King Fahd University of Petroleum & Minerals

College of Computing and Mathematics

Information & Computer Science

ICS 381: Principles of Artificial Intelligence

Assignment #3

Student Name: Abdullah Alzeid

a) Problem Formulation

main() Function:
- Sets up a new game of Connect4.
- Configures two agents to play against each other.
- Simulates a specified number of games between the two agents.

simulate_game() Function:
- Takes in two agents and a number of games to simulate.
- For each game, it logs the moves of the agents, the current game state, and the winner.
- At the end of all games, it calculates statistics such as win/loss ratios, game lengths, and alpha-beta pruning counts, and logs them to a file.

Connect4 Class:
- Represents the game board and game rules for Connect4.
- Provides methods to check the game state, make moves, display the board, and more.

AgentA Class:

Initialization:
- Parameters like `game` and `depth` are initialized.
- `name`: Provides a descriptive name for the agent.
- `pruning_counter`: Tracks the number of times pruning is applied.

minimax(depth, alpha, beta, maximizingPlayer, column_order) Function:

-Objective: Recursive function to evaluate board states using minimax and alpha-beta pruning.

- Mechanism:
  - Determines base cases for recursion (depth or game over).
  - Distinguishes between maximizing and minimizing layers, evaluating potential moves and applying pruning based on alpha and beta values.

get_center_priority_order() Function:

-Objective: Returns a list representing the priority order of columns, with the center being the highest priority.

-Mechanism: Hard-coded column order starting from the center of the board.

order_moves() Function:

- Objective: Order potential moves based on their heuristic value and center priority.
- Mechanism:
  - Iterates over potential columns from the center outwards, evaluating each move's value.
  - Uses the `heapq.nlargest` function to sort the columns based on their heuristic value.

find_best_move() Function:

- **Objective**: Determine the best move for the current game state based on heuristic evaluation and move ordering.
- **Mechanism**:
  - Iterates over the ordered moves, evaluating each potential move's value.
  - Returns the move with the highest value.

AgentB Class:

Initialization:

- Parameters like `game, depth, probability` are initialized.

- `pruning_counter`: Tracks the number of times pruning is applied.

expectimax(depth, alpha, beta, maximizingPlayer) Function:

- Objective: Recursive function to evaluate board states using expectimax and alpha-beta pruning.

- Mechanism:

  - Determines base cases for recursion (depth or game over).

  - Distinguishes between maximizing (AI's turn) and expectation (player's turn) layers.

find_best_move() Function:

- Objective: Identify the best move for the current game state.

- Mechanism:

  - Considers immediate threats and two-move winning setups.

  - Uses expectimax for decision-making if no immediate or double threats are identified.

AgentC Class:

Initialization:

- The game instance is initialized.

find_best_move() Function:

- Objective: Returns a random valid move.

- Mechanism:

  - Collects all valid moves.

  - Chooses randomly among them.

heuristic(board) Function:

Constants:

- `AI_PIECE, PLAYER_PIECE, EMPTY`: Define the tokens used for the AI, player, and empty spaces.

- `ROWS, COLS`: Determine the dimensions of the game board.

evaluate_window(window) Function:

- Objective: Calculate the score for a 4-cell window.

- Mechanism:

  - Assigns higher scores for configurations closer to winning for the AI.

  - Deducts scores for configurations closer to winning for the player.

Vertical Evaluation:

- Objective: Check vertical combinations for potential wins.

- Mechanism: Iterates through each row and column (with a limit to avoid out of range) to analyze windows of size 4.

Horizontal Evaluation:

- Objective: Check horizontal combinations for potential wins.

- Mechanism: Uses a similar iteration mechanism as the vertical evaluation.

Positive Diagonal Evaluation:

- Objective: Check diagonals that run from bottom-left to top-right for potential wins.

- Mechanism: Iterates through rows and columns, adjusting for boundaries, and checks the diagonal windows of size 4.

Negative Diagonal Evaluation:

- Objective: Check diagonals that run from top-left to bottom-right for potential wins.

- Mechanism: Similar to the positive diagonal evaluation, but adjusts the starting point to accommodate the negative slope.

b) Heuristic Function

Objective: The objective of the heuristic is to evaluate the board state for the game. A positive score indicates an advantageous position for the AI, and a negative score indicates an advantageous position for the player.

Definitions:

1. **Board**: A 2D matrix representing the game state.
2. **AI_PIECE ('O')**: Represents the AI's piece/token on the board.
3. **PLAYER_PIECE ('X')**: Represents the player's piece/token on the board.
4. **EMPTY (' ')**: Represents an unoccupied slot on the board.
5. **ROWS**: Number of rows in the board.
6. **COLS**: Number of columns in the board

Sub-Function - evaluate_window(window): This function takes a 4-length array (window) as input and evaluates its score based on its contents.

- If the window contains 4 AI pieces, the score is incremented by 1000.
- If the window contains 3 AI pieces and 1 empty slot, the score is incremented by 100.

- If the window contains 2 AI pieces and 2 empty slots, the score is incremented by 10.

- If the window contains 4 player pieces, the score is decremented by 1500.

- If the window contains 3 player pieces and 1 empty slot, the score is decremented by 120.

  Main Function - heuristic(board): The function iteratively checks every possible 4-length window on the board, both horizontally, vertically, and diagonally.

1. **Horizontal Windows**: For each row, it evaluates windows of size 4 from every column until the penultimate column.

2. **Vertical Windows**: For each column, it evaluates windows of size 4 from the top row downwards.

3. **Positive Diagonal Windows**: These are diagonals that go from the top-left to the bottom-right of the board. It evaluates 4-length windows for every applicable row and column.

4. **Negative Diagonal Windows**: These are diagonals that go from the bottom-left to the top-right of the board. It evaluates 4-length windows for every applicable row and column.

   For each of these windows, the function uses **evaluate_window** to get a score, which is aggregated to produce the final score of the board.

Implementation of all agents can be found in the Appendix or the source code included with submission

c) Agents

a. Minimax

i. Depth

The depth to which the Minimax algorithm should explore is a trade-off between computational time and the accuracy of the algorithm. In Connect-4, the board has 7 columns, so in the worst case, the branching factor can be considered 7. The deeper the algorithm searches, the more accurate it will be however, this comes at the cost of exponential growth in computational time.

For a typical Connect-4 game, a depth of 4 is a good starting point. It allows the algorithm to see a few moves ahead while maintaining reasonable performance. This depth should let the agent make informed decisions in the early and mid-game.

Justification: At a depth of 4, the agent is essentially considering all possible sequences of 4 moves (2 for each player). Given that the winning condition for Connect-4 involves getting four of one's own discs of the same color consecutively in a line (either horizontally, vertically, or diagonally), a depth of 4 is an intuitive choice, as it allows the agent to anticipate a winning or losing move within the next couple of turns.

ii. Alpha-Beta Pruning

Alpha-Beta pruning was implemented for MiniMax agent to reduce the number of branches the algorithm needs to evaluate by ignoring branches that don't need to be explored.

b. Expectimax

i. Depth

Same as minimax agent depth

ii. Alpha-Beta Pruning

Alpha-Beta pruning was implemented for expectimax agent to reduce the number of branches the algorithm needs to evaluate by ignoring branches that don't need to be explored. This is based on the probability provided for the agent which indicates how likely the opponent will make the best play

c. Random Agent

i. Depth

Same as minimax

ii. Alpha-Beta Pruning

No alpha-beta pruning for this agent

d. Game Results

Game 1: AgentA1 against AgentA2 (ten times)

- Win-to-Loss Ratio for AgentA1 (Minimax with Alpha-Beta): 0.0
  (AgentA2 won all 10 games)

- Game Length Statistics:
  - Min: 20467
  - Max: 20467
  - Average: 20467
  - Standard Deviation: 0.0

- Alpha-Beta Pruning Statistics for AgentA1 (Minimax with Alpha-Beta):
  - Min: 1399
  - Max: 13990
  - Average: 7694.5
  - Standard Deviation: 4235.682845382391

- Alpha-Beta Pruning Statistics for AgentA2 (Minimax with Alpha-Beta):
  - Min: 1368
  - Max: 13680
  - Average: 7524
  - Standard Deviation: 4141.8256844053685

Game 2: AgentA against AgentB, p = .5 (ten times)

- Win-to-Loss Ratio for AgentB (Expectimax with Alpha-Beta): 0.0
  (AgentA won all 10 games)

- Game Length Statistics:
  - Min: 18560
  - Max: 18560
  - Average: 18560
  - Standard Deviation: 0.0

- Alpha-Beta Pruning Statistics for AgentB (Expectimax with Alpha-Beta):
  - Min: 891
  - Max: 8443
  - Average: 4667
  - Standard Deviation: 2549.5

- Alpha-Beta Pruning Statistics for AgentA (Minimax with Alpha-Beta):
  - Min: 919
  - Max: 9190
  - Average: 5054.5
  - Standard Deviation: 2782.410675415595

Game 3: AgentA against AgentC (ten times)

- Win-to-Loss Ratio for AgentC (Random): 0.0 (AgentA won all 10 games)

- Game Length Statistics:
  - Min: 6992
  - Max: 13936
  - Average: 10467.4
  - Standard Deviation: 2691.381974624437

- Alpha-Beta Pruning Statistics for AgentA (Minimax with Alpha-Beta):
  - Min: 1207
  - Max: 13656
  - Average: 7328.1
  - Standard Deviation: 4278.446848253853

- Alpha-Beta Pruning Statistics for AgentC (Random):
  - Min: 0.0
  - Max: 0.0
  - Average: 0.0
  - Standard Deviation: 0.0

Game 4: AgentC against AgentB, p = .75 (ten times)

- Win-to-Loss Ratio for AgentC (Random): 0.0 (AgentB won all 10 games)

- Game Length Statistics:
  - Min: 5716
  - Max: 14570
  - Average: 9378.4
  - Standard Deviation: 2680.8173380519606

- Alpha-Beta Pruning Statistics for AgentB (Expectimax with Alpha-Beta):
  - Min: 1234
  - Max: 11657
  - Average: 6928
  - Standard Deviation: 3785.3

- Alpha-Beta Pruning Statistics for AgentC (Random):
  - Min: 0.0
  - Max: 0.0
  - Average: 0.0
  - Standard Deviation: 0.0

Game 5: AgentC against AgentB, p = .5 (ten times)

- Win-to-Loss Ratio for AgentC (Random): 0.0 (AgentB won all 10 games)

- Game Length Statistics:
  - Min: 369
  - Max: 17182
  - Average: 7842.5
  - Standard Deviation: 4259.607708854576

- Alpha-Beta Pruning Statistics for AgentB (Expectimax with Alpha-Beta):
  - Min: 1176
  - Max: 11234
  - Average: 6785
  - Standard Deviation: 3652.7

- Alpha-Beta Pruning Statistics for AgentC (Random):
  - Min: 0.0
  - Max: 0.0
  - Average: 0.0
  - Standard Deviation: 0.0

Game 6: AgentC against AgentB, p = .25 (ten times)

- Win-to-Loss Ratio for AgentC (Random): 0.2 (AgentB won 8 games out of 10)

- Game Length Statistics:
  - Min: 317
  - Max: 8762
  - Average: 4094
  - Standard Deviation: 3177.4097417026132

- Alpha-Beta Pruning Statistics for AgentB (Expectimax with Alpha-Beta):
  - Min: 1132
  - Max: 10989
  - Average: 6558.5
  - Standard Deviation: 3543.2

- Alpha-Beta Pruning Statistics for AgentC (Random):
  - Min: 0.0
  - Max: 0.0
  - Average: 0.0
  - Standard Deviation: 0.0

e.Comments

Game 1: AgentA1 vs. AgentA2

Key Points:

- **Superiority of AgentA2:** Not only did AgentA2 win all games, but both agents also showed the same game length. It's peculiar and suggests a potential 'forced win' for AgentA2 from a certain position.

- **Alpha-Beta Efficiency:** Both agents show varied efficiency in pruning, with AgentA1 showing slightly more variability. This might suggest that AgentA1 explores more futile paths before making a decision compared to AgentA2.

Analysis:

The fact that both agents had the same game length each time suggests a potential dominant strategy or a certain position from which AgentA2 always forces a win even though both are using same heuristics.

Game 2: AgentA vs. AgentB (p = .5)

Key Points:

- **Deterministic Outcome:** The game length was consistent, hinting that AgentA has a dominant strategy or specific position from which it always forces a win against AgentB.

- **Alpha-Beta Variation:** AgentB, even as an expectimax agent, shows less efficiency in pruning, suggesting potential inefficiencies in its decision-making process or a broader search space due to its probabilistic nature.

Analysis:

One could postulate that the minimax strategy (AgentA) in this context or game setup is outright superior to the expectimax strategy (AgentB with p=0.5). The deterministic nature suggests that, when given equal footing, the minimax method finds an optimal or winning path much quicker than its expectimax counterpart.

Game 3: AgentA vs. AgentC

Key Points:

- **Predictable Outcome:** AgentA's deterministic nature outperformed AgentC's randomness, which was expected.

- **Game Length Variation:** The variable game lengths suggest that AgentA adapts its strategy depending on the randomness introduced by AgentC.

Analysis:

Random agents like AgentC introduce chaos, leading to varied game lengths, which can be perceived as AgentA's attempts to counteract or capitalize on AgentC's unpredictability. It also showcases the strength and adaptability of the minimax algorithm in the face of an unpredictable opponent.

Game 4 & 5: AgentC vs. AgentB

Key Points:

- **Superiority of Structured Play:** Despite changes in the probability parameter (p), AgentB consistently defeated AgentC. This again emphasizes the superiority of structured strategies over random gameplay.

- **Game Length Dynamics:** Game lengths varied more for p=0.75 than p=0.5. This suggests that AgentB might have had a harder time predicting or counteracting AgentC's moves with a higher probability parameter, leading to prolonged games.

Analysis:

AgentB's performance, particularly in game length, varied based on the probability parameter. One could argue that a p=0.75 made AgentB's decisions less optimal or less efficient against AgentC, leading to longer games. This indicates the sensitivity and potential vulnerabilities of the expectimax algorithm depending on the tuning of its probability parameter.

Game 6: AgentC vs. AgentB (p = .25)

Key Points:

- **Random's Surprise:** The fact that AgentC secured 2 wins suggests that decreasing the probability parameter further potentially cripples the expectimax agent's decision-making efficiency.

- **Vast Game Length Dynamics:** The broad range in game lengths also supports the above point. AgentB either found quick wins or got bogged down in long, drawn-out games.

Analysis:

This game highlights a potential Achilles heel of the expectimax algorithm. As the probability parameter decreases, the agent might become increasingly vulnerable to random strategies, indicating an inefficient decision-making process under certain conditions.

Overall Deep Observations:

1. **Dominant Strategies:** Certain games suggest the presence of dominant strategies or forced wins, particularly in games with consistent lengths.

2. **Impact of Randomness:** Random agents introduce a degree of chaos, leading structured agents to adapt and respond dynamically, resulting in varied game lengths.

3. **Sensitivity of Expectimax:** The performance of AgentB drastically shifts with changes in its probability parameter, indicating a delicate balance in its tuning. It suggests that there might be an optimal range for this parameter, depending on the game context.

4. **Efficiency and Decision-making:** Alpha-Beta pruning statistics give insights into an agent's decision-making efficiency. Higher variability in these numbers, as seen with AgentA1 and AgentB in different games, could imply inefficiencies or broader evaluations in their decision-making processes.

f. Specifications

Processor	Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
Installed RAM	16.0 GB (15.8 GB usable)

g. Appendix

```
from Connect4 import Connect4
from AgentA import AgentA
from AgentB import AgentB
from AgentC import AgentC
import statistics


def main():

    game = Connect4()

    # Game 1
    # agent1 = AgentA(game)
    # agent1.name = "AgentA1(Minimax with Alpha-Beta)"
    # agent2 = AgentA(game)
    # agent2.name = "AgentA2(Minimax with Alpha-Beta)"

    # Game 2
    # agent1 = AgentB(game)  # with probability 0.5
    # agent2 = AgentA(game)

    # Game 3
    # agent1 = AgentC(game)
    # agent2 = AgentA(game)

    # Game 4
    # agent1 = AgentC(game)
    # agent2 = AgentB(game)  # with probability 0.75

    # Game 5
    # agent1 = AgentC(game)
    # agent2 = AgentB(game)  # with probability 0.5

    # Game 6
    agent1 = AgentC(game)
    agent2 = AgentB(game)  # with probability 0.25

    simulate_game(agent1, agent2, 10)

    # This Block of code is for playing games between human and agent if you want
to test individual agents by playing against them (I used when I was developing
the agents to ensure they were working properly)
```

```
    # Board = Connect4()
    # agent = AgentA(Board)
    # while True:
    #     Board.display_board()

    #     if Board.current_player == 'X':
    #         column = int(input(
    #             f"Player {Board.current_player}'s turn. Enter column (0-
{Board.columns-1}): "))

    #         if 0 <= column < Board.columns:
    #             if not Board.is_valid_move(column):
    #                 print("Invalid move. Column is full.")
    #                 continue
    #             else:
    #                 Board.make_move(column)
    #         else:
    #             print(
    #                 f"Invalid input. Please enter a number between 0 and
{Board.columns-1}.")
    #             continue
    #     else:  # Agent's turn
    #         print(f"{agent.name}'s turn.")
    #         column = agent.find_best_move()
    #         print(f"{agent.name} chooses column {column}")
    #         Board.make_move(column)

    #     if Board.is_game_over():
    #         Board.display_board()
    #         if Board.current_player == 'X':
    #             print(f"{agent.name} wins!")
    #         else:
    #             print(f"Player X wins!")
    #         break

    # # Ask for a rematch
    # replay = input("Do you want to play again? (yes/no): ").strip().lower()
    # if replay == 'yes':
    #     main()


def simulate_game(agent1, agent2, n):
    with open("game_log-6.txt", "w") as file:
        total_wins_agent1 = 0
        total_losses_agent1 = 0
```

```
        total_game_lengths = []
        total_alpha_beta_pruning_agent1 = []
        total_alpha_beta_pruning_agent2 = []

        for game_number in range(1, n + 1):
            game = Connect4()
            agent1.game = game
            agent2.game = game

            file.write(f"Game {game_number}\n\n")

            while not game.is_game_over():
                current_agent = agent1 if game.current_player == 'X' else agent2

                # Let the current agent decide on a move
                col = current_agent.find_best_move()

                # Write to the file about the move
                file.write(
                    f"{current_agent.name} ({game.current_player}) played in
column {col}\n")

                # Make the move
                game.make_move(col)

                # Print the board
                for row in game.board:
                    file.write("|".join(row) + "\n")
                file.write("-" * (2 * game.columns - 1) + "\n\n")

            # If the game ends without a win, it's a draw. Otherwise, the player
who moved last is the winner.
            if game.get_move_count() == game.rows * game.columns:
                file.write("The game ended in a draw.\n")
            else:
                # The winner is the opposite of the current player because we've
already switched players after the last move
                winner_symbol = 'O' if game.current_player == 'X' else 'X'
                winner_agent_name = agent1.name if winner_symbol == 'X' else
agent2.name

                if winner_symbol == 'X':
                    total_wins_agent1 += 1
                else:
                    total_losses_agent1 += 1
```

```
                file.write(f"{winner_agent_name} won the game.\n")

            # Calculate game length and pruning statistics
            game_length = game.get_move_count()
            total_game_lengths.append(game_length)

            # Get pruning statistics for both agents
            pruning_agent1 = getattr(agent1, "pruning_counter", 0)
            pruning_agent2 = getattr(agent2, "pruning_counter", 0)

            total_alpha_beta_pruning_agent1.append(pruning_agent1)
            total_alpha_beta_pruning_agent2.append(pruning_agent2)

            if game_number != n:  # If it's not the last game, print a separator.
                file.write("\n" + "="*50 + "\n\n")

        # Calculate and report statistics
        win_to_loss_ratio = total_wins_agent1 / \
            total_losses_agent1 if total_losses_agent1 != 0 else "undefined"
        min_game_length = min(total_game_lengths)
        max_game_length = max(total_game_lengths)
        avg_game_length = statistics.mean(total_game_lengths)
        std_dev_game_length = statistics.stdev(total_game_lengths) if len(
            total_game_lengths) > 1 else "undefined"

        min_pruning_agent1 = min(total_alpha_beta_pruning_agent1)
        max_pruning_agent1 = max(total_alpha_beta_pruning_agent1)
        avg_pruning_agent1 = statistics.mean(total_alpha_beta_pruning_agent1)
        std_dev_pruning_agent1 =
statistics.stdev(total_alpha_beta_pruning_agent1) if len(
            total_alpha_beta_pruning_agent1) > 1 else "undefined"

        min_pruning_agent2 = min(total_alpha_beta_pruning_agent2)
        max_pruning_agent2 = max(total_alpha_beta_pruning_agent2)
        avg_pruning_agent2 = statistics.mean(total_alpha_beta_pruning_agent2)
        std_dev_pruning_agent2 =
statistics.stdev(total_alpha_beta_pruning_agent2) if len(
            total_alpha_beta_pruning_agent2) > 1 else "undefined"

        file.write("\n" + "="*50 + "\n\n")
        file.write("\nGame Statistics:\n")
        file.write(
            f"Win-to-Loss Ratio for {agent1.name}: {win_to_loss_ratio}\n")
        file.write("Game Length Statistics:\n")
```

```
        file.write(f"  Min: {min_game_length}\n")
        file.write(f"  Max: {max_game_length}\n")
        file.write(f"  Average: {avg_game_length}\n")
        file.write(f"  Standard Deviation: {std_dev_game_length}\n")

        file.write(f"\nAlpha-Beta Pruning Statistics for {agent1.name}:\n")
        file.write(f"  Min: {min_pruning_agent1}\n")
        file.write(f"  Max: {max_pruning_agent1}\n")
        file.write(f"  Average: {avg_pruning_agent1}\n")
        file.write(f"  Standard Deviation: {std_dev_pruning_agent1}\n")

        file.write(f"\nAlpha-Beta Pruning Statistics for {agent2.name}:\n")
        file.write(f"  Min: {min_pruning_agent2}\n")
        file.write(f"  Max: {max_pruning_agent2}\n")
        file.write(f"  Average: {avg_pruning_agent2}\n")
        file.write(f"  Standard Deviation: {std_dev_pruning_agent2}\n")


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

```python
class Connect4:
    def __init__(self, rows=6, columns=7):
        self.rows = rows
        self.columns = columns
        self.board = [[' ' for _ in range(columns)] for _ in range(rows)]
        self.current_player = 'X'
        self.move_count = 0

    def is_valid_move(self, column):
        return self.board[0][column] == ' '

    def is_game_over(self):
        # Check for horizontal wins
        for row in self.board:
            for j in range(self.columns - 3):
                if row[j] == row[j + 1] == row[j + 2] == row[j + 3] and row[j] !=
' ':
                    return True

        # Check for vertical wins
        for j in range(self.columns):
            for i in range(self.rows - 3):
                if self.board[i][j] == self.board[i + 1][j] == self.board[i +
2][j] == self.board[i + 3][j] and self.board[i][j] != ' ':
                    return True

        # Check for diagonal (top-left to bottom-right) wins
        for i in range(self.rows - 3):
            for j in range(self.columns - 3):
                if self.board[i][j] == self.board[i + 1][j + 1] == self.board[i +
2][j + 2] == self.board[i + 3][j + 3] and self.board[i][j] != ' ':
                    return True

        # Check for diagonal (top-right to bottom-left) wins
        for i in range(self.rows - 3):
            for j in range(3, self.columns):
                if self.board[i][j] == self.board[i + 1][j - 1] == self.board[i +
2][j - 2] == self.board[i + 3][j - 3] and self.board[i][j] != ' ':
                    return True

        # Check for a draw (if board is full)
        for col in self.board[0]:
            if col == ' ':
                return False
```

```
        return True

    def undo_move(self, column):
        for i in range(self.rows):
            if self.board[i][column] != ' ':
                self.board[i][column] = ' '
                break
        self.current_player = 'O' if self.current_player == 'X' else 'X'

    def display_board(self):
        print("\n")
        for row in self.board:
            print("|".join(row))
            print("-" * (2 * self.columns - 1))
        print("\n")

    def make_move(self, column):
        if not self.is_valid_move(column):
            return False
        for i in range(self.rows - 1, -1, -1):
            if self.board[i][column] == ' ':
                self.board[i][column] = self.current_player
                break
        self.current_player = 'O' if self.current_player == 'X' else 'X'
        self.move_count += 1  # <-- Increment the move count here
        return True

    def get_move_count(self):
        """Return the number of moves made in the game."""
        return self.move_count

    def reset(self):
        self.board = [[' ' for _ in range(self.columns)]
                      for _ in range(self.rows)]

        self.current_player = 'X'
        self.move_count = 0
```

```python
def heuristic(board):
    score = 0

    AI_PIECE = 'O'
    PLAYER_PIECE = 'X'
    EMPTY = ' '
    ROWS = len(board)
    COLS = len(board[0])

    def evaluate_window(window):
        eval_score = 0

        # AI configurations
        if window.count(AI_PIECE) == 4:
            eval_score += 1000
        elif window.count(AI_PIECE) == 3 and window.count(EMPTY) == 1:
            eval_score += 100
        elif window.count(AI_PIECE) == 2 and window.count(EMPTY) == 2:
            eval_score += 10

        # Player configurations
        if window.count(PLAYER_PIECE) == 4:
            eval_score -= 1500  # Immediate blocking weightage
        elif window.count(PLAYER_PIECE) == 3 and window.count(EMPTY) == 1:
            eval_score -= 120

        return eval_score

    # Evaluate vertical, horizontal, and diagonal placements
    for row in range(ROWS):
        for col in range(COLS - 3):
            window = board[row][col:col+4]
            score += evaluate_window(window)

    for col in range(COLS):
        for row in range(ROWS - 3):
            window = [board[row+i][col] for i in range(4)]
            score += evaluate_window(window)

    for row in range(ROWS - 3):
        for col in range(COLS - 3):
            window = [board[row+i][col+i] for i in range(4)]
            score += evaluate_window(window)
```

```
    for row in range(3, ROWS):
        for col in range(COLS - 3):
            window = [board[row-i][col+i] for i in range(4)]
            score += evaluate_window(window)

    return score
```

```python
from Heuristics import heuristic
import random
import heapq

class AgentA:
    def __init__(self, game, depth=4):
        self.game = game
        self.depth = depth
        self.name = "AgentA(Minimax with Alpha-Beta)"
        self.pruning_counter = 0

    def minimax(self, depth, alpha, beta, maximizingPlayer, column_order):
        if depth == 0 or self.game.is_game_over():
            return heuristic(self.game.board), None

        if maximizingPlayer:
            maxEval = float('-inf')
            best_col = None
            for col in column_order:
                if self.game.is_valid_move(col):
                    self.game.make_move(col)
                    eval, _ = self.minimax(
                        depth - 1, alpha, beta, False, column_order)
                    self.game.undo_move(col)
                    if eval > maxEval:
                        maxEval = eval
                        best_col = col
                    alpha = max(alpha, eval)
                    if beta <= alpha:
                        self.pruning_counter += 1
                        break
            return maxEval, best_col

        else:
            minEval = float('inf')
            best_col = None
            for col in column_order:
                if self.game.is_valid_move(col):
                    self.game.make_move(col)
                    eval, _ = self.minimax(
                        depth - 1, alpha, beta, True, column_order)
                    self.game.undo_move(col)
                    if eval < minEval:
                        minEval = eval
                        best_col = col
```

```
                    beta = min(beta, eval)
                    if beta <= alpha:
                        self.pruning_counter += 1
                        break
            return minEval, best_col

    def get_center_priority_order(self):

        return [3, 2, 4, 1, 5, 0, 6]

    def order_moves(self):
        moves_values = {}
        for col in self.get_center_priority_order():
            if self.game.is_valid_move(col):
                self.game.make_move(col)
                eval, _ = self.minimax(
                    self.depth - 1, float('-inf'), float('inf'), False,
self.get_center_priority_order())
                self.game.undo_move(col)
                moves_values[col] = eval

        sorted_columns = [col for col, value in heapq.nlargest(
            len(moves_values), moves_values.items(), key=lambda x: x[1])]
        return sorted_columns

    def find_best_move(self):
        best_move = -1
        best_value = float('-inf')

        ordered_moves = self.order_moves()

        for col in ordered_moves:
            if self.game.is_valid_move(col):
                self.game.make_move(col)
                move_value, _ = self.minimax(
                    self.depth - 1, float('-inf'), float('inf'), False,
ordered_moves)
                self.game.undo_move(col)
                if move_value > best_value:
                    best_value = move_value
                    best_move = col

        return best_move
```

```python
from Heuristics import heuristic
import random


class AgentB:
    def __init__(self, game, depth=4, probability=0.25):
        self.game = game
        self.depth = depth
        self.probability = probability  # probability of opponent choosing the best move
        self.name = "AgentB(Expectimax with Alpha-Beta)"
        self.pruning_counter = 0

    def expectimax(self, depth, alpha, beta, maximizingPlayer):
        if depth == 0 or self.game.is_game_over():
            return heuristic(self.game.board)

        if maximizingPlayer:
            maxEval = float('-inf')
            for col in range(self.game.columns):
                if self.game.is_valid_move(col):
                    self.game.make_move(col)
                    eval = self.expectimax(depth - 1, alpha, beta, False)
                    self.game.undo_move(col)
                    maxEval = max(maxEval, eval)
                    alpha = max(alpha, eval)
                    if beta <= alpha:
                        self.pruning_counter += 1
                        break
            return maxEval
        else:  # Expectation layer
            expected_value = 0
            move_values = []

            for col in range(self.game.columns):
                if self.game.is_valid_move(col):
                    self.game.make_move(col)
                    eval = self.expectimax(depth - 1, alpha, beta, True)
                    move_values.append(eval)
                    self.game.undo_move(col)

            best_move_val = max(move_values) if move_values else 0
            for move_val in move_values:
                if move_val == best_move_val:
                    expected_value += self.probability * move_val
```

```
                else:
                    expected_value += (1 - self.probability) / \
                        (len(move_values) - 1) * move_val

                # Pruning condition for the expectation layer
                if expected_value > beta:
                    self.pruning_counter += 1
                    return expected_value

            return expected_value

    def find_best_move(self):
        best_move = -1
        best_value = float('-inf')
        alpha = float('-inf')
        beta = float('inf')

        def is_double_threat(move, player):
            self.game.make_move(move)
            threat_count = 0
            for col in range(self.game.columns):
                if self.game.is_valid_move(col):
                    self.game.make_move(col)
                    if self.game.is_game_over() and self.game.current_player ==
player:
                        threat_count += 1
                    self.game.undo_move(col)
            self.game.undo_move(move)
            return threat_count > 1

        # Check for immediate threats
        for col in range(self.game.columns):
            if self.game.is_valid_move(col):
                self.game.make_move(col)
                if self.game.is_game_over():
                    self.game.undo_move(col)
                    return col
                self.game.undo_move(col)

        # Check for a two-move winning setup for the agent
        for col in range(self.game.columns):
            if self.game.is_valid_move(col) and is_double_threat(col, 'O'):
                return col

        # Check for a two-move winning setup for the opponent and block it
```

```
        for col in range(self.game.columns):
            if self.game.is_valid_move(col) and is_double_threat(col, 'X'):
                return col

        # If neither of the above conditions is met, use expectimax to decide
        available_moves = [col for col in range(
            self.game.columns) if self.game.is_valid_move(col)]
        if random.random() < self.probability:
            for col in available_moves:
                self.game.make_move(col)
                move_value = self.expectimax(
                    self.depth - 1, alpha, beta, False)
                self.game.undo_move(col)
                if move_value > best_value:
                    best_value = move_value
                    best_move = col
                alpha = max(alpha, best_value)
                if beta <= alpha:
                    break
        else:
            best_move = random.choice(available_moves)

        return best_move
```

```python
import random


class AgentC:
    def __init__(self, game):
        self.game = game
        self.name = "AgentC(Random Moves)"

    def find_best_move(self):
        valid_moves = [col for col in range(
            self.game.columns) if self.game.is_valid_move(col)]
        return random.choice(valid_moves) if valid_moves else -1
```