**Connect Go! Report**

**Introduction:**

The documentation presented herein delves into the intricate workings of two Java programs: the MyPlayer class and the Main class. These programs serve as exemplars of a player endowed with artificial intelligence prowess, showcasing the utilization of the minimax algorithm with alpha-beta pruning within the dynamic landscape of a game environment.

**Problem Statement:**

At the crux of these programs lies the quintessential challenge of empowering an artificial player with the cognitive faculties requisite for strategic decision-making in a game setting. The MyPlayer class encapsulates this endeavor, aiming to demonstrate adept maneuvering and astute gameplay leveraging sophisticated algorithms. The orchestration of the game simulation by the Main class serves as the stage upon which the prowess of the artificial player is showcased, inviting observers to witness the intricacies of AI-driven gameplay.

**Methods or Approach Used:**

**MyPlayer Class:**

The foundation of the MyPlayer class rests upon the robust architecture of the minimax algorithm augmented by alpha-beta pruning. This algorithmic framework endows the artificial player with the capability to traverse the intricate game tree, evaluating myriad potential game states to discern the optimal course of action. Augmenting this algorithm is a meticulously crafted evaluation function, meticulously designed to distill the nuances of game state assessment into a cohesive metric of desirability.

**Main Class:**

Serving as the gateway to the game simulation, the Main class meticulously orchestrates the interplay between the game board, player entities, and the overarching game loop. From initializing the game environment to overseeing the progression of gameplay and ultimately adjudicating the game's denouement, the Main class acts as the custodian of the gaming experience. Its role extends beyond mere execution, offering users a veritable window into the realm of AI-driven decision-making within the context of gameplay.

**Detailed Instruction on How to Run the Code:**

**Compilation:**

Embark upon the compilation journey by navigating to the directory housing the Java files (MyPlayer.java and Main.java).

Execute the compilation ritual with the following incantation:

javac MyPlayer.java Main.java

**Running:**

Traverse the realms of execution by invoking the Main class with the following invocation:

java Main

Embark upon a voyage of discovery as you traverse the annals of the game simulation, guided by the beacon of on-screen instructions.

Navigate the labyrinthine twists and turns of the game environment, taking note of the salient milestones and pivotal moments that shape the narrative arc of gameplay.

**Conclusion:**

In the culmination of this documentation journey, one finds themselves enriched with a deeper understanding of the intricacies underlying AI-driven gameplay. The MyPlayer class and its symbiotic relationship with the Main class offer a glimpse into the potential of artificial intelligence to revolutionize the gaming landscape, ushering in an era of nuanced decision-making and strategic acumen.

**Future Work:**

Looking towards the horizon, the realm of possibilities for further refinement and enhancement beckons. Future endeavors may entail the refinement of the evaluation function, imbuing it with greater sensitivity to contextual nuances and game dynamics. Furthermore, the integration of additional game features and rules could serve to enrich the gaming experience, offering players a more immersive and engaging journey through the game environment. Exploration of alternative algorithms and strategies also holds promise, opening new vistas for the evolution of AI-driven gameplay in diverse gaming scenarios.

**More details on Connect Go!**

**Overview:**

The **MyPlayer** class serves as a representation of a player in a game environment. It implements the **PlayerInterface**, providing essential methods to interact with the game state and make decisions regarding moves. This class utilizes the **minimax algorithm** with alpha-beta pruning to determine the best moves for the player.

**Constructors:**

• **MyPlayer(int player\_letter, int maxDepth):** Constructs a player object with the specified player letter and maximum search depth for the minimax algorithm. The player letter indicates whether the player is assigned the **RED** or **YELLOW** color in the game.

**Methods:**

**getMoveColumn(State board)**

• **Description:** This method is responsible for determining the column where the player wants to make their move.

• **Parameters:**

• **board**: The current state of the game board represented by the State object.

• **Returns:** An integer representing the column index where the player intends to place their piece.

• **Algorithm:** Utilizes the minimax algorithm with alpha-beta pruning to select the best move based on the current state of the game board.

**getPlayerID()**

• **Description**: Retrieves the player's identification, which corresponds to their assigned color (RED or YELLOW).

• **Returns:** An integer representing the player's identification.

**minimax(State board, int depth, int alpha, int beta, boolean maximizingPlayer)**

• **Description:** Implements the minimax algorithm with alpha-beta pruning to recursively search through the game tree and determine the optimal move for the player.

• **Parameters:**

• **board**: The current state of the game board.

• **depth:** The current depth of the search tree.

• **alpha:** The alpha value for alpha-beta pruning.

• **beta**: The beta value for alpha-beta pruning.

• **maximizingPlayer**: A boolean indicating whether the current player is maximizing or minimizing.

• **Returns:** An array containing the optimal move's evaluation score and the corresponding column index.

**evaluate(State board)**

• **Description**: Evaluates the current state of the game board to determine its desirability for the player. This method serves as the heuristic evaluation function for the minimax algorithm.

• **Parameters:**

• **board:** The current state of the game board.

• **Returns:** An integer representing the evaluation score of the game state.

**Main Class:**

**Overview:**

The Main class acts as the entry point for the program, providing a demonstration of the **MyPlayer** class's functionality within a game environment.

**Methods:**

**main(String[] args)**

• **Description**: The main method orchestrates the execution of the program. It initializes the game board, creates player objects, and manages the game loop until a terminal state is reached.

• **Parameters:**

• **args:** Command-line arguments (not utilized in this program).

• **Behavior:**

1. Initializes the game board with a specified number of rows and columns.

2. Constructs two player objects using the MyPlayer class, each assigned a different player letter (RED or YELLOW) and a maximum search depth for the minimax algorithm.

3. Iterates through a game loop wherein each player selects their moves using the minimax algorithm with alpha-beta pruning.

4. Continues the game loop until the game reaches a terminal state (e.g., a player wins or the board is full).

5. Prints the final state of the game and terminates execution.

This comprehensive documentation provides detailed insights into the structure, functionality, and behavior of both the MyPlayer and Main classes within the context of a game environment. It covers the constructors, methods, parameters, return types, and algorithms utilized in the program, facilitating a clear understanding of its implementation.

**Compilation:**

To compile the provided programs, you need to have a **Java Development Kit (JDK)** installed on your system. Follow these steps to compile the programs:

1. **Open a Terminal or Command Prompt**: Navigate to the directory containing the Java files (MyPlayer.java and Main.java).

2. **Compile the Programs**: Use the **javac** command to compile both Java files:

**javac MyPlayer.java Main.java**

This command compiles both MyPlayer.java and Main.java and generates corresponding bytecode files (MyPlayer.class and Main.class).

3. **Verify Compilation**: Check for any compilation errors displayed in the terminal. If there are no errors, the compilation is successful.

**Running:**

Once the programs are successfully compiled, you can run the main class Main to execute the demonstration of the MyPlayer class within a game environment. Follow these steps to run the programs:

1. **Execute the Main Class**: Use the java command to run the Main class:

**java Main**

This command executes the Main class, initiating the game simulation.

2. **Follow On-Screen Instructions**: The program prompts the players to input their moves. The MyPlayer class automatically selects its moves using the minimax algorithm with alpha-beta pruning. The game progresses until a terminal state is reached.

**Output:**

During the execution of the programs, you can observe the following output:

• **Game Progress**: The program displays the current state of the game board, indicating the positions of the players' pieces after each move.

• **Player Moves:** Information regarding the moves made by each player is printed on the console. For example, "Player RED, enter column (0-6):" prompts the player with the RED color to input their move column.

• **Game Outcome**: Upon reaching a terminal state (e.g., a player wins or the board is full), the program prints a message indicating the end of the game. For example, "Game over!" signifies the conclusion of the game simulation.

• **Error Messages:** If any errors occur during program execution (e.g., invalid input or unexpected behavior), relevant error messages are displayed in the console to assist in troubleshooting.

**Summary:**

The provided documentation outlines the steps for compilation, running, and expected output of the programs. By following these instructions, users can successfully compile and execute the programs to observe the demonstration of the **MyPlayer class** utilizing the **minimax algorithm** with **alpha-beta pruning** in a game environment. Additionally, the documentation provides insights into interpreting the output and handling potential errors during program execution.