Digital Empowerment Pakistan

Python Programming

Submitted By:

Farwa Rubab

Red-Blue Nim Game

1.Introduction:

The Red-Blue Nim Game is a strategic two-player game involving piles of red and blue marbles. The game has two versions: Standard and Misère. In both versions, players take turns removing marbles from the piles, and the game's outcome depends on the rules specific to the version being played. Implementing this game in Python involves creating an interactive command-line interface, applying artificial intelligence using the MinMax algorithm with Alpha Beta Pruning, and managing game mechanics.

2. Game Rules

2.1 Standard Version

In the Standard Version, players lose if either pile is empty on their turn.

2.2 Misère Version

In the Misère Version, players win if either pile is empty on their turn.

2.3 Scoring

Scoring is determined by the marbles left at the end of the game:

- Each red marble left: 2 points.
- Each blue marble left: 3 points.

3. Command Line

3.1 Usage

To start the game, use the command line. Here's an example of how to invoke the game:

python red_blue_nim.py <num-red> <num-blue> <version> <first-player> [<depth>]

3.2 Parameters

<num-red>: Number of red marbles.

<num-blue>: Number of blue marbles.

<version>: 'standard' (default) or 'misere'.

<first-player>: 'computer' (default) or 'human'.

<depth>: Search depth for AI (optional).

4. Game Flow

4.1 Turn Order

The game alternates between the human and computer players until the game ends.

4.2 Human Move

The program prompts the human player for their move and validates the input. For example:

Your move: Enter the number of red marbles to take: 1

Enter the number of blue marbles to take : 2

The program checks if the move is valid (i.e., not exceeding the available marbles).

4.3 Computer Move

The program determines the computer's move using the MinMax algorithm with Alpha Beta Pruning to ensure optimal decision-making.

5. MinMax Algorithm

5.1 Overview

The MinMax algorithm is used to optimize decision-making by exploring all possible moves and their outcomes, aiming to minimize the possible loss for a worst-case scenario.

5.2 Alpha Beta Pruning

Alpha Beta Pruning improves the efficiency of the MinMax algorithm by eliminating branches in the game tree that do not need to be explored, thus reducing the computation time.

5.3 Move Ordering (Standard)

In the Standard Version, the moves are ordered as:

Pick 2 red marbles.

Pick 2 blue marbles.

Pick 1 red marble.

Pick 1 blue marble.

5.4 Move Ordering (Misère)

In the Misère Version, the moves are ordered inversely:

Pick 1 blue marble.

Pick 1 red marble.

Pick 2 blue marbles.

Pick 2 red marbles.

6. Depth Limited Search (Extra Credit)

6.1 Purpose

Depth-limited search is implemented to enable faster decision-making by limiting the depth of the game tree explored by the MinMax algorithm.

6.2 Evaluation Function

A heuristic evaluation function is used to evaluate nonterminal game states, providing an estimate of the desirability of a position.

6.3 Extra Credit Details

For extra credit, describe the reasoning behind the evaluation function, including how it balances immediate gains versus long-term strategy.

7. End of Game

7.1 Game Over Conditions

The game ends when either pile is empty.

7.2 Scoring Calculation

The final score is calculated based on the remaining marbles:

Each red marble left: 2 points.

Each blue marble left: 3 points.

8. Implementation Details

8.1 Modules

The game is implemented with the following modules:

Command-line parsing: Handles user input and game configuration.

Game mechanics: Manages the game state and rules.

Human and computer moves: Facilitates player turns and move validation.

AI decision-making with MinMax and Alpha Beta Pruning: Implements the AI logic for optimal play.

9. Demonstration

9.1 Walkthrough

Here's a walkthrough of a sample game:

Game Setup:

python red_blue_nim.py 5 3 standard human 3

This command starts a game with 5 red marbles, 3 blue marbles, in standard mode, with the human player going first and the AI using a search depth of 3.

Turn 1 - Human Move:

- The human player chooses to take 1 red marble and 1 blue marble.
- Red marbles left: 4
- Blue marbles left: 2

Turn 2 - Computer Move:

- The AI calculates the optimal move using the MinMax algorithm with Alpha Beta Pruning.
- The computer strategically decides to take 2 blue marbles.
- Red marbles left: 4
- Blue marbles left: 0

Game Over:

- The game ends since the blue pile is empty.
- Final score is calculated based on remaining marbles.

This score is derived from the remaining marbles:

- 4 red marbles left: 4 * 2 = 8 points.
- 0 blue marbles left: 0 * 3 = 0 points.

Explanation by code:

Command-line Parsing Module (`cli.py`)

```
import argparse

def parse_arguments():

    parser = argparse.ArgumentParser(description="Red-Blue Nim Game")

    parser.add_argument('num_red', type=int, help='Number of red marbles')

    parser.add_argument('num_blue', type=int, help='Number of blue marbles')

    parser.add_argument('--version', type=str, choices=['standard', 'misere'],
default='standard', help='Game version')

    parser.add_argument('--first-player', type=str, choices=['computer', 'human'],
default='computer', help='First player')

    parser.add_argument('--depth', type=int, help='Search depth for AI (optional)')

    return parser.parse_args()
```

Game Mechanics Module (`game.py`)

```
class Game:

    def __init__(self, num_red, num_blue, version, first_player):

        self.num_red = num_red

        self.num_blue = num_blue

        self.version = version

        self.current_player = first_player

    def is_game_over(self):

        return self.num_red == 0 or self.num_blue == 0

    def apply_move(self, red_taken, blue_taken):
        if red_taken <= self.num_red and blue_taken <= self.num_blue:
            self.num_red -= red_taken
            self.num_blue -= blue_taken
            self.current_player = 'computer' if self.current_player == 'human' else 'human'
        else:
            raise ValueError("Invalid move")
    def get_score(self):
        return (self.num_red * 2) + (self.num_blue * 3)
```

Human Player Module (`human.py`)

```
class HumanPlayer:
    def get_move(self):
        while True:
            print(f"Your turn. Red marbles: {game.num_red}, Blue marbles: {game.num_blue}")
            try:
                num_red = int(input("Enter number of red marbles to take: "))
                num_blue = int(input("Enter number of blue marbles to take: "))
                if num_red <= game.num_red and num_blue <= game.num_blue and (num_red > 0
or num_blue > 0):
                    return num_red, num_blue
                else:
                    print("Invalid move. Try again.")
            except ValueError:
                print("Invalid input. Please enter integers.")
```

Computer Player Module (`ai.py`)

```
class ComputerPlayer:
    def __init__(self, depth=None):
        self.depth = depth if depth else float('inf')
    def minmax(self, game, depth, alpha, beta, maximizing_player):
        if depth == 0 or game.is_game_over():
            return self.evaluate(game), None
        best_move = None
        if maximizing_player:
            max_eval = float('-inf')
            for move in self.get_possible_moves(game):
                game.apply_move(*move)
                eval, _ = self.minmax(game, depth - 1, alpha, beta, False)
                game.undo_move(*move)
                if eval > max_eval:
                    max_eval = eval
                    best_move = move
                alpha = max(alpha, eval)
                if beta <= alpha:
                    break
            return max_eval, best_move
        else:
            min_eval = float('inf')
            for move in self.get_possible_moves(game):
                game.apply_move(*move)
                eval, _ = self.minmax(game, depth - 1, alpha, beta, True)
                game.undo_move(*move)
                if eval < min_eval:
                    min_eval = eval
                    best_move = move
                beta = min(beta, eval)
                if beta <= alpha:
                    break
            return min_eval, best_move
    def get_possible_moves(self, game):
        if game.version == 'standard':
            return [(2, 0), (0, 2), (1, 0), (0, 1)]
        else:
            return [(0, 1), (1, 0), (0, 2), (2, 0)]
    def evaluate(self, game):
        return game.get_score()
    def get_move(self, game):
         move = self.minmax(game, self.depth, float('-inf'), float('inf'), True)
        return move
```

Main Program (red_blue_nim.py)

```
from cli import parse_arguments

from game import Game

from human import HumanPlayer

from ai import ComputerPlayer

def main():

    args = parse_arguments()

    game = Game(args.num_red, args.num_blue, args.version, args.first_player)

    human = HumanPlayer()

    computer = ComputerPlayer(args.depth)

    while not game.is_game_over():

        if game.current_player == 'human':

            red_taken, blue_taken = human.get_move()

        else:

            red_taken, blue_taken = computer.get_move(game)

        try:

            game.apply_move(red_taken, blue_taken)

        except ValueError as e:

            print(e)

            continue

        print(f"Current state: {game.num_red} red marbles, {game.num_blue} blue marbles")
    print(f"Game over! Final score: {game.get_score()}")

if __name__ == "__main__":

    main()
```

Detailed Explanation

1. **Command-line Parsing**:
   - The `parse_arguments` function is called to parse the command-line arguments.
   - The parsed arguments are used to initialize the game state.
2. **Game Loop**:
   - The game loop continues until the game is over.
   - Depending on the current player, either the human or computer makes a move.
   - The `apply_move` method is called to update the game state based on the move.
   - The current state of the game is printed after each move.
3. **End of Game**:
   - When the game is over, the final score is calculated and displayed.