Senior Thesis in Mathematics

Algorithms and Artificial Intelligence for 2048

Author:
Neel Adhiraj Kumar

Advisor:
Dr. Vin de Silva

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

April 10, 2018

Abstract

To better understand the abilities of pattern-finding and strategy, which lie at the heart of intelligence, we simulate them by creating algorithms and artificial intelligence programs to play the single player puzzle board game *2048*. After understanding the game and discussing its value as a domain for artificial intelligence research, we investigate three classes of algorithms: blind, heuristic, and monte carlo tree search, in increasing order of complexity. We run each algorithm many times on the game and evaluate the performance. The blind algorithms move without looking at the board and are hence unsurprisingly unsuccessful, but give us a standard against which to compare 'real' intelligence. Heuristic algorithms, though not quite 'intelligent', employ strategies that intuitively seem like they should be better than 'blind' approaches. Yet, they fail to outperform the blind algorithms. Monte Carlo Tree Search uses random (unintelligent) play to find 'intelligent' moves, and reaches the 4096 tile despite having no in-built knowledge of the game. This approach is discussed and then extended by searching only up to a limited depth of moves, which proves to be both quicker and 'more intelligent'. Possible extensions and applications of this technique are considered.

Acknowledgements

This thesis is a result of the invaluable guidance I have been afforded by my teachers over the years. In particular, I am indebted to Dr. Parag Mehta and Professor Shahriar Shahriari, who each in their own way, introduced me to mathematics as an aesthetic activity and an end in itself, and set impossibly high standards for teaching and mentorship. Without either, I may never have embarked upon one of the most challenging and enriching journeys of my life - majoring in mathematics.

My fascination with intelligence and the study of consciousness (including artificial intelligence) was immeasurably deepened by Douglas Hofstadter's masterpiece, *Gödel, Escher, Bach: An Eternal Golden Braid*. The philosophy underlying my research is largely due to him. He was recommended to me by Professor Stephan Garcia, who in teaching me the beauty of mathematics demanded more intelligence from me than I thought I had, for which I am forever grateful.

I would also like to extend my appreciation and thanks to students in the mathematics department at Pomona College, whose passion and brilliance has provided an endless spring of inspiration. I could not have asked for better peers to collaborate with, and have learned immensely from their company.

Finally, this work and all the research that went into it would be entirely impossible without Professor Vin de Silva. His expertise as both a researcher and a teacher allowed me to meaningfully and productively explore an area I had long been fascinated with, and his understanding helped make a challenging process incredibly rewarding. Any merit here is due to him, all mistakes and shortcomings are mine alone.

Contents

Chapter 1

Introduction

Pattern-finding is at the core of intelligence, in the words of Douglas Hofstadter, scientist, writer, and artificial intelligence researcher [3]. Everything from concrete sensory experience to abstract mathematics relies on our minds recognising and representing 'meaningful' patterns, whether in sensory data or mental constructs. One promising way to study this most abstract and profound of abilities 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 at Go, a game many orders of magnitude more complex than chess [8]. 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 [9]. Games are a convenient and consequently rich domain for AI research. They have limited and clearly defined environments, making them easier to model and program than real world situations. Success at games can involve strategy, planning, and problem solving, components of what we could call intelligence. The question of what intelligence 'really' is, is murky philosophic territory. However, games tend to have a natural, in-built measure of performance - usually a game score, or a winner, giving us a proxy for measuring something related to and involved in intelligence. Developing programs to play games involves all the classical topics in AI: knowledge representation, 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. Human-players certainly do not play by calculating all possibilites. Developing AI for the game involves modelling and simulating abstract pattern finding skill and strategy, and allows us to try out a variety of AI techniques.

Chapter 2 discusses the mechanics of the *2048* game, and its success and value as a framework for studying AI. Chapter 3 describes the implementation of the game in software. Chapter 4 reviews literature on the subject. The next three chapters describe three classes of algoritihms, in increasing order of complexity, and evaluate their performance. Chapter 5 covers Blind Algorithms, which move without looking at the board, providing a standard against which to compare 'real' intelligence. Chapter 6 describes Heuristic Algorithms which

implement simple, intuitive strategies. Chapter 7 discusses Monte Carlo Tree Search algorithms, which both look at the game board and explore the game tree before choosing their move. Chapter 8 concludes.

Chapter 2

2048

2048 is a single-player 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 [7]. *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 a viral sensation.

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.

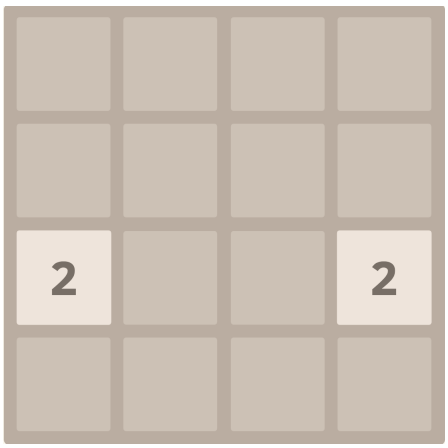


Figure 2.1: Example of a starting game board

- 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.

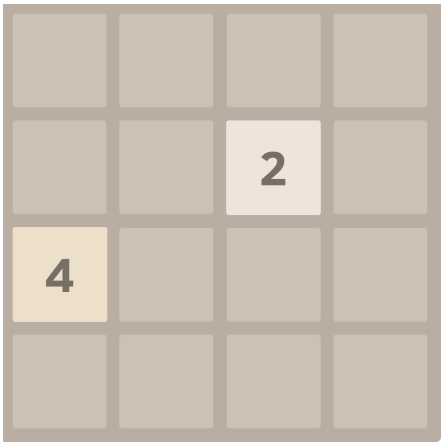


Figure 2.2: The game board from Figure 1.1 after a left swipe

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.

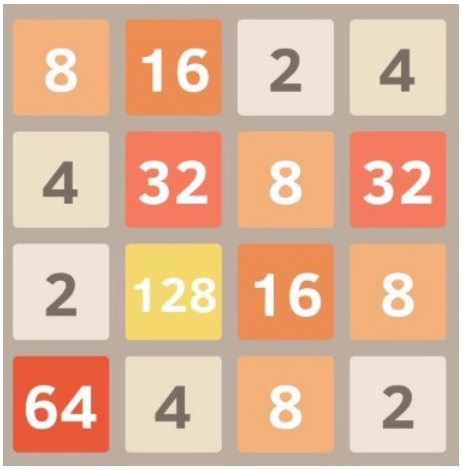


Figure 2.3: Example of a 'game over' board with no moves left

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. Players can try to build higher and higher tiles, and with artificial intelligence techniques, tiles up to $65,536$ have been achieved [10].

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. 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) can only have a 0.1 probability of making it to 131072, since it requires spawning a 4 at the right moment.

There game also has a score function: **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

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

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 $16 = 2^4$ 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 minima and the maxima is based on whether all the 4 tiles used to build the tile in question were randomly generated or built using 2 tiles (respectively).

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.

Figure 2.4: An example game board

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

2.2 Why we care

2.2.1 Why is it so addictive?

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. It became a viral phenomenon and many explanations have been offered for its popularity [4].

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 hundred moves and even a trivial strategy can score a few thousand 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 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, intuitive 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 impacts game strategy. 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 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

The purpose of this thesis is not to play the game, but develop players for the game. We are in a sense playing the game at one level of generality. 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 (using algorithms and artificial intelligence) that are successful across all instances of the game. This requires skillfully making decisions that move you towards a chosen goal (for instance, maximising score), 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 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 involve meaningful trade-offs applicable to other domains. The exponential increase in difficulty in creating higher and higher tiles additionally provides ample room to test vastly

different levels of performance.

Chapter 3

Implementation

This thesis implements the game in Python. The implementation began as an expansion of starter code for an introduction to computer science course [5], 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, 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 [10]. We calculate the result of left moves on all possible (4-tile) rows in the game and store the results in a table. Each time we have to make a move on the board, we convert it to the symmetric left move case for each row, and look up the resulting row using the table. For instance, when our original move is left, we simply look up each row of the board in the table and replace them with the results. If our original move is right, we reverse each row and look up the result in the table, and then reverse it back.

Each row (of 4 tiles) is encoded as a 16 (unsigned) bit integer, which gives us 4 bits to encode every tile. This again allows us to encode 16 values, giving us the same 15 powers of 2 and the empty tile. We can use the integer representing the starting row as the index of an array and the integer representing the row resulting from the left move as the value corresponding to the index (much like a hash table). This allows us to compute moves very quickly without having to check for tile merging subtleties during each move.

The source code is given as an appendix.

Chapter 4

Related Work

The first artificial intelligence program for *2048* published online was a Mini-max search with alpha-beta pruning [6]. 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 AI uses an evaluation function that 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 the mini-max approach of maximising the score in the worst case scenario seems reasonable, it 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. [10] 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. [11] 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 a serious attempt to play the game successfully. Instead, they will provide a benchmark against which to compare the performance of later algorithms, allowing us to distinguish 'intelligent' techniques from merely lucky ones. Due to their simplicity, they are also useful for understanding the basics of algorithmic gameplay. Later 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, and score over 2000 points on average. Some instances even managed to cross 7000.

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

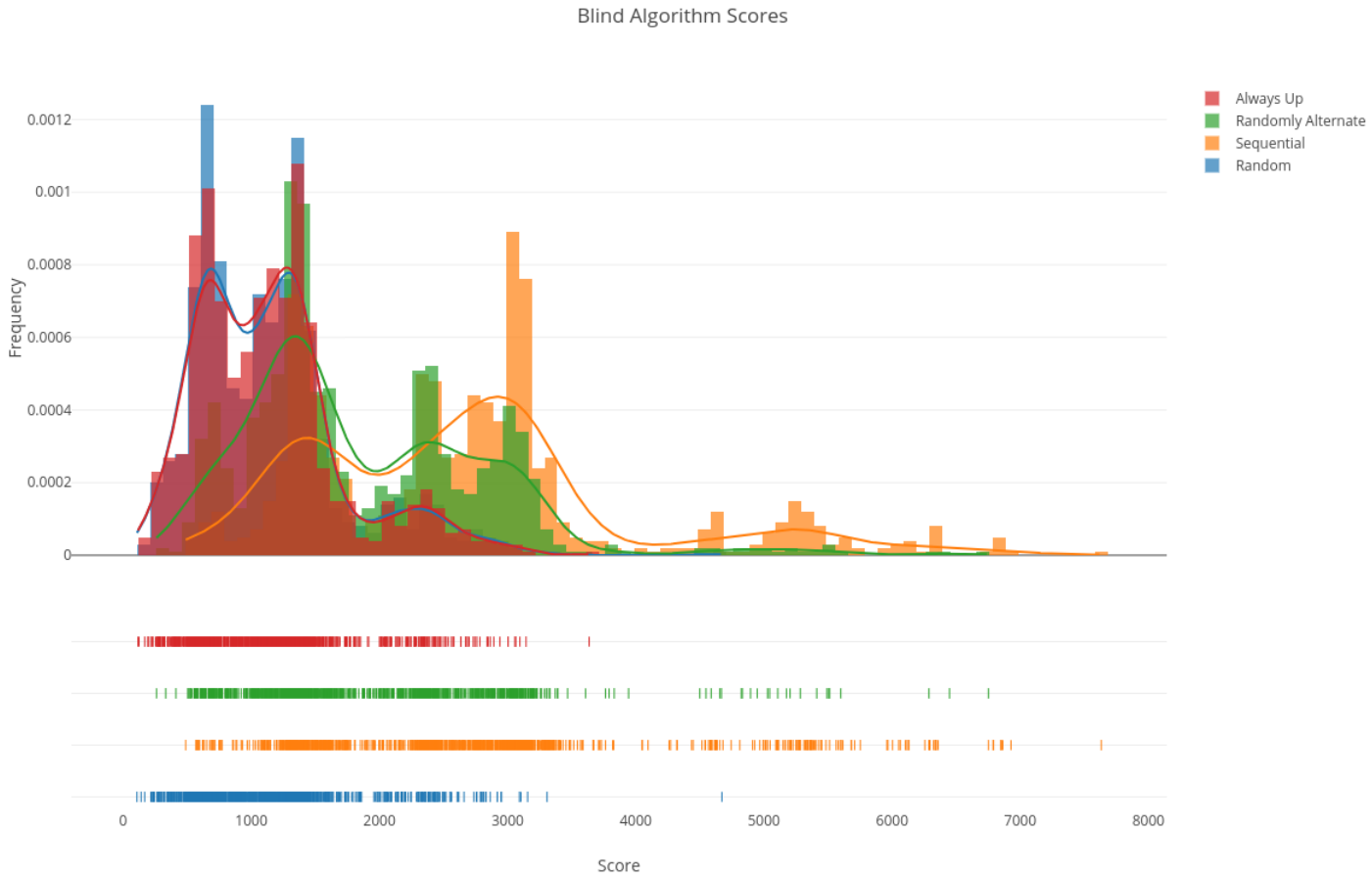


Figure 5.1: Blind Algorithm Score Distribution (1000 trials each)

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 Descriptions

6.1.1 Max-in-Corner (Unique)

This is the first algorithm that has some reasoning 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 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 may be able to merge) to occupy positions with more adjacent tiles, decreasing the possibility of getting stuck.

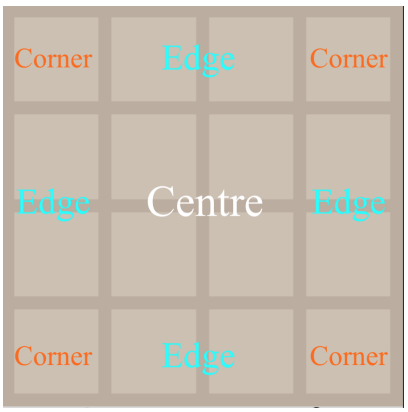


Figure 6.1: The 3 kinds of tiles on a 2048 board

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.

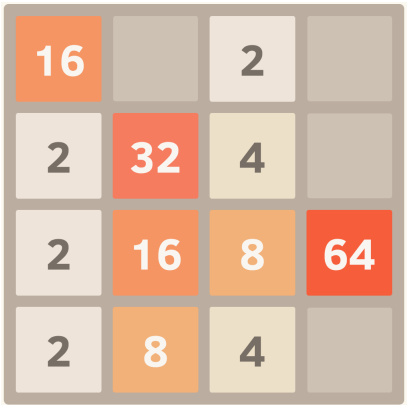


Figure 6.3: The 64 tile is closest to the bottom-right corner and the right wall. The second closest corner is on the top-right

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 Max-in-Corner (All)

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

Max-in-corner (Unique)

if no unique maximum tile **then** choose random move
else
  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
    towards opposite corner with weight 1
  end if
end if

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:
    towards closest corner add weight 2
    towards opposite corner 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

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.

6.1.3 Greedy

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.

Greedy

for all moves **do**
    calculate score increase from making move
end for
choose move corresponding to the highest score increase

6.1.4 Greedy Max-in-Corner

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.

6.1.5 Greedier Max-in-Corner

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 choose Greedy to choose among moves that keep our maximum tiles in their corners (i.e. the one that most increases our score next turn). If no such move exists, we simply use Greedy to choose the move that most increases our score next turn.

6.1.6 Greedy-for-Empty

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 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
else pick from weighted moves vector with probabiltiy weighted by the weight of the move
end if

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

6.2 Performance

Neither of the Max-in-Corner algorithms perform statistically better than a blind algorithm. This strategy is likely too simple to be successful across a game. Perhaps in many cases it is better to bring maxima together rather than move them to a corner. Our probability weights are also not trained in any way.

The Greedy algorithm also 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. Still, it is surprising that this approach does not even outperform blind approaches. When possible, the algorithm will choose to merge high-value tiles (since this creates a high

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

Algorithm	Average	Minimum	Maximum	Standard Deviation
Sequential	2650	490	7630	1257.52
Max-in-Corner (Unique)	1410	100	6460	813.38
Max-in-Corner (All)	2210	270	7290	1216.62
Greedy	1270	170	3450	593.02
Greedy Max-in-Corner	2290	290	7810	1243.59
Greedier Max-in-Corner	1760	150	8120	1284.25
Greedy-for-Empty	1250	290	4800	575.32

Table 6.1: Heuristic Algorithms Score Statistics (1000 trials each)

Algorithm	Minimum	Maximum
Sequential	64	512
Max-in-Corner (Unique)	16	512
Max-in-Corner (All)	32	512
Greedy	16	256
Greedy Max-in-Corner	32	512
Greedier Max-in-Corner	16	512
Greedy-for-Empty	32	512

Table 6.2: Heuristic Algorithms Largest Tile Statistics (1000 trials each)

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.

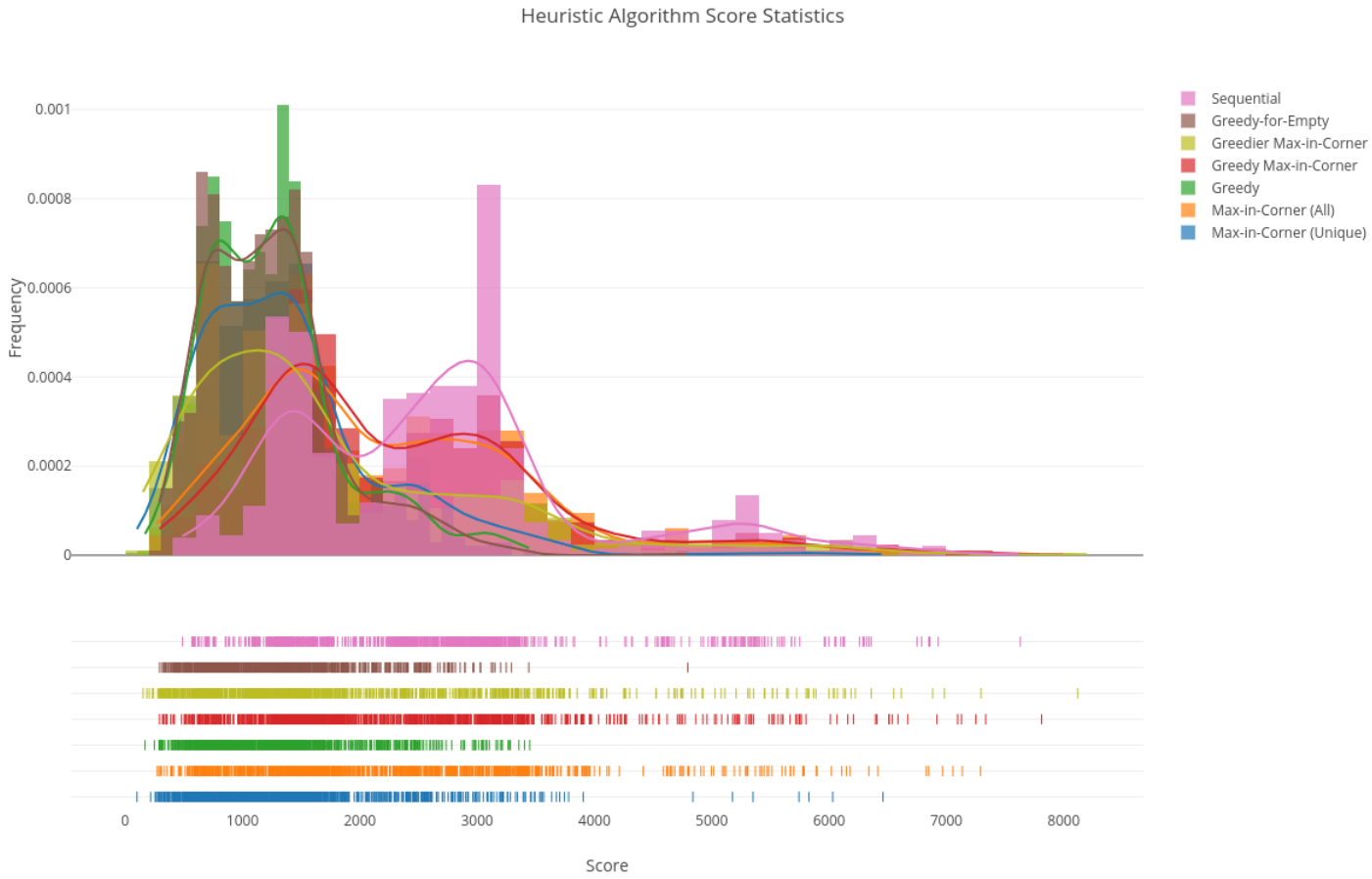


Figure 6.4: Heuristic Algorithm Score Distribution (1000 trials each)

Chapter 7

Monte Carlo Tree Search

7.1 Basic

7.1.1 Description

Finding an effective way to evaluate moves can be a useful approach to game strategies. 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 (MCTS) 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, record the score, and repeat this 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 play.

Monte Carlo (n)

```
save original board state
for all moves do
    for i = 1 to n do
        reset board to original
        make move
        play game randomly till end
        add to score for this move
    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, our sampling distribution converges to the 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.

We can safely simulate different threads in parallel making it straightforward to execute this algorithm concurrently using many processes.

7.1.2 Performance

Algorithm	Average	Minimum	Maximum	Standard Deviation
Sequential	2650	488	7630	1257.52
Greedy	1270	260	6750	934.69
Monte Carlo (5 Threads)	10480	1100	32650	7302.49
Monte Carlo (10 Threads)	15750	5580	36200	7331.86
Monte Carlo (20 Threads)	23500	3028	56960	10859.11

Table 7.1: Monte Carlo Algorithms Score Statistics (50 trials each)

Despite its simplicity, the algorithm is quite successful. Whereas none of the blind or heuristic algorithms crossed 512, the Monte Carlo algorithm manages to reach 2048 even with 5 threads, occasionally reaching 4096 with 20 threads. We can think of this as a long-term Greedy algorithm, which instead of choosing the move that maximises our score now,

Algorithm	Minimum	Maximum
Sequential	64	512
Greedy	16	256
Monte Carlo (5 Threads)	256	2048
Monte Carlo (10 Threads)	512	2048
Monte Carlo (20 Threads)	256	4096

Table 7.2: Largest Tile Statistics (50 trials each)

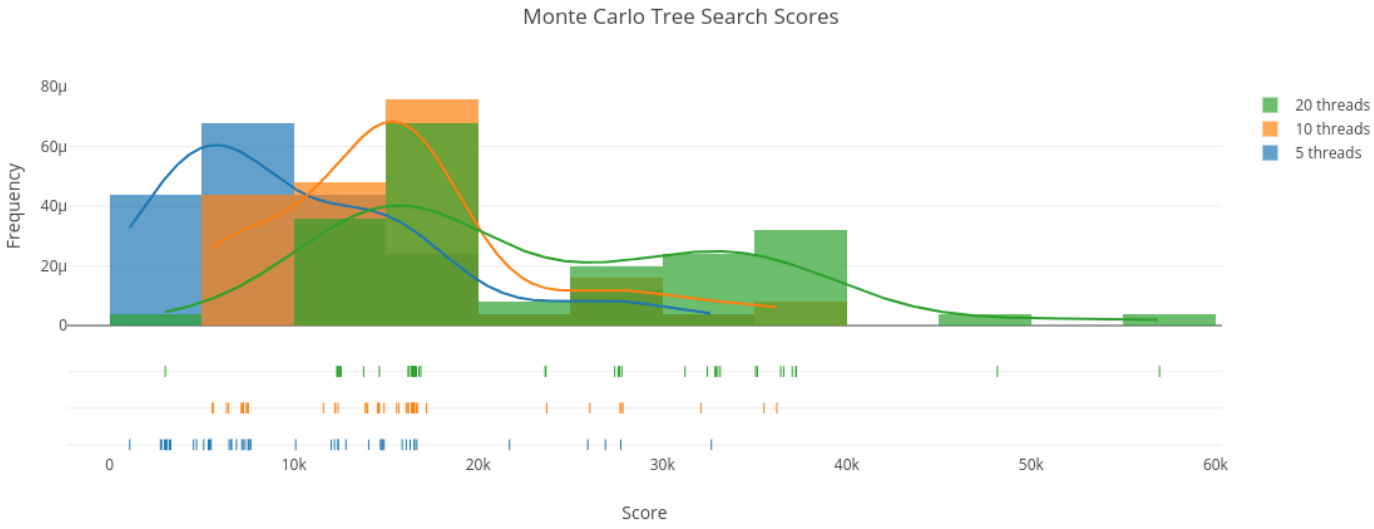


Figure 7.1: Monte Carlo Tree Search Score Distribution (50 trials each)

chooses the move that is statistically most likely to maximise our score at the end of the game. Being greedy for the score at the end of the game proves to be much more fruitful than being greedy for the score in the next move.

In this algorithm 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. This is akin to much of the learning we see in nature, including in evolution, where starting with essentially random guesses, we start to slowly create competence, by prioritising the better guesses in some way or another. However, instead of actually making the random mistakes, we are simulating them. This is something humans do all the time, when we consider the possible outcomes of a decision in our head, for instance. Though humans are really good at modelling and simulating the most salient aspects of even highly complex situations, computers are better at simulating everything within a limited domain. Hence, though this strategy is quite infeasible for human players, a computer playing *2048* can use MCTS to beat the game.

MCTS does not involve any in-built, domain specific knowledge of the game, for instance in the form of heuristics or evaluation functions that reward certain board positions. The algorithm only looks at the score. This makes it applicable to many other domains which can be modelled using trees, in particular those where we may not have existing knowledge or strategies or may not want to pre-emptively bias the algorithm with our intuitions.

Another useful feature is that the algorithm can run for an arbitrary amount of time and return a result, by simply increasing the number of threads for each move. As we can see from our results, the longer it runs the better the result tends to be, since more threads give us more samples and hence a better estimate of the game tree resulting from each move. 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 which 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. Moreover, the impact of the move we are trying to evaluate has been buried under the 'noise' of many random moves. Each of these evaluations gives us little information about the move we're interested in but is computationally demanding. If we try and improve performance by increasing the number of threads we soon reach a computational barrier. Still, with some optimised implementations of the game, we can make multiple moves a second even using hundreds of thousands of threads. The next section discusses one possible improvement to the algorithm.

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 we are stuck), 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 faster, allowing us to use more threads to get a better evaluation of the move.

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 (random) moves (depending on depth) after each move, and pick the highest, a more medium-term kind of greed. 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 useful representors

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
```

of where our move is going to lead us, since too many moves have been made afterwards and we are not playing randomly.

7.2.2 Performance

Algorithm	Average	Minimum	Maximum	Standard Deviation
Sequential	2650	488	7630	1257.52
Monte Carlo (5 Threads)	10480	1100	32650	7302.49
Monte Carlo (10 Threads)	15750	5580	36200	7331.86
Monte Carlo (5,3)	15200	5710	28080	5476.15
Monte Carlo (5,5)	16380	5580	33160	7234.43
Monte Carlo (5,10)	12930	2950	31120	6065.18
Monte Carlo (5,20)	13130	3076	26970	5174.11
Monte Carlo (10,3)	19870	7320	36290	7452.32
Monte Carlo (10,5)	22190	6740	37100	8982.07
Monte Carlo (10,10)	21710	6480	36920	7671.94
Monte Carlo (10,20)	18570	3060	36650	9137.19

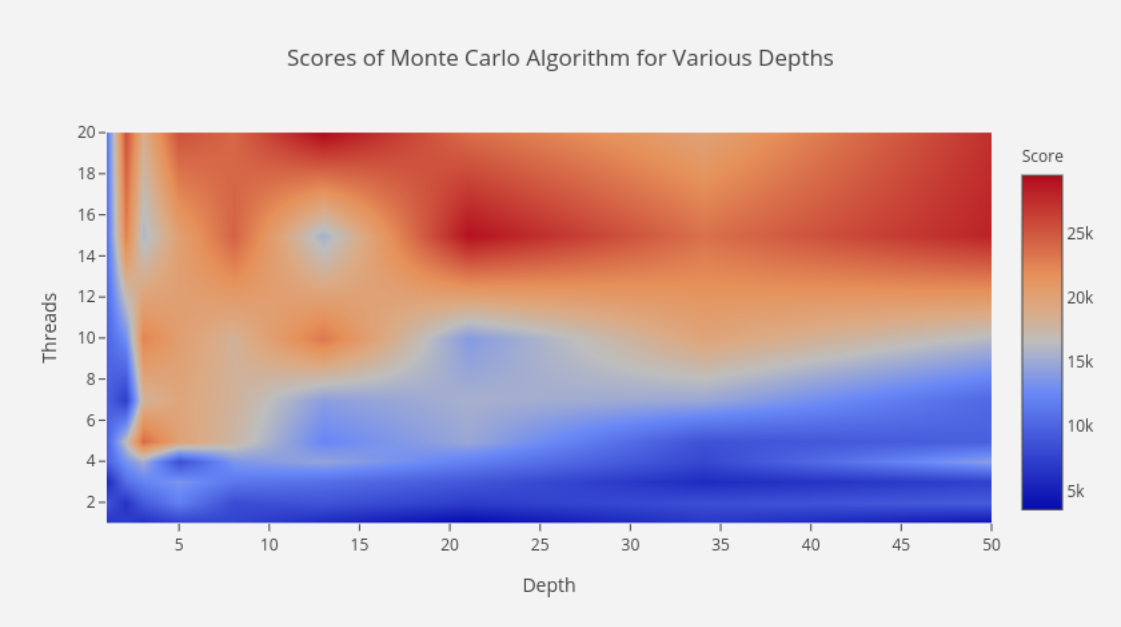
Table 7.3: Monte Carlo Algorithms Scores (50 trials each)

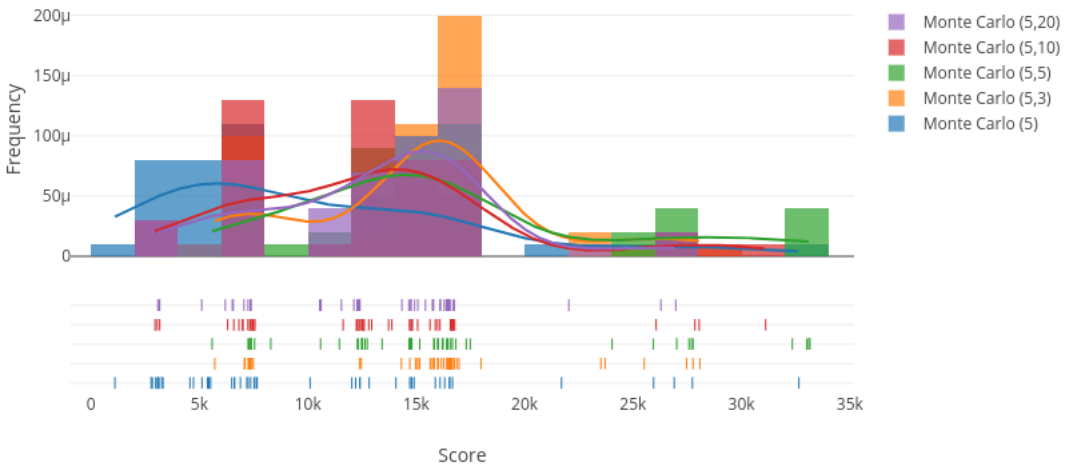
Evaluating game states resulting from fewer random moves after our current game state turns out to be not just faster, but better. Low depths perform atleast as well, and in some cases better, than larger depths. Particularly with fewer threads, low depths seem to outperform higher depths. A possible explanation is that for larger depths the number of possible game states is much higher. Hence the space we are sampling from is much larger. Our few threads cannot really give us an accurate representation of where we're headed. With a smaller depth, we are sampling from an (exponentially) smaller space, and hence sample more accurately from the underlying distribution with the same number of samples. We get a better idea of where we're headed with fewer threads.

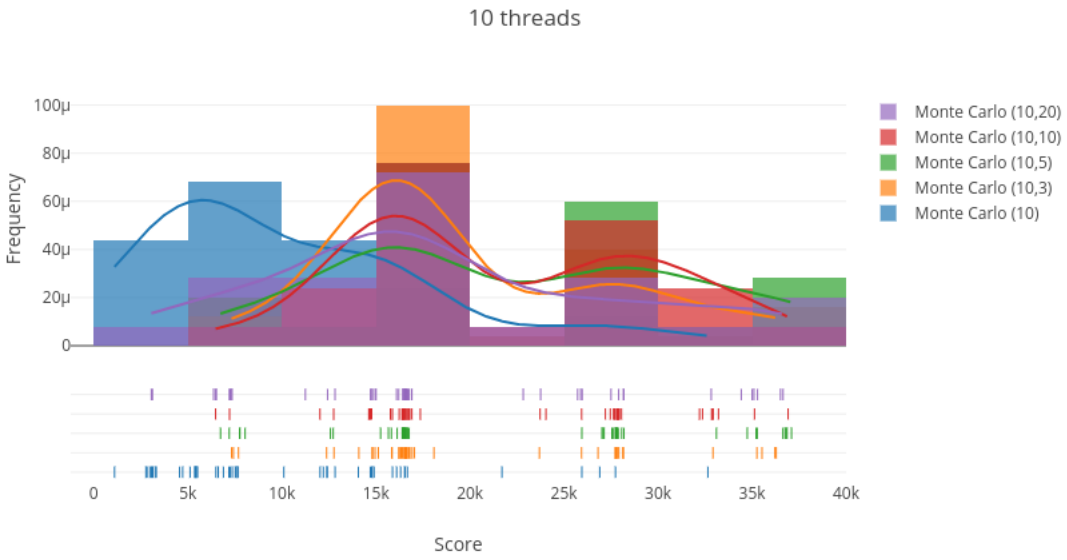
Algorithm	Minimum	Maximum
Sequential	64	512
Monte Carlo (5)	256	2048
Monte Carlo (10)	512	2048
Monte Carlo (5,3)	512	2048
Monte Carlo (5,5)	512	2048
Monte Carlo (5,10)	256	2048
Monte Carlo (5,20)	256	2048
Monte Carlo (10,3)	512	2048
Monte Carlo (10,5)	512	2048
Monte Carlo (10,10)	512	2048
Monte Carlo (10,20)	256	2048

Table 7.4: Largest Tile Achieved (50 trials each)

The information about the move we are interested in is not buried under the noise of the effects of many random moves. We are not wasting valuable computation time searching parts of the game tree reached after many bad moves that we will (hopefully) avoid.


Scores of Monte Carlo Algorithm for Various Depths
Threads
Depth
Score
25k
20k
15k
10k
5k


Frequency
Score
Monte Carlo (5,20)
Monte Carlo (5,10)
Monte Carlo (5,5)
Monte Carlo (5,3)
Monte Carlo (5)


10 threads
Frequency
100µ
80µ
60µ
40µ
20µ
0
0
5k
10k
15k
20k
25k
30k
35k
40k
Score
Monte Carlo (10,20)
Monte Carlo (10,10)
Monte Carlo (10,5)
Monte Carlo (10,3)
Monte Carlo (10)

Chapter 8

Conclusion

Intelligence, including the abstract ability to detect and meaningfully manipulate patterns, is yet poorly understood. The field of artificial intelligence offers us a useful approach to understanding this ability - by simulating it. While anything like human intelligence is still far out of the question for our programs, in limited domains (such as games), artificial intelligence techniques can demonstrate comparable, if not superior performance to expert humans. By developing such programs and analysing them, we can begin to put together the pieces that go into 'intelligence', and slowly build our way up.

Our blind algorithms are clearly unintelligent since they move without looking at the board, but their performance provides a standard against which we to compare other algorithms. We learn that reaching 512 is no mark of skill, since several blind algorithms occasionally stumble upon that tile. The first serious attempt at making successful players is with heuristic algorithms which implement intuitive strategies - keep the maximum tile in a corner, maximise the score in the next move, or maximise the number of empty tiles on the board. These algorithms are unsuccessful, failing to outperform even blind approaches, indicating that the 'intelligence' that *2048* requires is not trivial.

Monte Carlo Tree Search (MCTS) proves to be a promising and powerful approach. With just the rules of the game and the score available to us, using our ability to simulate the game, we are able to detect successful moves. The strategy is analogous to (unintelligently) trying things out for a while in our heads, choosing the approach that seems to work best (in the end), going with it for one step, and then trying things out in our heads again. Its generality makes it potentially applicable to many other problems, and allows for many possible extensions of the technique.

To begin with, we can optimise the search by eliminating bad moves once we discover them instead of searching them again and again, or at least by weighting good moves higher. This will require us to in some form 'remember' the tree we have so far searched and use that to prune the tree we are yet to search. This will be much more effective if we account for the symmetries in the board, making our tree exponentially smaller. Though the algorithm is domain-independent and starts with no knowledge of the game, there is no reason it should not over time accumulate knowledge and cultivate expertise.

We could even apply the Monte Carlo Tree Search recursively. Instead of evaluting the move we want using random moves, we could evaluate it using a Monte Carlo Tree Search. To avoid long recursive chains, we would require each succeeding search to be a strictly

smaller depth than the preceding one. This approach will allow us to get a much better idea of where the move we are evaluating might take us, but is also much more computationally demanding.

Another possible approach is searching until we reach a certain threshold in score difference between the moves, instead of searching till a certain depth of moves. This would allow us to keep searching until we find the best move(s), and move quickly when there is a clear winner.

A more minor variation on our existing algorithms might be to choose the move that maximises the minimum (or maximum) score, rather than the move that maximises the average score. The conservative (or optimistic) approach that consider the worst (or best) case might be more successful than simply maximising the average case.

Though it does not require any in-built knowledge of the game, MCTS could maybe still benefit from it. We could combine the search with heuristics that help us both search the space more efficiently and evaluate the board more accurately. For instance, rather than using the score, we could develop another measure of how good a game state is based on the arrangement of tiles on the board. This can be valuable when evaluating boards that are not yet stuck, or that might be close to many big merges.

The MCTS algorithm lends itself well to discrete-state games since they can be conveniently modelled as trees. AlphaGo and AlphaGo Zero use MCTS in combination with neural networks trained to evaluate game states to exhibit superhuman performance at Go [8] [9]. However, trees as data structures have much more general use. With the increasing need for search and optimisation algorithms (particularly those that are 'unsupervised') it is possible that this techique (and its many possible extensions) will become much more popular, especially in combination with other approaches. Since it is straightforward to implement in parallel, it is 'future-proof' in the sense that it will become almost proportionally quicker as the number of processors in computers grows. As its application extends into more complex domains, Monte Carlo Tree Search may offer us valuable insight into how 'intelligence' arises from simple statistical processes.

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] Douglas R. Hofstadter. *Fluid Concepts and Creative Analogies: Computer Models Of The Fundamental Mechanisms Of Thought*. Basic Books, 1 edition edition, March 1996.

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

[5] 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.

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

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

[8] 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.

[9] 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.

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

[11] 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.

Appendix A

Source Code

Python code for the game and most of the algorithms discussed in this thesis.

```
'''
utility:
main
play_mode
make_board
print_board
to_score
pause
clear
get_key_press
get_piece
place_piece
place_random - input board, output: board with random tile spawn
board_full
have_lost
possible_moves
to_move_list
move
swipe
transform_board - input: board, direction, output: board
automate
find_max
countEmpty
MonteCarloEval
'''

import random
import time
import numpy as np
import matplotlib.pyplot as plt
```

```
import math
import copy
from tkinter import *
from functools import partial
from tabulate import tabulate

pausetime = 0 #time to pause between AI moves (for visibility)
board_size = 4
N = 4 #board_size
move_table = [0]*65535

def main():

    #Creating my new 4x4 board
    board = make_board(N)

    #Getting the game started with two pieces on the board
    place_random(board)
    place_random(board)

    print_board(board)

    print("Press p to play the game using arrow keys");
    print("Press a to automate gameplay using one of the in-built programs");
    print("Press s to get statistics for one of the in-built programs");
    print("Press q to quit");

    while True:

        #Gets the key pressed and stores it in the key variable
        key = ord(input("Choose game mode: "))

        #play case ('p')
        if key==112:
            play_mode(board)

        #automate case ('a')
        if key==97:
            automate(board)

        #statistics case ('s')
        if key==115:
            getStatistics()

        #Quit case ('q')
```

```
        else:
            print("Game Finished!");
            quit()
            break

def play_mode(board):
    """
    Game mode that allows the user to play using arrow keys
    """
    while True:

        key = get_key_press()

        if key == 65:
            swipe(board, 0)

        elif key == 66:
            swipe(board, 1)

        elif key == 67:
            swipe(board, 2)

        elif key == 68:
            swipe(board, 3)

        print_board(board)

        #Check to see if I've lost at the end of the game or not
        if have_lost(board):
            maxval = max(max(row) for row in board)
            score = to_score(board)
            print("Game over. Final score %d; highest tile %d." % (score, maxval))
            print("You lost! Would you like to play again? (y/n)");
            if (input() == 'y'):
                main();
            else:
                quit()

def make_board(N):
    """
    Utility function that returns a new N x N empty board
    (empty spaces represented by '*')
    Arg N: integer - board dimensions
    (must be greater than or equal to 1)
    """
```

```
    assert N >= 1, "Invalid board dimension";
    assert type(N) == int, "N must be an integer";
    #return np.array([0 for x in range(N) for x in range(N)]);
    #represent board as N*N length array
    return np.array(['0' for x in range(N) for x in range(N)]);
    #represent board as 16 hexadecimal entries

def print_board(board):
    """
    Utility function that prints out the state of the board
    Arg board: board - the board you want to print
    """

    score = to_score(board)

    print("Your score: "+str(score));

    vertical_edge = "";
    for i in range(N+2):
        vertical_edge += "-\t";
    print(vertical_edge);

    for y in range(N):
        row = "";
        for x in range(N):
            cell = get_piece(x,y,board) #find the right index on the board
            if cell==0: row+= '0'
            else: row += str(2**cell)
            row += "\t";
        print("|\t" + row + "|");
        if y is not N-1: print("")
    print(vertical_edge);

def _to_score(c):
    #input: cell in binary
    #output: score from cell
    if c <= 1:
        return 0
    return (c-1) * (2**c)

def to_score(board):
    #input: board
    #output: board score
    #need to adjust for generated 4s
    cell_scores = [_to_score(get_piece(x,y,board)) for x in range(N) for y in range(N)]
```

```
    return(sum(cell_scores))

def pause(seconds):
    """
    Utility function that pauses for the given amount of time
    Arg seconds: a float or integer - number of seconds to pause for
    """
    time.sleep(seconds);

def clear():
    """Utility function that clears the terminal GUI's screen - takes no arguments"""
    try:
        try:
            #For Macs and Linux
            os.system('clear');
        except:
            #For Windows
            #REPORTED BUG: Sometimes does not work on 64 bit Windows
            os.system('cls');
    except:
        #If nothing else works, a hacky, non optimal solution
        for i in range(50): print("")

def get_key_press():
    """Utility function that translates pressed key into its character ascii value
    takes no arguments"""
    return ord(getch.getch());

def get_piece(x, y, board):
    """
    Utility function that returns the int power of 2 of the piece at a
    given (x,y) coordinate on the given board
    Returns the piece if the request was valid and None if the request was not valid
    Arg x: integer - x coordinate
    Arg y: integer - y coordinate
    Arg board: board - the board you wish to get the piece from
    """

    #Ensure that x and y are both integers (use assert)
    assert type(x) == type(y) == int, "Coordinates must be integers"

    #size constraint on x and y
    #Checking that the (x,y) coordinates given are valid for the N x N board
    if x >= N or y >= N or x < 0 or y < 0:
        return None
```

```
    #Getting the piece on the board
    piece = int(board[4*y+x],16) #converting to int from hexadecimal
    return piece

def place_piece(piece, x, y, board):
    """
    Utility function that places piece at (x,y) on board if possible
    Will overwrite the current value at (x,y), no matter what that piece is
    Returns True if the piece is placed successfully and False otherwise
    Arg piece: string - represents a piece on the board (as an integer power of 2)
    Arg x: integer - x coordinate
    Arg y: integer - y coordinate
    Arg board: board - the board you wish to place the piece on
    """

    #Ensure that x and y are both integers (use assert)
    assert type(x) == type(y) == int, "Coordinates must be integers"


    #Checking that the (x,y) coordinates given are valid for the board
    if x >= N or y >= N or x < 0 or y < 0:
        return False

    #Placing the piece on the board as a hexadecimal string
    board[4*y+x] = hex(piece)[2:]
    return True

def place_random(board):
    """
    Helper function which is necessary for the game to go to the next move
    Returns True if a piece is placed and False if the board is full
    Places a 2 (90%) or 4 (10%) randomly on the board in an empty space
    Arg board: board - the board you wish to place the piece on
    """

    #Check if the board is full and return False if it is
    if board_full(board): return False;

    #random.random() generates a random decimal between [0, 1)
    # Multiplying by 100 generates a number between [0, 100)
    generated = random.random() * 100;

    #Assign to_place according to my generated random number
```

```
    if generated < 90:
        to_place = 1

    else:
        to_place = 2

    #keeps track of whether a randomly generated empty spot has been found
    found = False

    while not found:
        #Generate random (x,y) coordinate that we can try to put our new value in at
        random_y = int(random.random() * N)
        random_x = int(random.random() * N)

        #If empty, we have found a spot to place our random piece
        found = get_piece(random_x, random_y, board) == 0

    #Place the piece at the randomly generated (x,y) coordinate
    place_piece(to_place,random_x,random_y,board)

    return True

def board_full(board):
    """
    Utility function that returns True if the given board is full else False
    Arg board: board - the board you want to check
    """
    for piece in board:
        if piece == '0':  return False; #check string equality here

    return True;

def have_lost(board):
    """
    Helper function which checks at the end of each turn if the game has been lost
    Returns True if the board is full and no possible turns exist and False otherwise
    Arg board: board - the board you wish to check for a losing state
    """
    moves = possible_moves(board)
    if len(moves)==0: #if no moves possible
        return True
    else:
        return False

def possible_moves(board):
```

```
    """
    Utility function that, given a board, will return a list of possible moves
    Arg board: board - the board you wish to check for possible moves
    """
    N = board_size
    possible = [False, False, False, False];
    #which moves are possible (up,left, bottom, right)

    #Check every (x,y) position on the board to see which moves are possible
    for y in range(N):
        for x in range(N):
            piece_at_xy = get_piece(x, y, board); #check each piece
            if piece_at_xy == None: #not a valid piece, go to next iteration
            #logically should never happen
                continue;
            elif piece_at_xy == 0: #find where existing pieces can move to
                continue;

            else:
                if (piece_at_xy == get_piece(x+1, y, board) or
                    piece_at_xy == get_piece(x-1, y, board)):
                #has an equal piece to the right or left
                    possible[3] = True;
                    possible[1] = True;
                if (piece_at_xy == get_piece(x, y+1, board) or
                    piece_at_xy == get_piece(x, y-1, board)):
                #has an equal piece down or up
                    possible[2] = True;
                    possible[0] = True;

                if get_piece(x+1, y, board)== 0: #has an empty spot to the right
                    possible[3] = True;
                if get_piece(x-1, y, board)== 0: #has an empty spot to the left
                    possible[1] = True;
                if get_piece(x, y+1, board)== 0: #has an empty spot below
                    possible[2] = True;
                if get_piece(x, y-1, board)== 0: #has an empty spot above
                    possible[0] = True;

    moves = to_move_list(possible)
    return moves;

def to_move_list(possible):
    """helper function for possible_moves function
    returns a list of the possible moves"""
```

```
    moves = []
    if possible[0]:
        moves.append(0)
    if possible[1]:
        moves.append(1)
    if possible[2]:
        moves.append(2)
    if possible[3]:
        moves.append(3)
    return moves

def move(row):
    """
    Utility function that moves the inputted row towards the left
    Returns row after movement
    """
    #don't need for each direction - do 1 and use symmetry

    #has been replaced by more efficient look up table (see movequick function)

    n = len(row)

    mergeable = [True]*n

    for i in range(len(row)):
        row[i]=int(row[i],16) #convert from hexadecimal to int

    for i in range(n): #going through each element in row

        while True:

            val = row[i] #current tile
            last = None

            if val==0: #empty tile
                break

            if i>0:
                last=row[i-1] #last tile

            if last==None: #if nowhere to move, go to next tile
                break

            elif last==0: #move to blank tile
                row[i-1] = val
```

```
                row[i] = 0
                i = i-1

            elif mergeable[i] and (val==last):
                row[i]=0 #move this tile out
                row[i-1] = val + 1 #double the last tile
                mergeable[i-1] = False
                i = i-1

            else: #unequal tile to the left or already merged
                break

    for i in range(len(row)):
        row[i]=hex(row[i])[2:] #convert from int to hexadecimal

    return row

def swipe(board, direction):
    """
    inputs a board and direction
    outputs board after swiping in the given direction
    """
    #tracks whether any action was taken
    action_taken = False

    #valid_direction = (direction == 0 or direction == 1 or direction == 2 or directio
    #assert valid_direction, "Invalid direction passed in";  #Logical debug case

    original = copy.deepcopy(board) #store original board

    #transform to a symmetric LEFT swipe case
    board = transform_board(board, direction)

    #swipe for each row in the transformed board (in the same direction)
    #use pre-generated look up table to do this
    for i in range(4):
        row = board[4*i:4*i+4]
        newrow = movequick(row)
        board[4*i:4*i+4] = newrow

    #transform back to original
    board = transform_board(board, direction)

    #if we did something this move
    if board is not original:
```

```
        place_random(board)

    return board

def transform_board(board, direction):
    """
    transforms board to the symmetric left move case for any direction
    (i.e., moving the input board in direction is equivalent to
        moving the outputted board left)
    """
    #change to use indexing rather than calcualtionso
    if direction==1:
        return board

    if direction==3:
        indices = [3,2,1,0,7,6,5,4,11,10,9,8,15,14,13,12]
        return board[indices]

    if direction == 0:
        indices = [0,4,8,12,1,5,9,13,2,6,10,14,3,7,11,15]
        return board[indices]

    if direction == 2:
        indices = [12,8,4,0,13,9,5,1,14,10,6,2,15,11,7,3]
        return board[indices]

def automate(board):
    """runs the game automatically using the chosen AI"""

    key = int(input("Which AI would you like to run? "));
    num = input("How many games would you like the AI to play? ")

    start =  time.clock()
    scores = []
    max_tiles = []

    for i in range(int(num)):

        while True:

            global pausetime
            #pause(pausetime); #for better visibility of watching AI play

            #possible moves
            moves = possible_moves(board)
```

```
#check if lost
if len(moves)==0: #if no move is possible
        scores.append(to_score(board)) #add this score to our score list
        current_max = findMax(board)[0][0]
        max_tiles.append(current_max)
        print_board(board) #for debugging

        if(i!=int(num)-1): #unless we're at the last iteration
            #Creating a new 4x4 board
            board = make_board(board_size)
            #Getting the game started with two pieces on the board
            place_random(board)
            place_random(board)
        break

#if only one move possible then just move
if len(moves)==1:
    board = swipe(board, moves[0])

else:
    #run the chosen AI
    #get moves from the AIs program
    #implement them
    #randomise AI (input '1')

    if key == 1:
        move = randomise(board)

    #sequential AI (input '2')
    elif key == 2:
        move = sequential(board)

    #random alternating AI (input '3')
    elif key == 3:
        move = random_altern(board)

    #always up else left (input '4')
    elif key == 4:
        move = alwaysup(board)

    #AI5 (input '5')
    elif key == 5:
        move = AI5(board)
```

```
            #AI6 (input '6')
            elif key == 6:
                move = AI6(board)

            #AI5 (input '7')
            elif key == 7:
                move = AI7(board)

            #AI8 (input '8')
            elif key == 8:
                move = AI8(board)

            #AI9 (input '9')
            elif key == 9:
                move = AI9(board)

            #AI10 (input '10')
            elif key == 10:
                move = AI10(board)

            #AI11 (input '11')
            elif key == 11:
                move = AI11(board)

            #AI12
            elif key == 12:
                move = AI12(board)

            #AI13
            elif key == 13:
                move = AI13(board)

            board = swipe(board,move) #make move based on AI input

end = time.clock()
print("AI "+str(key))
print("Time taken for "+str(num)+" tries: "+str(end-start))

#compute score statistics
print("The average score is: "+str(np.mean(scores)));
print("The minimum score is: "+str(np.min(scores)));
print("The maximum score is: "+str(np.max(scores)));
print("The standard deviation of scores is: "+str(np.std(scores)));
#plt.hist(scores,bins=20)
#plt.show()
```

```
    #plt.close()
    log_scores = np.log(scores)
    print("The average log score is: "+str(np.mean(log_scores)));
    print("The standard deviation of log scores is: "+str(np.std(log_scores)));
    #plt.hist(log_scores,bins=20)
    #plt.show()
    #plt.close()
    print(max_tiles)
    print("The average max tile is: "+str(np.mean(max_tiles)));
    print("The minimum max tile is: "+str(np.min(max_tiles)));
    print("The maximum max tile is: "+str(np.max(max_tiles)));
    print("The standard deviation of the max tile is: "+str(np.std(max_tiles)));
    #plt.hist(max_tiles,bins=10)
    #plt.show()
    #plt.close()
    main()

def findMax(board):
    """input: board
    output: vector with maximum tile and it's locations [[max, x, y],[max, x, y],...]
    finds the maximum tile and all its locations (since not necessarily unique)
    Helper function for AI
    """
    max_val = 0;
    max_vec = [];

    #find the biggest tile and store its locations
    for i in range(N):
        for j in range(N):
            current = get_piece(i, j, board);
            if current > max_val:
                max_val = current; #update our max
                max_vec = []; #empty out locations of previous max
            if not current<max_val: #if we found a new location for our previous max
                max_vec.append([2**current, i, j])
                #store max val and it's location in an array
    return max_vec;

def countEmpty(board):
    """input: board
    output: int with number of empty tiles on the board
    Helper function for AI
    """
    count = 0
    #find the empty tiles
```

```
    for i in range(N):
        for j in range(N):
            current = get_piece(i, j, board);
            if current == 0:
                count += 1
    return count;

def MonteCarloEval(board,n,move,d=0):
    """after making move on board, plays randomly n times for d moves
    (till the end if d=0)
    and returns score"""

    current_board = copy.deepcopy(board)

    move_score = 0

    for j in range(n): #run n times

        board = copy.deepcopy(current_board) #reset board state
        movecount = 0

        board = swipe(board, move) #make the move

        while not have_lost(board): #play game to end randomly

            movecount += 1
            if(d>0):
                if movecount>d:
                    break

            moves_now = possible_moves(board)
            move_now = random.choice(moves_now) #move randomly
            board = swipe(board, move_now)

        move_score += to_score(board) #record score

    return move_score

#board as string of length 16
#each tile a hex

def convertRowtoBits(row):
    """input: a row of 4 hexadecimals
    output: row encoded in bits"""
    #encode row as a 16 bit unsigned integer
```

```
    num = np.uint16(int(row[0],16)*4096 +
        int(row[1],16)*256 +16*int(row[2],16)+int(row[3],16))
    #num = np.uint16(int(row,16))
    return num

def convertBitstoRow(bits):
    """input: a row of 4 integers encoded as a 16 bit unsigned int
    output: row as hexadecimals"""
    #can eliminate this function if look up table stores hex and not int
    row=[0,0,0,0]
    bits = np.binary_repr(bits) #converts to binary
    while(len(bits)<16):
        bits = '0'+bits
    for i in range(N):
        row[i]=hex(int(bits[4*i:4*i+4],2))[2:]
        #store as decimal integer converting from binary
    return row

def movequick(row):
    """uses lookup tabe to quickly find what row will be after left move
    input: row as a list of decimal powers
    outputs: a list of decimal tile powers"""
    start = convertRowtoBits(row)
    return(move_table[start])

def generate_move_table():
    for i in range(65535): #each row (encoded as 16 bits unsigned integer)
        #each entry of table as a string of length 4 with 4 hex
        row = convertBitstoRow(i)
        newrow = move(row) #generate ith entry of table
        move_table[i] = newrow #convert to bit representation


def randomise(board):
    """1: instead of the player playing the game,
    randomly choose one of the 4 moves each turn"""
    moves = possible_moves(board)
    return random.choice(moves)

def alwaysup(board):
    """4: automates gameplay with each move up, else left, else down, else right"""
    #repeated computation (automate already calculates possible moves)
    moves=possible_moves(board)
    return moves[0]
```

```
def AI7(board):
    """greedy algorithm that chooses that automates gampeplay by choosing
    the move that most increases the score every turn"""
    N = board_size

    current_board = copy.deepcopy(board)
    current_score = to_score(board)
    delta_score = [0,0,0,0] #change in score with each move

    moves = possible_moves(board)

    for i in range(len(moves)): #for each move possible
        new_board = swipe(current_board, moves[i]) #make move
        delta_score[i] = to_score(new_board) - current_score #record change in score

        best = delta_score.index(max(delta_score))
        return moves[best]


def AI9(board):
    """a slight modification of AI8 (a greedy version of AI6)"""
    N = board_size

    max_vec = findMax(board) #vectors with all maxima
    preferred = [] #list of most preferred moves
    moves = possible_moves(board)

    for current_max in max_vec: #for each max
        x = current_max[1] #location of max
        y = current_max[2]
        #corner case
        #give high priority to corners
        #(i.e. corner move preference has infinitely more weight than edge move)
        #for corner, max must be (0,0), (n-1,0), (0,n-1) or (n-1,n-1)
        if (x == 0 or x == N-1) and (y == 0 or y == N-1):
            if x==0 and 1 in moves:
            #left edge and left move possible
                preferred.append(1)
            if x==N-1 and 3 in moves:
            #right edge and right move possible
                preferred.append(3)
            if y==0 and 0 in moves:
            #if we're in the top corner and moving up is also possible
                preferred.append(0)
```

```
            if y==N-1 and 2 in moves:
            #if we're in the bottom corner and moving down is also possible
                preferred.append(2)

    #once done building preference and weights for all maxima, choose greedily
    current_board = copy.deepcopy(board)
    current_score = to_score(board)
    delta_score = [0,0,0,0] #change in score with each move

    preferred = list(set(preferred)) #remove duplicates

    if len(preferred)>0: #if we have a move preference
        moves = preferred

    for i in range(len(moves)): #for each preferred move
        new_board = swipe(current_board, moves[i]) #make move
        delta_score[i] = to_score(new_board) - current_score #record change in score
        score = current_score

    best = delta_score.index(max(delta_score))
    return moves[best]

def AI10(board):
    """greedy algorithm that chooses that automates gampeplay by
    choosing the move that maximises empty tiles on the board"""
    current_board = copy.deepcopy(board) #save current_board
    moves = possible_moves(board)
    empty = [0,0,0,0]

    for i in range(len(moves)): #calculate empty tiles for each move
        swipe(current_board, moves[i])
        empty[i] = countEmpty(board)

        best = empty.index(max(empty)) #if multiple max, chooses the first one
        return moves[best] #make the move that gives us the most empty tiles at the end

def AI11(board):
    """for each move: play game to m moves randomly n times, choose
    the starting move that leads to the best score """

    n = 10 #number of times to play the game to end for each

    original_board = copy.deepcopy(board) #save board state now
    final_score = [0,0,0,0]
    moves = possible_moves(board)
```

```
    for i in range(len(moves)): #for each possible move
        final_score[i] = MonteCarloEval(board,n,moves[i]) #record score
        board = copy.deepcopy(original_board)

    #final_scores = np.array(final_score)
    #print(final_scores/n)
    best = final_score.index(max(final_score)) #if multiple max, chooses the first one
    return moves[best] #make the move with the best final score on average

def AI12(board, d=3,n=10):
    """for each move: play game to m moves randomly n times, choose the
    starting move that leads to the best score """

    #d = 3 #depth of search
    #n = 5 #number of times to play the game to end for each

    original_board = copy.deepcopy(board) #save board state now
    final_score = [0,0,0,0]
    moves = possible_moves(board)

    for i in range(len(moves)): #for each possible move
        final_score[i] = MonteCarloEval(original_board,n,moves[i],d) #record score
        board = copy.deepcopy(original_board)

    #final_scores = np.array(final_score)
    #print(final_scores/n)
    best = final_score.index(max(final_score)) #if multiple max, chooses the first one
    return moves[best] #make the move with the best final score on average

def AI13(board,d=3,t=5):
    """tries to keep the maximum tiles in corners (preferred move)
    uses montecarlo evaluation to choose between moves on equal preference
    built for a 4x4 board
    if unique max: (exactly like AI5)
        if in corner: stay in corner (hard preferences)
        if on edge: try and move to closest corner (weighted preferences)
        if in center: move towards nearest corner (weighted preferences)
    if not unique max:
        calculate preferences for each max and choose probabilistically
    """

    original_board = copy.deepcopy(board) #save board state now
    final_score = [0,0,0,0]
    moves = possible_moves(board)
```

```
preferred = [] #list of most preferred moves
N = board_size
max_vec = findMax(board) #vectors with all maxima

for current_max in max_vec: #for each max
    x = current_max[1] #location of max
    y = current_max[2]
    #corner case
    #give high priority to corners
    #(i.e. corner move preference has infinitely more weight than an edge move)
    #for corner, max must be (0,0), (n-1,0), (0,n-1) or (n-1,n-1)
    if (x == 0 or x == N-1) and (y == 0 or y == N-1):
        if y==0 and 0 in moves:
        #if we're in the top corner and moving up is also possible
            preferred.append(0)
        if x==0 and 1 in moves:
        #left edge and left move possible
            preferred.append(1)
        if y==N-1 and 2 in moves:
        #if we're in the bottom corner and moving down is also possible
            preferred.append(2)
        if x==N-1 and 3 in moves:
        #right edge and right move possible
            preferred.append(3)

preferred = list(set(preferred)) #remove duplicates

#once done building preference and weights for all maxima
if len(preferred)>0: #if we have a move preference
    if len(preferred)==1:
        return preferred[0]

    else:
        moves = preferred #choose from preferred moves

for i in range(len(moves)): #for each possible move
    final_score[i] = MonteCarloEval(original_board,t,moves[i],d) #record score
    board = copy.deepcopy(original_board)

final_scores = np.array(final_score)
print(final_scores/t)
best = final_score.index(max(final_score)) #if multiple max, chooses the first one
return moves[best] #make the move with the best final score on average
```

```
generate_move_table() #generate the table
#COMMENT OUT ABOVE LINE AFTER RUNNING ONCE

main()
```