Nim Game Analyser

Dakshesh

October 2, 2025

Abstract

This report outlines the design and implementation of a command-line tool for analyzing the game of Nim. The application determines if a given game state is a winning or losing position, provides the optimal move for winning positions, and caches results in an SQLite database for efficiency. The core analysis is based on Bouton's theorem, utilizing the Nim-sum (bitwise XOR sum) of the heap sizes. The project is modular, separating core logic, database handling, and user interaction, resulting in a clean and maintainable codebase.

Contents

1 Introduction

The game of Nim is a two-player mathematical strategy game where players take turns removing items from distinct heaps. The player to take the last item wins. The game is notable for having a complete mathematical theory, which allows any game state to be definitively classified as either a ”winning” or ”losing” position.

This project implements a Python tool that leverages this theory to provide players with an instant analysis of any game state. The primary goals of the project were to:

- Implement the core Nim game logic based on the Nim-sum calculation.
- Develop a command-line interface (CLI) for user input.
- Integrate a database to cache analysis results, improving performance for repeated queries.
- Structure the application into logical, decoupled modules.

2 Project Architecture

The application is logically divided into four Python modules, each with a distinct responsibility. This separation of concerns makes the system easier to understand, test, and extend.

2.1 Game Logic (`game_logic.py`)

This module forms the core of the analyser. It contains the mathematical functions necessary to evaluate a Nim game state. It is responsible for:

- **Calculating the Nim-Sum:** Computing the bitwise XOR sum of all heap sizes. A Nim-sum of zero indicates a losing position, while any other value indicates a winning one.
- **Determining the Optimal Move:** For a winning position, this module identifies a move that will leave the opponent in a losing position (i.e., a state with a Nim-sum of zero).

2.2 Database Management (`database.py`)

To avoid re-calculating the analysis for previously seen game states, this module provides a caching layer using an SQLite database. Its key functions are:

- Setting up the database and a ‘game$_s$$tates$‘$tableonfirstrun$.
- Storing the results of a new analysis, including the state, Nim-sum, and optimal move.
- Querying the database to check for a pre-existing analysis of a given state.

2.3 Main Entrypoint (`main.py`)

This module serves as the user-facing part of the application. It orchestrates the program flow:

- It uses Python’s ‘argparse‘ library to parse the heap sizes provided as command-line arguments.
- It first queries the database for a cached result.
- If no cached result is found, it calls the ‘game$_l$$ogic$‘$moduletoperformanewanalysis$.
- The result (either new or cached) is then formatted using the ‘utils‘ module and printed to the console.

2.4 Utilities (`utils.py`)

This small module contains helper functions, primarily for formatting the final analysis into a human-readable string for display.

3 Usage

The tool is run from the command line, with the sizes of the heaps passed as space-separated integer arguments.

3.1 Example Command

To analyze a game with heaps of size 10, 12, and 3, the user would run:

```
python main.py 10 12 3
```

3.2 Sample Output

For a winning position, the output clearly states the optimal move:

```
--- Nim Game Analysis ---
Game State: [10, 12, 3]
Nim-Sum (XOR Sum): 5
Position Type: Winning
Optimal Move: Take 7 from heap 1
```

If the position is a losing one, the tool indicates that no winning move is available:

```
--- Nim Game Analysis ---
Game State: [10, 12, 6]
Nim-Sum (XOR Sum): 0
Position Type: Losing
Optimal Move: No winning move available.
```

4 Conclusion

The Nim Game Analyser successfully meets all project goals. It provides accurate, instantaneous analysis of game states through a simple command-line interface. The modular architecture makes the code robust, and the implementation of database caching provides a significant performance enhancement for recurring states.

Possible future improvements could include the development of a graphical user interface (GUI) for interactive gameplay or extending the logic to support other impartial games based on the Sprague-Grundy theorem.