Using Dynamic Programming To Create More Efficient Solvers for Games

David Zagaynov and Lawrence Zhao

December 2020

Video Walkthrough: Click here

1 Introduction

The goal of GamesCrafters is to strongly solve different games through the use of computational game theory. However, certain games have state space sizes that are so large that it can take multiple days for a traditional solver to traverse the entire game tree. One such game is Connect-4. This paper looks into the practicality of using a bottom-up dynamic programming (DP) approach to solve Connect Four and other games in general. The implementation of the solver requires two main components: a DP Solver that can be generalized for all games and a hashing algorithm specific to each game. In previous semesters, we had attempted to come up with efficient and tight hash functions for Connect-4. This semester, we decided to work from the other end, designing and implementing the DP solver and seeing how much of an improvement in runtime we can get in solving the simple game of Tic-Tac-Toe. The hash function we use for Tic-Tac-Toe may serve as a blueprint for coming up with an effective hash for Connect-4 in later research.

2 Dynamic Programming Solver

All GamesCrafters are required in their first semester to code a traditional solver that traverses game trees from the top-down, calculates the primitive values of game states at the bottom of the tree, and then works its way back up to fully solve the game. All GamesCrafters are also required in their first semester to code a solver that utilizes memoization, preventing the solver from wasting time repeating calculations for game states it has already solved. In addition to those two types of solvers, the theory team has long been interested in a dynamic programming approach to solving games.

Dynamic programming solves problems by breaking them down into smaller sub-problems, which seems highly applicable to solving games as they are solved by solving the "sub-problem" of child positions. The main issue is ordering the

game states in such a way so that all child positions are solved before the solver attempts to solve their parent positions. That is why having an effective hash function is so imperative. If done correctly, dynamic programming should be able to avoid the overhead of traversing the game tree from the top-down, instead starting directly from the bottom and working its way up, thereby even exceeding the speed improvements achieved through memoization.

3 Hash Function Requirements

A hash function must meet three requirements in order to be effective for a DP solver: 1) form a 1-to-1 correspondence between games states and numbers, 2) have a maximum range that is equal to the state space size of the game, and 3) form a monotonically increasing/decreasing relationship between parent positions and children positions.

The first requirement is necessary because we need to be able to both hash positions and unhash their hash values. We have to be able to unhash the hash values so that we can derive the original positions and calculate what moves are available in order to determine what their children positions are, and then we need to be able to hash the children positions in order to determine what sub-problems the DP algorithm should look at. The second requirement is actually a derivative of the first requirement, as otherwise there would be hash values that do not map to valid game states. The third requirement is necessary because we need to be able to solve the sub-problems of child positions in order so that they are all solved before the DP solver attempts to solve their parent positions. This means that loopy games can not be applied to the tier solver in its current form. This also eliminates all random hash functions, such as built-in hash functions.

Many previous GamesCrafters in previous semesters have attempted to come up with a hash function that meets all three requirements for Connect-4, to no avail. Coming up with a hash that satisfies all three requirements turns out to be surprisingly difficult. This semester, we had initially set out in hope of building on previous work and making progress on a Connect-4 hash function, but as progress proved to be difficult, we decided to shift our focus to proving that there is a pot of gold at the end of the rainbow. Instead, we decided to use Professor Dan Garcia's optimal hash for Tic-Tac-Toe (Link here) to show that a DP solver does indeed lead to a runtime improvement.

4 Hashing Tic-Tac-Toe

The optimal hash for Tic-Tac-Toe takes a tier-based approach. For each number of X's and O's, it calculates the number of possible board rearrangements using the rearranger function:

$$R(X, O, S) = \frac{S!}{X!O!(S - X - O)!}$$

where X is the number of X's on the board, O is the number of O's on the board, and S is the number of spaces on the board (normally 9).

Given a position with a certain number of X's and O's on the board, the hash function starts by adding up the rearranger function for all combinations of fewer X's and O's, forming the bias term. This ensures that child positions always have larger values than their parents, as child positions always have more X's and O's than their parents. The hash function then seeks to find the positions specific rearranger number within the number of possible rearrangements. It does this by traversing through the board position by position and considering only the positions it hasn't traversed through yet. It then adds the following values to the bias for each position:

$$\begin{cases} R(A, B, C-1) + R(A, B-1, C-1) & \text{if } X \\ R(A, B, C-1) & \text{if } O \\ 0 & \text{if empty} \end{cases}$$

where A represents the number of X's to still traverse, B represents the number of O's to still traverse, and C represents the number of positions to still traverse. There is also a special caveat where if there are only entirely X's, entirely O's, or entirely blanks to traverse, the hash function stops adding to the bias entirely.

This setup also ensures that the maximum range for the hash function is equal to the state space size of Tic-Tac-Toe and there is a 1-to-1 correspondence between board positiosn and hash values. We can then unhash the hash values by storing the positions in order in a list according to their hash values.

5 Results

We coded all three solvers and ran 100 trials of each of them on Tic-Tac-Toe. The average runtime for the traditional solver was about 4.978 seconds, which is actually somewhat slow for a game as small and simple as Tic-Tac-Toe. The average runtime for the memoized solver was 0.074 seconds, an enormous improvement. Initially, the average runtime for the DP solver was about 0.218 seconds, which is actually slower than the memoized solver. However, we realized that the DP solver had a significant bottleneck recomputing the hash values for the same positions, so it could be optimized by memoizing the hash values. After implementing memoization in the DP solver, the average runtime dropped to about 0.067 seconds, which is a minor improvement over the memoized solver.

6 Final Comments

At the end of the day, the improvement of the DP solver over the memoized solver was only about 10%, but when one consider that the solver for Connect-4 can take multiple days to run to completion, a decrease of 10% runtime can save several hours. It should also be noted that previous work on a Connect-4 hash function also attempted to take a tier-based approach, but struggled to find

a closed-form rearranger function that could be applied in the same way that Professor Garcia's rearranger function was applied to Tic-Tac-Toe. Despite Tic-Tac-Toe and Connect-4 being similar in that they are both dart-board games, Connect-4 has a factor of gravity to it that forces all pieces to fall as far as they possibly can. This makes the rearranger function more complex, but should future GamesCrafters pick up on this work, they should start by trying to find a closed-form rearranger function that can be applied tier-by-tier.