A Variation of Nim

Team E

Data Structures II

Derik Dreher: 230 119 887

Sofia Jones: 230 136 084

Isayha Raposo: 230 133 508

March 31, 2021

Introduction

The following is a report that provides a detailed description and analysis of our team project: a variation of Nim with a working computer player. Nim is a two-player game of strategy that has great significance in the field of combinatorial game theory [1]. In particular, the *Sprague-Grundy theorem* states that "*every impartial game under the normal play convention is equivalent to a one-heap game of Nim, or to an infinite generalization of Nim*" [2] is based upon the (mathematically proven) "normal-play" variation of Nim (where the player to make the last move wins).

This project investigates the "misère" variation (that in which the player to make the last move *loses*) of multi-"heap" (henceforth referred to as *piles*) Nim, with some particular constraints. More specifically, the game is played as follows:

Excerpt from `README.md` *(in project files):*

- The game features n piles where $n > 1$, and each pile initially contains some arbitrary number of objects (> 0)
- Two players take turns removing $y > 0$ objects from a single pile x (both y and x are chosen by the player at each turn), where $y <=$ the number of objects in x
- A player *loses* the game if, *after their turn*, any one of the following is true:
  - All n piles are empty
  - There exists 3 piles each with 2 objects remaining AND all other piles are empty
  - There exists 1 pile with 1 object remaining AND 1 other pile with 2 objects remaining AND a third distinct pile with 3 objects remaining AND all other piles are empty
  - (Optional) Some fourth game state that has been specified as a losing state is reached (in the demonstration lecture, this state was: There exists 2 piles each with 2 object remaining AND another 2 piles each with 1 objects remaining)

The Solution

Instructions

Part I: Running The Game

Excerpt from `README.md` (in project files):

- To run the program, simply run

  ```
  nim_variation.py
  ```

  either using the command `python` on the command line (while in the same directory as the source files), or the IDE of your choice

- For reference, this program was written in Python 3.8.6 and 3.9 (due to having multiple contributors with differing versions), and requires the package `tabulate`, which can be installed using Pip by running

  ```
  pip install tabulate
  ```

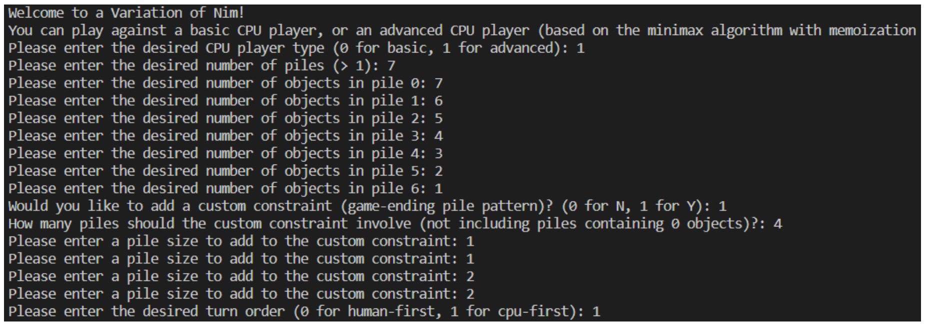
Part II: Playing The Game

The project provides a very simple text-based user interface that allows players to both set up and play a round of the Nim variant described in the introduction above.

Upon running the project, players are greeted and given a choice between playing against a *basic* computer player or an *advanced* computer player. The difference between the two is explained briefly below and in detail in the report section `Project/Algorithm Overview`.

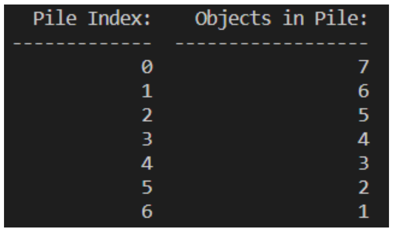
Each and every selection required of the player by the project is made by prompting the player to enter an integer value representing some specific choice. In this case, the project will prompt the player to either enter `0` to play against the basic computer player or `1` to play against the *advanced* computer player. The project will then prompt the player to enter the desired (initial) number of piles, followed by the desired number of object(s) initially within each of said piles.

Following this, the project will ask the player if they would like to add an additional custom constraint (game-ending pile pattern) to the game. If so, the player is prompted to first enter the number of pile(s) the constraint consists of, followed by the number of object(s) in each. To enter the constraint given in the demonstration lecture (the game state consisting of four non-empty piles where two of said piles contain two objects each and the other two piles contain a single object each), the player would enter `4` (for the number of piles the constraint should involve), followed by `2`, then `2`, then `1`, then `1` (for the number of objects in each of the 4 piles). Lastly, the project will ask the player if either they or the computer player should have the first turn:



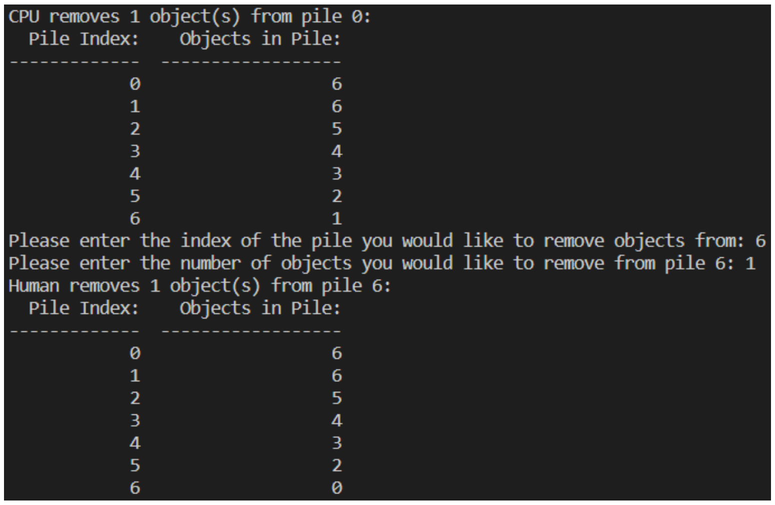
```
Welcome to a Variation of Nim!
You can play against a basic CPU player, or an advanced CPU player (based on the minimax algorithm with memoization
Please enter the desired CPU player type (0 for basic, 1 for advanced): 1
Please enter the desired number of piles (> 1): 7
Please enter the desired number of objects in pile 0: 7
Please enter the desired number of objects in pile 1: 6
Please enter the desired number of objects in pile 2: 5
Please enter the desired number of objects in pile 3: 4
Please enter the desired number of objects in pile 4: 3
Please enter the desired number of objects in pile 5: 2
Please enter the desired number of objects in pile 6: 1
Would you like to add a custom constraint (game-ending pile pattern)? (0 for N, 1 for Y): 1
How many piles should the custom constraint involve (not including piles containing 0 objects)?: 4
Please enter a pile size to add to the custom constraint: 1
Please enter a pile size to add to the custom constraint: 1
Please enter a pile size to add to the custom constraint: 2
Please enter a pile size to add to the custom constraint: 2
Please enter the desired turn order (0 for human-first, 1 for cpu-first): 1
```

Upon selection of the starting player (human or computer), the game will automatically begin, and the project will print a table that represents the initial game state (with pile indexes on the left, and pile contents on the right):



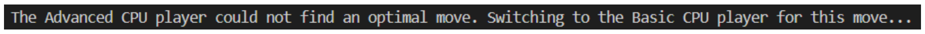
```
  Pile Index:    Objects in Pile:
-------------  ------------------
            0                   7
            1                   6
            2                   5
            3                   4
            4                   3
            5                   2
            6                   1
```

The project will continue to print the game state for each move made by each player, forming a log of past game states and moves. This will continue until either player loses.



```
CPU removes 1 object(s) from pile 0:
  Pile Index:    Objects in Pile:
-------------  ------------------
            0                   6
            1                   6
            2                   5
            3                   4
            4                   3
            5                   2
            6                   1
Please enter the index of the pile you would like to remove objects from: 6
Please enter the number of objects you would like to remove from pile 6: 1
Human removes 1 object(s) from pile 6:
  Pile Index:    Objects in Pile:
-------------  ------------------
            0                   6
            1                   6
            2                   5
            3                   4
            4                   3
            5                   2
            6                   0
```

For some of the computer player's turns, if the player is playing against the *advanced* computer player, the project will indicate that the advanced computer player could not find a move:



```
The Advanced CPU player could not find an optimal move. Switching to the Basic CPU player for this move...
```

The advanced computer player uses the minimax algorithm with memoization and alpha-beta pruning to evaluate game states. Game states are evaluated in a boolean manner (**good** game states are assigned a value of 1; **bad** game states are assigned a value of -1), which means that no **good** game state is better or worse than any other **good** game state, and vice versa. The *advance* computer player will only return future game states that evaluate to 1, and that are within a single move of the current game state. If all game states within a single move of the current game state evaluate to -1, all possible moves are non-optimal, and the computer player's next move can, in theory, be chosen arbitrarily. Thus, in these cases, the basic computer player logic, which is relatively simple, is used to find a move instead.

This leads us to the next section of the project report, which gives a detailed description of the logic/algorithms utilized in both the computer player(s) and the game itself.

Algorithm Overview

Algorithm 1 Minimax with Alpha-Beta Pruning and Memoization

```
$\alpha \leftarrow -\infty$
$\beta \leftarrow \infty$
$choices \leftarrow []$
function MINIMAX(state, iteration, cpu_turn, blacklist, $\alpha$, $\beta$)
    if game_over(state, blacklist) then
        return 1 if cpu_turn else -1
    if cpu_turn then
        $children \leftarrow generate\_children(state)$
        $max\_eval \leftarrow -\infty$
        for child in children do
            if child in cache then
                $eval \leftarrow cache[child, cpu\_turn]$
            else
                $eval \leftarrow minimax(child, iteration + 1, false, blacklist, \alpha, \beta)$
                $cache[child, cpu\_turn] \leftarrow eval$
            $max\_eval \leftarrow maximum(max\_eval, eval)$
            $\alpha \leftarrow maximum(\alpha, max\_eval)$
            if $\alpha$ is greater than or equal to $\beta$ then
                break
            if iteration is 0 and eval is 1 then
                add child to choices
        return max_eval
    else
        $children \leftarrow generate\_children(state)$
        $min\_eval \leftarrow \infty$
        for child in children do
            if child in cache then
                $eval \leftarrow cache[child, cpu\_turn]$
            else
                $eval \leftarrow minimax(child, iteration + 1, true, blacklist, \alpha, \beta)$
                $cache[child, cpu\_turn] \leftarrow eval$
            $min\_eval \leftarrow minimum(min\_eval, eval)$
            $\beta \leftarrow minimum(\beta, min\_eval)$
            if $\beta$ is less than or equal to $\alpha$ then
                break
            if iteration is 0 and eval is 1 then
                add child to choices
        return min_eval
```

Algorithm 2 Evaluation

```
function GAME_OVER(state, blacklist)
    if state in blacklist then
        return true
    if all pile zero in state then
        return true
    return false
```

Algorithm 3 Generate All Possible Moves

```
function GENERATE_CHILDREN(parent)
    $children \leftarrow []$
    for pile_index in parent do
        for reduction from 1 to parent[pile_index] + 1 do
            $copy \leftarrow parent$
            $copy[pile\_index] \leftarrow copy[pile\_index] - reduction$
            add copy to children
    return children if not empty
```

Algorithm Correctness

The minimax algorithm intends to provide an agent with an intelligent decision. The algorithm will recursively construct a game tree until some terminal state is reached (in this case, a terminal state is that in which the game is considered to be over); upon this state, a static evaluation will occur taking into account the depth, and therefore the player expected to make a move at the state [**Algorithm 1, Line 4**]. This allows the agent to either maximize its probability to win, or minimize its probability to lose.

Our implementation very closely follows the standard recursive minimax algorithm. Most importantly, static evaluation occurs *only* in the case of an terminal state (either a game state that is in the blacklist or the game state in which no non-empty piles remain) [**Algorithm 2, Lines 2-5**]. In the case that minimax is called for the maximizing player, our evaluation will return a high value (1); contrarily, if minimax is called for the minimizing player, the same evaluation will return a low value (-1) [**Algorithm 1, Line 5**].

Following evaluation, the heuristics of the children are evaluated at the parent in which the most advantageous child is selected depending on whether the depth is a minimizing or maximizing state [**Algorithm 1, Line 15/Line 31**]. For example, if we evaluate the heuristic at a state with four other possible next states (children) and the depth relevant to the maximizing player, we would inherit the highest heuristic from any of the children (ideally 1 in this case).

This recursively occurs until we reach depth 0, in which we add all evaluated states at depth 0 with the ideal heuristic value to a list of choices [**Algorithm 1, Line 19-20/35-36**] Given that evaluation of the heuristic *only occurs at game-ending states*, and given that the heuristic is inherited from the bottom-up, we can conclude that any decision the agent makes will be the most optimal decision possible.

Heuristics in this case are binary in nature; for our algorithm, only states with an advantageous heuristic are added to the list of choices. In the case that no advantageous choices are given, an arbitrary choice is made as moves are not relative - i.e. no bad move is worse than another bad move. Therefore, in cases where a choice cannot be found, the program will default to the original artificial agent, as described below.

Basic Computer Player Algorithm

Our original, "basic" computer player uses an algorithm based off of both the mathematics behind standard variations of multi-pile Nim, and off of edge cases observed through testing. For example, in the misère variation of multi-pile Nim (without any constraints), assuming both players play optimally and the nim-sum (the integer result of applying bitwise-XOR to the number of objects within each of the piles) of the initial game state is non-zero, the first player should always win. Edge cases we accommodated for include cases such as those in which the number of non-empty piles remaining is less than or equal to two.

Our team eventually switched to a minimax-based methodology after learning about minimax and alpha-beta pruning in lecture, and after realizing that adding constraints to Nim, such as those in the variation described throughout this paper, force the computer player to become much more modular and complex than initially assumed, if it is to always be correct. For example, ignoring edge cases, our original computer player arbitrarily chooses the first means of reducing the Nim-sum of the current game state to 0 it finds (if possible). Normally this is perfectly acceptable, but when playing with constraints, this means that one means of doing so may be less likely to lead the computer player to a constrained state (and thus a loss) later in the game than another means of doing so.

We realized we needed to enable the computer player to "look ahead". However, for the sake of simply showing our progress throughout the project development, we decided to leave the original computer player in as an option for an opponent. We also decided to have the newer, "advanced" computer player hot-swap back to the original computer player when it determines that no optimal move exists, as the latter already contained handles for making an arbitrary move, among other features.

A very simple description of the original computer player logic is as follows:

```
function cpu_plays(piles, non_empty_pile_indexes, blacklist)
    calculate the nim-sum of the current game state
    if the nim-sum calculated is not equal to 0
        if the number of non-empty piles remaining is greater than 2
            if all but 1 non-empty piles remaining contain exactly 1 object each
                make the move that makes, or keeps,
                the number of piles containing exactly 1 object each, odd
            else
                consider moves that change the nim-sum to 0
                if the corresponding state(s) is/are blacklisted
                    save the blacklisted state for future reference
                else
                    make the move
        else
            if the number of non-empty piles remaining is equal to 1
                consider removing all but one object from the final pile
                if the corresponding state is blacklisted
                    save the blacklisted state for future reference
                else
                    make the move
            else
                if any 1 of the 2 non-empty piles remaining contains just one object
                consider removing all of the objects from the opposite remaining pile
                    if the corresponding state is blacklisted
                        save the blacklist state for future reference
                    else
                        make the move
                else
                    consider the move that changes the nim-sum to 0
                    if the corresponding state is blacklisted
                        save the blacklisted state for future reference
                    else
                        make the move

    if a blacklisted state has been saved for future reference
        find and consider a move that forces the opposite player
        to enter the blacklisted state on the next turn if they
        are to reduce the nim-sum to 0, or makes reducing the
        nim-sum to 0 potentially difficult (increase the nim-sum
        as much as possible)
            if such a move exists and the corresponding state is not blacklisted
                make the move

    consider any and all arbitrary moves
        if the corresponding state is not blacklisted
            make the move

    make the losing move (handle loss)
```

Data Structures Employed

1. **Trees** (Game/Search Trees): The recursive calls of the minimax algorithm described above generates a game tree, which is used to search for the best move at a given game state by considering future game states.
2. **Hash Maps** (Dictionaries): The game state blacklist is stored as a hash map/dictionary. The minimax algorithm also uses a cache stored as a dictionary/hash map for memoization purposes.
3. **Arrays**: Current game states are stored as arrays of integers (where indexes refer to each pile and elements refer to the number of objects within each pile).

Algorithm Complexity Analysis

Given some initial state $s \to [p_1, p_2, ..., p_k]\, for\, k, p_k \in Z$, the maximum number of moves in each state $m \to \sum_{i=1}^{k} s_i$, and the maximum number of states, or depth $d \to m + 1$, we infer that classical minimax is upper-bounded by m^d.

There were multiple optimizations made to the base algorithm. *Alpha-beta pruning* provides a noticeable advantage in both best-case and average-case time complexity given optimal move ordering, reducing best case from m^d to $m^{d/2}$ but does not affect worse-case. However, memoization was another optimization made; by caching previously evaluated states using multisets, we can detect isomorphic states, and can logarithmically reduce states *as they are evaluated* such that our worse-case complexity becomes $m \cdot log(m)^d$

Therefore, our optimized algorithm is upper-bounded by:

$$O(m) = m \cdot log(m)^d$$

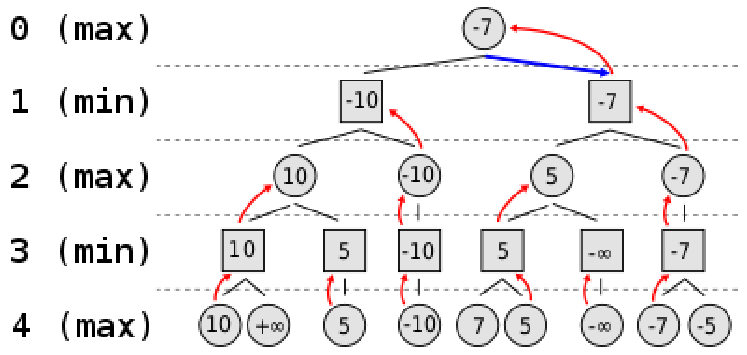


Figure 1: The tree above has a depth of 4 and 2 choices, b value of 2. $O(2(^4))$ is the complexity for this tree

References

[1] https://en.wikipedia.org/wiki/Nim

[2] https://en.wikipedia.org/wiki/Sprague%E2%80%93Grundy_theorem

[3] https://en.wikipedia.org/wiki/Alpha\%E2\%80\%93beta_pruning

[4] https://www.youtube.com/watch?v=l-hh51ncgDI

[5] https://en.wikipedia.org/wiki/Minimax_theorem

[6] https://en.wikipedia.org/wiki/Minimax