A Memoization Minimax Solver for a Variation of Nim

Data Structures II Course Project

By

Mason Legere
Son Chau
Kai Lan

University of Northern British Columbia
Department of Computer Science

March 2021

Contents

1 Introduction

The Game of Nim is a traditional and well studied situation within combinatorial game theory. Further, through the celebrated Sprague–Grundy theorem[2] it was shown that impartial combinatorial games are equivalent to a one stack game of Nim. Because of this connection there has been much work on studying solutions for determining if a given game state within the traditional sense is winning as well as the study of different variations of the Game of Nim. The latter point is exactly what we are doing within this project which provides an algorithm that can theoretically solve both the misère and standard games of Nim with additional losing (resp. winning) conditions. By this we mean that beyond the typical losing condition of misère play where no beads are remaining, we can also support arbitrary losing states such that if a player makes a move *to* one of these states they lose the game. In particular, to meet the requirements of the project our algorithm supports the following losing states:

- all piles are empty.
- 3 piles each with 2 objects, and all other piles are empty.
- 1 pile with 1 object, 1 pile with 2 objects, 1 pile with 3 objects, all other piles are empty.
- 2 piles with 1 object, 2 piles with 2 objects, all other piles empty.

However, the algorithm presented will naturally extend to other losing (resp. winning) conditions. While within the traditional game of Nim Grundy numbers can be used to this project also contains code for a terminal interface to play the game as well as PyPi format making installation easy to run. These points of consideration will be discussed in depth in the following section. Further, we will also describe the general project format, libraries and data structures used as well as possible improvements that could be made. Following this there is a dedicated section to analysis of the algorithm used for making the CPU's moves.

1.1 Program Design

This project provides a terminal user interface that allows some dynamic feedback without having large dependencies. Additionally, the actual program could be decoupled from the interface and used separately as well. The front end display and input is handled by the `Blessed` library which is a higher level abstraction of the older `Curses` library written in `C`. Functionally, the program is very simple with essentially two layers of abstraction. Namely, there is an instantiated `GameInstance` that is provided with the initial state of the board upon construction. This instance has the methods `next()` and `player_move(state)` used to get the CPU's next move and modify the game instance state respectively. Beyond the handling of keyboard movements, there is not a lot more from a high-level that was needed to implement this project. A visual representation of this design can be seen within Figure 3.

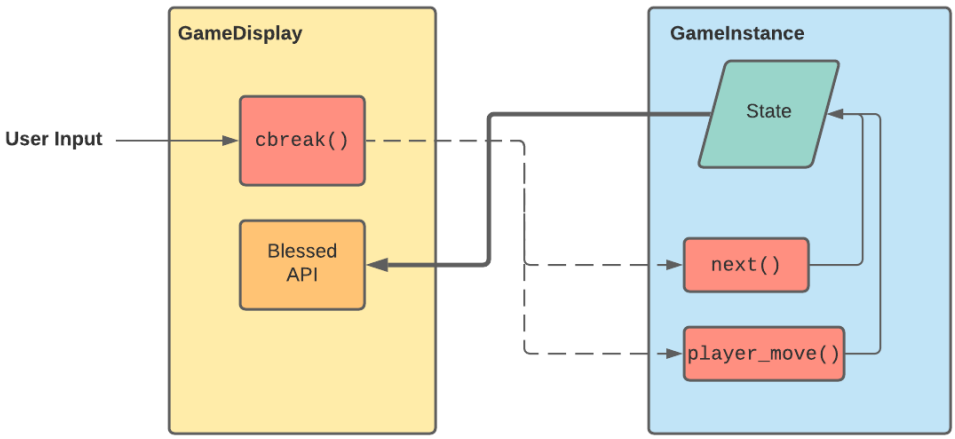

Figure 1: High-level view of the architecture used in the project

1.2 Data Structures

Within the program the state of the board at any given time is solely represented by a list of non-negative integers representing the number of beads per peg. For example, a valid initial state would be represented as `[1,2,3,2]` which, within our terminal interface is seen in Figure 2.

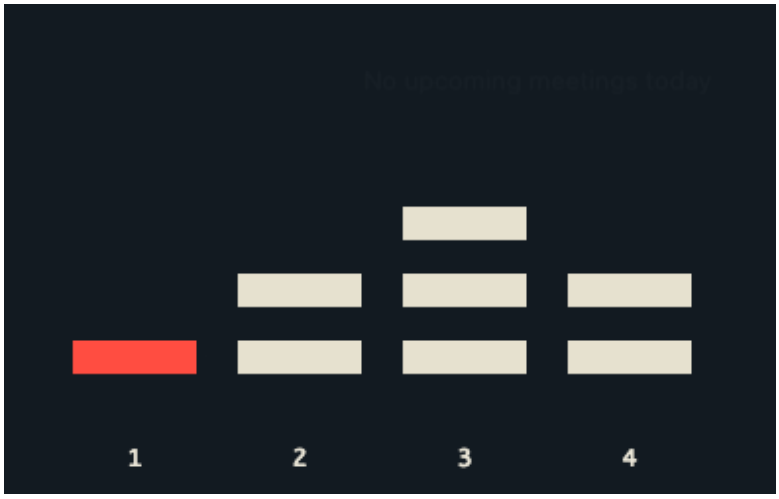

Figure 2: Terminal user interface display subsection using the initial board state of `1,2,3,2`

Working with these integer lists is the basis of the rest of the project and algorithm. Notice that however, one optimization that we will perform that many game states are *equivalent*. That is, two game states represented by lists are equivalent if they differ only by a permutation. For example `[1,2,3]` ~ `[2,1,3]` where ~ denotes the equivalence relation between the permutation group G_n with n being the length of the initial game state. Further, as we will see within the algorithm (where we apply memorization to save on time complexity) we also need to use Python's `set` and `dict` classes. Not that CPython actually implements set as a dictionary with dummy values as seen in the CPython source code [1]. This allows us to check to see if a state has been seen previously in constant time. Finally, we mention that as we are representing game states as lists, they are not hashable by default in Python as they are mutable types. For this reason when working in cases where we want to hash a given state, we first sort it and map it to a tuple (non-mutable type) allowing it to be hashed. While the sorting does create some additional work upfront, using a `frozenset()` removes duplicate values. To get

[1] CPython set source code: https://hg.python.org/releasing/3.6/file/tip/Objects/setobject.c

around this you would have to map original list to a list of tuples to avoid duplicate values (or a similar solution) which would add linear time for the conversion, double the amount of memory required for caching values and also neglects the fact that `frozenset()` takes roughly $4n$ time complexity to create the set along with increased memory consumption. Therefore, our simplistic solution of mapping to sorted tuple was deemed to be more appropriate. This discussion will also be made more clear once the algorithm is presented.

1.3 Installation and Run Guide

One of the goals of this project was to make it easy to install and run without dealing with script installations or direct downloads. To this end, assuming that you have both `pip` and `Python3.7+` installed, our project can be installed directly with

Listing 1: Installation Command

```
#!/bin/bash
pip install nimmy
```

To run the project, simply provide the initial state with the flag `-s` or `--state` followed by a string list as displayed below in Listing 2. Additionally, if you would like the CPU to play first you can use the optional flag `--cpu`.

Listing 2: Run Example

```
#!/bin/bash
nimmy -s '1,2,3,2' --cpu
```

Upon running the above command your terminal should transform into the terminal interface created with `Blessed`.

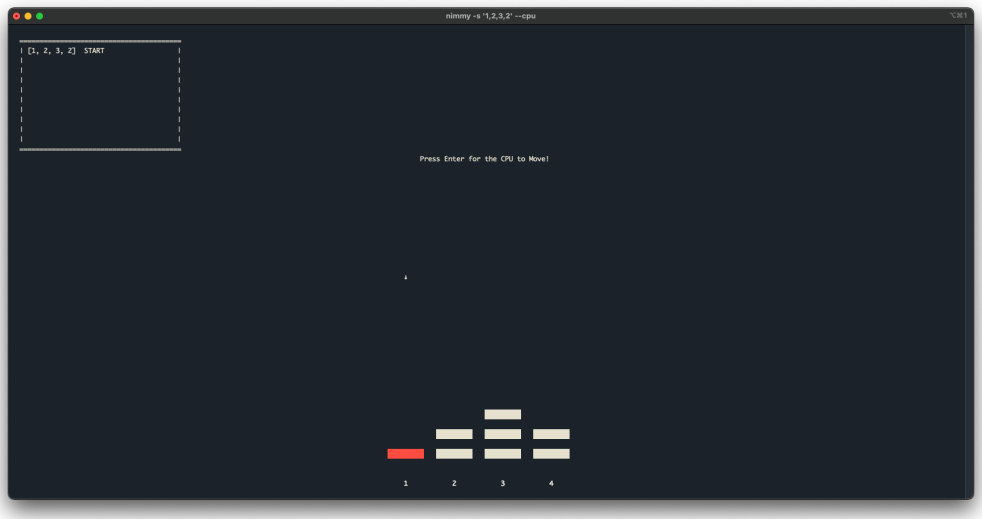

Figure 3: Output of the program upon running Listing 2.

Once inside the game you can quit at anytime by pressing 'q'. If it is your turn to move you can use any typical arrows keys (wasd, up right, etc) to move around the cursor and select the number of beads you would like to remove from a given pile. Once you have selected the number of beads, press 'ENTER' to remove them and update the game state. In the event it is the CPU's turn, you also press 'ENTER' in order for it to make a move. For more information, please see the project web page Project Page which was made public as of the project due date.

2 Algorithm Analysis

Within this section we describe the algorithm used for generating the CPU's move within the program. This algorithm using three components:

- Minimax optimization with Alpha-beta pruning[1]
- Memoization of previously seen game states
- Equivalence of game states that differ only by a permutation

This is accomplished by using a hash table to store previously seen game states and sorting said game states before caching such that equivalence is preserved. Finally, alpha-beta pruning was used to allow for the pruning of game states that are guaranteed to be losing across a family of equivalent states. This of course does not change the asymptotic complexity but is known to lead to performance increases. At this point we present our solution in Algorithm 1. Notice that this solution will return back a list of game states, starting from the initial state such that it leads to optimal play. Therefore, in order to actually pull the next move from the algorithm you must call it as:

Listing 3: Calling Algorithm For Next Move

```
AlphaBetaPruning(initial_state, alpha, beta, True)[1][1]
```

Where the index notation of `[1][1]` is in reference to that found in Python's syntax. Now that that algorithm has been defined fully, in the next section we will describe the time and space complexity associated with Algorithm 1.

2.1 Correctness

While there will be some thought required for time complexity, correctness of the solution follows without much effort. Because the mini-max algorithm with alpha-beta pruning is known to play optimally this algorithm is guaranteed to lead to optimal play. Additionally, because our algorithm adds nothing to the alpha-beta pruning beyond memoization of previous states as well as considerations between the equivalence of game states - our solutions therefore also leads to optimal play.

2.2 Complexity Analysis

Referring to Algorithm 1 we see that for any given initial state we will at most transverse all of it's possible children recursively. That is, all the possible game states that be be reached from the given initial state. However, as mentioned previously there are several optimizations that can be done to speed up our algorithm. Namely, we only ever consider invariant versions of the states - meaning that permutations of possible states do not come into affect when doing complexity analysis. Furthermore, due to the implementation of memorization we never compute the same state twice. To this end we can now present the time and space complexity of our algorithm based upon this observation.

Given an initial input state $[n_1, n_2, \ldots, n_k]$ we note that for each pile n_i there can be $n_i + 1$ possible positions it can be in throughout the game. Then altogether without counting for permutations, there are $\prod_{i=1}^{k}(n_i + 1)$ possible states that can occur throughout the game. Therefore, because we cache each invariant state throughout the algorithm in the form (state, player), and there are only two options for the player, namely true or false, we find that in the worst case we will cache $2 \cdot \prod_{i=1}^{k}(n_i + 1)$ states. Consequently, our space complexity is given by

Algorithm 1: Algorithm with procedure

```
Result: Next Optimal Move to be played given initial state
Procedure AlphaBetaPruning(state, alpha, beta, maximizing_player)
    state_list ← list(state)
    if state is a terminating node and not the initial state then
        if maximizing_player then
            return 1, state_list ;
        else
            return −1, state_list ;
        end
    end
    if maximizing_player then
        bound ← −∞ ;
        path ← list() ;
        for each unique child C of state do
            if (C, False) in cache then
                (child_bound, child_state) ← cache(C, False);
            else
                (child_bound, child_state) ← AlphaBetaPruning(C, alpha, beta, False) ;
                cache(C, False) ← (child_bound, child_state);
            end
            if child_bound > bound then
                bound ← child_bound ;
                path ← child_state ;
            end
            alpha ← max(alpha, child_bound) ;
            if alpha ≥ beta then
                break ;
            end
            return bound, concat(state_list, path) ;
        end
    else
        bound ← ∞ ;
        path ← list() ;
        for each unique child C of state do
            if (C, True) in cache then
                (child_bound, child_state) ← cache(C, True);
            else
                (child_bound, child_state) ← AlphaBetaPruning(C, alpha, beta, True) ;
                cache(C, True) ← (child_bound, child_state);
            end
            if child_bound < bound then
                bound ← child_bound ;
                path ← child_state ;
            end
            alpha ← min(alpha, child_bound) ;
            if beta ≤ alpha then
                break ;
            end
            return bound, concat(state_list, path) ;
        end
    end
Result: A branch of the game tree played optimally given the initial state
```

$$\text{space complexity} \sim \mathcal{O}\left(\prod_{i=1}^{k}(n_i+1)\right)$$

Now that we have considered time complexity, we now consider the space complexity associated with out algorithm. As before we see that actually using methods of dynamic programming the worst-case analysis is easy to perform due to the fact that we do not have to account for possible repetitions. To this end, we note that we will need to compute at most compute the outcome of $2\cdot\prod_{i=1}^{k}(n_i+1)$ game states. Further, as these games states are recursively dependent on their children this amounts to adding at most $2\cdot\prod_{i=1}^{k}(n_i+1)$ elements to the cache. In order to avoid adding permutations of game states, which of course are equivalent, we also need to move states into invariant formats before the addition. This is done by sorting states before being added to the cache. As mentioned previously, while one could convert the lists into dictionaries rather than sorting, because we are working with very small lists (maximum size 9 for our examples) sorting is actually more memory and time efficient. However, asymptotically hashing would only add a linear factor rather than our log-linear factor. Therefore, with this in mind we see that our solution's time complexity can be bounded above by

$$\text{time complexity} \sim \mathcal{O}(k\log(k)\prod_{i=1}^{k}(n_i+1))$$

This time complexity is much better than that given if a naive mini-max implementation was used which would be exponential in time. Furthermore, our additional optimization to sort states before adding them to the cache not only simplifies error analysis greatly, it also makes the algorithm much more efficient than any other direct computations. Finally we note that for our purposes of playing against another computer or human, memorization is excellent as computations done at the start of the game can be used repeatedly throughout. Meaning that after the initial move the computers response should be fast without delay.

3 Conclusion

Within this report we have provided an algorithm to solve a variation of the Game of Nim using alpha-beta pruning of the game tree, memoization of previous game states, and additionally, looking for equivalence across game states that are equivalent with respect to the rules of Nim. From this we were exposed to creating terminal interfaces in `Blessed` as well as the general ideas behind combinatorial game theory. While there are several features of our project that we would like to add in the future, such as providing the user the ability to replay the game, we are happy with the final outcome of out project. Furthermore, since modeling computer players in competitive games is a popular ongoing research field, it is also worth learning other alternative algorithms such as SSS* and comparing them without our own solution.

References

[1] Stuart Russell and Peter Norvig. “Artificial intelligence: a modern approach”. In: (2002).

[2] Richard Sprague. “Über mathematische kampfspiele”. In: *Tohoku Mathematical Journal, First Series* 41 (1935), pp. 438–444.