Results from the Theory Group 2019

Spencer McCall

Abstract—Spring and Fall 2019, the theory group has predominantly focused on developing efficient mappings from state-space to database index. In this report, I outline and evaluate the motivating algorithm, describe the requirements for future work for mapping (hash) functions that are amenable to this algorithm, and provide two extensions that I developed this semester.

I. DYNAMIC PROGRAMMING AS A SOLVER

The primary intuition was developed Spring 2019 and summarised in the group report thereof (so I will not spend too much time on it). In brief, we seek to replace the normal, recursive descent solver with a dynamic ascent solver. This is done to better take advantage of easy parallelism, cache coherency, and is done in conjunction with some prior intuition (e.g., computing successors directly from the mapping).

A. Python Code

While the actual working solution is written in GoLang, the Python version is significantly more readable.

```python
values = []

def solve(state):
    temp = [values[c] for c in child(state)]
    return WIN if LOSS in temp else LOSS

def solve_all():
    for i in range(MAX_STATE):
        values.append(solve(i))
```

This, in practice, has a few performance improvements, primarily designed this Fall 2019. For example, by encoding Wins as a 1, and Losses as 0, the return statement in `solve` can be replaced with an arithmetic expression, which reduces branching overhead and has shown some modest improvements. Similarly, `child` is designed for our test games to never create a new object, or data structure: all computation done in finding successors is done on the integer.

In GoLang we make a few more modest improvements (use of unsigned integers to reduce silent bugs, parallelisation of `solve_all` using chunk sizes equal to the greatest common factor among the tiers, etcetera).

B. Evaluation

To compare “apples-to-apples,” I coded both of this algorithm, and the memoized recursive descent algorithm, in Python, with only the performance modifications that were available to both. Note, there are no modifications to the recursive descent algorithm not amendable to the dynamic programming algorithm, to my knowledge.

This implies the true algorithm (to which more improvements have been made), which additionally benefits more greatly from being run in a low-level language, could only perform relatively better than these tests indicate. The results are summarised in the two following plots. The first shows the two algorithms’ absolute performances, next to one another, as a function of the size of the state space of the (subtraction) game. The second shows the relative speed-up of the dynamic programming algorithm, over the same axis.

![Fig. 1. Solving a basic subtraction game.](image1.png)

![Fig. 2. Solving a basic subtraction game.](image2.png)

As shown in both figures, the dynamic programming algorithm generally (but not always) outperforms the memoized recursive ascent algorithm. Notably, it also has less variance, with respect to small perturbations in the game tree structure. Notice as the “starting index” of a subtraction game changes, with the subtraction set (eg., 1 and 2 in our in-class example, although, in this test less trivial) staying fixed, the reachable positions could dramatically change. The performance of the new algorithm is more robust, relative to this.
II. SUFFICIENT MAPPING FUNCTIONS

Last semester, Zoe and I found two necessary conditions for this algorithm to behave properly, and one condition that utilizes its capacity to exploit cache performance well. The necessary, for mapping function $H$:

1) $H(state) > H(child)$ $\forall child \in state$
2) $H(child)$ must have a closed-form representation, given $H(state)$ and a move.

The first is to ensure DP-solvability. The second allows child not to have to create new data structures.

The non-necessary: either the hash of a state and its children must be at most some $\epsilon$ distance away, or the children of two $\epsilon$-close states must all be at most $\epsilon$ away from each other.

In other words, for a cache with sufficient associativity (in practice, how you would define $\epsilon$), either solve must stay within a block (with high likelihood), or each iteration in solve_all must stay in the same blocks.

III. EXTENSIONS AND IMPROVEMENTS

A. Accounting for Ties / Draws

The pseudo-code given in section I-A does not account for ties. While changing the return statement in solve is not particularly difficult, adjusting the modification given is slightly more difficult. I recognized that the definition of winning, losing, and tie positions are symmetric, and present the following generalisation:

$$V = WIN - \min(children)$$

where all possible values are encoded from zero to WIN. For example, in the Win / Loss case, this gives

$$V = \begin{cases} 1 - 0 = 1 & 0 \in children \\ 1 - 1 = 0 & 0 \notin children \end{cases}$$

which gives precisely what we want. For ties, let 0 be a loss, 1 be a tie, and 2 be a win. This fails when we include draws, however, because draws do not “become” ties, or vice versa. Instead, introduce Draw-Wins and Draw-Losses.

We make the assumption that tieing is better than being on the fringe of a draw, and that forcing the opponent to the fringe is better than tieing. This may be an incorrect assumption, but it seems reasonable given the group’s assumptions about remoteness and winning versus losing positions (ie., take the tie if and only if, suppose the game would end, it would likely result in you losing).

B. Sprague-Grundy Values

Currently, Gamesman solvers only account for wins, losses, ties and draws. For impartial games, it would be interesting to, in the future, analyse more complicated values, like Sprague-Grundy values. As such, I’ve written an alternate solve function that does this.

The Sprague-Grundy function notably requires use of the minimal excluded value (mex) function, for which I provide a novel solution. The mex function calculates the smallest natural number not in a given set (in this case, the values of the children). Most solutions take two passes over the set (2n time), whereas mine only requires one pass.

My solution first creates a bitstring of all zeros, and iterating through the values $i$ of the children – sets the $i^{th}$ bit of this string to 1. This gives a bitstring that represents the boolean function of whether each natural number is in the set. We can initialise a bitstring of finite length because we know the mex will be at most the number of children (number of elements in the set).

My solution then negates the bitstring, and uses a de Bruijn sequene to, in constant time, find the index of the first 1 in the string. This gives the smallest index of a zero in the original string, which is the smallest integer not present in the set.