T.C.

MARMARA UNIVERSITY

FACULTY of ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

Artificial Intelligence CSE 4082

Project 3

Ömercan Sabun – 150119555

Senanur Yılmaz – 150119801

SOS Game

1. Introduction

For this project, we present an adaptation of SOS, incorporating specific rule modifications and enhancements. The key features of this modified version include:

- **5x5 Board with Initial 'S' Symbols:**
  - The game is played on a square grid with a size of 5x5.
  - Initial 'S' symbols are strategically placed in the corners of the board, shaping the starting configuration.
- **Restricted Turn Alternation upon Creating 'SOS':**
  - Unlike the traditional SOS game, in our modified version, once a player successfully creates an 'SOS,' their turn is restricted, and the next move alternates to the opponent.

These alterations add a layer of strategic complexity to the game, requiring players to balance the creation of 'SOS' sequences with their turn management. The objective remains consistent: score as many 'SOS' sequences as possible, with the player achieving the highest score declared as the winner. In the event of a fully filled board, the player with the most 'SOS' sequences secures victory. If both players have an equal number of 'SOS' sequences, the game results in a draw.

This project not only implements the modified SOS game but also introduces an artificial intelligence (AI) player capable of engaging in dynamic gameplay against human opponents or other AI players. The AI player utilizes the minimax algorithm with alpha-beta pruning and incorporates heuristic evaluation methods to make intelligent moves, enhancing the overall gaming experience.

2. Implementation Overview

This report presents an analysis of the "SOS" game based on the provided Python code. The game involves two players taking turns to fill a game board, aiming to create SOS sequences, which are strings of the same symbols (S or O).

1. **Board Class:**
   - **Properties:**
     - **size**: Determines the size of the board (default is 5x5).
     - **board**: Represents the game board as a matrix.
   - **Methods:**
     - **initialize_board**: Sets the board to its initial state, starting with "S" symbols at the corners.
     - **display_board**: Prints the board to the screen.
     - **is_valid_move**: Checks if a specific move is valid.
     - **make_move**: Executes a specific move on the board.
     - **is_full**: Checks if the board is full.
2. **Player Class:**

- **Properties:**
  - **name**: Represents the player's name.
  - **score**: Tracks the player's score.
- **Methods:**
  - **make_move**: Abstract method containing the logic for making a move. Subclasses implement this method.

3. **Human Player Class:**
   - **Methods:**
     - **make_move**: Takes input from the human player, ensuring reliable input handling for potential errors.
4. **AI Player Class:**
   - **Properties:**
     - **depth**: Depth level used in the Minimax algorithm.
     - **heuristic**: Intuitive function used to evaluate game states.
   - **Methods:**
     - **make_move**: Selects the best move for a specific player using the Minimax algorithm.
     - **h1** and **h2**: Two different intuitive functions used to evaluate the game state.
     - **minimax**: Implements the Minimax algorithm to choose the best move.
5. **Game Playing and Control (play_game, main):**
   - **play_game Function:**
     - Initiates the game and manages the game loop.
     - Displays the state of the board after each move and updates player scores.
     - Declares the winner or a tie when the game ends.
   - **main Function:**
     - Offers the user the option to choose the game mode (Human vs. Human, Human vs. AI, AI vs. AI).
     - Starts the game in the selected mode.
     - Asks the user if they want to play again after the game ends.
6. **General Overview:**
   - The code adopts a clear and organized object-oriented programming (OOP) approach with well-defined classes and methods.
   - The Minimax algorithm and intuitive functions are employed by the artificial intelligence to select the optimal move.
   - The code's usage and flow are designed in adherence to the fundamental rules of the game.
   - Game modes provide users with various gameplay experiences through different options.
   - The code uses a clear structure for readability and maintainability.
7. **Improvement Suggestions:**
   - Considering the potential decrease in performance, particularly with increased board size and high depth in the Minimax algorithm, optimizing the algorithm might be beneficial.

- Adding an option to change the size of the game board could enhance user experience, allowing them to play on larger or smaller boards.
- Improving the AI's strategy by enhancing the complexity of the intuitive functions could lead to a more challenging opponent.

3. Human vs Human Gameplay

In a Human vs Human gameplay scenario, two participants take turns playing the modified SOS game. Each player is prompted to input their moves, and the game state is displayed after each turn. Below is an explanation of the gameplay flow, input/output mechanisms, and sample interactions:

3.1 Gameplay Flow:

1. **Initialization:**
    - The game starts with the creation of a 5x5 game board, with initial 'S' symbols strategically placed in the corners.
    - Players are prompted to enter their names.
2. **Turn Alternation:**
    - Players take turns making moves.
    - The player with the first turn is prompted to choose their symbol ('S' or 'O').
    - Subsequently, each player is prompted to enter their move (row and column).
3. **Move Validation:**
    - The game checks the validity of each move.
    - If the chosen position is already occupied or out of bounds, players are asked to input a new move until a valid one is entered.
4. **Score Tracking:**
    - After each move, the game checks if a new 'SOS' sequence is created.
    - If yes, the scoring player gains points equal to the number of newly formed 'SOS' sequences.
    - The scoring player gets an additional turn.
5. **Game Termination:**
    - The game continues until the board is fully filled or until players decide to end the game.
    - The player with the highest score at the end is declared the winner.

3.2 Input Mechanisms:

- **Symbol Selection:**
    - Players are prompted to choose their symbol ('S' or 'O') at the beginning of the game.
- **Move Input:**
    - Players input their moves by entering the row and column numbers when prompted.

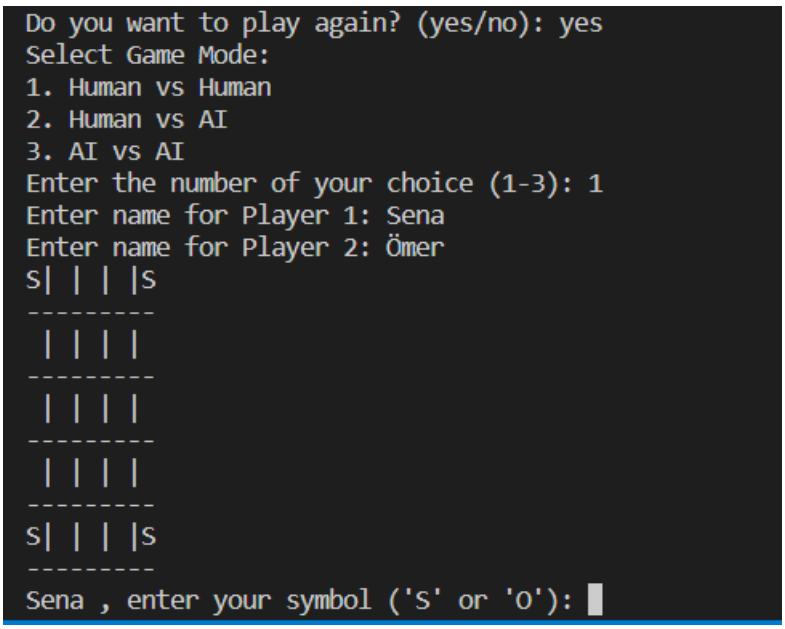
3.3 Output Mechanisms:

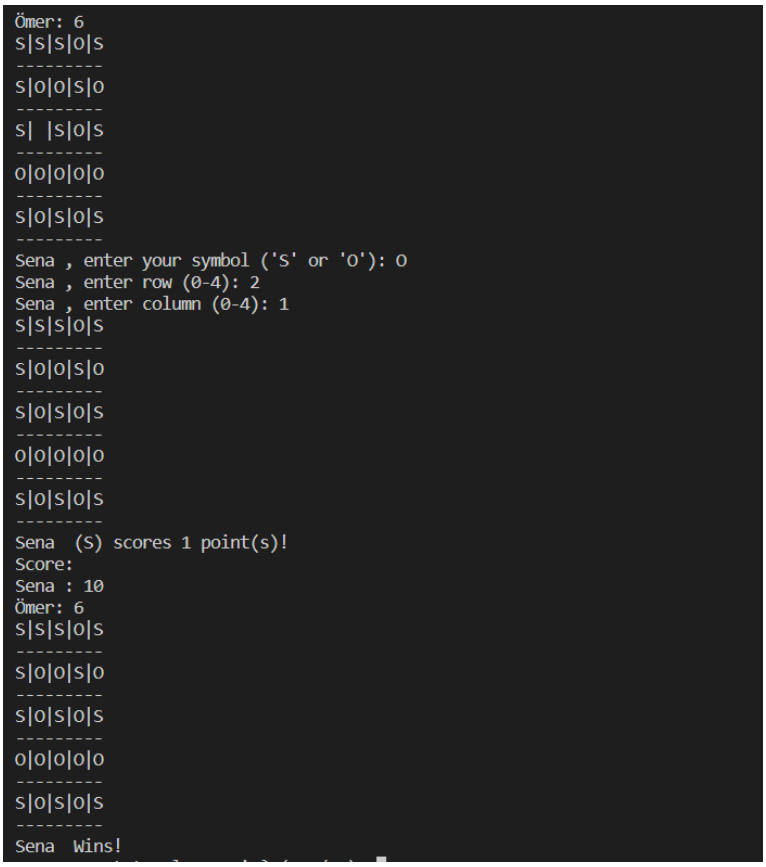
- **Board Display:**

    - After each move, the current state of the game board is displayed, showing the positions of 'S' and 'O' symbols.
- **Score Display:**
    - The scores of both players are shown after each move.
- **Game Outcome:**
    - The final result of the game (winner or draw) is displayed at the end of the gameplay.

Sample Interactions:

Sample 1:





4. Human vs AI Gameplay

4.1 Interaction

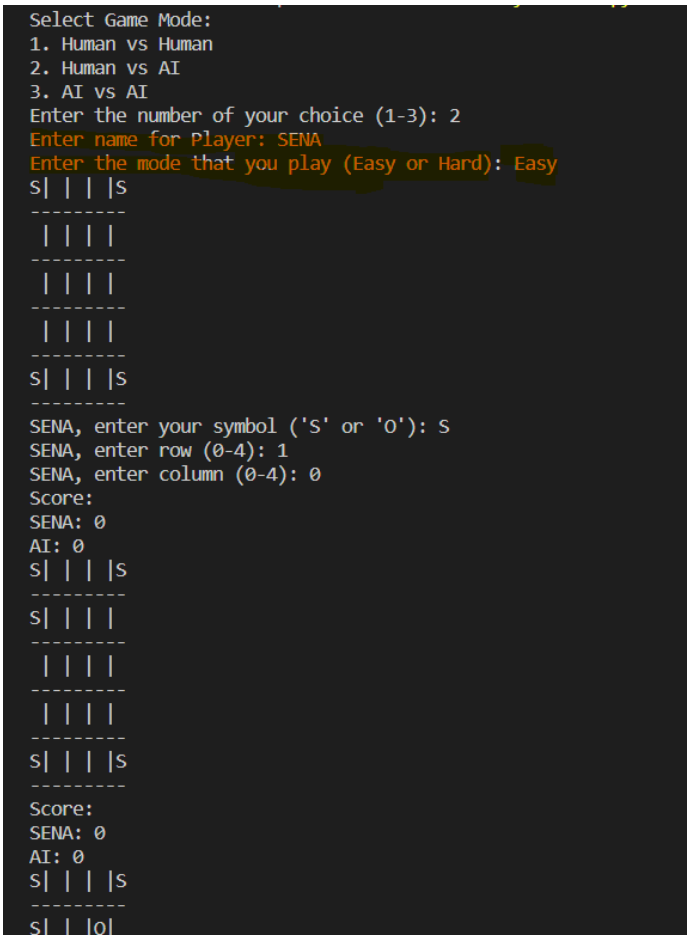
In the Human vs AI scenario, the **HumanPlayer** interacts with the AI player. The human player provides input for their moves, and the AI player utilizes the minimax algorithm to make strategic moves.

4.2 AI Decision-making

The AI player selects moves by evaluating the current state of the board using the specified heuristic. The minimax algorithm explores potential moves up to a specified depth, considering the possible outcomes of each move.

Sample Interactions:

Sample 1:


Select Game Mode:
1. Human vs Human
2. Human vs AI
3. AI vs AI
Enter the number of your choice (1-3): 2
Enter name for Player: SENA
Enter the mode that you play (Easy or Hard): Easy
S| | | |S

| | | |

| | | |

| | | |

S| | | |S

SENA, enter your symbol ('S' or 'O'): S
SENA, enter row (0-4): 1
SENA, enter column (0-4): 0
Score:
SENA: 0
AI: 0
S| | | |S

S| | | |

| | | |

| | | |

S| | | |S

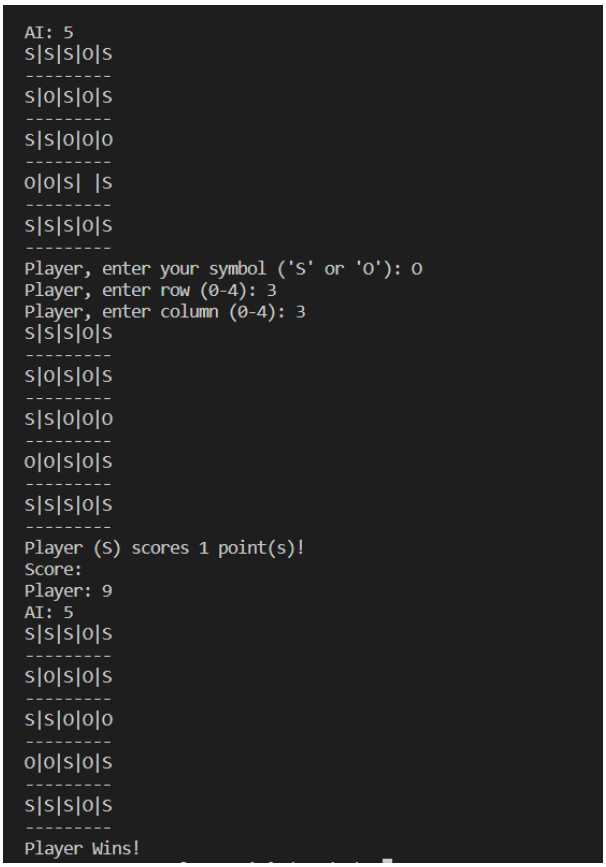
Score:
SENA: 0
AI: 0
S| | | |S

S| | |O|

```
S|S|O|S|S
---------
S|S|S|O|S
---------
S|O|S|S|
---------
S|O|S|O|S
---------
S|O|S|O|S
---------
SENA, enter your symbol ('S' or 'O'): O
SENA, enter row (0-4): 2
SENA, enter column (0-4): 4
S|S|O|S|S
---------
S|S|S|O|S
---------
S|O|S|S|O
---------
S|O|S|O|S
---------
S|O|S|O|S
---------
SENA (S) scores 1 point(s)!
Score:
SENA: 10
AI: 5
S|S|O|S|S
---------
S|S|S|O|S
---------
S|O|S|S|O
---------
S|O|S|O|S
---------
S|O|S|O|S
---------
SENA Wins!
```

Sample 2:

```
Do you want to play again? (yes/no): yes
Select Game Mode:
1. Human vs Human
2. Human vs AI
3. AI vs AI
Enter the number of your choice (1-3): 2
Enter name for Player: SENA
Enter the mode that you play (Easy or Hard): Hard
S| | | |S
---------
 | | | |
---------
 | | | |
---------
 | | | |
---------
S| | | |S
---------
Player, enter your symbol ('S' or 'O'): S
Player, enter row (0-4): 1
Player, enter column (0-4): 0
Score:
Player: 0
AI: 0
S| | | |S
---------
S| | | |
---------
 | | | |
---------
 | | | |
```



5. AI vs AI Gameplay

5.1 Interaction

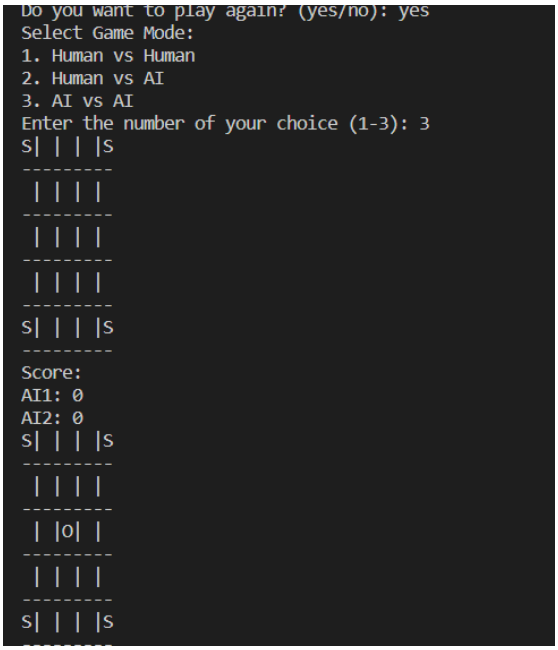
In the AI vs AI scenario, two instances of **AIPlayer** interact. Each AI player uses the minimax algorithm to decide its moves based on the specified heuristic.

5.2 Decision-making

Both AI players evaluate the board state independently, and their decisions are based on the heuristic specified. The minimax algorithm with alpha-beta pruning is used to optimize decision-making while exploring the game tree.

Sample Interactions:

Sample 1:


Do you want to play again? (yes/no): yes
Select Game Mode:
1. Human vs Human
2. Human vs AI
3. AI vs AI
Enter the number of your choice (1-3): 3
S| | | |S

| | | |

| | | |

| | | |

S| | | |S

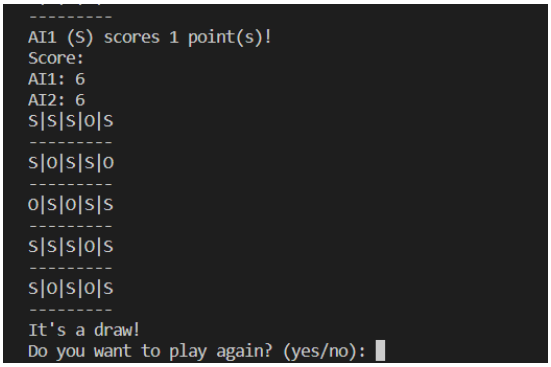
Score:
AI1: 0
AI2: 0
S| | | |S

| | | |

| |O| |

| | | |

S| | | |S



AI1 (S) scores 1 point(s)!
Score:
AI1: 6
AI2: 6
S|S|S|O|S

S|O|S|S|O

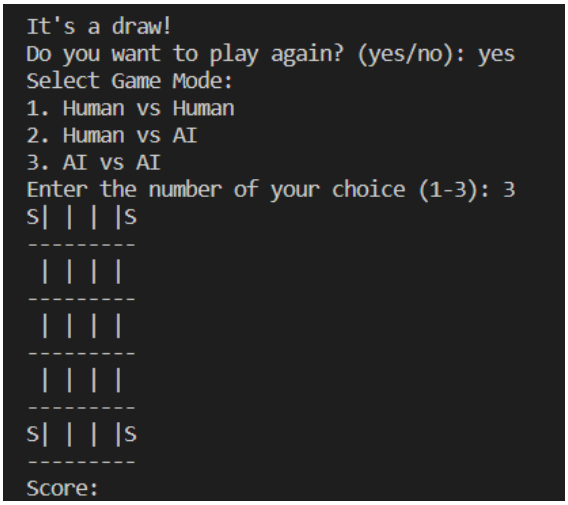
O|S|O|S|S

S|S|S|O|S

S|O|S|O|S

It's a draw!
Do you want to play again? (yes/no):

Sample 2:


It's a draw!
Do you want to play again? (yes/no): yes
Select Game Mode:
1. Human vs Human
2. Human vs AI
3. AI vs AI
Enter the number of your choice (1-3): 3
S| | | |S

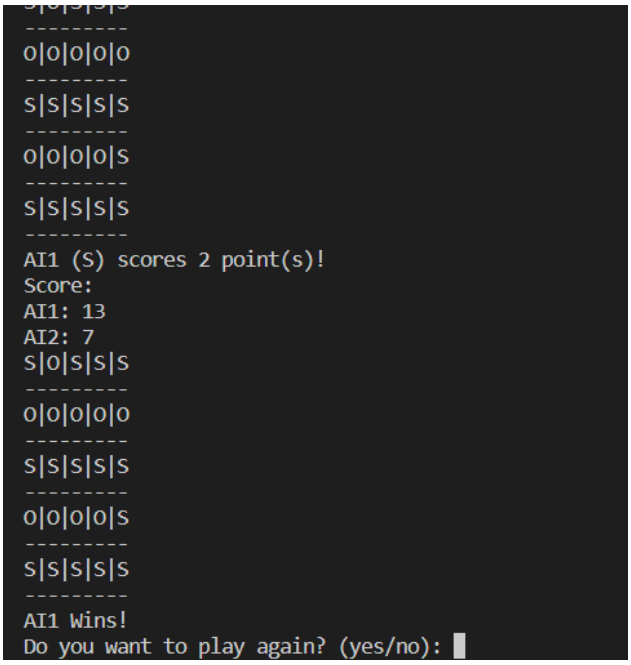
| | | |

| | | |

| | | |

S| | | |S

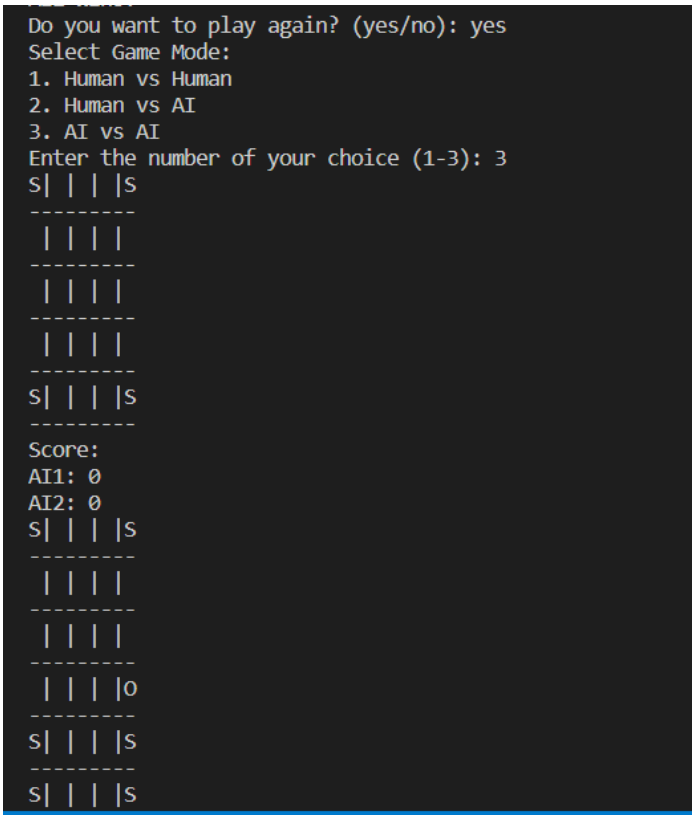
Score:


----
o|o|o|o|o
----
s|s|s|s|s
----
o|o|o|o|s
----
s|s|s|s|s

AI1 (S) scores 2 point(s)!
Score:
AI1: 13
AI2: 7
s|o|s|s|s
----
o|o|o|o|o
----
s|s|s|s|s
----
o|o|o|o|s
----
s|s|s|s|s

AI1 Wins!
Do you want to play again? (yes/no):

Sample 3:


Do you want to play again? (yes/no): yes
Select Game Mode:
1. Human vs Human
2. Human vs AI
3. AI vs AI
Enter the number of your choice (1-3): 3
S| | | |S

| | | |

| | | |

| | | |

S| | | |S

Score:
AI1: 0
AI2: 0
S| | | |S

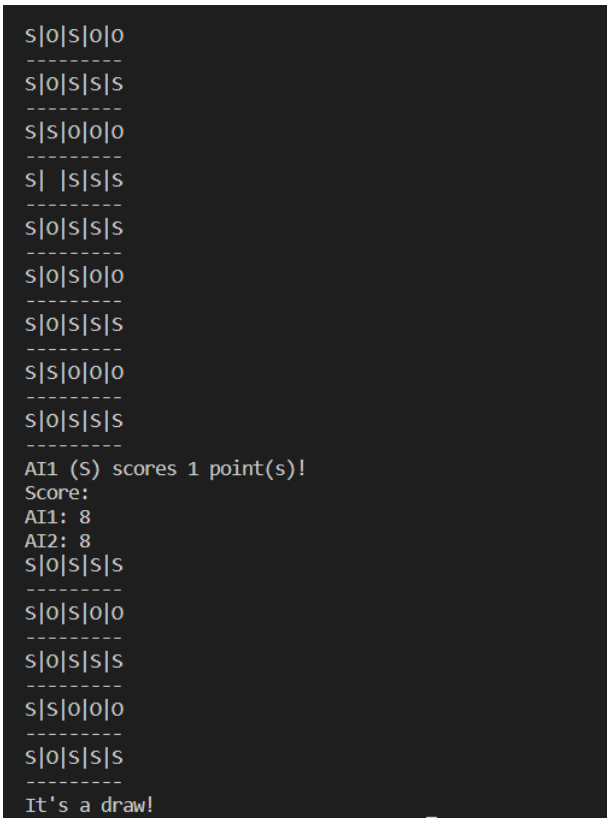
| | | |

| | | |

| | | |O

S| | | |S

S| | | |S



6. Conclusion

In summary, we've successfully implemented a modified version of the SOS game, introducing strategic elements with a 5x5 board, 'S' symbols, and turn restrictions. The project, designed with a clear object-oriented approach, offers engaging gameplay in both Human vs. Human and Human vs. AI modes.

The AI player, powered by the Minimax algorithm and intuitive evaluations, enhances the gaming experience by providing a challenging opponent. While achieving our primary goals, potential improvements include optimizing the Minimax algorithm for better performance and enabling users to customize the board size.

Overall, our SOS project not only meets its objectives but also delivers an enjoyable gaming experience, combining strategy and artificial intelligence.

CODES

Board Class:

- **init Method:**
  - Initializes the board with a size of 5x5 and creates a matrix representing the board.
  - Initializes the board with the starting configuration with 'S' letters in the corners.
- **initialize_board Method:**
  - A helper method that creates the initial state of the board.
- **display_board Method:**
  - Prints the current state of the board.
- **is_valid_move Method:**
  - Checks if the specified row and column constitute a valid move on the board.
- **make_move Method:**
  - Makes a move at the specified row and column with the given symbol; returns True if the move is valid.
- **is_full Method:**
  - Checks if the board is full.

```
# this class creates game board. Its default size is 5x5 with the letters 'S' in the corners.
class Board:
    def __init__(self):
        self.size = 5
        self.board = [[' ' for _ in range(self.size)]
                      for _ in range(self.size)]
        self.initialize_board()

    # bring the board to its initial state. There is the letter 'S' in the corners.
    def initialize_board(self):
        self.board[0][0] = self.board[0][4] = self.board[4][0] = self.board[4][4] = 'S'

    # prints the current state of the board on the screen.
    def display_board(self):
        for row in self.board:
            print('|'.join(row))
            print('-' * 9)

    # checks the specified row and column is a valid move.
    def is_valid_move(self, row, col):
        return 0 <= row < self.size and 0 <= col < self.size and self.board[row][col] == ' '

    # attempts to make a move to the specified row and column with the specified symbol.
    def make_move(self, row, col, symbol):
        if self.is_valid_move(row, col):
            self.board[row][col] = symbol
            return True
        return False

    # check if the board is full.
    def is_full(self):
        return all(all(cell != ' ' for cell in row) for row in self.board)
```

Player Class:

- **init Method:**
    - Stores the player's name and score.

```python
class Player:
    # each player has a name and score.
    def __init__(self, name):
        self.name = name
        self.score = 0


class HumanPlayer(Player):
    # allows a human player to make moves. It receives symbol and move information from the user.
    def make_move(self, board):
        symbol = input(
            f"{self.name}, enter your symbol ('S' or 'O'): ").upper()
        while symbol not in ['S', 'O']:
            print("Invalid symbol. Please enter 'S' or 'O'.")
            symbol = input(
                f"{self.name}, enter your symbol ('S' or 'O'): ").upper()

        while True:
            try:
                row = int(input(f"{self.name}, enter row (0-4): "))
                col = int(input(f"{self.name}, enter column (0-4): "))
                if board.make_move(row, col, symbol):
                    return row, col
                else:
                    print("Invalid move. Try again.")
            except ValueError:
                print("Invalid input. Enter integers for row and column.")
```

HumanPlayer Class:

- **make_move Method:**
    - Takes input from the user for symbol and move.
    - Includes a loop to correct invalid inputs.

AIPlayer Class:

- **init Method:**
    - Stores properties such as depth, heuristic, and player name.
- **make_move Method:**
    - Takes input for the symbol from the user.
    - Computes the best move using the Minimax algorithm and updates the board.
- **h1 Method:**
    - A simple heuristic function that calculates the number of 'SOS' sequences.
- **h2 Method:**
    - Calculates the number of 'SOS' sequences and applies a penalty for each 'O' on the board.
- **minimax Method:**
    - A recursive method that implements the Minimax algorithm with alpha-beta pruning.

play_game Function:

- **play_game Function:**
  - Manages the game loop between two players.
  - Displays the board state and controls the moves.
  - Updates player scores by calculating the number of 'SOS' sequences.

```
class AIPlayer(Player):
    # constructor method of AI player class. It has features such as depth, heuristic function and name.
    def __init__(self, depth=4, heuristic=None, name="AI"):
        super().__init__(name)
        self.depth = depth
        self.heuristic = heuristic if heuristic else self.h1

    # AI player make a move. This method use Minimax algorithm.
    def make_move(self, board):
        # generate all valid moves
        valid_moves = [(i, j) for i in range(board.size)
                       for j in range(board.size) if board.is_valid_move(i, j)]

        # evaluate each move
        scores = []
        for move in valid_moves:
            row, col = move
            board.make_move(row, col, 'S')  # try 'S'
            score_s = self.minimax(board, self.depth, float(
                '-inf'), float('inf'), False, 'S')[0]
            board.board[row][col] = ' '  # undo the move

            board.make_move(row, col, 'O')  # try 'O'
            score_o = self.minimax(board, self.depth, float(
                '-inf'), float('inf'), False, 'O')[0]
            board.board[row][col] = ' '  # undo the move

            # choose symbol that gives the higher score
            if score_s > score_o:
                symbol = 'S'
                score = score_s
            else:
                symbol = 'O'
                score = score_o

            scores.append((score, symbol, move))

        # sort moves by their scores
        scores.sort(reverse=True)

        # choose one of top moves randomly
        top_scores = [score for score in scores if score[0] == scores[0][0]]
        _, symbol, move = random.choice(top_scores)

        # make the chosen move
        row, col = move
        board.make_move(row, col, symbol)
        return row, col
```

Main Program:

- Asks the user to choose the game mode (Human vs Human, Human vs AI, AI vs AI).
- Starts the relevant game mode based on the choice and takes player names.
- Initiates and continues the game until completion.

```python
    # heuristic 1
    @staticmethod
    def h1(board):
        # count the number of 'SOS' sequences
        count = 0
        for i in range(board.size):
            for j in range(board.size):
                # check for vertical 'SOS'
                if i + 2 < board.size and board.board[i][j] == board.board[i + 2][j] == 'S' and board.board[i + 1][j] == 'O':
                    count += 1
                # check for horizontal 'SOS'
                if j + 2 < board.size and board.board[i][j] == board.board[i][j + 2] == 'S' and board.board[i][j + 1] == 'O':
                    count += 1
                # check for diagonal 'SOS'
                if i + 2 < board.size and j + 2 < board.size and (
                        board.board[i][j] == board.board[i + 2][j + 2] == 'S' and board.board[i + 1][j + 1] == 'O' or
                        i == 0 and board.board[i][j] == board.board[i +
                                                                   2][j + 2] == 'S' and board.board[i + 1][j + 1] == 'O'
                ):
                    count += 1
                # check for reverse diagonal 'SOS'
                if i - 2 >= 0 and j + 2 < board.size and board.board[i][j] == board.board[i - 2][j + 2] == 'S' and board.board[i - 1][j + 1] == 'O':
                    count += 1

        return count

    # heuristic 2
    @staticmethod
    def h2(board):
        # count number of 'SOS' sequences and penalize for each 'O' symbol on board.
        count = AIPlayer.h1(board)
        for row in board.board:
            count -= row.count('O')
        return count
```

```python
#  finds the best move with the minimax algorithm.
def minimax(self, board, depth, alpha, beta, maximizing_player, symbol):
    if depth == 0 or board.is_full():
        # return empty move when game is over.
        return self.heuristic(board), (0, 0)

    valid_moves = [(i, j) for i in range(board.size)
                   for j in range(board.size) if board.is_valid_move(i, j)]

    if maximizing_player:
        max_eval = float('-inf')
        best_move = None
        for move in valid_moves:
            row, col = move
            board.make_move(row, col, symbol)
            eval = self.minimax(board, depth - 1, alpha,
                                beta, False, symbol)[0]
            board.board[row][col] = ' '  # undo the move
            if eval > max_eval:
                max_eval = eval
                best_move = move
            alpha = max(alpha, max_eval)
            if beta <= alpha:
                break
        return max_eval, best_move if best_move is not None else (0, 0)
    else:
        min_eval = float('inf')
        best_move = None
        for move in valid_moves:
            row, col = move
            board.make_move(row, col, 'O' if symbol == 'S' else 'S')
            eval = self.minimax(board, depth - 1, alpha,
                                beta, True, symbol)[0]
            board.board[row][col] = ' '  # undo the move
            if eval < min_eval:
                min_eval = eval
                best_move = move
            beta = min(beta, min_eval)
            if beta <= alpha:
                break
        return min_eval, best_move if best_move is not None else (0, 0)
```

```python
# starts the game and controls the flow of the game.
def play_game(player1, player2):
    board = Board()
    current_player = player1

    while True:
        board.display_board()
        row, col = current_player.make_move(board)

        sos_count = AIPlayer.h1(board)
        if sos_count > (player1.score + player2.score):
            board.display_board()
            player_added_score = sos_count - (player1.score + player2.score)
            print(
                f"{current_player.name} ({'S' if current_player == player1 else 'O'}) scores {player_added_score} point(s)!")
            current_player.score += player_added_score

        print("Score:")
        print(f"{player1.name}: {player1.score}")
        print(f"{player2.name}: {player2.score}")

        if board.is_full():
            board.display_board()
            if player1.score > player2.score:
                print(f"{player1.name} Wins!")
                break
            elif player2.score > player1.score:
                print(f"{player2.name} Wins!")
                break
            else:
                print("It's a draw!")
                break


        current_player = player2 if current_player == player1 else player1
```

```python
# selects the game mode and starts the game.
def main():
    while True:
        print("Select Game Mode:")
        print("1. Human vs Human")
        print("2. Human vs AI")
        print("3. AI vs AI")

        choice = input("Enter the number of your choice (1-3): ")

        if choice == '1':
            player1_name = input("Enter name for Player 1: ")
            player2_name = input("Enter name for Player 2: ")
            play_game(HumanPlayer(player1_name), HumanPlayer(player2_name))
        elif choice == '2':
            while True:
                player_name = input("Enter name for Player: ")
                choice_mod = input("Enter the mode that you play (Easy or Hard): ")
                if choice_mod.lower() == 'easy':
                    play_game(HumanPlayer(player_name), AIPlayer(
                        heuristic=AIPlayer.h1, name="AI"))
                    break
                elif choice_mod.lower() == 'hard':
                    play_game(HumanPlayer("Player"), AIPlayer(heuristic=AIPlayer.h2, name="AI"))
                    break
                else:
                    print("Invalid mode. Please enter 'Easy' or 'Hard'.")
        elif choice == '3':
            player1 = AIPlayer(heuristic=AIPlayer.h1, name="AI1")
            player2 = AIPlayer(heuristic=AIPlayer.h2, name="AI2")
            play_game(player1, player2)
        else:
            print("Invalid choice. Exiting.")
            break

        restart = input("Do you want to play again? (yes/no): ").lower()
        if restart != 'yes':
            print("Exiting.")
            break


if __name__ == "__main__":
    main()
```