Assignment 2

Game Playing Problem

Max possible score:

- 4308: 100 Points [+20 Points EC]
- 5360: 100 Points [+20 Points EC]

Task 1

Max: [4308: 100 Points, 5360: 100 Points]

Your task is to build an agent to play a modified version of nim (called red-blue nim against a human player). The game consists of two piles of marbles (red and blue). On each players turn they pick a pile and remove one marble from it. If on their turn, either pile is empty then they win. The amount they win is dependent on the number of marbles left (2 for every red marble and 3 for every blue marble). So if on the computer player turn, it has 0 red marbles and 3 blue marbles, it wins 9 points.

Your program should be called red_blue_nim and the command line invocation should follow the following format:

red_blue_nim.py <num-red> <num-blue> <first-player> <depth>

- <num-red> and <num-blue> are required. (Number of red and blue marbles respectively)
- <first-player> can be
  - computer - computer starts the game followed by human [default option if <first-player> is not given]
  - human - human starts the game followed by computer
- <depth> only used if depth limited search (Extra Credit) is implemented.

On Computer turn, the program should use MinMax with Alpha Beta Pruning to determine the best move to make and perform the move.
On Human turn, the program should use a prompt to get the move from the human user and perform the move. The program should alternate between these turns till the game ends (when the players run out of either red or blue marbles). Once the game ends, calculate who won and their final score and display it to the user.

Extra Credit (20 Points):

If your program determines computer move by using depth limited MinMax search with alpha beta pruning then you will be given 20 points extra credit. You will need to come up with a eval function to use with the program also. Please submit a text file describing the reasoning behind your eval function for full credit.

How to submit

Implementations in C, C++, Java, and Python will be accepted. Points will be taken off for failure to comply with this requirement unless previously cleared with the Instructor.

Create a ZIPPED directory called <net-id>_assmt2.zip (no other forms of compression accepted, contact the instructor or TA if you do not know how to produce .zip files).

The directory should contain the source code for the task (no need for any compiled binaries). Each folder should also contain a file called readme.txt, which should specify precisely:

- Name and UTA ID of the student.
- What programming language is used for this task. (Make sure to also give version and subversion numbers)
- How the code is structured.
- How to run the code, including very specific compilation instructions, if compilation is needed. Instructions such as "compile using g++" are NOT considered specific if the TA needs to do additional steps to get your code to run.
    - If your code will run on the ACS Omega (not required) make a note of it in the readme file.
- Insufficient or unclear instructions will be penalized.
- **Code that the TA cannot run gets AT MOST 75% credit (depending on if the student is able to get it to run during a Demo session)**.

```
/*
Assignment 2 - Game Playing Problem
--------------------------------------------------------------------------
-----------------
INFO:
--------------------------------------------------------------------------
-----------------
 The code provided is a Java program that simulates a game called Red
Blue Nim.
 It is a two-player game where the players take turns removing marbles
from two piles of marbles, one pile of red marbles and one pile of blue
marbles.
 The winner of the game is the player who takes the last marble, and the
score is calculated based on the number of marbles taken by each player
during the game.

 The program uses the MinMax algorithm with alpha-beta pruning to find
the best move for the computer player.
 The program takes command line arguments to specify the number of red
and blue marbles, whether the computer player goes first, and the depth
of the MinMax algorithm.
 If the computer player does not go first, the program prompts the human
player to enter their move using the console.

 The program consists of the following methods:
        - main: The entry point of the program. Parses command line
arguments and plays the game.
        - parseArgs: Parses the command line arguments and sets the
values of red, blue, turnComputer, and depth.
        - play_red_blue_nim: Plays the game using a while loop that
continues until there are no more marbles left.
 -> In each iteration, it alternates between the computer player and the
human player, and updates the number of marbles left and the score
accordingly.
 -> If it is the computer player's turn, it calls the MinMax algorithm to
find the best move.
 -> If it is the human player's turn, it prompts the user to enter their
move using the console.

 The program uses the Scanner class from the java.util package to read
input from the console.
 It also uses printf to print formatted strings to the console.

 Overall, the program demonstrates how to implement the MinMax algorithm
with alpha-beta pruning to solve a simple game, and how to parse command
line arguments and read input from the console in Java.
*/

import java.util.Scanner; // import Scanner class from java.util package

public class red_blue_nim { // define a public class called red_blue_nim1

    // define static variables red, blue, turnComputer and depth
    static int redMarble;
```

```
    static int blueMarble;
    static boolean turnComputer;
    static int depth;

    public static void main(String[] args)  // main method, entry point
of the program
    {
        System.out.println("        ASSIGNMENT 2     \n------------------
-----------\n***** RED BLUE NIM GAME ***** \n----------------------------
-");
        parseArgs(args);  // parse command line arguments
        play_red_blue_nim(); // play the game
    }

    static void parseArgs(String[] args) // method to parse command line
arguments
    {
        redMarble = Integer.parseInt(args[0]); // extract the value of
red from the first command line argument
        blueMarble = Integer.parseInt(args[1]);     // extract the value
of blue from the second command line argument
        turnComputer = (args.length >= 3 && args[2].equals("human")) ?
false : true;   // check if the third command line argument is "human" to
determine if computer plays first
        depth = (args.length >= 4) ? Integer.parseInt(args[3]) : -1; //
set the value of depth to the fourth command line argument, or to -1 if
it does not exist
        if (args.length < 3) { // if no player is specified, print a
message indicating that computer will play first by default
            System.out.println(" -> No player specified. Computer will
play first by default."); }
    }

    static int[] MinMax(int d, boolean computerTurn) // This function
takes two parameters: d, an integer representing the depth of the search
tree,
    // and computerTurn, a boolean representing whether it is the
computer's turn to play or not.
    {
        // Initialize the best move and best score to be returned.
        int[] bestMove = new int[2];
        int bestScore = (computerTurn) ? Integer.MIN_VALUE :
Integer.MAX_VALUE;
        // Initialize the alpha and beta values for alpha-beta pruning.
        int alpha = Integer.MIN_VALUE;
        int beta = Integer.MAX_VALUE;

        if (d == 0 || redMarble == 0 || blueMarble == 0)  // Check if
this is a leaf node or if one of the players has no more moves left.
        {
            bestScore = (computerTurn) ? (2 * redMarble + 3 *
blueMarble) : -(2 * redMarble + 3 * blueMarble); // Calculate the score
for this node.
        }
```

```
        else
        {
            // Loop through all possible moves for the current player.
            for (int i = 0; i <= redMarble; i++)
            {
                for (int j = 0; j <= blueMarble; j++)
                {
                    if (i + j == 0) continue; // Skip if both players are
not making any move.
                    // Calculate the remaining number of red and blue
balls after the move.
                    int tempRed = redMarble - i;
                    int tempBlue = blueMarble - j;
                    int tempScore = (computerTurn) ? Integer.MIN_VALUE :
Integer.MAX_VALUE; // Initialize the temporary score for this move.
                    if (computerTurn)
                    {
                        int[] result = MinMax(d - 1, false); //
Recursively call this function to get the score for the next level of the
search tree.
                        tempScore = 2 * i + 3 * j + result[1];  //
Calculate the score for this move by adding the score for the next level
to the current move's score.
                        // If this move's score is better than the
previous best score, update the best score and best move.
                        if (tempScore > bestScore)
                        {
                            bestScore = tempScore;
                            bestMove[0] = i;
                            bestMove[1] = j;
                        }

                        alpha = Math.max(alpha, bestScore); // Update
alpha for alpha-beta pruning.
                        if (beta <= alpha) break; // If beta is less than
or equal to alpha, break the loop and do alpha-beta pruning.
                    }
                    else
                    {
                        int[] result = MinMax(d - 1, true);  //
Recursively call this function to get the score for the next level of the
search tree.
                        tempScore = -(2 * i + 3 * j + result[1]); //
Calculate the score for this move by subtracting the score for the next
level from the current move's score.
                        if (tempScore < bestScore) // If this move's
score is better than the previous best score, update the best score and
best move.
                        {
                            bestScore = tempScore;
                            bestMove[0] = i;
                            bestMove[1] = j;
                        }
```

```
                        beta = Math.min(beta, bestScore); // Update beta
for alpha-beta pruning.
                        if (beta <= alpha) break; // If beta is less than
or equal to alpha, break the loop and do alpha-beta pruning.
                    }
                }

                if (beta <= alpha) break; // If beta is less than or
equal to alpha, break the loop and do alpha-beta pruning.
            }

            double random = Math.random(); // Generate a random move to
make Computer choose sub optimal moves to make game interesting.
            if (random <=0.999)
            {
                // make a random move
                int redMove = 0;
                int blueMove = 0;
                while (redMove + blueMove == 0 || redMove > redMarble ||
blueMove > blueMarble) {
                    redMove = (int)(Math.random() * (redMarble + 1));
                    blueMove = (int)(Math.random() * (blueMarble + 1));
                }
                bestMove[0] = redMove;
                bestMove[1] = blueMove;
            }
        }
        return new int[]{bestMove[0], bestMove[1], bestScore}; // Return
the best move along with its score.
    }

    static void play_red_blue_nim() // method to play the game
    {
        int turn = (turnComputer) ? 1 : 0; // initialize turn to 1 if
computer is playing first, or to 0 if human is playing first
        int score = 0;  // initialize score to 0
        int winner = -1;  // initialize winner to -1
        while (redMarble > 0 && blueMarble > 0)
        {
            if (turn == 1) // computer's turn
            {
                int[] bestMove = MinMax(depth, true); // find the best
move for the computer using the MinMax algorithm with depth 'depth'
                int redMove = bestMove[0]; // extract the number of red
marbles to take from the best move
                int blueMove = bestMove[1];   // extract the number of
blue marbles to take from the best move
                redMarble -= redMove;      // update the number of red
marbles left by subtracting the red marbles taken
                blueMarble -= blueMove; // update the number of blue
marbles left by subtracting the blue marbles taken
                int roundScore = 2 * redMove + 3 * blueMove; // calculate
the score for the round based on the number of marbles taken
```

```
                score += roundScore;  // update the total score by adding
the score for the current round
                System.out.printf("\n -> Computer took %d Red & %d Blue
Marbles.\n\n -> Current Score is : %d\n", redMove, blueMove, score); //
print a message indicating the computer's move and the current score
                System.out.printf("\n -> Remaining Marbles are : %d Red
& %d Blue.\n", redMarble, blueMarble);  // print a message indicating the
remaining number of red and blue marbles
                turn = 0;  // set the turn to 0 to indicate it is the
human's turn
            }
            else // human's turn
            {
                Scanner scanner = new Scanner(System.in);  // A Scanner
object is created to read input from the console.
                // Variables are created to hold the human's move and
determine whether it is valid.
                int redMove = -1;
                int blueMove = -1;
                boolean validMove = false;

                while (!validMove) // A while loop is used to ensure that
the human enters a valid move.
                {
                    System.out.print("\n -> Human`s Turn: "); // The
human is prompted to enter their move.
                    // The human's input is read and processed.
                    try
                    {
                        redMove = scanner.nextInt();
                        blueMove = scanner.nextInt();
                        // The move is deemed valid if both the red and
blue moves are non-negative and the total move is greater than 0,
                        // but less than or equal to the number of
remaining marbles.
                        if (redMove >= 0 && blueMove >= 0 && redMove +
blueMove > 0 && redMove <= redMarble && blueMove <= blueMarble)
                        {
                            validMove = true;
                        }
                        else
                        {
                            System.out.println(" !! Invalid move. \n ->
Please try again.");
                        }
                    } catch (Exception e)
                    {
                        // If the human enters an invalid input format,
they are prompted to enter a valid input.
                        System.out.println(" !! Invalid input format. \n
-> Please enter two non-negative integers separated by a space.");
                        scanner.nextLine();
                    }
                }
```

```
                // The number of red and blue marbles remaining is
updated based on the human's move.
                redMarble -= redMove;
                blueMarble -= blueMove;
                // The round score is calculated and added to the total
score.
                int roundScore = 2 * redMove + 3 * blueMove;
                score += roundScore;
                // The human's move, total score, and remaining marbles
are printed to the console.
                System.out.printf("\n -> Human took %d Red & %d Blue
Marbles. \n \n -> Current Score is: %d\n", redMove, blueMove, score);
                System.out.printf("\n -> Remaining Marbles are : %d Red
& %d Blue.\n", redMarble, blueMarble);
                turn = 1;  // It is now the computer's turn.
            }
        }
        if (redMarble == 0) // If there are no more red marbles left, the
computer wins and adds 3 times the number of blue marbles to its score.
        {
            winner = 1;
            score += 3 * blueMarble;
        }
        else   // Otherwise, the human wins and adds 2 times the number
of red marbles to their score.
        {
            winner = 0;
            score += 2 * redMarble;
        }
        System.out.printf("\n -> Game is Over !! \n\n -> %s WON the Game
with a Score of %d.\n", (winner == 1) ? "Computer" : "Human", score); //
Print out the winner of the game and their final score.
        System.out.println(" \nExiting the Game /-/-\n");
    }
    static int eval() // This method calculates the value of the current
game state for the computer player.
    {
        // Calculate the total value of red and blue marbles.
        int redValue = redMarble * 2;
        int blueValue = blueMarble * 3;
        return (redValue + blueValue) * ((turnComputer) ? 1 : -1); //
Multiply the total value by 1 if it's the computer's turn or -1 if it's
the human's turn.
    }
}
```