the human player's ships. Fuzzy logic allows the AI to handle uncertainty and make better decisions based on partial information.
- **Minimax Algorithm**: Utilized to optimize the AI's moves by evaluating the potential outcomes and choosing the best possible move. This algorithm considers the possible responses from the human player and selects the move that minimizes the maximum potential loss.

Methodology

The overall user interface consists of a menu window and a gaming window. The menu window is designed using tkinter and the gaming window is designed using pygames.

Menu

The main menu of the game application, designed using the Tkinter library, features six buttons divided into two categories: Mode Buttons and Gameplay Buttons. Each button, when pressed, opens the main game window and passes a unique parameter based on the selected mode or gameplay demonstration.

Mode Buttons

- **Easy**: Here, a human player competes against the Minimax-based AI. Pressing the "Easy" button launches the game with the Minimax AI set to challenge the human player. The AI mode is set to Minimax AI.
- **Medium**: In this mode, a human player competes against the Basic AI. Pressing the "Medium" button launches the main game window with the Basic AI set to play against the human player. The parameter passed is the AI mode set to Basic AI.
- **Hard**: This mode pits a human player against the Fuzzy Logic-based AI. The "Hard" button starts the main game window with the Fuzzy Logic AI. The parameter passed sets the AI mode to Fuzzy Logic AI.

Gameplay Buttons

- **Battle 1(Basic AI vs Fuzzy Logic AI)**: Demonstrates a game between Basic AI and Fuzzy Logic AI. Pressing the ”Battle 1” button launches a game showcasing the competition between these two AIs, with parameters set accordingly.
- **Battle 2 (Basic AI vs Minimax AI)**: Showcases a game between Basic AI and Minimax AI. The ”Battle 2” button starts a demonstration game with parameters set for each AI respectively.
- **Battle 3 (Fuzzy Logic AI vs Minimax AI)**: This button initiates a demonstration game between Fuzzy Logic AI and Minimax AI, highlighting their strategic differences and capabilities.

Fig. 1 shows the initial menu.

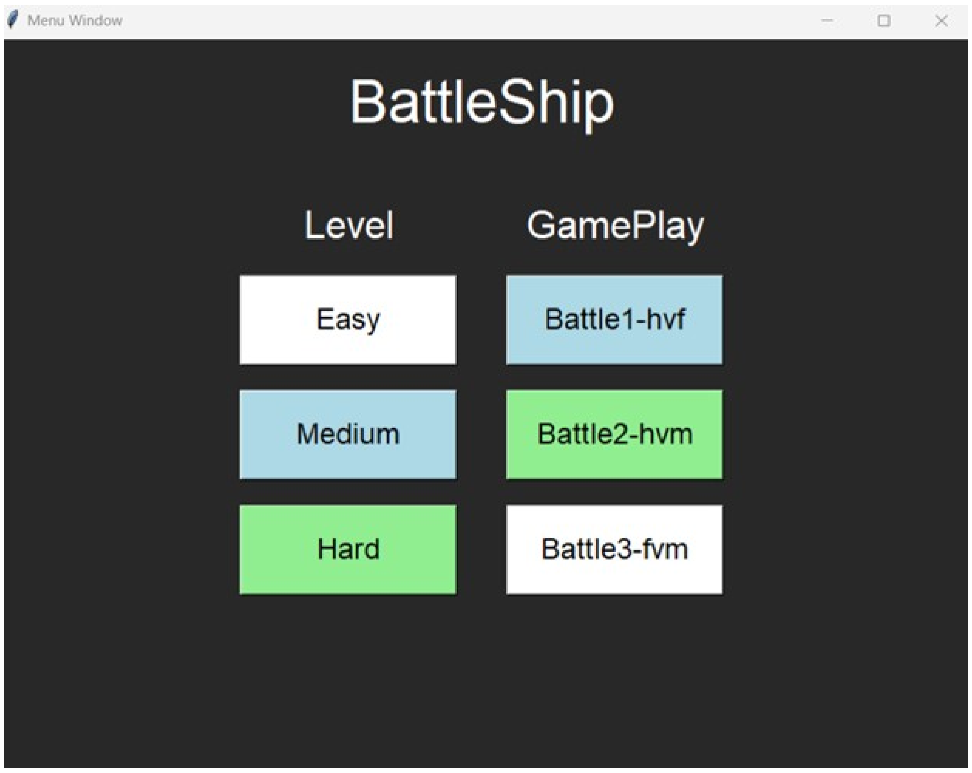

Figure 1: Modes and gameplay menu

Game Interface

The game interface consists of three grids and a results section:

- **Your Attacking Grid (Top Left)**: Displays the human player’s attempts to hit the AI’s ships.
- **Your Own Grid (Bottom Left)**: Shows the positions of the human player’s ships.
- **Computer’s Attacking Grid (Bottom Right)**: Displays the AI’s attempts to hit the human player’s ships.
- **Results Section (Top Right)**: Shows the statistics of the game, including hits and sinks for both players.

The human player places five ships on their grid using a drag-and-drop interface. The game enforces rules to prevent ships from overlapping and ensures they are placed within the grid boundaries. Fig. 2 shows the initial grids and ships.

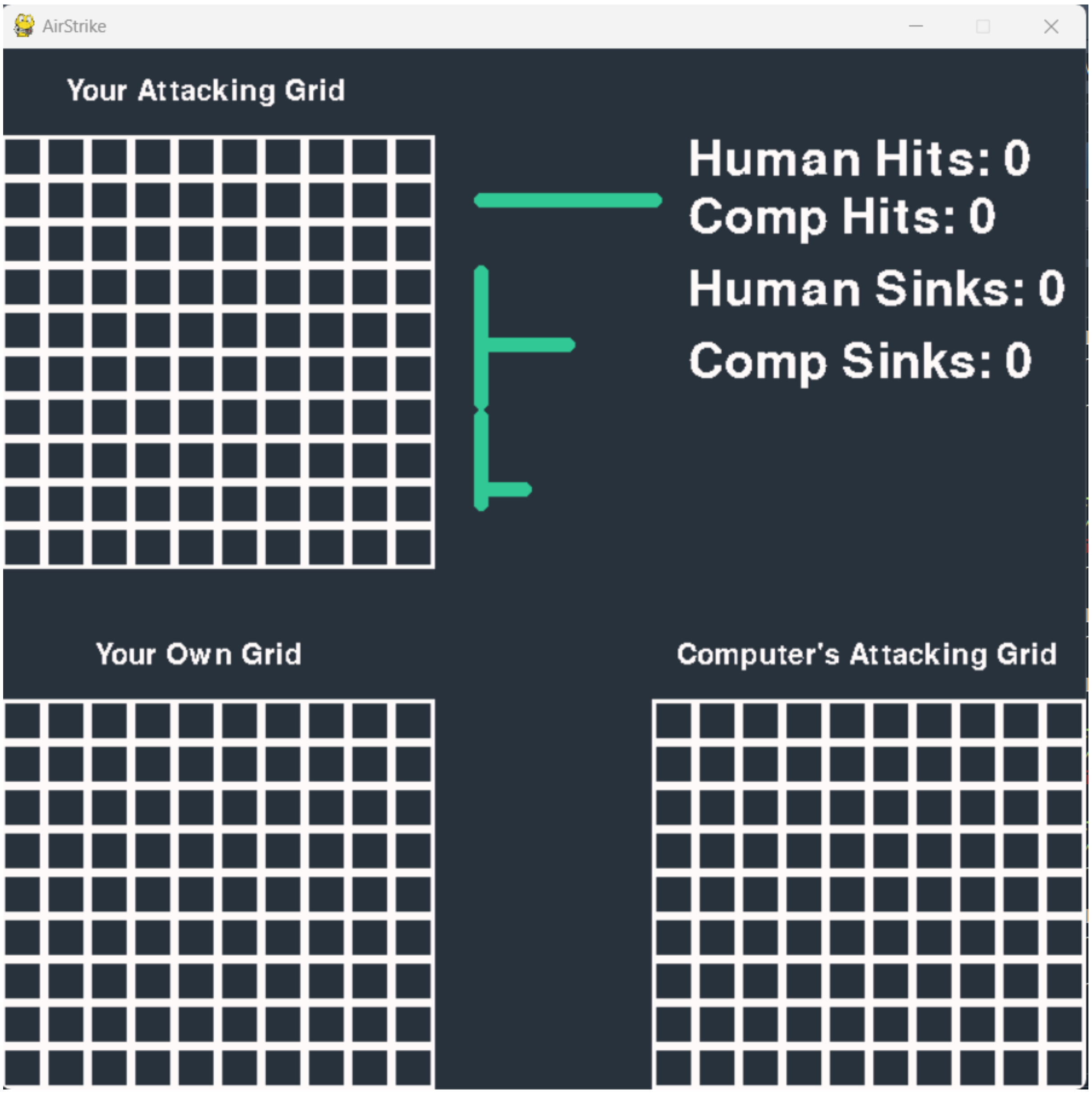

Figure 2: Initial Setup for Battleship AI

Genetic Algorithms for Ship Placement

The AI uses a Genetic Algorithm to strategically place its ships. The steps involved in this process are:

- **Initialization**: A population of possible ship placements is generated randomly. This is done by creating multiple valid ship placements, ensuring that the placements do not overlap and are within the grid boundaries.
- **Selection**: The fitness of each placement is evaluated based on how well it avoids clustering ships together. The fitness function calculates a score for each placement, considering factors such as distance between ships, edge and corner penalties, and grid coverage. Higher fitness scores are given to placements that spread ships out and avoid edges and corners.
- **Crossover**: Pairs of placements are combined to produce new offspring placements. The crossover process exchanges segments of two parent placements to create new child placements, inheriting characteristics from both parents.
- **Mutation**: Random changes are made to some placements to introduce variability. The mutation process randomly alters the position or orientation of ships in a placement, ensuring that new configurations are

explored.

- **Evaluation**: The new placements are evaluated, and the best ones are selected for the next generation. This iterative process continues for a specified number of generations, gradually improving the quality of ship placements.
- **Termination**: The algorithm runs for a set number of generations or until the placements converge to an optimal solution. The best placement is selected as the final configuration for the AI's ships.

Fig. 3 shows the ship pacement using genetic algorithm.

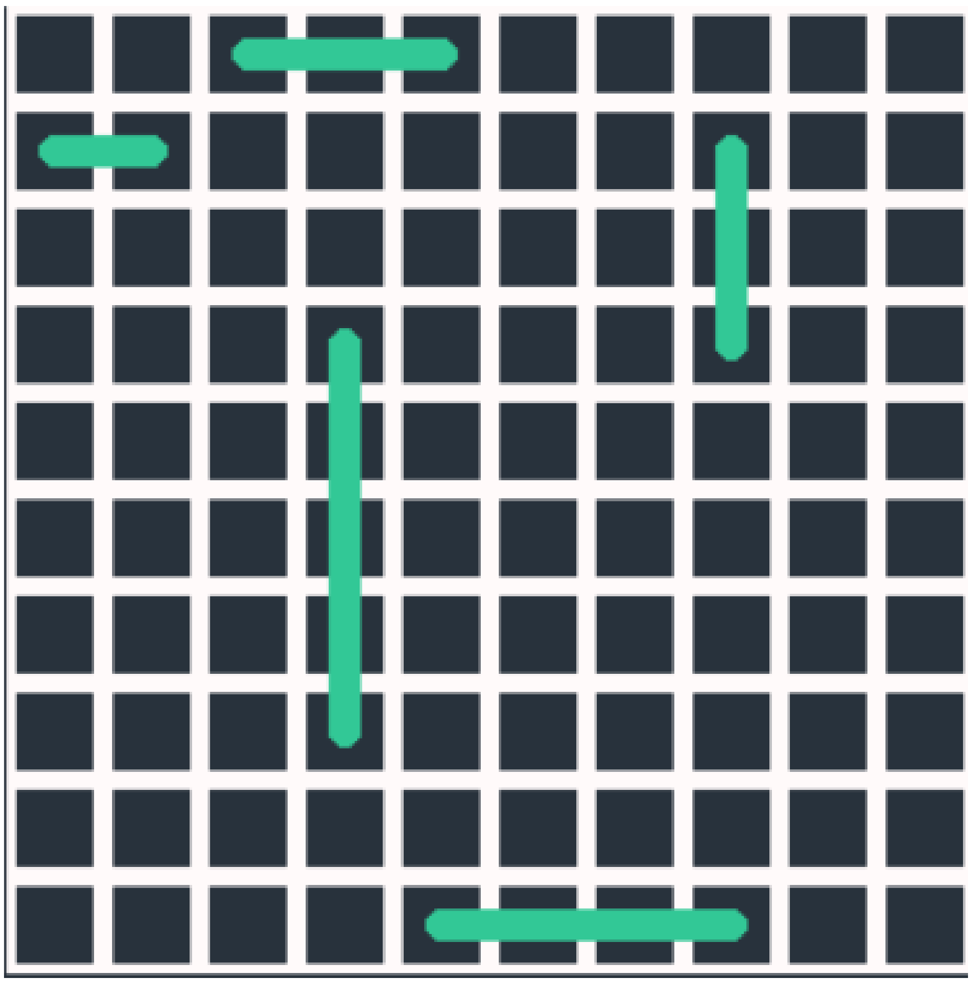

Figure 3: ship Placement using Genetic Algorithm

Algorithm 1 Genetic Algorithm for Battleship AI

```
Initialize GA parameters: population_size, mutation_rate, generations, grid_size
Function initialize_population(pop_size, ship_sizes, board_size)
    population ← []
    for each individual in pop_size:
        placement ← []
        for each ship in ship_sizes:
            Randomly place the ship on the board
            while placement is invalid: retry
        Add ship to placement
        population.append(placement)
    return population
Function valid_ship_placement(existing_placements, new_ship)
    Extract new_ship parameters
    if out-of-bounds or overlapping: return False
    return True if valid
Function fitness(placement)
    Initialize total_distance, penalties, coverage_bonus
    Calculate positions occupied by each ship
    Adjust fitness_score using distance, penalties, and bonuses
    return fitness_score
Function select_parents(population, fitnesses)
    Select two parents based on fitnesses using weighted random choice
    return selected parents
Function crossover(parent1, parent2)
    Randomly choose a crossover point
    Combine parents to form child1 and child2
    Validate and return children
Function mutate(individual, mutation_rate, board_size)
    for each ship in individual:
        if random value < mutation_rate: mutate ship's position
        while new placement is invalid: retry
    return individual
Function evolve_population(population, generations, mutation_rate, board_size)
    for each generation:
        Calculate fitness for each individual
        next_population ← []
        for half of population_size:
            Select parents, crossover, mutate children
            Add children to next_population
        Replace population with next_population
        Validate the best individual in population
        if valid: return best placement
        else return invalid placement message
```

Search Techniques for AI Moves

The AI employs a combination of heuristic and pattern-based search techniques to make informed moves:

- **Heuristic Search**: The AI prioritizes cells adjacent to known hits. It examines cells that are immediately adjacent and two cells away from hits to identify potential ship locations.
- **Pattern-Based Search**: When no adjacent hits are available, the AI uses a checkerboard pattern to explore the grid systematically. This approach ensures efficient coverage of the search space and increases the probability of hitting a ship.

Game Play

The game starts with the human player placing their ships using a drag-and-drop interface. Once the ships are placed, the human player presses a key to start the game. The AI places its ships using the Genetic Algorithm and begins making moves based on the search techniques described.

The human player and the AI take turns making moves, attempting to hit and sink each other's ships. The game continues until one player has sunk all of the opponent's ships, at which point the winner is declared.

The combination of Genetic Algorithms, search techniques, fuzzy logic, and the minimax algorithm creates a challenging and engaging experience for the human player, making the AI a formidable opponent in the Battleship game. A demonstration of gameplay with red as sunk,orange as hits and blue as miss is shown in Fig. 4.

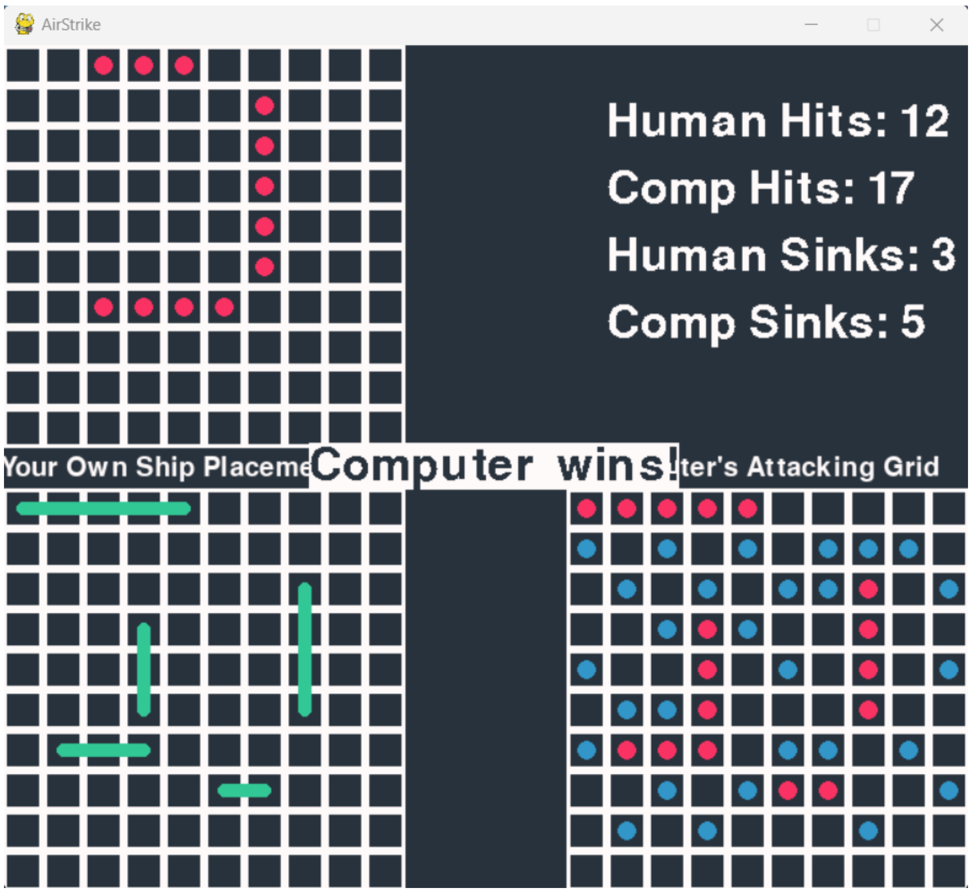

Figure 4: Game Play

Basic AI

The Basic AI in the game operates using a straightforward yet effective strategy designed to challenge the human player. Here's an overview of its operation:

- **Initialization:** The game initializes by setting up the players and the game state, which includes deciding the turn order.

- **Ship Placement:** The AI places its ships automatically on the grid using a predefined logic to ensure valid placements without overlaps and within the grid boundaries.
- **Gameplay Loop:**
  - **Decision Making:** At each turn, the Basic AI evaluates the game board to decide its next move. Initially, if no previous hits are guiding its decisions, it selects targets randomly to maximize coverage and potentially discover parts of ships.
  - **Targeting Strategy:** Once a hit is achieved, the AI switches to a more focused approach. It begins to target adjacent cells (i+1, i-1, i+10, i-10 and i+2, i-2, i+20, i-20 for grid-based adjacency). This strategy helps in trying to sink the discovered ship efficiently.
  - **Random Switch:** If the AI detects that surrounding hits have all been sunk (i.e., no adjacent hits lead to further damage), it reverts to random targeting to find new targets, thus alternating between strategic focus around known hits and broad random searches to discover new ships.
  - **Result Tracking:** The game continuously tracks each move's outcome—hit, miss, or sink—and updates the game board and state accordingly.
- **End of Game:** The game concludes once all the ships of either side are sunk. The surviving side, either the AI or the human player, is declared the winner.

This methodology allows the Basic AI to provide a consistent and adaptive challenge, making each game scenario unique and engaging. The AI's ability to switch between random searches and targeted attacks ensures a dynamic gameplay experience, keeping the human player engaged and strategizing throughout the game.

Algorithm 2 Heuristic Search for Battleship AI

```
Function basic_ai()
  Determine the current player's search grid:
    search ← player1.search if player1_turn else player2.search
  Identify unknown cells and hit cells:
    unknown ← [index $i$ for each square in search if square = "U"]
    hits ← [index $i$ for each square in search if square = "H"]
  Identify unknown cells with neighboring hits (1-cell distance):
    unknown_with_neighbouring_hits1 ← []
    for each $u$ in unknown do
      if ($u+1 \in$ hits) or ($u-1 \in$ hits) or ($u+10 \in$ hits) or ($u-10 \in$ hits)
        unknown_with_neighbouring_hits1.append($u$)
    end if
  Identify unknown cells with neighboring hits (2-cell distance):
    unknown_with_neighbouring_hits2 ← []
    for each $u$ in unknown do
      if ($u+2 \in$ hits) or ($u-2 \in$ hits) or ($u+20 \in$ hits) or ($u-20 \in$ hits)
        unknown_with_neighbouring_hits2.append($u$)
    end if
  Prioritize cells with hits within both 1-cell and 2-cell distances:
    for each $u$ in unknown do
      if $u \in$ unknown_with_neighbouring_hits1 and $u \in$ unknown_with_neighbouring_hits2
        make_move($u$)
        return $u$
      end if
    end for
  Otherwise, choose a random cell with a neighboring hit (1-cell distance):
    if unknown_with_neighbouring_hits1 is not empty then
      $num$ ← make_move(random choice from unknown_with_neighbouring_hits1)
      return $num$
    end if
  If no nearby hits, use checkerboard strategy:
    checker_board ← []
    for each $u$ in unknown do
      $row$ ← $u$ // 10
      $col$ ← $u$ % 10
      if ($row + col$) % $2 == 0$
        checker_board.append($u$)
      end if
    end for
    if checker_board is not empty then
      $num$ ← make_move(random choice from checker_board)
      return $num$
    end if
  Fallback to random AI if no other options:
    random_ai()
End Function
```

Fuzzy AI

The Fuzzy AI incorporates fuzzy logic to enhance decision-making during gameplay, providing a dynamic and adaptable opponent. Here is how the Fuzzy AI operates:

- **Initialization:** Similar to Basic AI, Fuzzy AI starts with setting up its ships on the game board, adhering to placement rules that ensure no overlapping and all ships are within the boundaries.
- **Fuzzy Logic System Setup:**
  - **Definition of Variables:** The AI defines fuzzy variables such as 'hits' and 'unknowns' which represent the counts of adjacent hits and unknown squares around a target, respectively.
  - **Membership Functions:** These variables are assigned membership functions (e.g., low, medium, high) to classify the degree to which a cell's characteristics belong to these categories.
  - **Rules Setup:** Fuzzy rules are established to dictate the targeting probability based on the defined variables. For example, a cell surrounded by more hits and fewer unknowns might have a higher probability of being targeted.
- **Decision Making:**
  - The AI evaluates each unknown cell on the board using its fuzzy logic control system, calculating the probability of targeting each cell based on its neighboring hits and unknowns.
  - The AI selects the cell with the highest calculated probability as its next move, aiming to optimize its chances of hitting a ship.
- **Adaptation and Targeting:**
  - **First Hit Search:** If no previous hits guide the AI, it uses fuzzy logic to determine the most probable initial targets, enhancing the chances of hitting a ship.
  - **Subsequent Strategy:** After a hit, similar to the Basic AI, it focuses on adjacent cells but integrates fuzzy logic to decide systematically rather than randomly, improving the hit accuracy and efficiency.
  - **Extended and Dynamic Targeting:** Depending on the results of ongoing hits, the Fuzzy AI dynamically adjusts its targeting strategy, sometimes expanding the search area beyond immediate neighbors to locate parts of larger ships or disconnected ship sections.
- **Game Progression and Conclusion:** The game progresses with alternating turns between the AI and the human player. The AI continues to apply its fuzzy logic system throughout the game to adapt to the evolving board state. The game ends when all ships of one side are sunk.

The use of fuzzy logic allows the AI to mimic human-like decision-making by dealing with uncertainties and partial truths effectively, making it a more challenging opponent compared to traditional deterministic AIs.

Algorithm 3 Fuzzy Logic Search for Battleship AI

```
Function fuzzy_ai()
    Determine the current player's search grid:
        search_grid ← player1.search if player1_turn else player2.search
    Initialize fuzzy variables and membership functions:
        hits ← Fuzzy variable representing the number of neighboring hits
        unknowns ← Fuzzy variable representing the number of neighboring unknowns
        probability ← Fuzzy variable representing the probability of a promising target
    Define fuzzy rules:
        rule1 ← IF hits = low AND unknowns = high THEN probability = medium
        rule2 ← IF hits = low AND unknowns = medium THEN probability = low
        rule3 ← IF hits = low AND unknowns = low THEN probability = low
        rule4 ← IF hits = medium AND unknowns = high THEN probability = high
        rule5 ← IF hits = medium AND unknowns = medium THEN probability = medium
        rule6 ← IF hits = medium AND unknowns = low THEN probability = low
        rule7 ← IF hits = high AND unknowns = high THEN probability = high
        rule8 ← IF hits = high AND unknowns = medium THEN probability = high
        rule9 ← IF hits = high AND unknowns = low THEN probability = medium
    Create fuzzy control system and simulation:
        probability_ctrl ← Control system with rules [rule1, rule2, ..., rule9]
        probability_simulation ← Simulation of probability_ctrl
    Evaluate each unknown cell to find the best target:
        max_probability ← 0
        best_index ← None
        for each cell in search_grid:
            if cell = "U":
                neighboring_hits ← Count of neighboring cells that are "H"
                neighboring_unknowns ← Count of neighboring cells that are "U"
                probability_simulation.input['hits'] ← neighboring_hits
                probability_simulation.input['unknowns'] ← neighboring_unknowns
                probability_simulation.compute()
                prob ← probability_simulation.output['probability']
                if prob > max_probability:
                    max_probability ← prob
                    best_index ← cell index
    return best_index (the most promising target cell index)
```

Minimax AI

The Minimax AI employs a more sophisticated decision-making process through the Minimax algorithm, which is typically used in two-player zero-sum games. Here's how it is applied in the game:

- **Initialization:** Like other AI modes, Minimax AI begins with ship placement and initialization of game parameters.
- **Decision Process:**
  - **Depth-Limited Search:** Minimax AI uses a depth-limited search to forecast potential future game states up to a certain depth. This helps in predicting the outcomes of various move sequences.
  - **Maximize and Minimize:** The AI alternates between maximizing its gain (trying to hit and sink ships) and minimizing the opponent's gain (avoiding moves that could lead to rapid retaliation).
  - **Evaluation Function:** The AI uses an evaluation function that considers the number of hits and potential hits, adjusted by strategic considerations like ship placement potential and avoiding known misses.
- **Strategic Play:**
  - After each move, the AI evaluates the game state by simulating potential future moves using the Minimax strategy, incorporating alpha-beta pruning to eliminate less optimal moves.
  - The AI chooses the move that leads to the best evaluated future game state, considering both its turn and the opponent's response.
- **Limitations:**
  - **Two Independent Grids:** The standard application of Minimax is less effective in environments like Battleship with two separate grids where the AI's moves on its grid do not directly affect the opponent's moves on theirs. This independence can lead to less impactful foresight and strategic depth.
  - **Computational Complexity:** Given the large number of possible moves and outcomes, Minimax can be computationally intensive, especially at higher depths, making it potentially slower than other simpler algorithms.
- **End of Game:** The game concludes once all the ships of one player are sunk. The effectiveness of the Minimax AI's strategy is then evaluated based on the game outcome.

Despite its potential computational drawbacks and the challenge posed by the game's structure, Minimax AI offers a robust, strategic depth that can challenge advanced players by simulating a more human-like opponent.

Algorithm 4 Minimax Algorithm with Alpha-Beta Pruning

```
1:  Function minmax_ai(depth)
2:  best_score ← $-\infty$
3:  best_move ← None
4:  available_moves ← get_available_moves()
5:  for each move in available_moves
6:      score ←simulate_and_evaluate(move, depth, False,
    $-\infty,\infty$)  ifscore>best_score
7&8:    best_score ← score
9:      best_move ← move
10: if best_move $\neq$ None
11:   num ← make_move(best_move)
12:   if player1_turn and player1.search[best_move] = ”H”
13:     in_sinking_mode ← True
14:     hit_stack.append(best_move)
15:   Return num
16: else
17:   Return basic_ai()
18: End Function
19: Function simulate_and_evaluate(move, depth, is_maximizing, alpha, beta)
20: original_search ← player.search[:]
21: original_player1_turn ← player1_turn
22: assumed_hit ← assume_hit(player, move, original_search)
23: simulated_search ← simulate_search(player, move, assumed_hit,
    original_search)
24: player1_turn ← not player1_turn
25: score ← minimax(depth - 1, not is_maximizing, alpha, beta,
    simulated_search)
26: player1_turn ← original_player1_turn
27: Return score
28: End Function
29: Function minimax(depth, is_maximizing, alpha, beta, search)
30: if depth = 0 or self.over
31:   Return evaluate(search)
32: if is_maximizing
33:   max_eval ← $-\infty$
34: available_moves ← get_available_moves()
35: for each move in available_moves
36:   eval ← minimax(depth - 1, False, alpha, beta, simulate_search(player,
    move))
37:   max_eval ← max(max_eval, eval)
38:   alpha ← max(alpha, eval)
39:   if beta $\leq$ alpha then break
40: Return max_eval
41: else
42:   min_eval ← $\infty$
43: available_moves ← get_available_moves()
44: for each move in available_moves
45:   eval ← minimax(depth - 1, True, alpha, beta, simulate_search(opponent,
    move))
46:   min_eval ← min(min_eval, eval)
47:   beta ← min(beta, eval)
48:   if beta $\leq$ alpha then break
49: Return min_eval
50: End Function
```

AI Comparisons

In the context of the Battleship game, the performance of each AI strategy varies significantly based on their inherent design and decision-making processes. Here, we compare the effectiveness of three AI strategies: Basic AI, Fuzzy Logic AI, and Minimax AI, based on the outcomes of a 10-game tournament between the AIs.

Performance Evaluation

- **Minimax AI:** Although theoretically robust due to its ability to simulate optimal game outcomes by thinking several moves ahead, Minimax AI faces practical challenges in Battleship. The separation of grids means its anticipatory capabilities are less impactful since the moves on one grid do not influence the state of the opponent's grid directly. This isolation reduces its strategic advantage, often making it less effective than the other two AIs in practice.
- **Basic AI:** The Basic AI employs a straightforward heuristic-based approach, primarily focusing on random moves with a simple follow-up strategy on hits. It tends to perform better than Minimax because it reacts faster without the computational overhead of deep lookahead. However, its lack of a sophisticated strategy means it can miss opportunities for more strategic play, making it less effective against more nuanced strategies like that of the Fuzzy Logic AI.
- **Fuzzy Logic AI:** Fuzzy Logic AI has shown the best performance among the three. It balances between the computational simplicity of Basic AI and the strategic depth akin to Minimax. By using fuzzy rules to determine the probability of hitting a ship based on nearby hits and unknown squares, it adapts dynamically to the changing conditions of the game board. This adaptability allows it to make smarter decisions without the need for extensive computation, leading to more effective gameplay.

Graphical Data

Below are the comparative performance graphs from the tournament, displayed side by side for a direct comparison:

As shown in Fig. 5, Fig. 6, and Fig. 7, the number of wins across 10 games demonstrates the superior performance of the Fuzzy Logic AI when compared to both the Basic and Minimax AIs.

The "Distribution of Shots per Game" graph shows the total number of shots fired in each game during the tournament. The X-axis represents the number of shots, ranging from the minimum possible (17) to the maximum encountered (200). The Y-axis indicates the frequency of games that ended with that specific number of shots. This distribution helps in understanding the efficiency and dynamics of gameplay, reflecting how quickly or slowly games tend to conclude based on the strategies employed by different AIs.

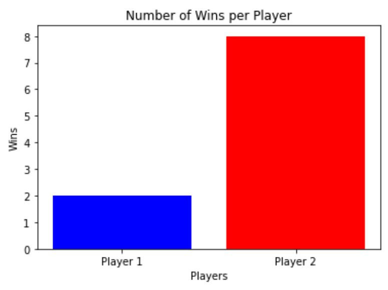

Figure 5: Basic vs Fuzzy AI

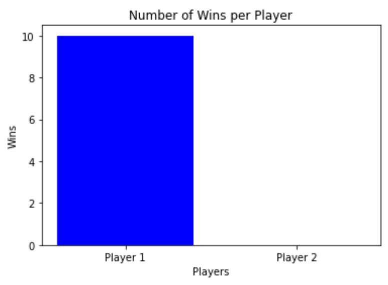

Figure 6: Basic vs Minimax AI

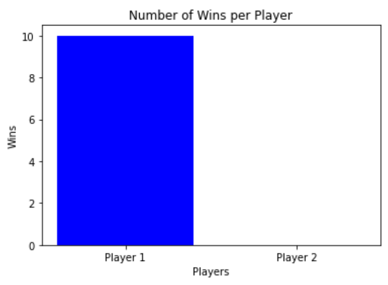

Figure 7: Fuzzy vs Minimax AI

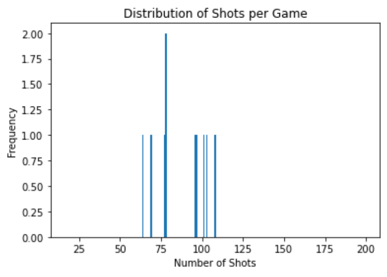

Figure 8: Shot Distribution for Basic AI vs Fuzzy Logic AI

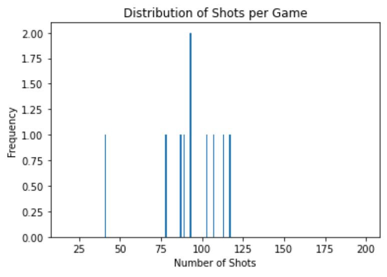

Figure 9: Shot Distribution for Basic AI vs Minimax AI

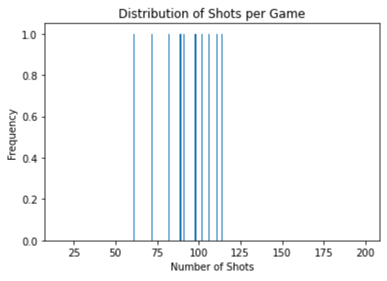

Figure 10: Shot Distribution for Fuzzy Logic AI vs Minimax AI

Fig. 8, Fig. 9, and Fig. 10 detail the distribution of shots per game for each AI matchup. These graphs provide insights into the efficiency and tactics of each AI, showcasing how each strategy influences the length and outcome of the games.

Contribution

1907033: Minimax Algorithm
1907049: Genetic Algorithm
Both: Heuristic Search, Fuzzy Logic, UI Design, Tournament.

Conclusion

Based on multiple gameplay sessions and evaluations, including a 10-game tournament, the Fuzzy Logic AI consistently outperforms both the Basic and Minimax AIs. Its ability to adapt its strategy dynamically based on the game state provides a significant advantage, allowing it to respond effectively to different situations without the computational burden seen in Minimax AI.For players seeking a challenging yet computationally efficient opponent, Fuzzy Logic AI is recommended. It strikes an optimal balance between strategic complexity and computational efficiency, making it the most suitable AI for competitive play in Battleship.