Senior Thesis in Mathematics

2048: Algorithms and Artificial Intelligence (version 0.2)

Author:
Neel Adhiraj Kumar

Advisor:
Dr. Vin de Silva

Submitted to Pomona College in Partial Fulfillment
of the Degree of Bachelor of Arts

February 17, 2018

Abstract

In this paper we investigate algorithmic and artificial intelligence approaches for playing the game *2048*. We use blind algorithms, heuristic algorithms, and monte carlo tree search methods, and evaluate their results. Through this, we aim to learn something about puzzle-solving, pattern finding, algorithms, and intelligence.

Contents

Chapter 1

Introduction

Pattern-finding is at the core of intelligence, according to Douglas Hofstadter, cognitive-scientist, artificial intelligence researcher, and Pulitzer-prize winning writer. Everything from concrete sensory experience to abstract mathematics relies on our minds recognising and constructing meaningful patterns. One way to study this most abstract and profound of human qualities is to simulate it, in other words develop artificial intelligence (AI) that can exhibit this ability.

The field of AI has recently seen significant progress. One example of this is the victory of AlphaGo over the human champion Lee Sedow at Go, a game many orders of magnitude more complex than chess [7]. The same research group recently developed AlphaGo Zero, a reinforcement learning AI that learns to play the game entirely on its own (without observing human plays) that beat AlphaGo 100-0 [8]. AI for games is itself a rich domain of research. Games provide a limited and clearly defined environment where we can observe, emulate, and evaluate (aspects of) intelligence. There tends to be a natural, in-built measure of performance - usually a game score, or a winner. Success at games can involve strategy, planning, problem solving, in other words, intelligence. However, they also tend to be limited in scope, making it possible to comprehensively model them. Developing AI for games involves all the classical AI topics: knowledge epresentation, searching, and learning. Hence, games provide a useful framework to build, test, and study AI.

This thesis investigates algorithmic techniques and AI approaches to the single-player sliding-block puzzle game, *2048*. Success in the game seems to require the strategic detection and manipulation of patterns on the board towards a set of goals. The number of possible board states is too high for brute-force searches, making the solution non-trivial, and human-players certainly do not play by calculating all possibilites. Hence, developing AI for the game not only involves modelling and simulating abstract pattern finding skill and to an extent, mathematical ability (itself a component of intelligence), but also allows us to try out a variety of AI techniques.

I start by describing the mechanics of the *2048* game, and discussing the game's addictiveness and value as a framework for studying AI. After describing the implementation of the game, I discuss various classes of algorithms and their performance.

Chapter 2

2048

2048 is a puzzle game developed by Gabriele Cirulli, released open-source and for free in March 2014. The game is a clone of *1024*, itself a clone of *Threes* that had been released earlier as a paid phone application [6]. *Threes* won several awards, and spawned dozens more clones, still available as mobile apps and on websites, all of which together have been downloaded and played millions of times. *2048*, in particular, became wildly popular.

2.1 Mechanics

The *2048* board is a 4×4 grid. Each entry on the grid is either a tile with a positive power of 2, or empty (no tile). The game starts with 2 randomly spawned tiles. Figure 2.1 shows an example.

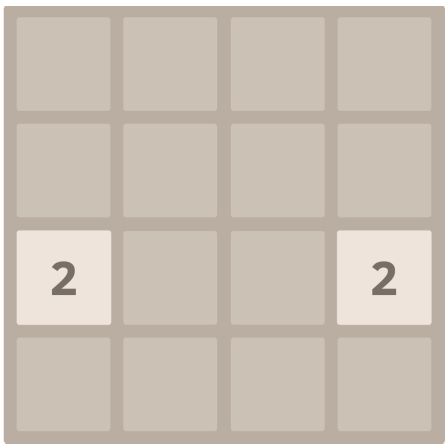

Figure 2.1: Example of a starting game board

- Players slide all the blocks on the board in one of four directions: up, down, left, right.
- Tiles with the same number combine when slid into another. There is a subtlety to tile merging that is explained in section 2.1.2.

- After each succesful move, a new random tile spawns on any of the empty spots on the grid (with equal probability).
- There is a 0.9 probability that the tile spawned will be a 2, and a 0.1 probability that it will be a 4.

Figure 2.2 shows the board in Figure 2.1 after a left swipe. The two 2 tiles combined to form a 4 tile, and a new 2 tile was generated in one of the empty spots on the grid.

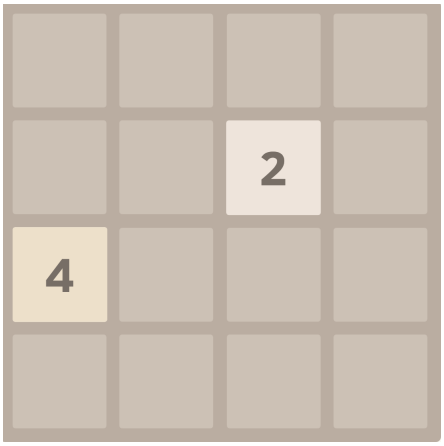

Figure 2.2: The game board from Figure 1.1 after a left swipe

The game ends when the player is stuck, meaning there are no more possible moves. This means that swiping in any of the four directions, will not cause any tile to merge or move. An example is shown in Figure 2.3.

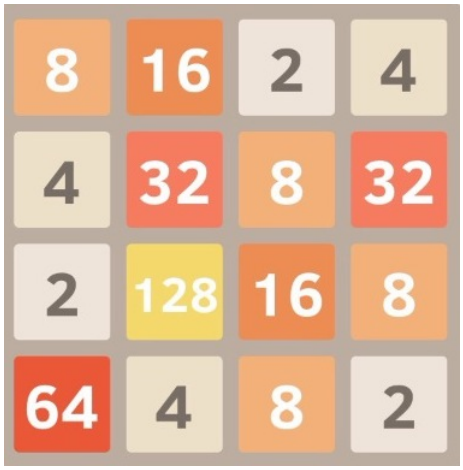

Figure 2.3: Example of a 'game over' board with no moves left

2.1.1 Objective

The official goal of the game is to succesfully create a 2048 tile. This is challenging because creating one requires first building 2 1024 tiles, which require 4 512 tiles, 8 256 tiles and so

on. The randomness in tile spawns prevents (at least human) players from being able to calculate all possibilities and develop optimal strategies. The game does not end, however, after a 2048 tile is built. In practice, players try to build higher and higher tiles, and with artificial intelligence techniques, tiles up to $65,536$ have been achieved [9].

As one builds higher and higher numbered tiles on the board, there are more and more different tiles on the board that cannot merge, increasing the chance of getting stuck. In fact, for this reason, there is a theoretical upper bound on the maximum tile.

Theorem 2.1 *On a generalised 2048 board with n spaces, the largest tile we can construct using $n-k$ spaces is 2^{n-k+1}.*

Proof

We prove by induction on $n-k$.

If $n-k=1$, we can construct a maximum tile of $4=2^2=2^{1+1}$ (by having it spawn).

Assume that with $n-k$ spaces, we can make a maximum tile of 2^{n-k+1}.

Then with $n-k+1$ spaces:

To build a 2^{n-k+2} tile, in the turn immediately before, I need two 2^{n-k+1} tiles to combine.

On the turn immediately before that, I have one 2^{n-k+1} and $n-k$ spaces with which to build another 2^{n-k+1} tile. By assumption, this is possible. Hence, I can build a 2^{n-k+2} tile.

However, to build a 2^{n-k+3} tile, in the turn immediately before, I need two 2^{n-k+2} tiles to combine.

On the turn immediately before that, I would have one 2^{n-k+2} and $n-k$ spaces with which to build another 2^{n-k+2} tile. By assumption, this is impossible. Hence, I cannot build a 2^{n-k+3} tile.

∎

Corollary 2.2 *The maximum tile possible in a standard 4×4 game of 2048 is $131072=2^{17}$.*

Note that even a 'perfect' algorithm (if it exists) has only a 0.1 probability of making it to 131072, since it requires spawning a 4 at the right moment.

There is also a score function built into the game. Every time a x tile is created, the score increases by x. Randomly generated tiles do not provide any score points.

Theorem 2.3 *The score contribution of the tile 2^k, is lower bounded by $(k-2)2^k$ and upper bounded by $(k-1)2^k$, for $k\geq2$.*

Proof Remember that when we create a 2^k tile, we get 2^k points added to our score. We prove by induction.

A 2 tile contributes nothing to our score.

A $4=2^2$ tile can either be generated randomly, which contributes $0=(2-2)2^2$ to our score,

or it can be the result of combining 2 2^1 tiles, which contributes $4=(2-1)2^2$ to our score.

Assume that creating a 2^k tile, adds at least $(k-2)2^k$ points to our score, and at most $(k-1)2^k$ points.

Then if we create a 2^{k+1} tile, we need to combine 2 2^k tiles. At the minimum, each of these have already contributed $(k-2)2^k$ to our score.

Now we combine them to get a further 2^{k+1} points.

Hence the total is at least: $2 \times (k-2)2^k + 2^{k+1} = (k-2)2^{k+1} + 2^{k+1} = 2^{k+1}(k-2+1) = 2^{k+1}(k-1)$.

Similarly, at most each of the 2 2^k tiles have contributed $(k-1)2^k$ to our score.

Now we combine them to get a further 2^{k+1} points.

Hence the total is at most: $2 \times (k-1)2^k + 2^{k+1} = (k-1)2^{k+1} + 2^{k+1} = 2^{k+1}(k-1+1) = 2^{k+1}k$.

∎

Corollary 2.4 *If the maximum tile is 2^k, the maximum game score possible is $2^{n+4}(n-1)$*

Proof The maximum score possible would be achieved if every tile on the grid is as large as possible.

Since the maximum tile is 2^k, the maximum score is achieved when all 16 tiles on the board are 2^k tiles.

Each tile contributes a maximum of $(k-1)2^k$ to the score. Hence all of them contribute a maximum of:

$$16 \times (k-1)2^k = 2^4 \times (k-1)2^k = (k-1)2^{k+4}$$

∎

This theorem gives us a way to understand the score function of the game, by thinking of the various maxima tiles in terms of their score contributions. The difference between the maxima and the minimima is based on whether all the 4 tiles used to build the tile in question were randomly generated or built using 2 2 tiles.

2.1.2 Tile Merging Subtlety

As mentioned earlier, there is a subtlety to the game's tile merging mechanics that can impact gameplay and strategy. On each move, a tile can only merge once, though it will always move as far as it can go.

For example, starting in Figure 2.4, if we swipe up, we end up at Figure 2.5. In particular, in the 3rd column, the newly formed 4 tile does not combine with the existing 4 tile, even though it is moving towards it, since it has already been combined this turn. Similarly, the newly formed 8 tile does not combine with the existing 8 tile.

2.2 Why we care

2048 is a game of mathematics and strategy. A succesful player detects meaningful patterns on the board and manipulates them strategically towards his goal (which may be changing and composed of many subgoals). There is no 'opponent', unlike in Chess or Go, which makes the game a personal, mental challenge, like a puzzle or a good mathematics problem.

Tile	Minimum	Maximum
4	0	4
8	8	16
16	32	48
32	96	128
64	256	320
128	640	768
256	1536	1792
512	3584	4096
1024	8192	9216
2048	18432	20480
4096	40960	45056
8192	90112	98304
16384	196608	212992
32768	425984	458752
65536	917504	983040
131072	1966080	2097152

Table 2.1: Score contributions of various tiles

Figure 2.4: An example game board

2.2.1 Why is it so addictive?

2048 became a viral phenomenon. Many explanations have been offered for its popularity [3].

Like many good games, it is easy to learn, but hard to master. The mechanics of the game are quite straightforward, elegant even. Players only have up to 4 move choices each turn - up, down, left, or right. It is actually quite difficult to lose the game in the first few dozen moves, and as we shall see, even a trivial strategy will score at least a few hundred points.

Tiles numbers increase quickly in the beginning, giving you a sense of progress and some confidence in your ability that keeps you trying as the game gets increasingly difficult. The exponential growth of numbers makes winning seem closer than it really is. For instance, if you get one 1024 tile, you feel like you've almost won, when in reality, you've just passed

Figure 2.5: The game board after moving up (with a tile spawn)

the halfway point to getting 2048.

Most moves are 'easy' in the sense that there is an obvious move (or set of moves) that is 'better' than others, given usual strategies. These can be played with little to no thought, keeping the game moving in between 'hard' moves, which can be quite tricky. This keeps the game engaging and fast-paced, as opposed to some puzzles where every move must be carefully considered.

The new tile spawns randomly on an empty tile, and its value is randomly chosen between 2 or 4 (the former is 9 times more likely). This stochasticity drastically changes the set of possible strategies for the game. After each move of the player, the game board can change in up to 28 different ways (there are always at least 2 tiles occupied on the board, the other 14 tiles can each spawn a 2 or a 4). Even with an almost full game board, different tile spawns may have drastically different implications for strategy. It is difficult for human players to predict the game state (i.e. the state of the game board) many moves ahead. Even computers cannot search all possible outcomes. Hence, playing the game becomes less a matter of calculating far ahead and more a matter of strategically mantaining and exploiting board symmetries, while trying to minimise the probability of catastrophe.

2.2.2 Domain for algorithms and artificial intelligence research

It is certainly possible, and perhaps worthwhile, to analyse the mathematics of the game and identify optimal moves in various cases. However, this is not the purpose of this thesis. Instead, we will focus on developing algorithmic techniques for the game, in a sense solving the game at one level of generality. We will only analyse the game in so far as it assists us in this goal. To be more specific, we are not interested in finding the best move to make in a particular instance (a given, particular board combination), as much as finding ways of playing the game that are successful across all instances of the game. This requires finding useful patterns and manipulating them strategically, an ability with far more general utility than the game of *2048*.

2048 is a useful domain for this problem for several reasons, many of which are what make it so addictive. The simplicity and mathematical elegance of the game simplifies modelling the game and analysing and developing algorithms. The fact that there is only one player removes complications arising from opponent matching and multiple algorithms working

together.

While the domain is certainly limited (what makes it seem so easy), within those limits, the game is actually quite demanding, and mathematically rich. Its stochasticity makes calculating game trees computationally prohibitive, meaning we must make our move without being able to fully calculate where it might lead us. Since brute-force calculation is impossible, we must choose between more and less computationally intensive approaches, which may involve meaningful trade-offs applicable to other domains. The exponential increase in difficulty in creating higher and higher tiles provides ample room to test vastly different levels of performance.

Chapter 3

Implementation

This thesis uses an implementation of the game in Python. The implementation began as an expansion of starter code for an introduction to computer science course [4], and has since been revamped several times.

The board is represented as 16 hexadecimal entries, one for each tile in a 4×4 board. Each hexadecimal entry can contain one of 16 values, allowing us to encode empty tiles and positive powers of 2 from 1 to 15. This is an efficient way to encode tiles up to 32768, which is enough for our purposes.

Following Robert Xiao's lead, we implement tile movements on the game using a look-up table of 16 bit (unsigned) integers [9]. When we have to make a move, we first use board symmetries to rotate the board so the same move is now on the left. For instance, when our original move is left, we do nothing. If our original move is right, we flip the board so that each horizontal row is reversed, and similarly rotate the board in the up and down cases. Now moving our original board in our intended direction will be equivalent to moving our new board left.

Then we consider each row on our new board (which we have to move to the left) individually. We encode the row as a 16 (unsigned) bit integer, which gives us 4 bits to encode every tile, allowing us to again encode 16 values, which gives us the same 15 powers of 2 and the empty tile. We use the pre-computed look-up table to see what this row gets transformed to after a left move. Once we have all the new rows, we rotate/flip the board back. This allows us to compute moves quickly without having to check for tile merging subtleties during each move.

The code is in the appendix.

A runnable version is on: https://github.com/nadhirajk/Thesis

Chapter 4

Related Work

The first artificial intelligence for *2048* published online was a Mini-max search with alpha-beta pruning [5]. This technique has been applied successfully for other discrete state space, perfect information, turn-based games like chess and checkers. It can be applied in a 1-player game like *2048* by treating the game as your opponent, always assuming it will spawn the worst possible tile. The state evaluation function rewards monotonicity (the values of tiles are either all increasing or decreasing horizontaly and vertically), smoothness (a small difference in value between adjacent tiles) and empty tiles. While it seems reasonable to maximise the score in the worst case scenario, this approach ends up being too cautious by giving too much weight to highly unlikely events since most tile spawns are not that bad. This approach combines human intelligence (choosing the heuristics for the evaluation function) with computation (searching the game tree to evaluate possibilities).

Robert Xiao implemented an Expectimax algorithm, which like Mini-max search, is a recursive, depth-limited tree search algorithm, with a similar evaluation function. [9] However, instead of maximising the minimum score, expectimax maximises the expected score (i.e. the average, weighted by probability). By making decisions based on what is likely, and using carefully optimized brute force search methods and a highly efficient implementation, the algorithm can score 8192 100% of the time, and 32768 36% of the time, with each move taking approximately 150 ms. The AI involves hard-coded intelligence in the form of heuristics similar to the minimax search above, and heavily relies on compuation to both optimise these heuristics and search the game tree. The same StackOverFlow thread where these results were posted describes several other algorithms, including top-down decision making algorithms modelling human player strategies, reinforcement learning, and monte carlo tree search [1].

Szubert and Jaskowski [2] first applied Temporal Difference learning, a form of reinforcement learning to the game, using 1 million games to train an afterstate-value function represented by a systematic n-tuple network. Yeh et al. [10] extended this by using 5 million training games to learn a larger systematic n-tuple system, scoring 142,727 on average. By employing three such n-tuple networks enabled at different stages of the game along with expectimax with depth 5, they were able to achieve 328,946 points on average. This program outperforms all the known 2048 programs up to date, except for Robert Xiao's, which is 100 times slower. These algorithms start with no previous knowledge of the game (for instance, as heuristics designed based on human strategies).

Chapter 5

Blind Algorithms

This chapter describes simple algorithms to play the *2048* game. These algorithms are 'blind', in the sense that they make moves without looking at the current state of the board. After brief descriptions, I discuss their performance. These algorithms are not meant to represent intelligence, but rather provide a performance benchmark for later algorithms, allowing us to distinguish 'intelligent' algorithms from merely lucky ones. Due to their simplicity, they are also useful for understanding the concept of algorithmic gameplay. Later, more sophisticated algorithms may use these in special cases.

Remember: the game ends when no more moves are possible. If there is only one move possible, we must make that move. Hence, all the algorithms described in this thesis assume there is more than one possible move.

5.1 Descriptions

5.1.1 Random

Choose a random (possible) move (uniformly).

5.1.2 Sequential

loop
    Choose right
    Choose down
    Choose left
    Choose up
end loop

If any of the moves above is impossible when it is performed, the sequence simply continues in the above order.

5.1.3 Randomly Alternate

loop

Choose randomly between up and down
Choose randomly between right and left
end loop

The algorithm randomly chooses a move from possible moves on each axis. Hence on alternate moves, the algorithm moves on alternate axes.

In other words, if we number each move, on odd numbered moves, we randomly choose between up and down, and on even numbered moves, we randomly choose between left and right. If the chosen move is impossible, the program simply makes the opposite direction move and then alternates to the other axis of movement.

5.1.4 Always Up

if up move possible **then** move up
else if left move possible **then** move left
else if right move possible **then** move right
end if

If none of these 3 moves are possible, we are forced to make a down move, and hence as stated before, the algorithm is unnecessary.

5.2 Performance

Algorithm	Average	Minimum	Maximum	Standard Deviation
Random	1140	108	4670	576.79
Sequential	2650	488	7630	1257.52
Randomly Alternate	1870	260	6750	934.69
Always Up	1140	116	3640	565.56

Table 5.1: Blind Algorithms Score Statistics (1000 trials each)

Algorithm	Minimum	Maximum
Random	16	512
Sequential	64	512
Randomly Alternate	32	512
Always Up	16	256

Table 5.2: Blind Algorithms Maximum Tile Statistics (1000 trials each)

Despite completely ignoring the board state before choosing their moves, these 'blind' algorithms manage to build 512 tiles (Sequential and Randomly Alternate), and score over 2000 points on average (Sequential). Some instances even managed to cross 6000.

Clearly, making it to 512 is no mark of skill or 'intelligence'. However, none of these algorithms even once achieved a 1024 tile (which requires a minimum score of 8192), let alone a 2048 .

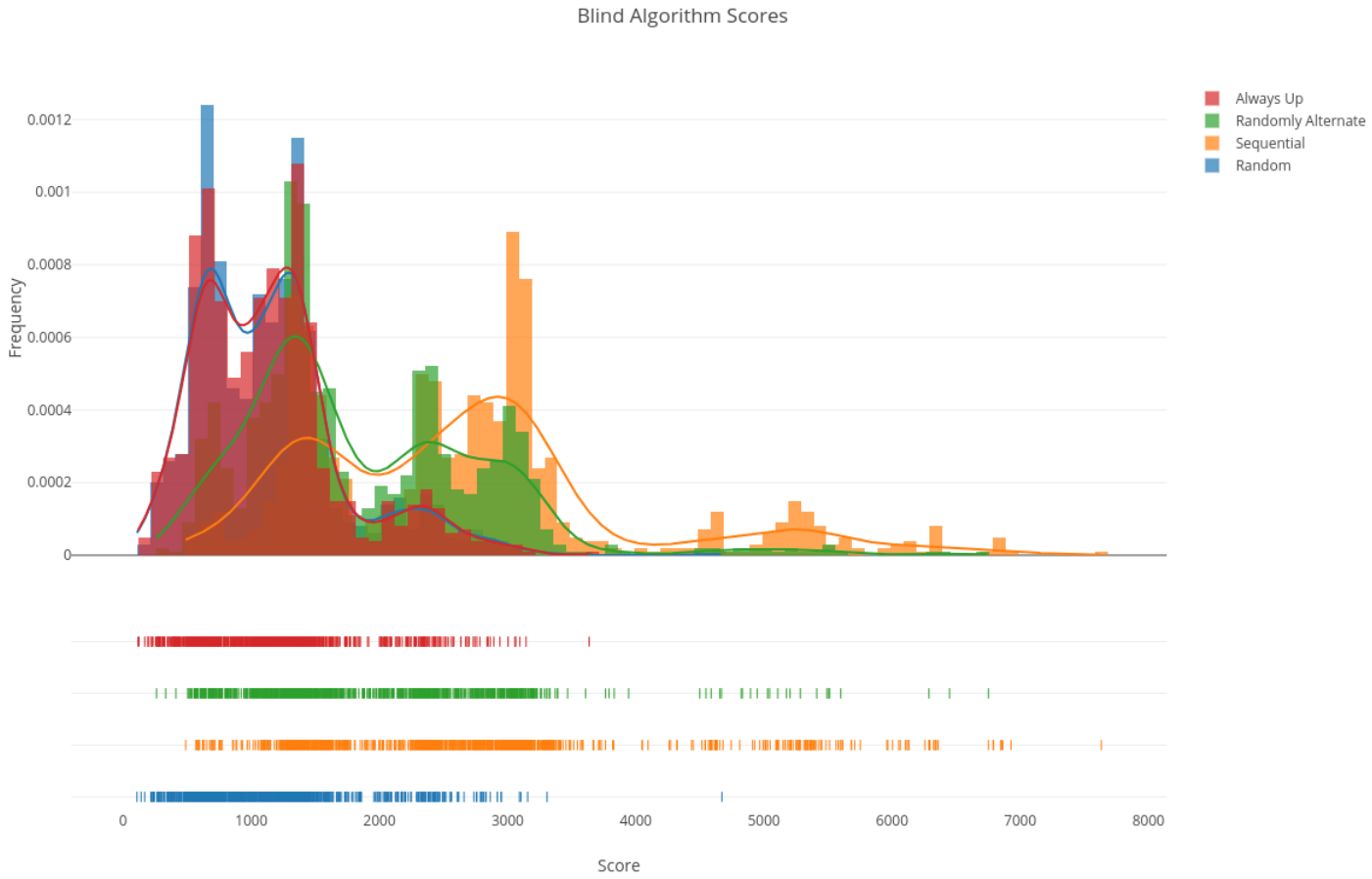

Figure 5.1: Blind Algorithm Score Statistics (1000 trials each)

Chapter 6

Heuristic Algorithms

This chapter describes algorithms that look at the tiles on the board, and move according to some strategy. These algorithms are simplistic models of heuristics human players often employ while playing the game, along with some simple strategies that, at least intuitively, seem like they should be better than 'blind' algorithms.

6.1 Max-in-Corner (Unique)

6.1.1 Description

This is the first algorithm that has some strategy behind its moves, even if only for limited cases. The intuition is to try and keep the biggest tile in a corner, a strategy most successful players seem to employ. This is because the after a new maximum tile has been built, it is (by definition) larger than all other tiles on the board, and hence cannot merge with them. By keeping it in a corner (where it only has 2 adjacent tiles), we leave more space for other tiles (which me may be able to merge) to occupy positions with more adjacent tiles, decreasing the possibility of getting stuck.

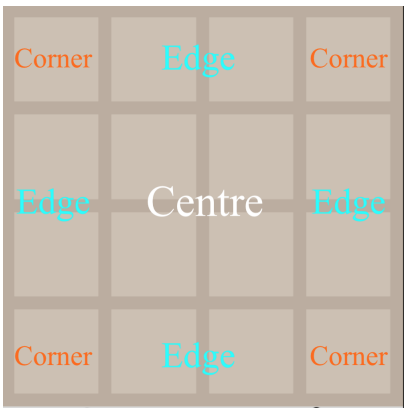

Figure 6.1: Centre tiles have 4 adjacent tiles, edge tiles have 3, and corner tiles have 2.

Each corner has 2 directions which we will call the corner directions. For instance, the top-right corner has the 2 corner directions: up and right. Moving in either of these directions

keeps the tile in the corresponding corner fixed.

Figure 6.2: Moving up or right will keep the 64 tile fixed

Each edge tile has a closest wall and a closest corner. The second closest corner is the other corner on the closest wall.

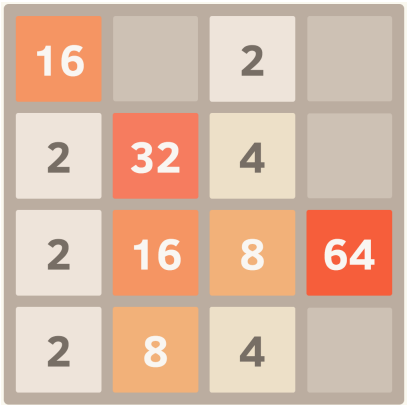

Figure 6.3: The 64 tile is closest to the bottom-right corner and the right wall. The second closest corner is on the top-right

If a tile is not in a corner or an edge, it must be one of the 4 center tiles. Each of those has a closest corner and a furthest corner (the opposite corner).

6.1.2 Performance

The average score is:1443.88
The minimum score is:312
The maximum score is:5752
The standard deviation of scores is:810.840246658
The average log score is:7.12993368721
The standard deviation of log scores is:0.552478950765
The average max tile is:126.72
The minimum max tile is:32

Max-in-corner (Unique)

if no unique maximum tile **then** choose random move
else ▷ if a move is impossible it is ignored in the random choice
  if maximum is in a corner **then** move randomly in one of the 2 corner directions
  else if maximum is on an edge **then** move randomly with following probability weights:
    towards closest corner with weight 4
    towards second closest corner with weight 1
    towards closest wall with weight 1
  else move randomly with following probability weights :
    towards closest corner with weight 4
    in each of the opposite corner directions with weight 1
  end if
end if

The maximum max tile is:512
The standard deviation of the max tile is:68.4710274496

6.2 Max-in-Corner (All)

6.2.1 Description

For simplicity, the previous algorithm works only in the case of a unique maximum tile. Here, we extend that algorithm to the case of multiple maxima. The intuition again is to try and keep the maxima in a corner. How this should be done depends on the 'preferences' of each of the maxima (with some stochasticity).

Essentially, we choose randomly among the moves that keep whatever maxima we have in the corners stationary.

If no such move exists (for instance, if we don't have any maxima in corners), then we try and bring maxima from the edges and center to a corner. To do this, we calculate probabiltiy weights for all possible moves. For each maximum, we add weights to prefer moves that would bring it to corner. Then we choose a move randomly based on these weights.

Note that since a move can be added multiple times to the preferred moves vector, if we have multiple maxima in corners, any move preferred by multiple maxima would be more likely to be sampled in the random selection.

6.2.2 Performance

The average score is:2239.0
The minimum score is:212
The maximum score is:6172
The standard deviation of scores is:1194.96709578
The average log score is:7.55492709775
The standard deviation of log scores is:0.600349270904

Max-in-corner (All)

initialise empty preferred moves vector
initialise empty weighted moves vector
for all maximum tiles **do**
if maximum is in a corner **then** add the 2 corner directions to the prefered moves vector ▷ the same move can be added multiple times to this vector
else if maximum is on an edge & preferred moves vector is empty **then** add following weights to the moves in the weighted moves vector:
towards closest corner add weight 4
towards second closest corner add weight 1
towards closest wall add weight 1
else if maximum is on a centre tile & preferred moves vector is empty **then** add following weights to the moves in the weighted moves vector:
in the closest corner directions add weight 2
in the opposite corner directions add weight 1
end if
end for
if preferred moves vector is not empty **then** pick randomly from preferred moves vector
else pick from weighted moves vector with probabiltiy weighted by the weight of the move
end if

The average max tile is:187.04
The minimum max tile is:16
The maximum max tile is:512
The standard deviation of the max tile is:105.057881189

Neither of the Max-in-Corner algorithms perform statistically better than a blind algorithm. This strategy is possibly too simple to be successful across a game. Perhaps it would be better to bring maxima together than move them to a corner.

6.3 Greedy

6.3.1 Description

This is a greedy algorithm, which means it uses the intuition that choosing a local optimum every turn will lead us towards a global optimum. Hence, it chooses the move with the largest score increase every turn, in the hope that this strategy will lead us to a high score in the long run.

6.3.2 Performance

The average score is:1500.04
The minimum score is:552
The maximum score is:3760

Greedy

```
initialise empty vector
save the current board state
for all possible moves do
    play the move on saved board
    save score in vector
end for
return game board to saved board
choose move corresponding to the highest score increase
```

The standard deviation of scores is:721.940162617
The average log score is:7.20359270583
The standard deviation of log scores is:0.467043453267
The average max tile is:140.8
The minimum max tile is:64
The maximum max tile is:256
The standard deviation of the max tile is:70.1084873607

Perhaps surprisingly, the Greedy algorithm performs worse than some of the Blind algorithms. This suggests that choosing the best move each turn is too short-sighted to perform well in the long-run.

When possible, the algorithm will choose to merge high-value tiles (since this creates a high score increase), which seems like the correct move to make since it creates space on the board. However, for the vast majority of moves when there are no high-value tiles that can be merged, this will lead to simply choosing the move with the most merges (since that is how we get points), which may sacrifice opportunities for better merges in a few more moves. Clearly *2048* is a game that requires thinking about more than just the next move.

6.4 Greedy Max-in-Corner

6.4.1 Description

This is a slight modification of the Max-in-Corner (All) algorithm to incorporate greediness. The goal is to see if greediness does better than randomness, in at least the cases where there is no obvious preferred move.

We use the same method as Max-in-Corner (All) to construct a list of preferred moves and a lower-priority weighted list of moves. However, instead of randomly choosing from our preferred moves, we greedily choose the preferred move that most increases our score. If there is no preferred move, we randomly sample from the lower-priority weighted list of moves, and make that move. If we have nothing in this list either, we simply make a random possible move.

Greedy Max-in-Corner

initialise empty preferred moves vector
initialise empty weighted moves vector
for all maximum tiles **do**
if maximum is in a corner **then** add the 2 corner directions to the prefered moves vector ▷ the same move can be added multiple times to this vector
else if maximum is on an edge & preferred moves vector is empty **then** add following weights to the moves in the weighted moves vector:
towards closest corner add weight 4
towards second closest corner add weight 1
towards closest wall add weight 1
else if maximum is on a centre tile & preferred moves vector is empty **then** add following weights to the moves in the weighted moves vector:
in the closest corner directions add weight 2
in the opposite corner directions add weight 1
end if
end for
if preferred moves vector is not empty **then** use Greedy to pick from preferred moves vector
elsepick from weighted moves vector with probabiltiy weighted by the weight of the move
end if

6.4.2 Performance

The average score is:2448.04
The minimum score is:552
The maximum score is:5752
The standard deviation of scores is:1203.90464672
The average log score is:7.67791369986
The standard deviation of log scores is:0.516924064614
The average max tile is:211.2
The minimum max tile is:64
The maximum max tile is:512
The standard deviation of the max tile is:112.137058995

6.5 Greedier Max-in-Corner

6.5.1 Description

This is a combination of the Max-in-Corner (All) algorithm with the Greedy algorithm, which incorporates greediness in even more cases than the previous algorithm, to see if there is a measurable impact. Here we use Greedy to choose among moves that keep our maximum

tiles in their corners. If no such move exists, we simply use Greedy to choose among possible moves.

Greedier Max-in-Corner

initialise empty preferred moves vector
for all maximum tiles **do**
if maximum is in a corner **then** add the 2 corner directions to the prefered moves vector ▷ the same move can be added multiple times to this vector
end if
end for
if preferred moves vector is not empty **then** use Greedy to pick from preferred moves vector
else use Greedy to pick from possible moves
end if

Then, we greedily choose the preferred move that most increases our score.

If there is no preferred move, we greedily choose the possible move that most increases our score.

6.5.2 Performance

The average score is:1616.84
The minimum score is:540
The maximum score is:5184
The standard deviation of scores is:821.016793007
The average log score is:7.27604088715
The standard deviation of log scores is:0.468583909708
The average max tile is:155.52
The minimum max tile is:64
The maximum max tile is:512
The standard deviation of the max tile is:84.0595598371

6.6 Greedy-for-Empty

6.6.1 Description

This algorithm tries to maximise the number of empty tiles on the board. Since we only lose the game if we get stuck, the intuiton is to minimise that possibility by keeping the board as open as possible, in the hope that the game is able to go on longer. All other things constant, it seems like a longer game would tend to have a higher score than a shorter one. It does this by every turn choosing the move that maximises the number of empty tiles on the board in the next turn. Since it always chooses the local optimum, it is also a greedy algorithm, except here it is trying to optimise empty tiles rather than the score.

Greedy-for-Empty

initialise empty vector
save the current board state
for all possible moves **do**
    play the move on saved board
    save number of empty tiles on the board in vector
end for
return game board to saved board
choose move corresponding to the maximum number of empty tiles

6.6.2 Performance

The average score is:1711.68
The minimum score is:604
The maximum score is:4736
The standard deviation of scores is:774.528022476
The average log score is:7.34590612325
The standard deviation of log scores is:0.448783217986
The average max tile is:164.48
The minimum max tile is:64
The maximum max tile is:512
The standard deviation of the max tile is:76.9358797961

Chapter 7

Monte Carlo Tree Search

7.1 Basic

7.1.1 Description

We do not yet have an obvious way to evaluate our moves. In previous algorithms, we use heuristics such as the increase in score or the number of empty tiles on the board after the move to choose one move over another. However, as we have seen, moves that are 'good' in this sense in the short-run do not necessarily lead to better performance in the long-run. Thus, we need some way to judge how good our moves are in the long-run.

Monte Carlo Tree Search is one possible approach. To evaluate a move, we make the move, and then, starting with the resulting position, play randomly till the end of the game a certain number of times. We call this number of times, the number of 'threads'. The intuition is that we can measure how good a board position is by seeing how well a random algorithm does on average starting from that position. All other things equal, a better board state should lead to a higher score through random play than a bad board state. Thus, we can use this method to evaluate the board states resulting from our moves. We do this for each possible move, and choose the move that gives us the highest score on average after random playing.

Monte Carlo (n)

```
save the current board state (call this original)
initialise vector of scores for each possible move
for each possible move do
    for i = 1 to n do
        reset board to original
        make move
        play game randomly till end
        record score in entry for this move in vector
    end for
end for
reset board to original
make move with the highest average score
```

The algorithm is called a Monte Carlo Tree Search since it uses Monte Carlo sampling to explore the game tree resulting from each move. The game tree is a directed graph whose nodes are game states and edges are moves, representing all possible game states resulting from the game state at the root node. Each move can lead to many different game states due to the random tile spawns. Thus, it is computationally infeasible to search every possible state resulting from every move till the end of the game.

Hence, we use Monte Carlo sampling. Instead of searching every possible outcome, we randomly sample from the space of all outcomes (by randomly choosing a move at each step). As the number of samples (threads) increases, the distribution we are sampling from converges to the actual distribution of all possible board states resulting from our starting node. Thus, even if we cannot account for every state, by taking enough samples we can use randomness to get a fair estimate of where the board state resulting from a given move will lead us, allowing us to evaluate moves.

7.1.2 Performance

n=5

20 trials:

Time taken for 10 tries: 940.3717909999999 (15 min)

The average score is:11179

The minimum score is:1280

The maximum score is: 23964

The standard deviation of scores is:5059.43870012

The average log score is:9.28874682895

The standard deviation of log scores is:0.63161803248

[1024, 1024, 256, 1024, 128, 1024, 512, 1024, 1024, 512, 1024, 1024, 512, 2048, 1024, 512, 1024, 2048, 512, 1024]

The minimum max tile is:128

The maximum max tile is: 2048

n=10

20 trials:

Time taken for 10 tries: 2848.3589260000003 (47 min)

The average score is:16735

The minimum score is:1236

The maximum score is:31496

The standard deviation of scores is:8936.49738992

The average log score is:9.49181662303

The standard deviation of log scores is:0.797716587686

[1024, 1024, 2048, 512, 128, 1024, 2048, 2048, 1024, 512, 1024, 2048, 1024, 256, 1024, 2048, 1024, 2048, 512, 1024]

The minimum max tile is:128

The maximum max tile is:2048

Despite its simplicity, the algorithm is quite successful. Whereas none of the blind or heuristic algorithms crossed 512, by playing 10 games till the end for each move ($n = 10$), we got 2048 in 6 out of 20 games.

Even with $n = 5$, in 20 games we managed to get to 2048 twice, and 1024 eleven times.

We use 'random play till the end a certain number of times' to evaluate moves. As we have seen in the section on Blind Algorithms, random play is not very successful and gets stuck quite quickly. Hence each time we are playing randomly to evaluate a move, we get stuck quite quickly. However, the actual game played with the moves we have evaluated lasts much longer. Thus, even though our random plays do not themselves exhibit intelligent play, they give us useful information that allows us to play more intelligently.

Notably, all this works without an explicit evaluation function. There is no human knowledge of what is good gameplay or what is a good game state built into the algorithm. It simply uses the game mechanics (including the in-built score function) to evaluate game states. We are not telling the algorithm to reward having maxima in corner or a more open board, for instance. We have no preconceived strategies. We simply let the algorithm explore the game tree resulting from each move and in each case, choose the move that leads to the best game tree on average. This is a useful technique generalisable to other cases where we may not have any ideas of what is a 'good' move, or may not want to be limited by previously discovered strategies.

Another useful feature is that the algorithm can run for an arbitrary amount of time and return a result. In general, the longer it runs the better the result will tend to be, since we will have more samples and hence a better estimate of the game tree resulting from each move. However, if computation is taking too long, we can have a timer that simply returns the best move found after a certain amount of time (assuming each move is evaluated k times before any move is evaluated $k + 1$ times). This gives the algorithm some flexibility, that can be quite useful.

There are, however, some limitations to this evalution. Though we evaluate a move by sampling from the space of all possible game states resulting from that move, since we are not playing randomly, we are never actually going to reach most of those states. Most of those game states are the result of 'bad' moves that our algorithm is likely to avoid. A possible improvement could be to somehow search the better move paths more often than the bad moves, since then we would be sampling from a space of game states closer to the ones we

are interested in. This would allow us to evaluate moves better while saving computation time.

The next section discusses another possible improvement.

7.2 Limited Depth

7.2.1 Description

Instead of sampling from the space of end game states resulting from a move (by playing randomly till the end), we could search the space of game states that come soon after making our move. Thus, after making a move, we play randomly for only a few moves (a limited depth) rather than till we get stuck. This will make the algorithm a lot faster, allowing us to potentially take more samples at each step to get a better evaluation of the move.

Evaluating game states resulting from fewer random moves after our current game state might not just be faster, but better. This way, the information about the move we are interested in is not buried under the noise of the effects of many random moves. Moreover, this space of game states will be much smaller than the space of end game states resulting from a given move, hence our technique will sample more accurately from the underlying distribution.

Monte Carlo Limit (n, d)

```
save the current board state (call this original)
initialise vector of scores for each possible move
for each possible move do
    for i = 1 to n do
        reset board to original
        make move
        play game randomly for d moves
        record score in entry for this move in vector
    end for
end for
reset board to original
make move with the highest average score
```

Another way to understand this algorithm is as a less greedy version of the Greedy algorithm. Instead of comparing scores immediately after each possible move, we compare scores a few moves (depending on depth) after each move (a slightly more long-term kind of greed), and pick the highest. However, since the space of all possible board states after even a few moves is quite large, instead of looking at every state, we randomly sample from it by playing randomly after making the move we want to evaluate. By comparing the average scores of these plays, we can get some sense of where this move is going to lead us. Whereas the greedy algorithm failed because simply looking at the score of the immediate next state did not tell us much about where this move was taking us, the previous algorithm

may be limited because the end game states resulting from random play are not accurate representors of where our move is going to lead us, since we are not playing randomly.

7.3 Performance

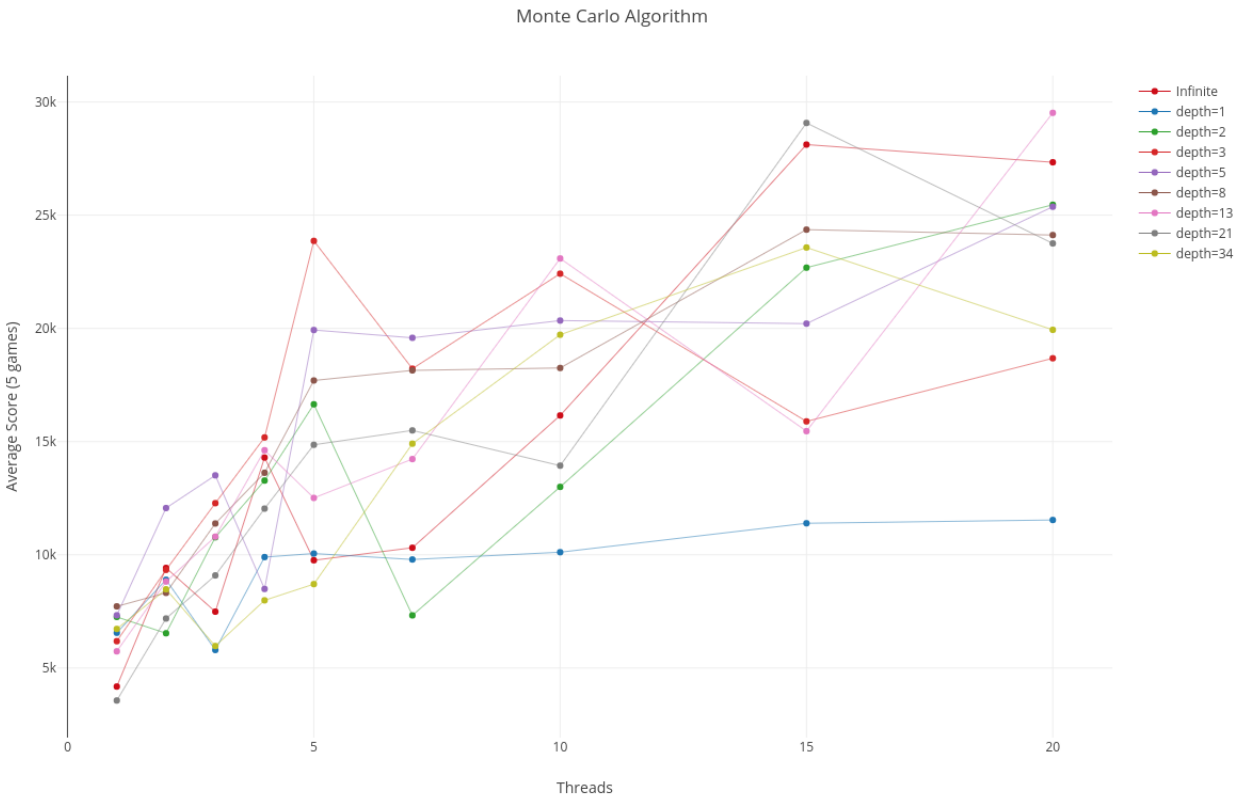

Figure 7.1: Performance of the Monte Carlo algorithm for different depths

Chapter 8

Conclusion

Bibliography

[1] logic - What is the optimal algorithm for the game 2048? - Stack Overflow.

[2] Temporal difference learning of N-tuple networks for the game 2048 - IEEE Conference Publication.

[3] Joshua A. Krisch. Why is the 2048 Game So Addictive?, April 2014.

[4] Kunal Mishra. 2048-Starter: A CS0 game for beginning students in Computer Science, developed by Paradigm Shift, October 2017. original-date: 2016-08-13T05:29:34Z.

[5] ovolve. 2048-AI: A simple AI for 2048, February 2018. original-date: 2014-03-11T15:03:00Z.

[6] Sarah Perez. Clones, Clones Everywhere 1024, 2048 And Other Copies Of Popular Paid Game Threes Fill The App Stores, March 2014.

[7] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[8] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, October 2017.

[9] Robert Xiao. 2048-ai: AI for the 2048 game, November 2017. original-date: 2014-03-19T07:00:52Z.

[10] Kun-Hao Yeh, I.-Chen Wu, Chu-Hsuan Hsueh, Chia-Chuan Chang, Chao-Chin Liang, and Han Chiang. Multi-Stage Temporal Difference Learning for 2048-like Games. *arXiv:1606.07374 [cs]*, June 2016. arXiv: 1606.07374.