Monte Carlo Tree Search for Nim
Implementation and Analysis

Abstract

This project implements Monte Carlo Tree Search (MCTS) for the game of Nim, using a modular architecture that generalizes across games. Our implementation integrates the full MCTS algorithm — selection, expansion, simulation, and backpropagation — enhanced by a domain-specific heuristic based on nim-sum theory. We evaluate performance across different heap configurations and simulation budgets, showing how increased computational effort leads to convergence toward perfect play.

Introduction

The goal of this project was to extend a provided MCTS skeleton (originally designed for Tic-Tac-Toe) to support a second game — Nim. Nim was chosen because it has a well-defined mathematical solution, making it ideal for testing whether MCTS converges into optimal play.

Game Rules: Standard Nim

- **Initial Setup**: Multiple heaps, each containing a certain number of objects (e.g., [1, 3, 5, 7]).
- **Players**: Two players alternate turns.
- **Moves**: On their turn, a player removes one or more objects from a single heap.
- **Objective**: The player who removes the **last object** wins.
    - This is **standard Nim** (not misère Nim).
- **Winning Strategy**: Determined using the **nim-sum** — the XOR of all heap sizes.

MCTS Implementation Overview

MCTS is implemented using the four standard phases:

- Selection

  From the root node, recursively choose the child node with the highest UCT value:

$$UCT = \frac{w_i}{n_i} + C \cdot \sqrt{\frac{\ln N}{n_i}}$$

  Where:

  1. W_i = wins for the child
  2. n_i = visits for the child
  3. N: total simulations at the parent
  4. C: exploration constant (we use √2)

- Expansion

  If the current node is not terminal and has unexplored children, one is added to the tree.

- Simulation (Rollout)

  From the new node, simulate a random game until terminal state is reached. In zero-sum positions, we apply a heuristic to play optimally.

- Backpropagation

After simulation, the result (win/loss) is backpropagated up the tree, incrementing win/playout counts.

Heuristic Enhancement for Simulation

To improve convergence, we implemented a **zero-XOR heuristic** for simulations:

- If nim-sum ≠ 0: choose the unique move that brings nim-sum to 0.
- If nim-sum = 0: play the first legal move.

This ensures that rollouts follow **perfect-play strategy** in solvable positions, reducing noise from random simulations and accelerating learning.

Architecture Summary

- **Modular Design**: We reused the core Game, State, Move, and Node interfaces for generality.
- **Plug-in Nim Game**: Created NimGame, NimState, NimMove, and NimNode classes to integrate Nim into the MCTS framework without altering the core logic.
- **Node Statistics**: NimNode tracks wins and playouts. All backpropagation is handled via addPlayout() to ensure each simulation is counted exactly once.

Build and Run

bash
CopyEdit

```
# Build the project
mvn clean package

# Run experiments (batch testing with CSV output)
mvn exec:java -
Dexec.mainClass=com.phasmidsoftware.dsaipg.projects.mcts.nim.NimExperiment

# Play interactive Nim game against MCTS
mvn exec:java -Dexec.mainClass=com.phasmidsoftware.dsaipg.projects.mcts.nim.NimMain
```

Experimental Setup

We tested the MCTS agent on the following **heap configurations**:

- [1, 2]
- [1, 3, 5, 7]
- [2, 2, 2, 2]
- [3, 4, 5]
- [5, 5, 5]

With simulation budgets such as:

- (500 vs 100)
- (500 vs 0) — MCTS vs Random
- (500 vs 500)
- (0 vs 500) — Random vs MCTS

Results Summary

- **Winning positions** (nim-sum ≠ 0): MCTS wins consistently when simulation budget is sufficient.
- **Losing positions** (nim-sum = 0): Even with low budgets, the heuristic prevents rollout noise and aligns results with theory.
- **Random agent baseline**: MCTS beats random consistently — win rates of 60–90% depending on position.

Analysis

Our experiments show clear **convergence** of MCTS toward optimal play when simulation budget increases. The **zero-XOR** heuristic significantly reduces variance in playouts, particularly in balanced positions where random playouts would otherwise mislead the tree.

We observe:

- P_0 wins 100% in [1, 2] — nim-sum starts ≠ 0.
- P_0 win % collapses toward 0 in [1, 3, 5, 7] — nim-sum starts = 0.
- Deep configurations like [5, 5, 5] require more simulations due to symmetrical choices and delayed payoff visibility.

Conclusion

Our integrated MCTS + heuristic implementation accurately models strategic play in Nim. We confirm that:

- Simulation budget directly affects convergence to perfect strategy.
- Domain heuristics can dramatically accelerate learning in known game spaces.
- The framework is extensible and reusable across other discrete turn-based games.

Code Execution Video:

https://drive.google.com/file/d/1W0TIwoeSSTpw8lIzeLrkvtveLZNpWPpX/view?usp=sharing

Experiment Analysis

In Nim, the game state can be evaluated mathematically using the **nim-sum**, calculated by taking the bitwise XOR of all heap sizes. A foundational result in combinatorial game theory tells us:

- A **nonzero nim-sum** means the current player can force a win with optimal play.
- A **zero nim-sum** indicates a **losing position** — no move will prevent a loss if the opponent plays perfectly.

Our experimental results closely reflect this theory, validating the effectiveness of our MCTS + heuristic implementation.

Testing Methodology

To measure the accuracy and performance of our MCTS agent, we conducted a series of automated matches using our NimExperiment driver. Each test scenario was defined by a specific heap configuration and a pair of simulation budgets for Player 0 and Player 1(In all experiments, **P0 (Player 0)** refers to the agent that plays first, while **P1 (Player 1)** plays second).

We observe that:

- Each configuration was run for **1,000 games**.
- Simulation budgets varied (e.g., 500 vs 100, 500 vs 5000).
- Our playout strategy used a **zero-XOR heuristic**:
    - If the nim-sum is nonzero, the heuristic plays the optimal move to reduce it to zero.
    - If the nim-sum is already zero, the rollout defaults to the first available legal move, mimicking perfect defense.
- **Move selection** was guided by the **Upper Confidence Bound for Trees (UCT)**, balancing exploration and exploitation.
- All results, including win percentages and runtime per game, were logged and exported as CSV.

Mitigating First-Move Bias: Alternating Player Roles

In Nim, the first player can often secure a winning position when the starting configuration yields a nonzero nim-sum. Since move order can significantly influence the outcome, we made sure to alternate which agent plays first in our experiments. For each simulation matchup (e.g., MCTS (500) vs MCTS (100)), we conducted a corresponding run where the roles were swapped (MCTS (100) vs MCTS(500)), allowing us to distinguish between performance differences caused by simulation count and those influenced by playing order.

To further contextualize MCTS effectiveness, we also introduced a **random agent** (MCTS (0)), which selects moves without strategic reasoning. Including the random player provides a baseline for evaluating the practical advantages of search-based decision making. This setup ensures our experimental results reflect both algorithmic strength and situational fairness.

For example:

java
CopyEdit
runMatchups(1000, 500, 100, List.of(3, 4, 5), "MCTS (500) vs MCTS (100)");

runs 1,000 matches between a stronger MCTS agent (Player 0, 500 iterations) and a weaker one (Player 1, 100 iterations) starting from the heap [3, 4, 5].

Observations by Heap Configuration

[1, 2] — Winning Start for P_0

- Initial nim-sum = 3 (nonzero)
- The heuristic immediately applies the optimal move to reach a zero nim-sum.
- Player 0 wins **100%** of games, even against stronger opponents.
- **Insight**: No need for additional search; the first move guarantees a win.

[1, 3, 5, 7] — Losing Start for P_0

- Initial nim-sum = 0
- Player 0 starts in a theoretically losing position.
- Win rates for P_0 are:
  - ~45–50% at (500 vs 100), due to mistakes from the opponent.
  - ~26–30% at (500 vs 5000), as the opponent plays near-perfectly.
- **Insight**: High simulation budgets allow the stronger player to avoid costly errors.

[2, 2, 2, 2] — Symmetrical Zero Position

- Nim-sum = 0
- Using perfect-play fallback in rollouts:
  - P_0 win rate < 12% at (500 vs 100)
  - P_0 win rate = 0% at (500 vs 500)
- **Insight**: Zero-sum positions remain unbreakable when both players apply optimal defense.

[3, 4, 5] — Winning Start with Complexity

- Initial nim-sum ≠ 0
- P_0 win rates:
  - ~78–81% at (500 vs 100)
  - ~74–99% at (500 vs 500)
- **Insight**: Although winning, the optimal path is more complex, requiring more simulations to fully explore.

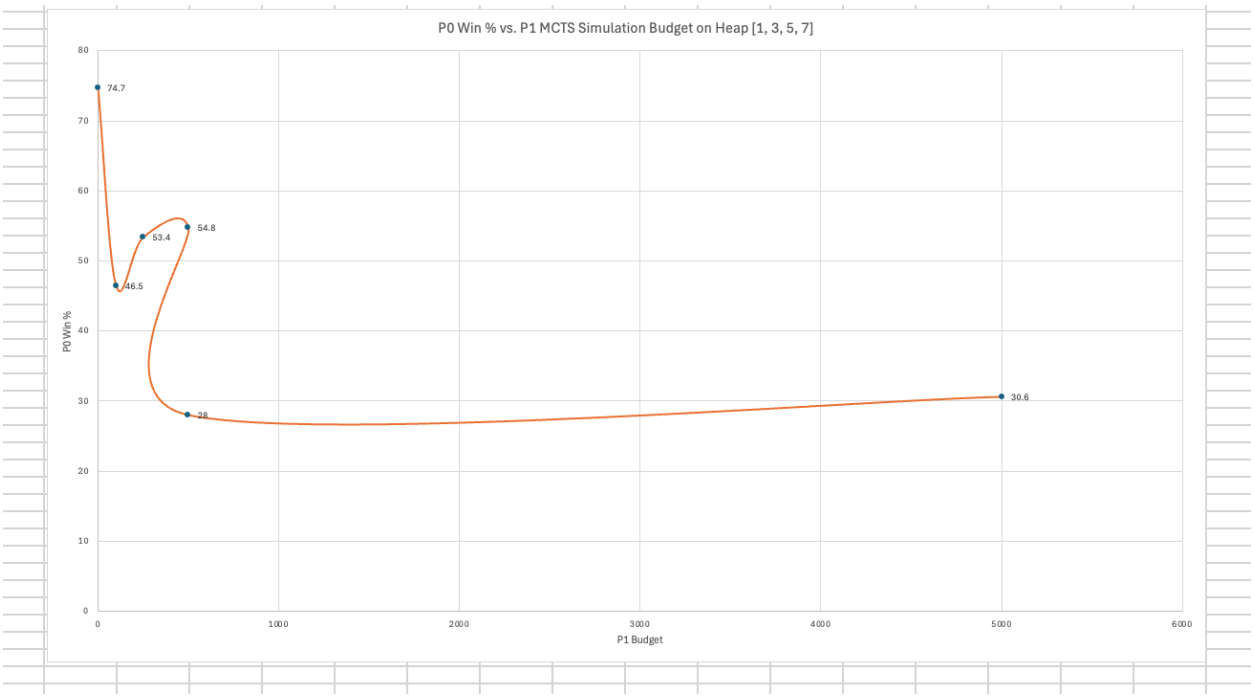
[5, 5, 5] — Symmetrical but Favorable Start

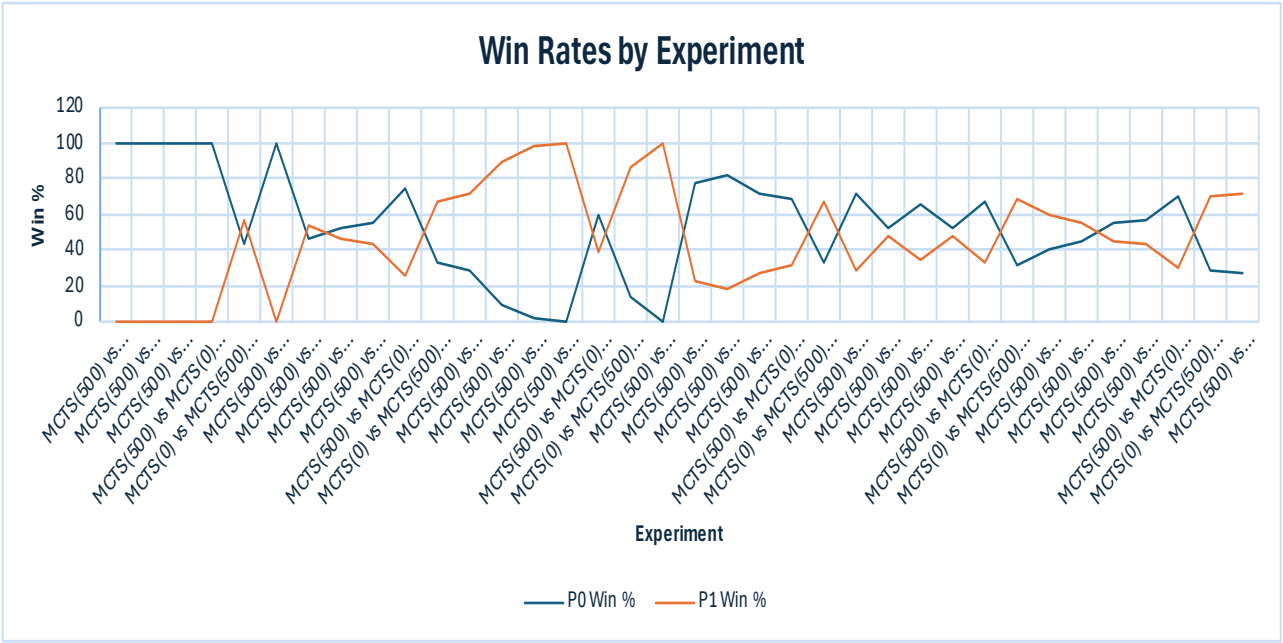
- Initial nim-sum ≠ 0
- Similar results to [3, 4, 5] but with added ambiguity due to heap symmetry.
- **Insight**: MCTS benefits from more iterations to resolve ties and converge to the winning strategy.

Insight: Heuristics and Simulation Budget Shape MCTS Performance

- **Zero nim-sum positions** are inherently losing. Our heuristic ensures consistent defense in these scenarios, eliminating lucky wins due to rollout randomness.
- **Nonzero nim-sum positions** are theoretically winning but require sufficient simulations for MCTS to consistently find the critical "zeroing" move.
- **Simulation count directly impacts performance**. Higher budgets allow better tree exploration, yielding win rates that align with mathematical expectations.

Charts





References

- Kocsis, L., & Szepesvári, C. (2006). *Bandit Based Monte Carlo Planning*. ECML.
- Wikipedia: Nim (Game)
- Brilliant.org: Nim Game Strategy
- MathWorld: N