ABSTRACT

2048 is a simple and intriguing sliding block puzzle game that has been studied for several years. Many complex solvers, often developed using neural nets are available and capable of achieving very high scores. We are, however, interested in using only basic heuristics, the kind that could be conceivably employed by human players without the aid of computation. A common way to implement a 2048 solver involves searching the game tree for the best moves, choosing a move and scoring the game board using some evaluation functions. The choice in heuristic evaluation function can dramatically affect the moves chosen by the solver. Furthermore, two or more possible moves can frequently produce the same score as evaluated by the same heuristic function, requiring either a random choice, or the use of a secondary or back up evaluation function which itself in turn may produce a tie. In this paper, we test the effectiveness of several basic heuristics in a simple 2048 solver. In order to test these, we create a system that takes basic predefined heuristic evaluation functions as input parameters, generates compositions from these functions with certain rules, and automatically tests all of them with a specified number of games. We find that compositions of evaluation functions that maximize empty spaces and monotonicity of tiles on the board – especially those that prioritize high numbers of empty spaces above prioritizing higher monotonicity – perform the best out of all compositions that we test.

CCS CONCEPTS

- Computing methodologies → Game tree search; Heuristic function construction; Discrete space search;

KEYWORDS

2048, Evaluation, Function, Heuristic, Search

1 INTRODUCTION

2048 is an incredibly popular game created by Gabriele Cirulli and released on March 9, 2014. Soon after release of the original game, it and various clones were downloaded tens of millions of times from various app stores, making 2048 popular and culturally relevant [4]. Completing the game has become a compelling artificial intelligence problem, with at least twenty papers published about the game at the time of this writing. Solvers were created as early as March 19, 2014 [13], and research continues to this day.

In this paper, we focus on solving 2048 using only basic heuristic functions which can be simple enough to be followed by a human player as a strategy. For most non-random evaluation functions, frequently more than one possible move will lead to the same heuristic value. Thus we also explore having a secondary heuristic break a tie, which may also lead to another tie, leading to the necessity for a third heuristic evaluator, etc. At some point a tie can be definitively broken by a terminal evaluator. We call each sequence of basic heuristic functions a “composition”, and we attempt to find the best performing composition in this paper.

2 BACKGROUND AND RELATED WORK

In this section, we cover the rules of 2048, mathematical theory and related work.

2.1 Rules of 2048

2048 is a single-player tile-based game with random elements. It takes place on a 4 × 4 board, and each game starts with two randomly placed tiles. A location on the board will be denoted by its row and column (both of which can be any integer between 0 and 3 inclusive). The player has an integer score.
A tile in the game has a location (a square on the grid) and a value (of the form \(2^k\) for \(k \geq 1\)). See (Fig. 1).

A turn in the game consists of the following: the player chooses a legal move; all tiles are shifted accordingly, merging with any adjacent tile of the same value; and a random tile is placed.

There are four possible moves the player can make: up, down, left, and right. A move is considered legal if it results in the movement of at least one tile and illegal otherwise. When a move is made, all tiles that can move (do not have an edge or another tile of a different value that also cannot move adjacent to them in that direction of movement) move in that direction as far as they can. Note that, when a move is executed, any tile in the row or column closest to the corresponding wall will be moved first, followed by the next row or column, and so on until all tiles in the row or column furthest from the corresponding wall are handled.

A merge happens when a tile with value \(v\) moves into another tile of value \(v\). If this happens, both tiles are removed, and a new tile is placed in the location of the stationary tile with value \(2v\). When this occurs, the player is given \(2v\) points. Note that merging two tiles always counts as tile movement.

After all tiles are shifted, a single tile is placed in a randomly-selected empty square. For any random tile placed (including the first two at the beginning of the game), there is a 90% chance that it has the value 2 and a 10% chance its value is 4.

The object of the game is to merge tiles until one with the value 2048 appears. However, it is possible to continue the game past obtaining the 2048 tile. When none of the four moves are legal, the game terminates. That is, the entire board is filled with tiles such that no two adjacent tiles have the same value.

### 2.2 Theory

The game will always terminate. In fact it is not possible, even with the best of conditions, to get a tile of value greater than \(2^{17}\) on a \(4 \times 4\) grid. This can be seen by noting that the largest tile obtainable from a \(1 \times 1\) grid is \(4 = 2^{(1+1)}\) (assuming that a four gets placed), the largest tile obtainable from a \(2 \times 2\) grid is \(32 = 2^{(4+1)}\) (assuming that a four is given at every stage and in the desired position), and finally the largest tile obtainable from a \(3 \times 3\) grid is \(1024 = 2^{(9+1)}\) (again, assuming that a four is given at every stage and in the desired position). Therefore, the largest tile possible on a board of size \(n \times n\) is \(2^{n^2+1}\).

Langerman and Uno show that the problem of determining if a tile of value \(T\) (where \(T\) is a power of two) can be made on a given board is NP-hard by a reduction from 3SAT [5]. They further show that the problem is inapproximable within a factor of \(\Theta(2^N)\). Abdelkader, Acharya, and Dasler showed that the related problem of determining if it is possible to achieve a 2048 tile on a given configuration of board when no new tiles are added remains NP-Hard [1]. Further, Mehta showed that the question of: “Given an initial configuration of a game board and an oracle, does there exist a sequence of moves to reach a certain configuration?” is PSPACE-Complete (reduced from Nondeterministic Constraint Logic) [7]. Epstein described an optimal strategy for an abstract version of 2048 [3]. Das and Paul analyzed an nth-dimensional version of 2048 and presented strategies for computers and human players [2].

### 2.3 Other Implementations

Many 2048 solving programs exist, both using machine learning and without it. None, however are simple enough to be used by a human player. We briefly summarize the machine learning and the non-machine learning solutions below, before moving on to the human performable basic heuristic analysis that is our own contribution.

Szubert and Jaśkowski approached 2048 in 2014, demonstrating how temporal difference learning and n-tuple networks can be applied to the game [10]. Oka and Matsuzaki exhaustively tested the effectiveness of generating different 6- and 7-tuple networks in a temporal difference learning system [8]. Later, Matsuzaki improved upon this work by considering interinfluence between generated networks, improving the performance [6]. Wu et al. explored multi-stage temporal difference learning and improved on Szubert and Jaśkowski’s network [11].

Robert Xiao et al. created a variable-depth expectimax 2048 solver in 2014. It uses a bitboard representation and efficient row and column transitions to speed up computation time. It also combines several heuristics –rewarding empty squares, large values on the edge, and potential merges–, and penalizing non-monotonic rows and columns whose weights are optimized using the CMA-ES algorithm [13][12]. Using our own implementation, it scored an average of 439,046 in 100 games, and it is the best-performing solver that does not use machine learning. Rodgers and Levine tested two strategies–Monte Carlo Tree Search and Average Depth-Limited Search. Average Depth-Limited outperformed Monte Carlo, though the expectimax solver made by Xiao et al. still scored higher; Rodgers and Levine attributed this to the fact that the expectimax solver’s optimizations allowed it to search deeper than their ADLS solver in the same time frame [9].

### 3 HEURISTICS

In this section, we outline all the basic heuristics and how we implement them as evaluators for this research.

Let \(M\) be the set of all possible moves and \(G\) be the set of all possible games. An evaluator is a function of the form \(e : G \times P(M) \rightarrow P(M)\). For each move in the input set, if that move is legal, the evaluator applies that move to the game, then applies certain criteria to assign a score to the new game state. It then creates a move or set of moves that produce the highest score.

#### 3.1 Nonterminal Evaluators

An nonterminal evaluator returns a set of moves with at least one element but with no more than the number of elements in the input parameter \(M\).

- **Greedy.** The greedy solver examines the official score of each game after applying each move from the input set and returns the moves that result in the highest score. Unsurprisingly, this method by itself is not guaranteed to result in the highest scoring game.

- **Empty.** The empty solver prioritizes moves that create the most empty tiles.

- **Uniformity.** The uniformity solver selects for moves that produce the most tiles of the same value. The uniformity score of a
Algorithm 1: The general evaluation procedure

Input: \( G \) - a game, \( M \) - a set of moves
Output: The best moves according to the specific evaluator

1: procedure EVALUATEGAME(G, M)
2: \( \text{bestScore} \leftarrow -1 \)
3: \( \text{bestMoves} \leftarrow \emptyset \)
4: for all \( \text{move} \in M \) do
5: \( G' \leftarrow G \)
6: apply \( \text{move} \) to \( G' \)
7: \( \text{curScore} \leftarrow \text{the score given by the evaluator's specific criteria} \)
8: if \( \text{curScore} > \text{bestScore} \) then
9: \( \text{bestScore} \leftarrow \text{curScore} \)
10: \( \text{bestMoves} \leftarrow \{\text{move}\} \)
11: else if \( \text{curScore} = \text{bestScore} \) then
12: \( \text{bestMoves} \leftarrow \text{bestMoves} \cup \{\text{move}\} \)
13: end if
14: end for
15: return \( \text{bestMoves} \)
16: end procedure

Figure 2: A 2048 game state exhibiting high monotonicity

The game \( G \) is:

\[
\sum_{v \in V_G} (n_v)^3
\]

where \( V_G \) is the set of all unique values present in any tile in game \( G \) and \( n_v \) is the number of tiles with value \( v \).

3.1.4 Monotonicity. We consider a game to have high monotonicity if the values are either non-increasing along all rows or non-decreasing along all rows and if the values are non-increasing along all columns or non-increasing along all columns, with the highest value being in one of the four corners (see 2). Algorithm 2 describes a simplified version of the process we use to score a single game state’s monotonicity.

Algorithm 2: Scoring a game’s monotonicity

Input: \( G \) - a game
Output: The game’s monotonicity score

1: procedure SCOREMONOTONICITY(G)
2: \( \text{best} \leftarrow -1 \)
3: \( \text{current} \leftarrow 0 \)
4: for \( i \leftarrow 1, 4 \) do
5: \( \text{for} \ \text{row} \leftarrow 0, 3 \) do
6: \( \text{for} \ \text{col} \leftarrow 0, 2 \) do
7: if \( G[\text{row}][\text{col}] \geq G[\text{row}][\text{col} + 1] \) then
8: \( \text{current} \leftarrow \text{current} + 1 \)
9: end if
10: end for
11: end for
12: for \( \text{col} \leftarrow 0, 3 \) do
13: \( \text{for} \ \text{row} \leftarrow 0, 2 \) do
14: if \( G[\text{row}][\text{col}] \geq G[\text{row} + 1][\text{col}] \) then
15: \( \text{current} \leftarrow \text{current} + 1 \)
16: end if
17: end for
18: end for
19: if \( \text{current} > \text{best} \) then
20: \( \text{best} \leftarrow \text{current} \)
21: end if
22: \( \text{Rotate the board 90 degrees clockwise} \)
23: end for
24: return \( \text{best} \)
25: end procedure

3.2 Terminal Evaluators

For each nonterminal evaluator, there is a chance that the output set may have multiple moves. Often, pairs of opposite directions (both left and right or both up and down) will be scored the same by a nonterminal evaluator. However, the solver must only choose one move per turn. There needs to be a guarantee that only one move will be chosen.

Terminal evaluators are guaranteed to return only one move, no matter how many moves are in their input sets.

3.2.1 Random. The random evaluator is a terminal evaluator that randomly selects a move from its input set and returns only that move.

4 STRATEGIES

A strategy is an ordered list of evaluators, denoted \( S = (E_1, E_2, \ldots, E_k) \). Strategies must follow these rules:

1. No evaluators are repeated
2. A strategy contains exactly one terminal evaluator
3. The terminal evaluator is the last evaluator in the strategy

Note that a strategy does not have to use every nonterminal evaluator in \( E \) as long as the rules are adhered to. Therefore, \( 1 \leq |S| \leq |E| \) for any strategy \( S \).
4.1 Generating All Strategies

The set of all unique strategies given a set of evaluators \( E \) is \( S_E \) and is generated from permutations of \( E \). A simplified version of the process to generate \( S_E \) is described in Algorithm 3. In the code, permutations of all nonterminal evaluators are generated first, then the terminal operator is inserted into position of every permutation and the nonterminal evaluators after are removed.

Algorithm 3 Generating all strategies from a set of evaluators

Input: \( E \) - a set of evaluators such that there is exactly one terminal evaluator and, if there are any nonterminal evaluators, each is unique

Output: \( S_E \)

1: procedure GenStrategies\((E)\)
2: \( P_E \leftarrow \) the set of permutations of \( E \)
3: \( S_E \leftarrow \emptyset \)
4: for all \( p \in P_E \) do
5: \( i \leftarrow \) the index of \( p \)'s terminal evaluator
6: \( p' \leftarrow p_0,i \)
7: if \( p' \notin S_E \) then
8: \( S_E \leftarrow S_E \cup \{p'\} \)
9: end if
10: end for
11: return \( S_E \)
12: end procedure

Note that, since some permutations are thrown out in the creation of \( S_E \) if \( P_E \) denotes the set of permutations of \( E \), such that \( 1 \leq |S_E| \leq |P_E| \).

4.2 How Strategies Are Used

A strategy is applied as follows: If the first evaluator in the list produces a single best legal move, then that move is made without further processing. If there is a tie, then the second evaluator is used with the input being the set of tied moves from the first, and so on.

Note that since, by definition, a strategy must contain a terminal evaluator, the move selection algorithm always terminates.

This flexible movement selection procedure allows us to automate performance testing of many different strategies.

5 EXPERIMENTS

5.1 Procedure

We generate \( S_E \) using the set \( E = \{\text{Monotonicity, Empty, Uniformity, Greedy, Random}\} \). Then, for each strategy composition, we run 1000 games, recording mean, standard deviation, median, and highest score for each. The results for all solvers were sorted by mean, standard deviation, median, maximum score, then name.

6 RESULTS

Table 1 shows the performances of each evaluator by itself, with the random evaluator used to resolve any ties. Table 2 shows the ten strategies that performed the best, sorted by mean.

The results of running all strategies 1000 times show that strategies that have \textit{empty} before \textit{monotonicity} are the most effective.

Table 1: Key And Individual Performances

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Evaluator</th>
<th>Mean</th>
<th>Std Dev</th>
<th>Median</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>Monotonicity</td>
<td>3177.17</td>
<td>1196.25</td>
<td>3218</td>
<td>7636</td>
</tr>
<tr>
<td>u</td>
<td>Uniformity</td>
<td>1079.36</td>
<td>516.63</td>
<td>1038</td>
<td>2996</td>
</tr>
<tr>
<td>e</td>
<td>Empty</td>
<td>1948.13</td>
<td>878.63</td>
<td>1652</td>
<td>6336</td>
</tr>
<tr>
<td>g</td>
<td>Greedy</td>
<td>1090.64</td>
<td>530.21</td>
<td>1044</td>
<td>3088</td>
</tr>
<tr>
<td>r</td>
<td>Random</td>
<td>1093.87</td>
<td>539.75</td>
<td>1058</td>
<td>4348</td>
</tr>
</tbody>
</table>

Table 2: Top Performances of All Strategies

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Mean</th>
<th>Std Dev</th>
<th>Median</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>emr</td>
<td>3947.83</td>
<td>1883.28</td>
<td>3442</td>
<td>12488</td>
</tr>
<tr>
<td>emur</td>
<td>3946.13</td>
<td>1857.78</td>
<td>3444</td>
<td>12248</td>
</tr>
<tr>
<td>eumr</td>
<td>3944.40</td>
<td>1861.71</td>
<td>3480</td>
<td>12688</td>
</tr>
<tr>
<td>egrm</td>
<td>3930.66</td>
<td>1824.02</td>
<td>3510</td>
<td>12872</td>
</tr>
<tr>
<td>eurmr</td>
<td>3925.01</td>
<td>1821.80</td>
<td>3458</td>
<td>12264</td>
</tr>
<tr>
<td>guemr</td>
<td>3919.58</td>
<td>1820.29</td>
<td>3462</td>
<td>12292</td>
</tr>
<tr>
<td>guemr</td>
<td>3895.64</td>
<td>1833.61</td>
<td>3444</td>
<td>12220</td>
</tr>
<tr>
<td>eugmr</td>
<td>3883.58</td>
<td>1820.29</td>
<td>3448</td>
<td>12376</td>
</tr>
<tr>
<td>eugmr</td>
<td>3879.13</td>
<td>1812.96</td>
<td>3474</td>
<td>12356</td>
</tr>
</tbody>
</table>

This is followed by strategies that use \textit{monotonicity} before empty, strategies that use \textit{monotonicity} but not \textit{empty}, strategies that use \textit{empty} but not \textit{monotonicity}, and, finally, strategies that use neither \textit{empty} nor \textit{monotonicity}. The ordering of median scores followed this trend as well. The top three highest performing strategies from this test were \{empty, monotonicity, random\}, \{empty, monotonicity, uniformity, greedy, random\}, and \{greedy, empty, uniformity, monotonicity, random\}.

7 CONCLUSIONS

Strategies that select for both boards with a large number of empty spaces then high monotonicity perform better than other strategies. Therefore, basic heuristic functions for 2048 solvers should prioritize high numbers of empty spaces and monotonicity, and they should prioritize having a high number of empty spaces over having high monotonicity. To translate the top strategy shown, \textit{emr}, to human instructions, we could write

Try first to do whatever move allows for most empty tiles. If there is more than one such move, then try to pick one that contributes to monotonicity of the board. If there are still more than one moves then just pick one of them at random.

REFERENCES