**Name: Rukh-e-Zahra**

**Explanation of The Code:**

* **Purpose:**

A variation of the depth-first search (DFS) method called depth-limited search restricts the search tree's depth to a predetermined degree. Among the goals of using depth-limited search to expedite decision-making are:  
  
**Efficiency:**

By limiting the search to a specific depth, depth-limited search lowers the amount of processing resources needed. This can make it possible, particularly in games with huge state spaces, to explore more game states in an acceptable amount of time.

**Faster Decisions:**

By concentrating on a small number of potential movements in the future, depth-limited search can aid in faster decision-making in real-time or turn-based games. This is essential when making a decision in a constrained amount of time.

* **Evaluation Function:**

A heuristic for estimating a game state's desirability is an evaluation function; it is particularly useful in situations when reaching terminal states in a fair amount of time is not possible. The evaluation function for the Red-Blue Nim game could be created using a number of different criteria, including:

**Number of Marbles:**

The function could determine how many red and blue marbles are left. For instance, if there are less marbles left, it could be deemed less beneficial as it could result in the game ending sooner.

**Piles Balance:**

It would be helpful to evaluate how the red and blue piles are balanced. Depending on the game strategy, an imbalance where one pile is noticeably larger may be favored or disfavored.

**Winning Probability:**

Based on the current situation, the evaluation function might calculate the probability of winning. For example, the function may rate a situation as highly favorable if the human player is about to make a move that would result in victory.

**Future Moves:**

Assessing the current state might be aided by examining potential future actions and their results. Positive evaluation would result from the current state pushing the opponent into a losing position or other more advantageous future states.

A simple example evaluation function might be:

def evaluate\_state(red, blue):

# Higher score if the opponent has fewer marbles

return (max\_red - red) + (max\_blue - blue)

* **Extra Credit Details:**

Fewer marbles signify a more difficult level of the game where fewer moves are needed to reach the finish, lower numbers typically translate into greater scores.

Assuming that the endgame (fewer marbles) is closer to a winning condition, this function is meant to give priority to game conditions that move the game closer to its conclusion.

We can boost performance and strategic depth in the game by incorporating these ideas to make decision-making more effective and efficient.

**End of Game:**

* **Human Wins:**

If the human player makes a move that results in the red and blue piles being empty at the conclusion of their turn, they win the game. After the human moves in the game loop, this is verified:

if red <= 0 and blue <= 0:

print("You win!")

break

* **Computer Wins:**

If, during its turn, the computer makes a move that causes the red and blue piles to be empty, the game is over and the computer has won. After the computer moves in the game loop, this is verified:

if red <= 0 and blue <= 0:

print("Computer wins!")

break

The game loop ends and the result (win or loss) is printed in both scenarios.

**Scoring Calculation:**

The number of marbles left at the end of the game could be used to determine the score. As an illustration:  
  
**Human's Final Score:**

The amount of marbles left in the computer's piles could determine the score if the human prevails.

**Computer's Final Score:**

The amount of marbles left in the human's heaps could determine the computer's score if it prevails.

if red <= 0 and blue <= 0:

if human\_turn:

print(f"Final Score: {computer\_red + computer\_blue} (Computer's remaining marbles)")

else:

print(f"Final Score: {red + blue} (Human's remaining marbles)")

**Implementation Details:**

To make the game easier to comprehend and maintain, it is divided into various components. The many modules and their duties are broken down as follows:

* **Command line praising:**

It typically involves using the argparse module or similar to parse arguments and initialize game settings.

* **Game mechanics:**  
    
  **Goal:**

The fundamental guidelines and principles of the game are covered in this module, along with how the game progresses and concludes.

Implementation: The methods print\_piles(), get\_player\_move(), and get\_computer\_move() as well as the main() function take care of this in the code that is provided. These procedures control how the game is played and how turns are transitioned between.

* **Human and Computer Moves:**

**Goal:**

This module manages the computer's and the player's input and decision-making processes.  
Execution:

**Human Moves:**

In response to a player's move, the get\_player\_move() function asks the user for input, verifies it, and modifies the game state.

**Computer Moves:**

The get\_computer\_move() function modifies the game state and chooses the computer's move using a straightforward random strategy.

* **AI Decision-Making with Minimax and Alpha-Beta Pruning**:

**Goal:**

Using increasingly complex AI decision-making techniques, like the Minimax algorithm and Alpha-Beta pruning, is the goal of this module. To find the best course of action, these methods are employed to investigate game trees and assess prospective moves.

**Minimax Algorithm:**

This algorithm uses a heuristic function to assess the value of game situations while investigating potential movements down to a specific depth. It seeks to reduce the worst-case scenario's potential loss.

**Alpha-Beta Pruning:**

By removing branches from the game tree that are not necessary to investigate because they have no bearing on the outcome, this optimization method increases the Minimax algorithm's efficiency.

Here’s a brief outline of how you might integrate these algorithms:

def minimax(depth, is\_maximizing):

if game\_over():

return evaluate\_game\_state()

if is\_maximizing:

best\_value = -float('inf')

for move in possible\_moves():

value = minimax(depth - 1, False)

best\_value = max(best\_value, value)

return best\_value

else:

best\_value = float('inf')

for move in possible\_moves():

value = minimax(depth - 1, True)

best\_value = min(best\_value, value)

return best\_value

* **Sample Game Walkthrough**

**Initial Setup:**

 Red marbles **:** 10

 Blue marbles **:** 15

**Game Start:**

#### **Human’s Turn**

**Prompt:**

Enter your move (1-10 red, 1-15 blue):

**Human Input:**

2 red 3 blue

The human decides to remove 2 red marbles and 3 blue marbles.

**Game State Update:**

**Red marbles left:**

10 - 2 = 8

**Blue marbles left:**

15 - 3 = 12

* **Game Output:**

**Current piles:**

8 red, 12 blue

#### **Computer’s Turn**

* **Computer's Decision:**

The computer randomly decides to remove 1 red marble and 2 blue marbles.

* **Game State Update:**

**Red marbles left:** 8 - 1 = 7

**Blue marbles left:** 12 - 2 = 10

**Game Output:**

* Computer removes 1 red and 2 blue marbles.
* Current piles: 7 red, 10 blue

#### **Human’s Turn**

* **Prompt:**

Enter your move (1-7 red, 1-10 blue):

* **Human Input:**
  + 1. red 4 blue

The human decides to remove 1 red marble and 4 blue marbles. And soo on…….

**Summary:**

In this game, the quantity of marbles in both piles decreased as the person and machine took turns moving in different directions. The human player won the game by making both piles empty in their final move. The game loop ran until both piles were emptied.

This walkthrough shows how moves impact the game state and how the game ends when both piles are exhausted, illustrating the interplay between the human and machine player throughout a typical gaming session.