Solving 2048

Prithvi Ramakrishnan
prithvir@stanford.edu

December 13, 2014

1 Introduction

2048 is a single-player puzzle game created in March 2014 by 19-year-old Italian web developer Gabriele Cirulli, in which the objective is to slide numbered tiles on a grid to combine them and create a tile with the number 2048. The game is conventionally considered a one player game, but it can be considered as a two-player game, where the player is playing against an adversary that places new tiles on the board. Call the player that moves the tiles Player 1 and the player that places tiles on the board Player 2. In the regular version of 2048, Player 2 simply plays randomly, arbitrarily choosing a vacant square on the board and placing either a 2 or 4 on that square (a four is placed instead of a two with probability 0.1). A score is given for the game, where whenever a tile is merged, the new value of the merged tile is added to the score.

Because of the game being developed as an open-source project, the repository has been forked multiple times, creating different variations on the game. From a game playing perspective where we consider the game as two different adversaries, there are two games that are particularly interesting. The first is a variation of the game where a simple artificial intelligence engine plays the role of Player 2, called Evil 2048. This game is particularly challenging; there isn't anyone who has claimed to reach the 2048 tile on the game.

The second, 8402 is a variation where Player one is played by an AI engine, and Player 2 is played by the user. In other words, the engine selects the directions to move, and the player selects the squares on which to place a new tile (and whether the tile to be placed is a two or a four.)

These games illustrate that as a two player game, the optimal strategy for either player is not easily determinable. The purpose of this project is to investigate strategies for both players, and try to converge on optimal strategies for both players, and to determine estimations or ranges for the score achieved in the game given optimal play from both sides.

This will involve building engines for both players, and playing them against a variety of engines found online, to find ways to appropriate the performance of the games.

Because of the game's huge popularity, there are several engines that play the game, with varying degrees of reported success, ranging up from 90% of a win rate. These engines will serve as good frameworks with which test an engine for player 2.

To add to the value of this project, we also develop a program to play the regular version of 2048, which will assume that the tile generation is done randomly.

2 Game Formulation

We formulate the game as a minimax tree as usual. To illustrate, we have the following examples of instances in the tree.

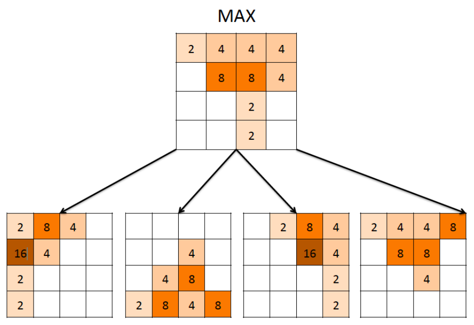

Figure 1: A sample of a max node in the game-tree that the algorithm searches

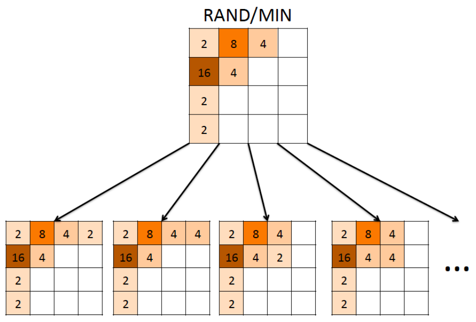

Figure 2: A sample of a min or rand node in the game-tree that the algorithm searches

For the second figure, depending on whether we're playing the game which is adversarial or in which the tile generation is random.

The top-level algorithm for both players will be a minimax algorithm. The state will store the following information:

- The board, a 4x4 array of integers representing the value of the tile at each location (giving a zero value to spots that don't have tiles on it)
- The score of the board

Most of the research and work will be focused on the pruning and on the evaluation function.

3 Evaluation Function

We compose several parts to the evaluation. The first uses an array to generate a very human playing algorithm. We define

$$A = \begin{bmatrix} 1 & 8 & 16 & 2048 \\ 1 & 4 & 32 & 1024 \\ 1 & 2 & 64 & 512 \\ 1 & 1 & 128 & 256 \end{bmatrix}$$

The evaluation function will be composed of the following features:

- $\sum_{i=1}^{4}\sum_{j=1}^{4} G_{ij}A_{ij}$, where A is defined as above and G is the grid of values on the tiles on the board. I might also consider different ways of determining a useful score for the closeness of the grid to this ideal grid A. For instance, one idea I've been experimenting with has been using the weighted harmonic mean of the values, i.e. $\left(\sum_{i=1}^{4}\sum_{j=1}^{4}\frac{1}{G_{ij}A_{ij}}\right)^{-1}$, which will basically weight the matching of the higher values disproportionately more to the ideal matrix, which is natural, given that as a human player one is much more concerned with the highest values being the right place.
- Booleans for each of the rows and columns, indicating whether the row or column is monotonic. A row is monotonic if the elements in it are either increasing or decreasing (with empty tiles being ignored). For instance, the row $[2, 0, 4, 8]$ is monotonic, but the row $[2, 0, 8, 4]$ is not. The intuition for this boolean is that rows and columns that are monotonic are much easier to merge, and so value should be placed on this.
- The number of possible merges in a state. We consider the number of possible merges as a heuristic. This is because we observed that the engine as it was was extremely greedy, and merged whenever possible. However, this was sometimes not desireable, as many game-players observe from playing the game as a human. Including this feature added a tradeoff, which improved performance in situations where precise moves are necessary. Prolonging merges is often useful because then it maintains freedom, since merges are always available to keep the game in progress.

4 Analysis

Even though we couldn't do very much work on the side of effective pruning, we perform an analysis of the algorithm, to determine the quality of pruning.

First, observe that the branching factor gets pretty high. At each ply (one move by either player), we have two cases:

- If the current player to move is player 1, the branching factor is between zero and four, since the number of moves that can be made is limited by the number of possible directions in which the tiles can be shifted (in some positions, certain moves are not possible, with the extreme case being when the game is over, in which there are no possible moves)
- If the current player to move is player 2, the branching factor is much larger, from zero to around thirty, since for each vacant tile, a new tile can be placed, and that tile can either be a 2 or a 4. Since there must be at least one grid location with a tile on it, there can be at most thirty moves.

This high branching factor for player 2 can become problematic, since, for instance, in order to achieve a depth of 6 ply (three moves on each player's side), the number of states that would need to be visited would be on the order of $(4 \times 30)^3 = 1,728,000$, which is already fairly large, especially if we want to try to use non-constant evaluation functions. However, this number isn't general, and is most applicable in the beginning of the game, where making precise moves isn't as crucial. In more crucial parts of the game, where precise moves are essential, there are likely to be fewer than four empty locations on the board, and the number of legal moves are likely to be between two and three at every move. In this case, we can have up to ten ply of search without exceeding a million explored states. To corroborate this vast range required search space for fixed depth searches, I checked the minimum and maximum move times for running a minimax search with no pruning for a depth of just four ply, on 400 games, and the minimum time needed to make a move was 0.009758 milliseconds, while the maximum time was 253.387821 milliseconds, with the quartiles being at 2.383335ms, 3.83748ms, and 5.558604ms, respectively. This emphasizes the degree to which the number of states needed to be searched at different points in the game varies.

5 Results

When running against a random tile generator, we obtained the following results, all being run on my personal macbook air:

- Number of trials: 1000
- Win rate: 97.4%, reaches 4096 with rate 86%
- Move time: average 4.18ms, with quartiles $\{2.98, 4.66, 7.01\}$
- Average depth: 5.6 ply
- Nodes searched per move: 56k
- Speed 13.4 million nodes per second.

When running against an adversarial tile generator, we obtained the following results, again all being run on my personal macbook air:

- Number of trials: 1000
- Average score: 3142 (on each game, the largest tile it reached was either a 256 or a 512)
- Move time: average 3.98ms, with quartiles $\{2.78, 4.97, 6.98\}$
- Average depth: 6.8 ply
- Nodes searched per move: 56.6k
- Speed 14 million nodes per second.

6 Future work and Optimizations

Much of the future work will be done in pruning. As described previously, in many games, there is value in making seven or eight precise moves in a row to avoid losing, and an engine that is intelligent enough to determine this would require a depth of much more than this optimistic value of ten ply. Of course, an obvious technique for reducing this search would be to employ alpha-beta pruning, but this is still unlikely to improve the depth by that much. For some more inspiration, I looked at the performance of some chess engines, where much deeper pruning techniques are often being innovated. Stockfish, an open-source chess engine currently ranked as the best in the world and one that is often hailed and sometimes criticized for using extremely aggressive pruning techniques, achieves 20-ply depth when running on my machine from the starting position within one or two seconds (on high-performance machines with specially designed architecture for running chess programs, Stockfish apparently regularly reaches 40-ply depth in fairly complex positions, but for a more useful comparison, I cloned the program on my local machine and ran it here). Noting that the branching factor of chess is around 35, which is greater than the maximum branching factor of 2048 on either player's side, this suggests that more aggressive pruning strategies could be effective here as well. Doing some research on pruning strategies in chess programming yielded techniques such as futility pruning, razoring, and others, that help develop unbalanced pruning strategies to effectively consider more promising ideas and charging down those paths before considering less promising ideas, and ruling out ideas that are not likely to produce positive outcomes.

There are also several optimizations that speed up the gameplay, which allows us to get further in the game tree. We describe two of these strategeies:

- We observe that the representation of the array is extremely inefficient. Since all the numbers are powers of two, we have only one significant digit in every integer in the array. Instead, we can replace every integer $G_{i,j}$ with $\log G_{i,j}$. The only change that has to be made is that instead of adding two tiles together when we merge tiles, we simply increment them. This allows numbers to represented as an integer from 1 to 15 (we assume that all the numbers are less than $2^16 = 65,536$, which is reasonable, given the quality of the program). Then, instead of each number taking up an `int` of space, each tile only takes 4 bits of space. Then, the entire board, with sixteen tiles, can be stored in 64 bits. Since this is the same size as a system register on most systems these days, this can be passed around extremely quicky, which would increase the speed dramatically.

- We also observe that the process of actually computing moves on the board is actually not immediate, and a lot of the time spent in the program is spent in computing results of combinations of states and actions. This can be sped up significantly, by simply caching all possible states and actions, for individual rows. In a naïve strategy, we can simply store the raw forms of each row. Since each of the four elements of a row can be stored as an integer from 1 to 16, we can cache all possible combinations that could construct a row in $16^4 = 65,536$ states. If each cache entry requires about five bytes, this will be stored in less than a megabyte of space, which is very reasonable.

With these optimizations, we expect significant improvements.