A Variation of NIM: A Report

Derik Dreher, Sofia Jones, Isayha Raposo

March 30, 2021

Introduction

Our project team has developed a computer player able to play the variation of NIM described in the project outline at a high enough capacity that we consider it to be flawless: The computer player seemingly always wins in all cases except those where winning is not possible through any conceivable combination of moves.

The Solution

Algorithm Overview

Our computer player actually has two levels, basic and advanced. The basic computer player is our original design, which was based on the mathematics/game theory behind NIM. This computer player is programmed in an entirely logical manner (it does not use any sort of modular, recursive algorithm) and is always able to win NIM games in which there are no constrained patterns such as those described in the project outline, as long as it goes first. This is because, in the version of NIM where the player with the last move loses, the player with the first move, when playing optimally, should always win, as long as the NIM-sum of the initial game state is not 0 (otherwise the opposite is assumed).

When constraints such as those specified in the project outline are added to the game, simply ensuring that the NIM-sum of the game state following the computer player's move is 0 (along with implementing some other logic for edge cases, such as when pile sizes are ¡ 2) is not sufficient enough to win each time. For this reason, we created our second design, the advanced computer player.

The advanced computer player is based off of the minimax search algorithm with alpha-beta pruning and memoization. If a move exists that is likely to either keep or put the computer player ahead, the advanced computer player can and will find it, especially in cases where the basic computer player would not.

When playing the game, you are given the choice of playing against either the basic or the advanced computer player. If you choose the latter, in cases where the advanced computer player is unable to find a move (such as cases in which there is no winning move), the basic computer player will be called to make a less-than-optimal move, if it can find one. If neither the basic nor the advanced computer play is able to find a move, a minimal, arbitrary move will be made (1 object will be removed from the first non-empty pile found). This also ensures that the game is actually able to end if the computer player is about to lose.

Game Pseudocode: (when playing against the advanced computer player)

```
get num of piles, pile counts and turn

while game ! over:
  if humans turn:
    get move from human

  if cpus turn:
    if using advanced CPU:
     get choice using minimax
     if choice does not exist:
       get move using basic CPU (using nim-sums and edge-case logic)
       if move does not exist:
         take 1 object from the first non-empty pile found
     else:
       convert choice to move

  execute move

  check 3 conditions to end game:
   if all piles empty:
     game over = true
   elif 3 piles each with 2 objects remaining & all other piles are empty:
     game over = true
   elif 1 pile with 1 object & 1 other pile with 2 objects &  a third pile with 3 objects
     game over = true

  if game over & cpus turn:
    Human wins
  else:
    CPU wins
```

Minimax Algorithm Pseudocode (advanced computer player):

```
choices = []
if cpus turn:
  generate children
  for child in children:
    if child, turn in cache:
      get cache
    else:
      run minimax next iteration
      update cache
    if new choice found:
      add choice
  return max result
```

```
else:
  if new choice found:
    add choice
  return minimum result
```

Algorithm Correctness

Data Structures Employed

1. Tree (Game/Search Tree): To choose an optimal move, a game tree is used. This tree is used to search for the best move and is generated using our *minimax* algorithm described above.
2. Hash Map (Dictionary): The game state blacklist is stored as a dictionary/hash map. The minimax algorithm also uses a cache stored as a dictionary/hash map.
3. Arrays: For game states

Algorithm Complexity Analysis

$O(b^m)$ (maximum search tree depth/sum of number of objects across all piles)