An Expectimax Approach to 2048

Seth Carroll and Jennifer LaPierre
College of Computer and Information Science, Northeastern University
Email: {carroll.se, lapierre.je}@husky.neu.edu

Abstract

We developed an Artificial Intelligence solution to solve the game 2048 by modeling the game as an adversarial search problem and implementing a depth-limited expectimax algorithm, automating the gameplay to employ the algorithm and win the game reliably. Experimenting with different variations on the evaluation function, we found that the algorithm performed best when a board state was evaluated by squaring each tile's value and multiplying it by board position-based weights. Additionally, we tested the algorithm with different search depths, finding that a depth of 5 was optimally efficient, balancing game performance and speed. With these configurations, the algorithm was able to achieve a 2048 tile (win the game) on 10/10 trials and a 4096 tile (substantially exceed a winning score) on 9/10 trials, which took an average of 51 seconds per trial to execute to completion.

Introduction

2048 is an open-source game initially written in JavaScript, HTML, and CSS in 2014 by an Italian teenage web developer, Gabriele Cirulli (Cirulli 2017). Shortly after the game's publication online, the game went viral, prompting iOS and Android application versions and multiple clones and spin-offs. The fundamental state space of the game is a 4x4 grid that is initially empty, save for two tiles. These tiles, which are assigned numerical values in powers of two (initially 2 or 4) are placed on the grid in random locations. A player interacts with the game state by providing one of four directional inputs, which cause all tiles that are unobstructed to move as far as they can on the grid in the given direction. When these movements cause two adjacent tiles of equal value (say 2 and 2) to collide, they merge, being replaced by a single tile of their combined value (in this case, 4). After each player move, the board inserts another tile of value 2 or 4, determined by a set probability of .90 and .10 respectively, in a random, open space on the grid. The object of the game is to combine enough exponentially valued tiles to successfully create a tile of value 2048 before the board becomes filled completely and no valid moves to decongest the board through combination of the existing tiles remain.

Due to the popularity of this game and its notorious difficulty for human players, we were intrigued at the possibility of employing artificial intelligence algorithms to solve the game more effectively and efficiently than could be done as a human player. To do this, we assessed the environment type of the game and determined that at any given player state it was fully observable and static. Additionally, the environment, while appearing deterministic in that the player's move predictably affects the board's existing tiles, is stochastic, given the random insertions of tiles between player moves. It was from this perspective that we came to a decision to model the game as an adversarial search problem. While there are not two literal players, there are two actors that act alternatingly in the world -- the player and the board's tile generator. Moreover, the two actors are acting as adversaries, but not to optimize their moves against each other. Instead, one is acting to optimize their moves, while the other is acting probabilistically with defined chances for each of its move. Given this model and understanding of the game, it made sense to attempt to solve the problem of winning the game using an implementation of an expectimax algorithm.

Background

Search algorithms, generally speaking, are algorithms designed to traverse a data structure, such as a graph, to locate desired information from within the data structure. Graph search entails taking a data set and modeling it as a graph or tree, in which nodes containing data are interconnected by edges that can be traversed to progress from data point to data point. A generic search algorithm, such as depth-first search, is employed to search through the graph or tree in order to find a specific desired result if it exists. Depth-first search in particular searches a tree, starting from the root node and then navigating down along a single branch from parent to child nodes until it reaches the branch's terminal end, before returning back up the tree to explore another branch to its full depth. In this way, it prioritizes reaching terminal states over exploring broadly across branches at shallower depths, as is done in breadth-first search.

For 2048, we decided to model the game progression from state to state as a tree for an adversarial search algorithm. Adversarial search algorithms are employed to plan and construct trees that model the effects of moves in a game between opposing forces. For example, in a game with two players, a tree would be constructed where the root node is a single player's move and the score that would accompany it, the next level of the tree are all of the possible moves of the second player after the first player's move, and from there, the third level of the tree is the set of options for the first player after the second player's move. This hypothetical tree continues on until reaching terminal states that end the game, should a game have terminal states. For example, terminal states in a variety of 2048 where the game ends when a 2048 tile is formed, a terminal state would be either forming a 2048 tile and winning or running out of available moves and therefore losing the game. When constructing the tree, an evaluation function must also be designed to ascribe scores to each of these move selections, allowing them to be compared. The effectiveness of this evaluation function will determine the accuracy of move selection, so an evaluation function must accurately represent and differentiate game states to determine which move and outcome is optimal. In a two-player game, we can assume an optimal move is one that will allow a player to maximize their score and chance of winning, and correspondingly, minimize the chance of their opponent winning.

However, 2048 is not a two-player game, and therefore the tree must be adapted to account for this. There are not two players in the game, but there are two alternating, adversarial actors, as noted above: the player trying to combine and minimize the number of tiles and the random insertion of tiles to fill the board. In this instance, the player desires to play optimally in order to win. The tile insertion element of the game, however, does not act optimally to minimize the player's chances of victory. It places tiles with a value of 2 or 4 in random open positions. There is an even chance that any open tile will be selected, and there is a 90% chance that the tile will be of value 2 and a 10% chance that the tile will be of value 4. Given this randomness, we cannot know exactly what move the "opponent" will make, but we can use probability to predict likely outcomes and act on that knowledge. This is where the expectimax algorithm comes in.

Expectimax algorithms use these game trees with two opponents, with the root node being the move the player should make, the next level being the set of all possible moves of the tile inserter weighted to account for the likelihood of that move occurring, and the level after that is the set of moves the player could make in response to each of those possible actions. After creating such a tree, the move a player should take can be determined by exploring this tree using depth-first search to find which move should be made on the current turn to have the highest chance of providing the best outcome, based on looking ahead to likely outcomes in future rounds of the game. Now, each player decision can be modeled by up to 4 valid directional moves, but each tile insertion move could be a placement of one of two values of tiles in one of up to 15 available spaces on the grid. Such a tree would rapidly explode in branching factor for each further level down in the tree searched. Therefore, the computation cost of searching further in the tree is exponential in terms of the branching factor and number of available moves of the player and tile inserter. Additionally, with alternating levels of the tree being chance nodes that are inherently impossible to perfectly predict, looking further and further into the future eventually loses its notable impact on improving move selection at the present moment. Combining these two factors, it makes sense to artificially limit the search algorithm to a certain depth of exploration, rather than at the exponentially large set of terminal states at the end of the game. This allows for an algorithm that balances speed and performance for efficiency. All of this together led to our selection of a depth-limited expectimax search algorithm to solve 2048, and our experimentation revolved around finding an effective evaluation function and optimally efficient depth of search.

Related Work

Like expectimax, minimax is an adversarial search strategy, but instead of assuming a statistical probability of the opponent's move, the algorithm assumes that the opponent will make the move that yields the maximum gain for itself (and thus the minimum gain for the player agent) (Russell and Norvig 2010). For each opponent move, it takes the minimum value of the successor states, and for each player move, as in expectimax, it takes the maximum value of the successor states. We chose not to use minimax for this project because the moves that the game makes are random, not purposeful, unlike in a game such as chess in which we could assume an opponent would be playing optimally.

Another possible strategy for solving a game such as 2048 is reinforcement learning. A reinforcement learning agent refines its decision-making over time based on the outcomes of transitioning to different states and the eventual values of terminal states reached (Russell and Norvig 2010). There are different types of reinforcement learning: a utility-based agent learns the utility of different states as it reaches them and selects actions in order to maximize the expected utility; a reflex agent learns an ideal action for each state and chooses actions based on the current state; and a Q learning agent considers the impact of taking a given action in a given state, and it can be configured to be more or less wary of taking new, unexplored actions as opposed to actions with known outcomes (Russell and Norvig 2010). Reinforcement learning is useful for problem spaces in which hypothetical next states are difficult to predict, such as robotics. Rather than calculating what the outcome(s) of an action may be, a reinforcement learning agent finds out when it actually gets there. We chose not to use reinforcement learning for 2048 because it is a relatively simple problem space for which the possible successors of a given state are limited and easy to predict.

Project Description

To initiate the experiment, we first cloned the open-source code for 2048 from the latest commit to the original repository from GitHub (Cirulli 2017). From there, we thoroughly examined the JavaScript that runs the game state and logic to understand the way the game is modeled in code and determined what additions and modifications would be needed to instantiate an expectimax algorithm to supersede player input controls and automate gameplay. We modified the ‘index.html’ page from which the game is run to add a ‘Run Trial’ button, credit the original creator of the game, and provide instructions to run the algorithm, and add script tags to access our custom code. This code included two novel JavaScript files and modifications to one preexisting file. We minimally modified the included ‘game_manager.js’ to allow the ‘Run Trial’ button to trigger the automated application of the expectimax code until the game came to a conclusion.

In our first novel file, ‘expectimax.js,’ we implemented the expectimax algorithm methods, which consume a game’s grid object and a depth of search and returns a move to be executed by the existing game manager. In the second file, ‘grid_modifications.js,’ we created a set of methods for a grid object necessary to run the expectimax algorithm. The first was a deep clone necessary for the expectimax algorithm to create hypothetical future grid states without modifying the existing, “true” board of the game. Most of the other methods were adaptions of game state management methods from the ‘game_manager.js’ file. Modifications to the grid in the default game are typically handled by the game manager on the grid object it contains. However, we needed our cloned grids to be able to conduct moves on themselves for evaluation. Therefore, we adapted game manager methods and provided them as methods of the grid object, removing any actions not relevant to modifying the grid. Finally, we implemented our evaluation function as a method of a grid object to evaluate the score of its current state.

Our expectimax algorithm was divided into three methods: one to run the search algorithm and evaluation function at successive depths recursively given a present player move, a second to initiate the expectimax search for all possible present moves and store the returned scores from each, and a third to select the best of these moves and return it to the game manager for execution. The evaluation function was implemented as a single method. After implementing the infrastructure for the search and a simple shell evaluation function and insuring they properly integrated with the existing game files as intended, we began our experimentation on several evaluation functions and depths of search.

Experiments

While developing our algorithm, we had to adjust our evaluation function through trial and error in order to develop one that could consistently win the game. First, we tried a simple evaluation function that returns the sum of the values of all tiles on the board, or a good score in the current moment. However, this proved to be a poor strategy that did not perform much better than choosing random moves, as it did not consider positioning or proximity of tiles. Then, we decided to square the values of each tile in order to incentivize tile combination (the squares of two 2s, for example, add up to less than the square of one 4), so for each tile, *score = score + (tile.value * tile.value).* This evaluation function performed much better. In a sample of 10 trials each on depths 4 and 6, it achieved a score of 2048 once on depth 4 and twice on depth 6, as shown in Table 1.

Trial	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Depth 4	512	2048	1024	512	1024	1024	512	512	1024	256
Depth 6	1024	1024	1024	1024	1024	2048	1024	512	1024	2048

Table 1: Highest tile value achieved by the "values squared" evaluation function on a given trial

At depth 6, the algorithm started to run more slowly, so increasing depth alone was not enough. We wanted our algorithm to be able to consistently achieve a 2048 tile while also being able to run efficiently. To further improve our evaluation function, we decided to add a multiplier to each tile based on position. The "values squared" strategy often separated tiles with the same or similar weights, making them more difficult to combine, so we added a "grid weight" multiplier so that tiles would be worth more the closer they were to the top right corner. This was inspired by the strategy we use when playing manually and have found effective, which roughly sorts the tiles by value, so they may combine more easily and avoid congesting the board. The formula for this was: for each tile, *score = score + (gridWeights[x][y] * tile.value * tile.value)*, with

```
gridWeights = [[8,  4, 2, 1],
               [16,  8,  4, 2],
               [32,  16,  8,  4],
               [64,  32,  16,  8]];
```

This new "grid weights" evaluation function proved effective at winning the game at certain depths, but as depth increased the game began to run considerably slowly. Now that we had settled on an evaluation function, we tested the algorithm at different search depths in order to determine the ideal balance of success and efficiency. Trials were run in Google Chrome version 65.0.3325.181 on a MacBook Pro (Retina, 13-inch, early 2015) with a 3.1 GHz Intel Core i7 and 8GB of RAM, running macOS 10.13.4. We ran 10 trials at each depth from 1 to 7 (inclusive). For each trial, we measured the time taken, final game score, and highest tile achieved. Time for each trial was recorded from when the algorithm was initiated until the game determined that there were no more valid moves (game over). In analyzing the results, we took the mean time and score across trials and noted the percentage of trials in which certain tile values were achieved.

As shown in Figure 1, all search depths 5 and under ran a game of 2048 in under a minute, while depth 6 took almost 3 minutes and depth 7 took almost 30 minutes. Game score also increased with depth, as shown in Figure 2. Note that the increases are greater from even depths to subsequent odd depths, indicating that evaluating one more player move produces more gain than evaluating one more board move.

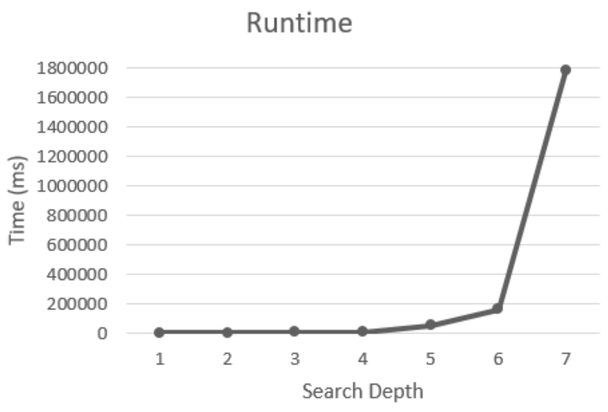

Figure 1: Mean time to run a game of 2048 at each search depth

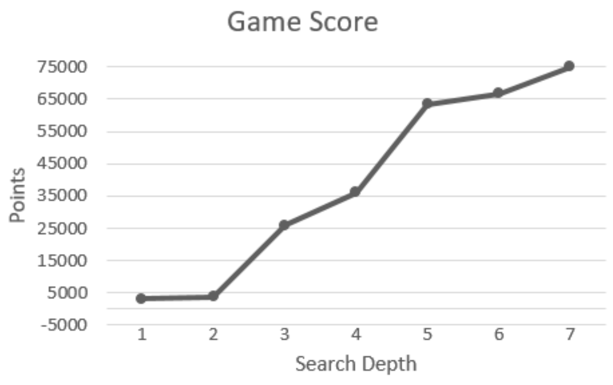

Figure 2: Mean game score at each search depth

In order to evaluate how much longer our algorithm took to run as depth increased, we divided score by time in order to find an efficiency ratio of points per millisecond, because more points take more moves to achieve, even at the same speed of playing. Depths 4 and under had about the same efficiency, which was better than the efficiency of the higher depths, as shown in Figure 3. However, this high efficiency can be explained by the fact that the algorithm lost very quickly at these depths, and points take slightly longer to achieve later on in the game. Depth 3 is sufficient to achieve 2048 at least some of the time, but depth 5 is necessary in order to achieve at least 2048 all the time, as shown in Figure 4. Depth 5 took about 51 seconds to run, on average, and had a 100% success rate at achieving at least a 2048 tile as well as a 90% success rate for achieving at least a 4096 tile, as shown in Figure 5. Depths of 6 and 7 performed slightly better than depth 5, but speed suffered greatly. Depth 7 took about 30 minutes to run, on average.

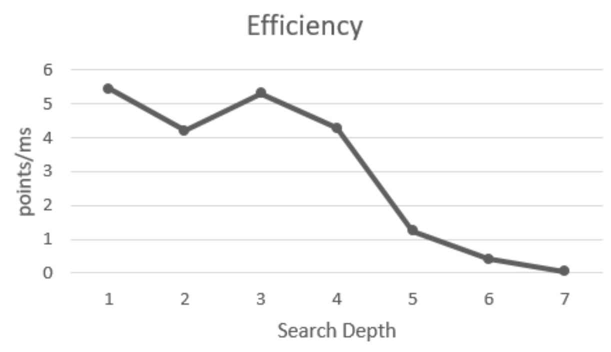

Figure 3: Efficiency at each search depth, calculated as game score over time taken

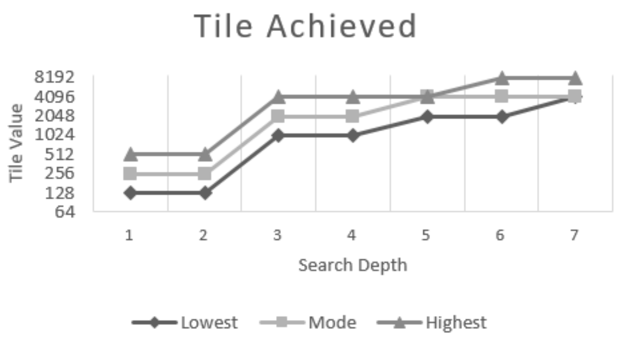

Figure 4: Minimum, maximum, and statistical mode of the highest tile achieved on each trial at each search depth

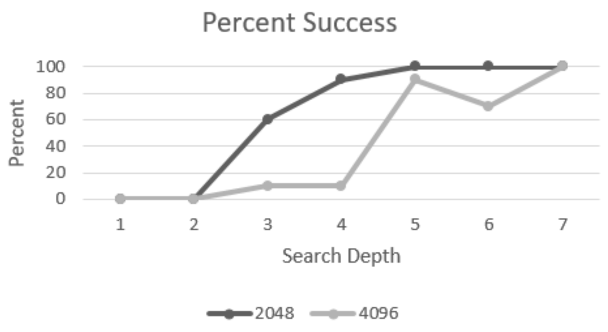

Figure 5: Percentage of trials in which each search depth achieved a 2048 tile and a 4096 tile

From a human perspective, the tile value achieved is more important than the game score, even though game score is a more granular indication of how well the game was played. After analyzing the results of our trials, we determined that 5 is the optimal depth for our algorithm, because it has a high success rate for not only 2048 but 4096 with minimal real-world time loss, and depths higher than that saw a great loss in efficiency.

Conclusion

With the right depth and evaluation function, expectimax is a surprisingly simple and effective strategy for solving 2048. We learned that incentivizing tile combination and weighting tile position was the most effective evaluation strategy, and that a search depth of 5 yielded the optimal balance of performance and speed. Depths smaller than 5 did not consistently achieve 2048, with the exception of depth 4 which reached 2048 in 9/10 trials. However, depth 4 only reached 4096 on 1/10 trials, while depth 5 had a 10/10 and 9/10 success rate for 2048 and 4096, respectively. Depths greater than 5 performed slightly better than depth 5, but also took significantly longer to run. Considering reaching a 4096 tile exceeds a win state, we determine that a depth of 5 is more than sufficient to reliably win 2048.

References

Cirulli, G. 2017. *2048*. https://github.com/gabrielecirulli/2048

Russell, S., and Norvig, P. eds. 2010. *Artificial Intelligence: A Modern Approach.* Upper Saddle River, NJ: Prentice Hall.

Appendix A

Full data sheet for trials of the "grid weights" algorithm at different depths. Time is given in milliseconds.

Depth		1	2	3	4	5	6	7
1								
Time		1849	2262	5095	15390	81254	118525	1595137
Game Score		3480	4004	13080	35496	73352	36052	60284
Best Tile		256	256	1024	2048	4096	2048	4096
2								
Time		904	434	3124	8074	56279	162201	1778148
Game Score		3572	1356	16932	35476	60328	80564	71596
Best Tile		256	128	1024	2048	4096	4096	4096
3								
Time		612	680	4014	13225	41909	210119	1789550
Game Score		3556	3404	32992	34468	57216	133028	80552
Best Tile		256	256	2048	2048	4096	8192	4096
4								
Time		670	1815	2439	4093	53348	121832	1779250
Game Score		4348	3748	21872	15228	80632	52164	78224
Best Tile		256	256	2048	1024	4096	4096	4096
5								
Time		301	640	3482	6697	53375	101803	1407858
Game Score		2448	2448	36048	34616	76416	32920	61280
Best Tile		256	256	2048	2048	4096	2048	4096
6								
Time		219	878	3131	5795	40885	146040	2410605
Game Score		1632	5612	34904	34360	60284	70124	129156
Best Tile		128	512	2048	2048	4096	4096	8192
7								
Time		369	753	1240	6524	41396	140537	1579571
Game Score		4636	6508	13280	40084	60372	61280	59740
Best Tile		512	512	1024	2048	4096	4096	4096
8								
Time		442	438	14277	6524	46350	130419	1294651
Game Score		3796	3500	50432	36952	60560	60428	60451
Best Tile		256	256	4096	2048	4096	4096	4096
9								
Time		101	346	4746	5207	21948	201766	1736418
Game Score		1164	3260	23788	34228	26944	112868	73192
Best Tile		128	256	2048	2048	2048	8192	4096
10								
Time		215	571	2460	7513	49885	91388	714270
Game Score		2800	4624	15744	59152	78988	27576	74932
Best Tile		256	256	1024	4096	4096	2048	4096