CSE 4110: Artificial Intelligence Laboratory

Magnet Wars : an AI integrated board game

By

Mahin Rashid Chowdhury
Roll: 1907021

MD. Rabby
Roll: 1907061

Department of Computer Science and Engineering

Khulna University of Engineering & Technology

Khulna 9203, Bangladesh

Magnet Wars : an AI integrated board game

By

Mahin Rashid Chowdhury

Roll: 1907021

MD.Rabby

Roll: 1907061

Submitted To:

Md. Shahidul Salim

Lecturer

Department of Computer Science and Engineering _____

Khulna University of Engineering & Technology Signature

Most. Kaniz Fatema Isha

Lecturer

Department of Computer Science and Engineering _____

Khulna University of Engineering & Technology Signature

" Department of Computer Science and Engineering

Khulna University of Engineering & Technology

Khulna 9203, Bangladesh

1 Objectives

1. Gain a deep understanding of game theory principles and how they can be applied to AI algorithms in gaming.

2. Study various AI techniques such as Minimax, Alpha-Beta Pruning, Fuzzy logic,Genetic Algorithm.

3. Implementation of a game integrating AI in python

2 Introduction

The field of artificial intelligence (AI) has significantly advanced in recent years, leading to the creation of sophisticated AI systems capable of performing complex tasks. One of the most engaging and challenging applications of AI is in the realm of games, where AI agents are developed to compete against human players or other AI agents. This project aims to explore the development, implementation, and analysis of AI agents designed to play strategic games, providing us with hands-on experience in AI methodologies and game theory.In recent years, AI agents have achieved remarkable success in mastering complex games, such as defeating world champions in chess, Go, and poker. These achievements highlight the power and potential of AI, driving further interest and research in the field. However, developing AI agents capable of excelling in strategic games remains a challenging task due to the vast number of possible moves, the need for long-term planning, and the unpredictability of human opponents.

3 Theory

To implement this game's AI, AI algorithms such as Minimax,Alpha-Beta pruning,Fuzzy logic is used.

3.1 Minimax algorithm

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that

opponent is also playing optimally.

- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.
- In this algorithm two players play the game, one is called MAX and other is called MIN. Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.

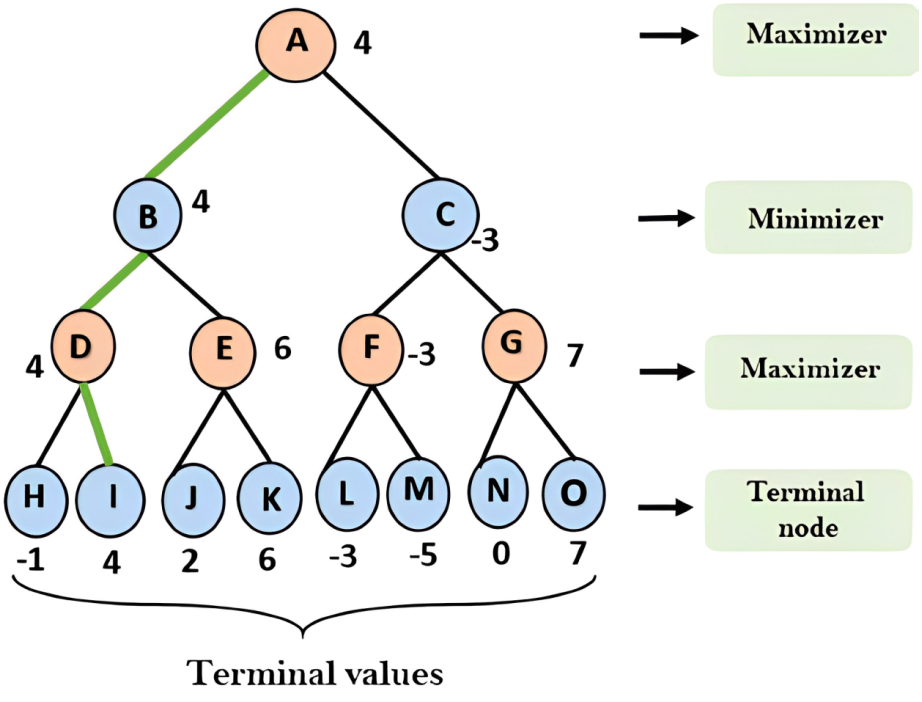

Figure 1: Simulation of Miinimax algorithm

3.2 Alpha-Beta Pruning

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm

- – **Alpha** (α): The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is $-\infty$.
  – **Beta** (β): The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is $+\infty$.

- The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

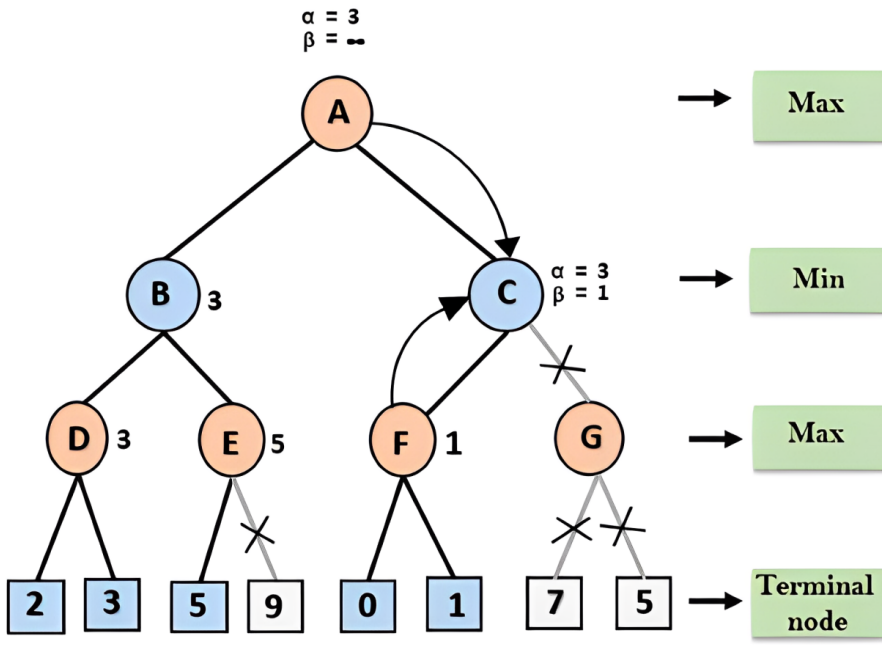

Figure 2: Simulation of Alpha-Beta Pruning algorithm

3.3 Mamdani Fuzzy Inference

Mamdani fuzzy inference, also known as the Mamdani-Assilian method, is one of the most commonly used approaches for implementing fuzzy logic. It was first introduced by Ebrahim Mamdani and Sedrak Assilian in 1975 to control a steam engine and boiler combination. This method is particularly popular due to its intuitive appeal and simplicity in capturing expert knowledge.

3.3.1 Key Components of Mamdani Fuzzy Inference

Mamdani fuzzy inference involves the following key components:

- **Fuzzification:** The first step is to convert crisp input values into fuzzy sets. This is done using membership functions that quantify the degree to which an input belongs to a fuzzy set. For example, if the input is temperature, the fuzzy sets might be "cold," "warm," and "hot," each with its own membership function.

- **Rule Base:** A collection of fuzzy rules forms the rule base. These rules are typically expressed in the form of IF-THEN statements. For example:

  > IF temperature is cold THEN fan speed is low. IF temperature is hot THEN fan speed is high.

  Each rule consists of an antecedent (the IF part) and a consequent (the THEN part), both of which are fuzzy statements.

- **Inference:** In the inference step, the fuzzy rules are applied to the fuzzified inputs to generate fuzzy outputs. This involves determining the degree of match between the fuzzified inputs and the antecedents of each rule, and then applying these matching degrees to the consequents.

- **Aggregation:** Since there are typically multiple rules, the outputs of these rules need to be combined or aggregated into a single fuzzy set. This step involves taking the union of the fuzzy sets generated by each rule.

- **Defuzzification:** The final step is to convert the aggregated fuzzy set back into a crisp output value. This process is known as defuzzification. Common methods for

defuzzification include the centroid method, which computes the center of gravity of the aggregated fuzzy set, and the maximum method, which selects the point with the highest membership value.

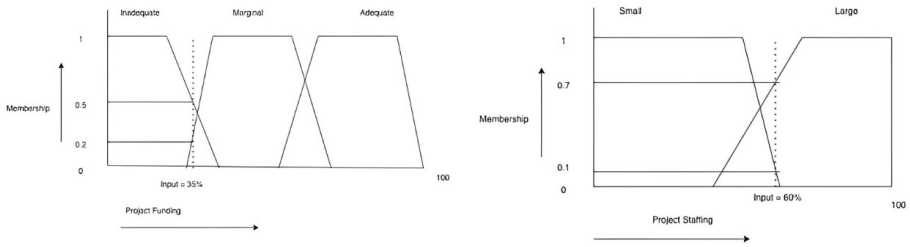

Figure 3: Fuzzy Inference

4 Game Design

4.1 Game UI

There is three options for a player to choose at first, he can choose to play as player vs player or player(White) vs AI or player(Black) vs AI.

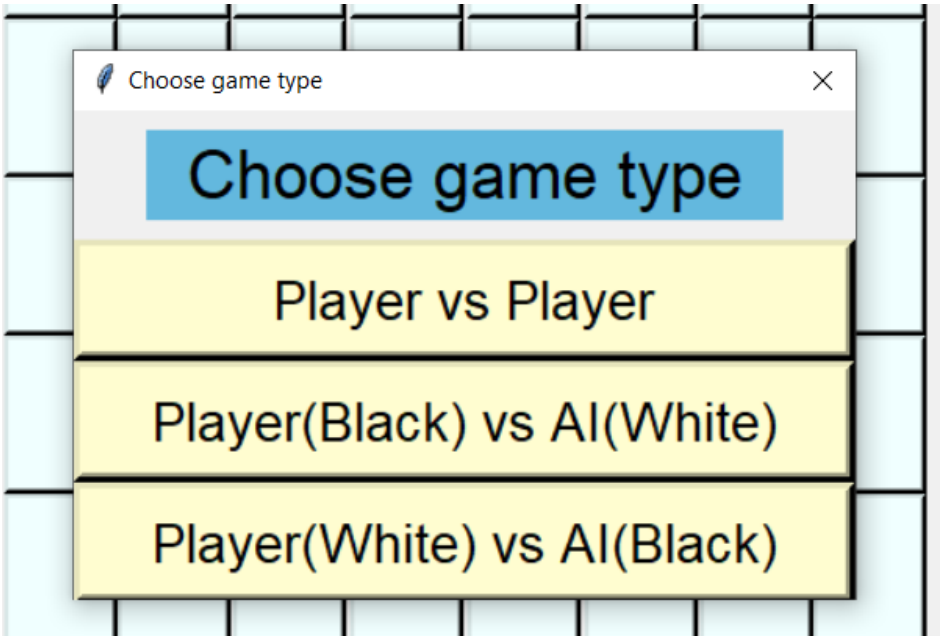

Figure 4: Choosing game type

The game contains a 8X8 grid , where a player can place his magnets to those grids.

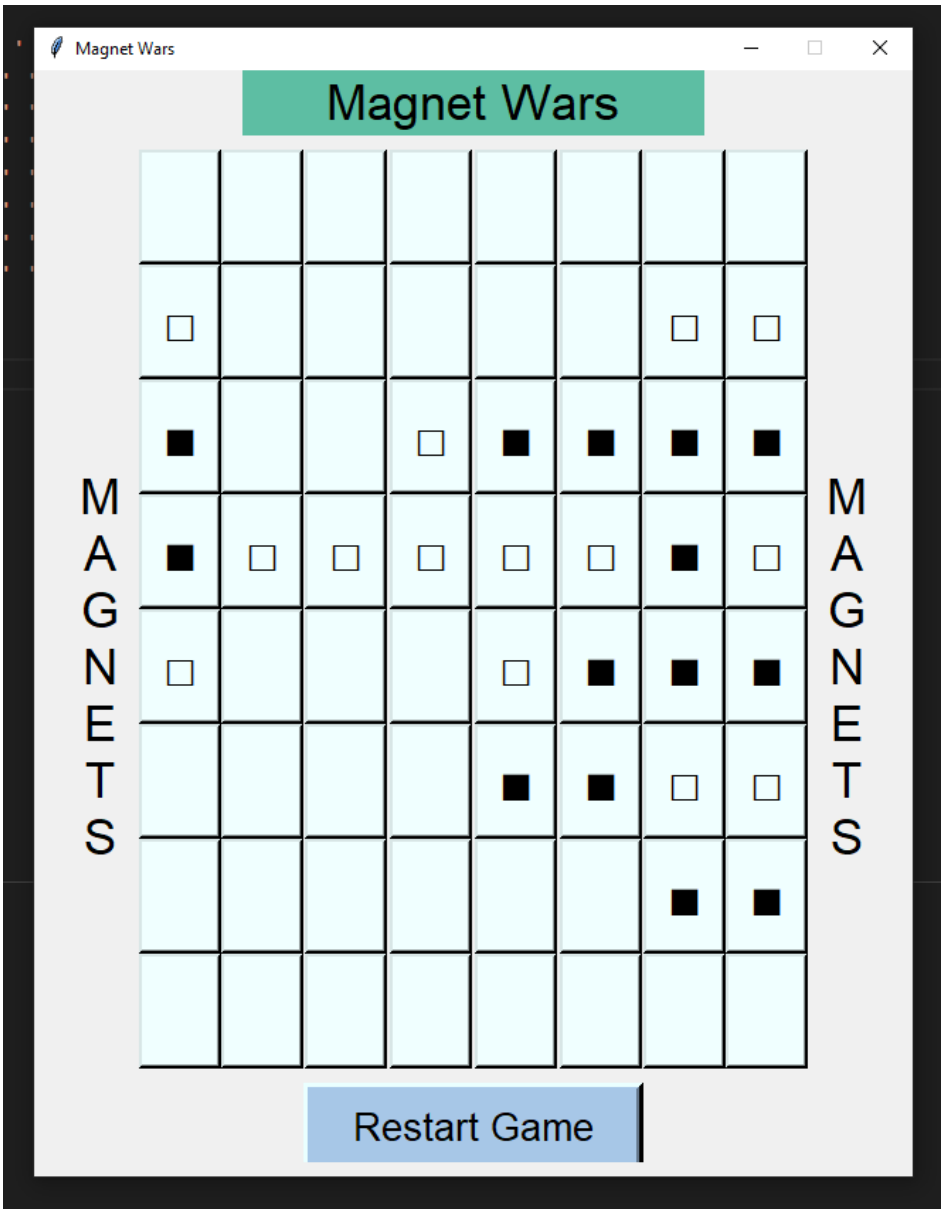

Figure 5: User Interface of game-grid

After the game finishes, a pop up box which contains the game verdict and point of the losing player using fuzzy inference will be showed.

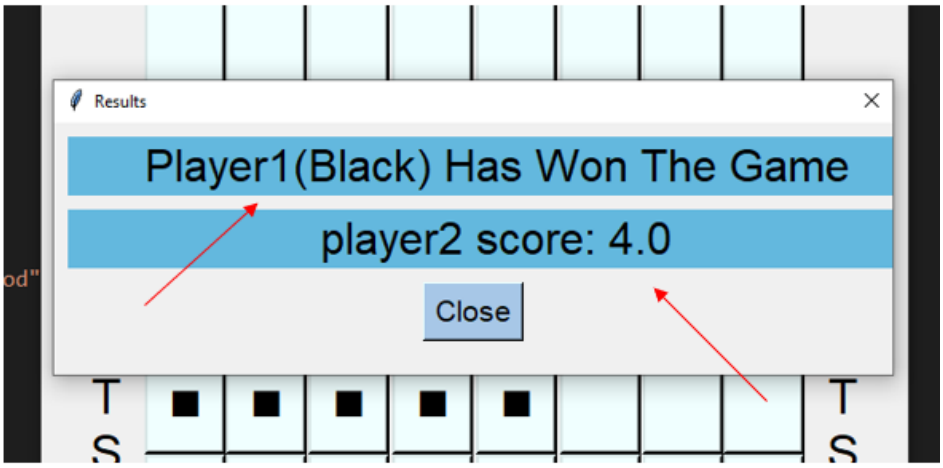

Figure 6: User Interface of game-verdict

4.2 Game Rules

This game rule is similar to the traditional 4-connect game. Magnet war is a 2-player competitive game where each player aims to build a "bridge" of 5 magnetic bricks within an 8x8 cave. One player's bricks are represented by ■, and the other player's by □. The rules are as follows:

1. The game board is initially empty.
2. Player ■ always starts, followed by player □, and they continue to alternate turns.
3. Bricks can only be placed on empty cells if they are directly next to the left or right wall, or adjacent to another brick (of any type) on the left or right.
4. The game is won when a player aligns 5 consecutive bricks in a row, column, or diagonal.
5. If the board is full and no player has won, the game ends in a tie.

Here some of the game-winning scenario is showed :

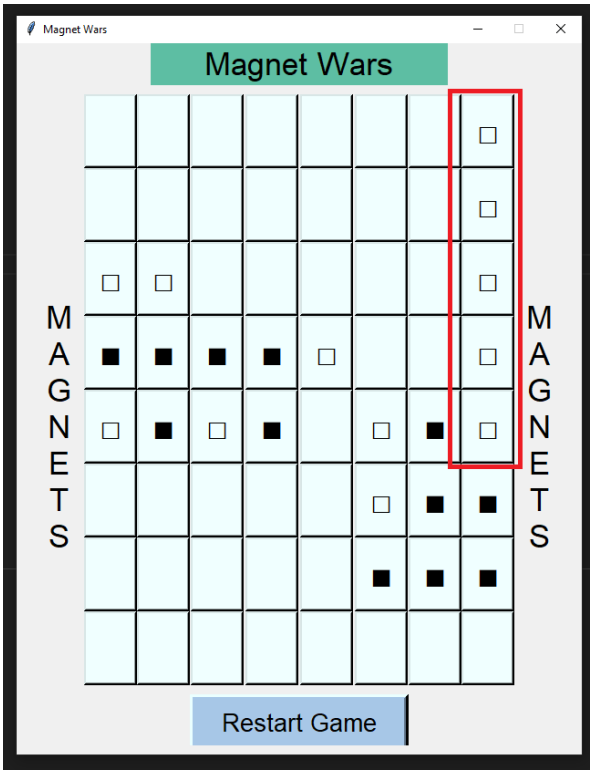

Figure 7: White Player could connect 5 magnets vertically.

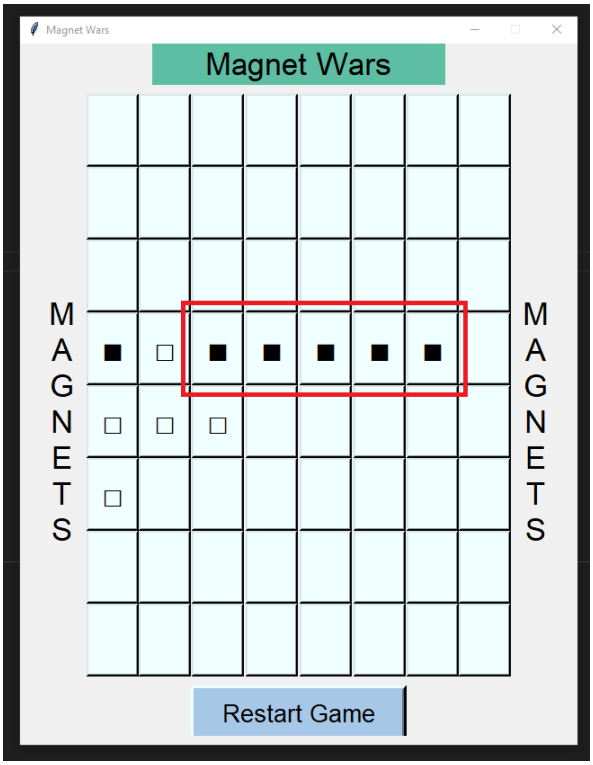

Figure 8: Black Player could connect 5 magnets horizontally.

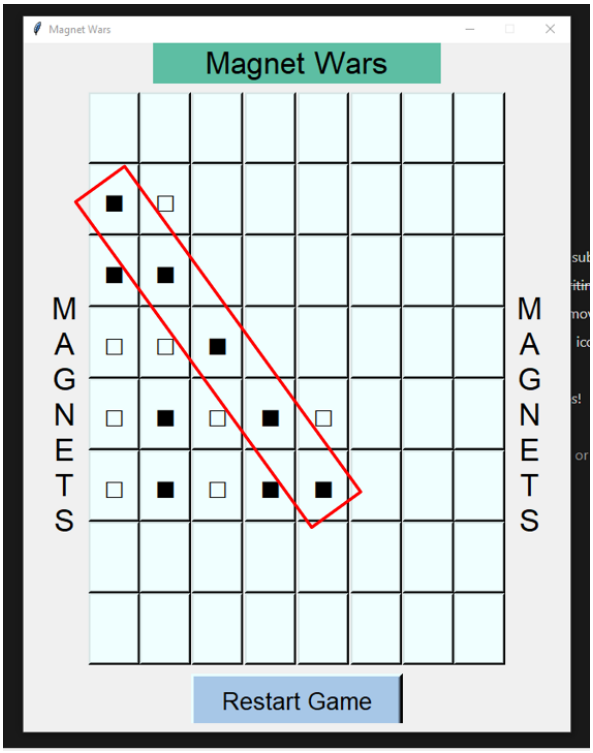

Figure 9: Black Player could connect 5 magnets diagonally.

4.3 Alpha-Beta Pruning in Minimax Algorithm for AI Decision Making

In this section, we describe the implementation of the Minimax algorithm with Alpha-Beta pruning used for determining the AI's move in the game. This method optimizes the decision-making process by eliminating branches that do not need to be explored, thus reducing the computational complexity.

4.3.1 Minimax Algorithm with Alpha-Beta Pruning

The Minimax algorithm is a recursive method used for decision making in game theory and artificial intelligence. It simulates all possible moves to a certain depth, evaluating each move and selecting the best one based on a heuristic evaluation function. Alpha-Beta pruning enhances the efficiency of this algorithm by pruning branches that cannot possibly influence the final decision.

4.3.2 Algorithm Description

The algorithm can be described by the following function:

```
def mini_max(current_board_position, depth_limit: int,
                max_turn: bool, depth: int, alpha, beta):
    if max_turn:  # If it's the max turn then the turn is 1
        turn = 1
    else:  # If it's the min turn then the turn is -1
        turn = -1

    if depth == depth_limit:  # If the depth limit is reached
        then return the evaluation of the board and the board
        return evaluate(current_board_position), current_board_position

    elif is_over(current_board_position, turn):
        return evaluate(current_board_position), current_board_position

    if max_turn:
        max_eval = -np.inf  # Max evaluation of the board
        best_move = [(0, 0)]

        for move in possible_moves(current_board_position):
            current_board_position[move[0], move[1]] = 1
            evaluation, tmp = mini_max(current_board_position,
                depth_limit, False, (depth + 1), alpha, beta)
            current_board_position[move[0], move[1]] = 0
            max_eval = max(max_eval, evaluation)  # Max evaluation

            if evaluation == max_eval:
                best_move = move

            alpha = max(alpha, max_eval)  # Alpha Beta Pruning
```

```
            if alpha >= beta:  # Pruning
                break

        return max_eval, best_move  # return the best move and
            the evaluation of the board
    else:
        min_eval = np.inf  # Min evaluation of the board
        best_move = [(0, 0)]  # dummy move

        for move in possible_moves(current_board_position):  # For all
                                                              the possible moves
            current_board_position[move[0], move[1]] = -1  # simulate the move
            evaluation, tmp = mini_max(current_board_position,
                                       depth_limit, True, (depth + 1), alpha, beta)
            current_board_position[move[0], move[1]] = 0  # undo the simulation
            min_eval = min(min_eval, evaluation)  # Min evaluation of the board

            if evaluation == min_eval:
                best_move = move

            beta = min(beta, min_eval)

            if beta <= alpha:
                break

        return min_eval, best_move
```

4.3.3 Algorithm Analysis

The `mini_max` function begins by setting the `turn` variable based on whether it is the maximizer's turn (represented by 1) or the minimizer's turn (represented by -1). The

function then checks if the current depth has reached the depth limit or if the game is over at the current board position. If either condition is met, it returns the evaluation of the board.

For the maximizer's turn, the function initializes `max_eval` to negative infinity and iterates through all possible moves. It simulates each move, calls `mini_max` recursively for the minimizer's turn, and then undoes the move. The `max_eval` is updated to the maximum of its current value and the evaluation returned by the recursive call. The `alpha` value is updated for Alpha-Beta pruning, and if `alpha` is greater than or equal to `beta`, the loop breaks to prune the remaining branches.

Similarly, for the minimizer's turn, the function initializes `min_eval` to positive infinity and follows the same process, updating `min_eval` and `beta`, and performing pruning when necessary.

This implementation ensures an optimal move selection while significantly reducing the number of nodes evaluated in the search tree, thereby enhancing the efficiency of the AI decision-making process.

4.4 Fuzzy Inference for score assesment

In this section, we describe the implementation of fuzzy logic to evaluate the player's performance in terms of their total moves and score. The fuzzy logic system assigns a final score between 1 and 100 based on the inputs.

4.4.1 Fuzzification

The fuzzification process involves defining membership functions for the input variables: *tot_moves* and *score*. The membership functions are defined as follows:

- **Total Moves:**
  - 0-10 : Bad
  - 8-20 : Average
  - 15-32 : Good
- **Score:**
  - 0-10 : Bad

- 10-50 : Average
- 40-100 : Good

4.4.2 Implementation

The following Python function implements the fuzzification and defuzzification logic:

```
Function fuzzy(tot_moves, score):
    Initialize moves membership values (bad, avg, good) to 0.0
    Normalize score by dividing by 5

    Determine membership values for tot_moves:
        If tot_moves <= 8:
            moves["bad"] = 1
        Else If 8 < tot_moves <= 10:
            moves["bad"] = (10 - tot_moves) / 2
            moves["avg"] = (tot_moves - 8) / 2
        Else If 10 < tot_moves <= 15:
            moves["avg"] = 1
        Else If 15 < tot_moves <= 20:
            moves["avg"] = (20 - tot_moves) / 5
            moves["good"] = (tot_moves - 15) / 5
        Else:
            moves["good"] = 1

    Determine membership values for score:
        If score <= 10:
            scores["bad"] = 1
        Else If 10 < score <= 20:
            scores["bad"] = (20 - score) / 10
            scores["avg"] = (score - 10) / 10
        Else If 20 < score <= 40:
            scores["avg"] = 1
```

```
    Else If 40 < score <= 60:
        scores["avg"] = (60 - score) / 20
        scores["good"] = (score - 40) / 20
    Else:
        scores["good"] = 1

Calculate fuzzy rules for bad, avg, fair, good:
    bad = max(min(moves["bad"], scores["bad"]),
        min(moves["bad"],scores["avg"]),min(moves["avg"],scores["bad"]))
    avg = max(min(moves["avg"],scores["avg"]),
        min(moves["bad"],scores["good"]),min(moves["good"],scores["bad"]))
    fair = max(min(moves["avg"], scores["good"]),
        min(moves["good"], scores["avg"]))
    good = min(moves["good"], scores["good"])

Initialize up and down to 0
For i from 0 to 100:
    If i <= 10:
        up += (bad * i)
        down += bad
    Else If 10 < i <= 50:
        up += (avg * i)
        down += avg
    Else If 50 < i <= 70:
        up += (fair * i)
        down += fair
    Else:
        up += (good * i)
        down += good

Calculate final_score = up // down
Return final_score
```

4.4.3 Analysis

The function begins by normalizing the input *score* and initializes dictionaries to hold the membership values for both *tot_moves* and *score*. The membership values are computed based on the predefined ranges for each fuzzy set.

Next, the function calculates the degrees of membership for each fuzzy rule and computes the combined membership values for the output fuzzy sets (bad, avg, fair, good).

Finally, the function uses these combined membership values to compute a weighted average, which represents the defuzzified final score. This score is an integer between 1 and 100, providing a comprehensive evaluation of the player's performance.

5 Result Analysis

In this section, we evaluate the performance of our AI using the Minimax algorithm with Alpha-Beta pruning. Our goal is to determine how effectively the AI can play the game and make strategic decisions.

5.1 Performance Metrics

- **Depth of Search**: A higher depth allows the AI to look further ahead, improving decision-making but increasing computational complexity. We tested various depths to balance performance and feasibility.
- **Win Rate**: The win rate against different levels of opponents (random moves, basic heuristics, and other AI) indicates the AI's effectiveness.
- **Average Decision Time**: The time taken by the AI to make decisions ensures it operates within acceptable limits for real-time play.

5.2 Experiment Setup

We conducted experiments with:

- **Board Size**: 8x8 grid

- **Depth Limits**: 3, 5, 7, and 9
- **Opponents**: Random Move Generator, Basic Heuristic AI, Advanced Heuristic AI

Each configuration was tested over 10 games.

5.3 Results

- **Depth of Search**:
  - Depth 3: Quick decisions but missed deeper strategies.
  - Depths 5 and 7: Balanced performance with improved win rates and reasonable decision times.
  - Depth 9: Slightly better win rates but impractical for real-time play due to long decision times.
- **Win Rate**:
  - Against random move opponents: 75% win rate across all depths.

6 Discussion

The game "Magnetic War" is played on an 8x8 board where players alternate placing magnetic bricks (■ and □). The objective is to align 5 bricks horizontally, vertically, or diagonally. Bricks can only be placed next to walls or existing bricks due to magnetic constraints.

The AI in this game employs the Minimax algorithm with Alpha-Beta pruning, a strategic decision-making approach that explores possible moves to maximize its chances of winning while minimizing the opponent's opportunities. This method ensures efficient computation by disregarding branches of the game tree unlikely to lead to a better outcome.

Additionally, fuzzy evaluation enhances the AI's decision-making process. By assessing game states based on total moves and current score, the AI gains a nuanced understanding of the game's progress beyond simple rule-based evaluations. This fuzzy logic allows for adaptive and context-aware gameplay decisions. Further improvement can be made to this game's AI by introducing algorithms such genetic algorithm or training a neural network.

7 Conclusion

In conclusion, the AI lab project on "Magnetic War" successfully integrates advanced algorithms and evaluation techniques to create a robust gameplay experience. By combining Minimax with Alpha-Beta pruning for strategic depth and fuzzy evaluation for adaptive decision-making, the project showcases effective AI-driven gameplay in a challenging adversary environment. Future directions may involve refining computational efficiency, exploring alternative algorithms, and expanding the game's complexity to further test the AI's capabilities in strategic decision-making and adaptive gameplay.