Comparing Approaches to Solve 2048, The Game

Eren Erisgen, Samantha Ballesteros

April 2024

Abstract

This experiment examines how three different problem-solving algorithms commonly used in artificial intelligence for searching for solutions compare when applied to the popular puzzle game 2048: Monte Carlo Tree Search, Expectimax, and Minimax. The game's premise is to combine numbered tiles to achieve a tile with a total of 2048. Game play can proceed past this, terminating when the 4x4 board is filled with tiles and no moves can be made. In nearly 3000 trials divided between the three algorithms, it is evident that there is a linear correlation between the number of moves completed and the total score achieved in a trial. The greater rate of change in total score compared to the number of moves made suggests that the given algorithm chose an action that resulted in a greater score increase. Analyzing the three algorithms with three metrics, the average score of the trials, the highest score achieved, and the slope of its trials when the total score is plotted against moves made allows for comparison between the Monte Carlo Tree Search, Expectimax, and Mininax algorithms. The Monte Carlo Tree Search resulted in a median score of 16,492, a high score of 60,628, and a slope of 18.54 points per move. Expectimax trials had a median score of 11,444, a high score of 34,620, and the lowest points per move of 16.39. Minimax had the highest median score of 18,048, a high score of 78,536, and a slope of 22.26 points per move. It should be noted that the slope was extracted using a linear regression model fitted to the trial data for each algorithm excluding outliers which were classified as data points that deviated from the first and third quartiles by more than 1.5 times the interquartile range. Thus, it seems that Minimax performed the best in solving the 2048 game.

Introduction

Rules of 2048

2048 is a popular online 'slide and merge' style game, developed by Gabriele Cirulli. The game is centralized on a 4×4 grid populated with small colored tiles. The game begins with two tiles that start as 2 but have a 10% chance of being 4[15]. The player can move these tiles in one of four directions: up, down, left, and right. However, there are legal and illegal moves that limit movement. Legal moves are any moves in which at least one tile slides, or merges into another tile, in one of the four directions[6]. When two tiles merge, with the same value v, the two tiles are removed and replaced with a new tile in the position of the stationary with a value of $2v$. Thus the tiles grow at a rate of 2^k $k >= 1$ [6]. This does give a hard upper limit as there is a finite amount of space. Therefore the game will always terminate, however, it's far more likely the player will get stuck in a terminal state. The terminal states for 2048 are when the player has no more legal moves.

2048 in Game Theory

2048 has elements of both combinatorial and stochastic puzzles. A Combinatorial puzzle is defined as a single-player sequential game with no randomness or hidden information [3]. These kinds of puzzles give us a great structure to design algorithms to beat a specific game. Trees are naturally a good representation of this game's search space since each state's four actions lead to four "branches" of potential future states, subdividing into a sequence of moves. [3]. However, 2048 violates one of the rules–randomness. The new tile placement is completely random along with the value within. A stochastic puzzle in contrast incorporates an element of randomness, and the outcome is non-deterministic. More specifically 2048 is NP-Hard (nondeterministic polynomial) amount of time to solve, along with PSPACE, or polynomial amount of space[2][3][7]. Both of these refer to computational complexity theory, neither of which we will cover in detail. Nonetheless, it's critical to understand the computational complexity constraints as it explains why the problem is one of decisions, not optimality. For example, just the initial state of the game alone has 360 possible combinations. Some of these starting states can be seen in Figure 1. This was calculated by taking the three combinations of starting tiles, $(2, 2)$, $(2, 4)$, and $(4, 4)$ times the number of combinations for the initial state, $\binom{16}{2} = 120$. Mathematically speaking, the highest plausible value for a tile is exactly 2^{17}, or $131,072$ [4][6].

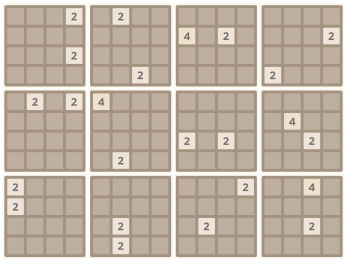

Figure 1: Sample of initial states [8]

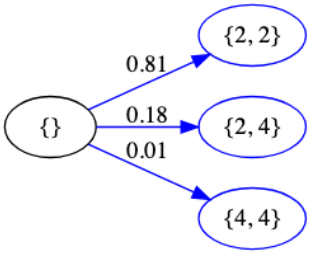

Figure 2: Initial values of starting tiles [8]

Background

2048 Strategies

The randomness in where tiles spawn, the value that those tiles take, and the large branching factor from the different actions that can be taken result in an extensively vast search space. Due to this, traditional tree search algorithms such as depth-first search (DFS) or breadth-first search (BFS) are ineffective and highly inefficient. Other methods that have been employed are machine learning neural networks as well as more conventional reinforcement learning systems [6]. However, both need vast amounts of training data and training time [12]. Thus, traditional tree search algorithms have the advantage of not requiring any training time albeit at a higher runtime memory cost. Hence, algorithms such as Monte Carlo Tree Search, Expectimax, and Minimax are better-suited approaches since decisions on what action to take from each state are made based on evaluation functions during runtime, instead of information gathered from previous experiences [12]. These evaluation functions, heuristics, are critical for the algorithm's efficiency. While the branching factor is low, the depth and randomness make evaluation game states increasingly difficult and expensive [13, 6]. A common evaluation function uses matrix multiplication between the values of the game board and a weight matrix (one that defines how desirable each position on the board is). The weight matrix being used in these experiments can be seen in Figure 3

Monte Carlo Tree Search

Unlike traditional tree search algorithms, Monte Carlo Tree Search is applicable in large spaces as it aims to explore only promising areas of the search space informed by its heuristic as opposed to BFS and DFS which attempt to exhaust the entire space. Monte Carlo Tree Search operates in four distinct phases: selection, expansion, sim-

ulation, and back-propagation. In the selection phase, the algorithm traverses the search space tree by using a best-first approach, prioritizing nodes that evaluate to the highest value. Expansion involves generating successor nodes from the possible actions to be taken from the current state, expanding the tree. During the simulation phase, different actions are taken to simulate the outcome of taking a series of actions. In back-propagation, the algorithm then uses information learned from simulations to update previous nodes to reflect the simulated outcomes[16] better. In regards to its applications to the 2048 game, there is a vast history of research for this algorithm. There is precedent for evaluation functions that can be applied to each state, and its application to these algorithms[9]. A key finding for this application is that back-propagation is especially important in generating a maximal solution [17]

Expectimax and Minimax

The Minimax algorithms is widely used in game theory and artificial intelligence, typically two-player zero-sum games with perfect information. The algorithm operates by applying an evaluation function for each state to quantify its desirability for each player and determines what action each player would make under the assumption they are playing rationally. Dating back over 70 years, Minimax has been used to solve popular games like chess and tic-tac-toe. [14]. Alpha-beta pruning is a modification to the algorithm that "prunes" branches in the search space that will not change the outcome of the algorithm and therefore do not need to be traversed. This technique has been widely used, and is proven to still generate the same outcome as Minimax without pruning [5]. The shortcoming of Minimax is that it doesn't fit for games without perfect information. Thus came Expectimax, an expansion on the Minimax algorithm. In 1966, Donald Michie introduced this variation of the original approach that incorporates chance nodes[11]. Thus, it can be applied to games with an element of randomness like Mahjong, Poker, and 2048. Like Minimax, Expectimax can be made more efficient by pruning, gamma-pruning in this case [10]. There is precedent for this approach being used for 2048. One study applied depth-limited Minimax and depth-limited Expectimax to 2048 and found that Expectimax at a search depth limit of 3 outperformed Minimax at a depth limit of 8 [18].

Approach

Evaluation Function

The matrix depicted in Figure 3 is the weight matrix used to evaluate the game state for a 4×4 board. These values prioritize a snake pattern; A common approach among real players. This strategy favors keeping higher-value tiles, which are less likely to merge turn by turn, to stay out of the way of smaller tiles. The tiles can then be sequentially merged to produce the greatest possible tile from those available on the game board. Figure 4 shows an example of this. From the upper left corner, tiles are in descending order along the path that "snakes" around the board. Since this evaluation function mimics a real-life strategy, it is possible to improve upon the weights assigned to each tile with the use of genetic algorithms, reinforcement learning, or machine learning to improve game outcomes.[1].

$$\begin{bmatrix} 10 & 8 & 7 & 6.5 \\ 0.5 & 0.7 & 1 & 3 \\ -0.5 & -1.5 & -1.8 & -2 \\ -3.8 & -3.7 & -3.5 & -3 \end{bmatrix}$$

Figure 3: Evaluation Matrix

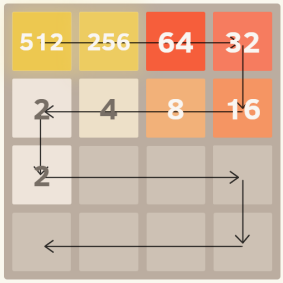

Figure 4: An example of the 2048 snake pattern strategy

Monte Carlo Tree Search

Monte Carlo Tree Search runs numerous simulations using randomly generated moves. During the simulated run, it will select a random move and apply the move to the simulated game state. Then it will return the move and score. These return values are used in conjunction with the other simulations. Thus allowing the agent to select the best move from all the simulated runs. This will repeat until the agent reaches a terminate state, where it will return.

Algorithm 1 Monte Carlo Simulation

```
1: function MonteCarlo(runs, game)
2:     while game.canContinue() do
3:         scores ← {0, 0, 0, 0}
4:         counter ← {0, 0, 0, 0}
5:         for i ← 1 to runs do
6:             (best_move, score) ← simulateOneGame(game)
7:             scores[best_move] += score
8:             counter[best_move] += 1
9:         end for
10:        best_move ← SelectBestMove(scores, counter)
11:        game.makeMove(best_move)
12:    end while
13: end function
```

Algorithm 2 Simulation of One Game

```
1: function SimulateOneGame(game)
2:     game_cpy ← game
3:     first_move ← −1
4:     while game_cpy.canContinue() do
5:         before_state ← game_cpy.state
6:         move_bank ← {UP, DOWN, LEFT, RIGHT}
7:         while before_state == game_cpy.state do
8:             random_pos ← DISTS[move_bank.size() − 1](rng)
9:             chosen_move ← move_bank[random_pos]
10:            MakeMove(game_cpy, chosen_move)
11:            if first_move = −1 then
12:                first_move ← chosen_move
13:            end if
14:            move_bank.erase(move_bank.begin() + random_pos)
15:        end while
16:    end while
17:    return (first_move, game_cpy.score)
18: end function
```

Search

Before explaining the Expectimax and Minimax codes, we need a search process for these algorithms to return to. The following pseudo-code is abstracted to reduce redundancy as they are largely the same. The main idea is to continue to select the best move scored by Expectimax or Minimax until it reaches a terminal state.

Algorithm 3 Search

```
1: function SEARCH(depth, game)
2:     while game.canContinue() do
3:         possibilities ← COMPUTEPOSSIBILITIES(game)
4:         scores ← {}
5:         for (move, weightedmoves) ← possibilities do
6:             totalScore ← 0
7:             for (probability, nextGame) ← weightedmoves do
8:                 score ← EXPECTIMAX(depth, nextGame, true)
9:                 score ← MINIMAX(depth, nextGame)
10:                totalScore += probability × score
11:            end for
12:            scores[move] ← totalScore
13:        end for
14:        bestMove ← SELECTBESTMOVE(scores)
15:        game.makeMove(bestMove)
16:    end while
17: end function
```

Expectimax

Expectimax is a variant of Minimax but it is used to maximize a given value. In the case of 2048 that would be to maximize the score of the agent. It does so by exploring a search tree up to a specific depth. During this search, it has two branches: maximizing and minimizing. The maximizing does exactly that, taking the max value between the current best score and the current score. The minimizing does something similar, however, it returns the average of the expected value with respect to the probability. Once it has reached its maximum depth, it will return recursively. This returned value is used by the search function to determine which move has the best future outcomes.

Algorithm 4 Expectimax Algorithm

```
1: procedure EXPECTIMAX(state, depth, maximizingPlayer)
2:     if depth = 0 or state is terminal then
3:         return EVALUATE(state)
4:     end if
5:     if maximizingPlayer then
6:         bestValue ← −∞
7:         for move in GENERATEMOVES(state) do
8:             value ← EXPECTIMAX(Result(state, move), depth − 1, False)
9:             bestValue ← max(bestValue, value)
10:        end for
11:        return bestValue
12:    else
13:        expectedValue ← 0
14:        totalProbability ← 0
15:        for move in GENERATEMOVES(state) do
16:            probability ← PROBABILITY(move)
17:            totalProbability += probability
18:            expectedValue += probability× EXPECTIMAX(Result(state, move), depth − 1, True)
19:        end for
20:        return expectedValue / totalProbability
21:    end if
22: end procedure
```

Minimax

Minimax is very similar to Expectimax, however, it has it assumes that the minimizing player is playing optimally. This is different from Expectimax as it assumes the adversary is random. In the case of 2048, it means the search algorithm will take the minimum between the best value and the current value. This is different than Expectimax as it took the average in the minimizing branch. Nonetheless, the search is nearly identical besides that small exception. Thus why a similar search function can be used for both algorithms.

Algorithm 5 Minimax Algorithm

```
1: procedure MINIMAX(state, depth, maximizingPlayer)
2:     if depth = 0 or state is terminal then
3:         return EVALUATE(state)
4:     end if
5:     if maximizingPlayer then
6:         bestValue ← −∞
7:         for move in GENERATEMOVES(state) do
8:             value ← MINIMAX(Result(state, move), depth − 1, False)
9:             bestValue ← max(bestValue, value)
10:        end for
11:        return bestValue
12:    else
13:        bestValue ← +∞
14:        for move in GENERATEMOVES(state) do
15:            value ← MINIMAX(Result(state, move), depth − 1, True)
16:            bestValue ← min(bestValue, value)
17:        end for
18:        return bestValue
19:    end if
20: end procedure
```

Experiment and Results

Experiment Design

To carry out our experiment an existing 2048 base game code in C++ was employed. Using a clone of the game allowed direct access to game mechanics instead of working through the graphical user interface. Additionally, this allowed the simulations to run concurrently as they did not rely on visual feedback. The code base did require some modifications to ensure the desired algorithms would work as intended. Additionally, there were trackers for key statistics needed for Analysis. Once all modifications were working and tested, a C script was created to run games concurrently by spawning batches of processes to run the simulations. The experiment was carried out on a 2021 M1 Pro processor with 16 GB of onboard memory. To gather all our data we will implement both a 2048 game with a solver as well as a script to continues run our solvers. The 2048 game will be based on an existing code base written in C++

based on a Python code base. This choice was made for two core reasons: speed and programming familiarity.

Selected Depths

Before starting the experiment, some small preliminary tests were done to find suitable parameters for our algorithms. The parameters are the depth for Minimax and Expectimax along with the number of random simulations for Monte Carlo Tree Search. To find suitable parameters for both, Expectimax and Minimax were run several times with varying depths. A similar approach was done for Monte Carlo Tree Search with the number of simulations. In an attempt to make sure it was a fair comparison, the total time for a game was used as a target. Additionally, the depth and number of simulations needed to at least be large enough to give the agents a fighting chance to win. However, the values couldn't be too large otherwise the time to complete a trial would be impracticable. Thus the depth of 10 for Expectimax and Minimax and the number of trials of 100 for Monte Carlo were selected. These values were suitable as they balanced speed, memory, and correctness evenly among the three algorithms.

Code Design

The code is comprised of three main components: the main solver, individual algorithms search methods, and the simulations script. The bulk of the code was the implementation of the individual algorithms. Luckily the forked code base provided an easy integration of the algorithms. These functions were used to select the best possible move given the current state of the game board. The game board was a C++ class that abstracted away all the game mechanics behind functions for the cardinal direction movements. Additionally, it had several helper functions that aided in the algorithm's implementation as well as statistic generation. The main solver was very rudimentary, it ran the search method for each move and logged the number of moves. The main solver also had several helper functions involved with the snake heuristic. Notably, a heuristic score function which all three of the algorithms used to determine the best move. Upon reaching a terminate state, the statistics were printed to standard out. Nonetheless, the main solver would continue to run the search algorithm until it reached a terminate state. This was crucial as the C script redirected these printed statistics into a CSV file.

The C script design is a basic master-worker model, where a master process spawns a batch of processes. Using the input parameters of the main solver exe-

cutable, the script can change the algorithm used. Additionally, it can control the depth and number of simulations. These parameters are passed from the script's input into the exec calls within the child processes. The parent controls all the child processes and waits for them to terminate before running the next batch. The batches were used to control the number of processes running to ensure each had enough resources and CPU time. Additionally, the script had an alarm that would activate after 6 minutes to kill any processes that failed to terminate. This was to ensure that the script doesn't get infinitely stuck on a process preventing the next batches from running.

Experiment Results

In total, 2941 simulations were run—approximately 1000 trials for each algorithm. If after six minutes a trial had yet to conclude, it would be terminated. This prevented the process executing that trial from consuming machine resources for excessive durations of time. It is possible that this affected the maximum score found by each algorithm, however, it should have minimal impact on the overall trends in the experiment's data. The C program aggregated data from each trial into CSV files. This included whether or not the simulation "won" (meaning it generated at least one 2048 tile), the final score of the simulation, the highest tile generated, the number of moves completed before reaching a terminal state, and the time the trial took in seconds. R scripts were then applied to the CSV files to produce visual representations of the data.

Monte Carlo Tree Search

957 simulations were successfully run employing Monte Carlo Tree Search. Of the simulations, the highest score produced was 60,628 from 3239 moves with a runtime of 22.337 seconds. Furthermore, additional statistics regarding the Monte Carlo Tree Search trials can be found in Table 1.

Means			Medians		
Score	Moves	Time	Score	Moves	Time
21900.44	1361.839	18.575	16492	1141.0	10.654

Table 1: Monte Carlo Data

Figure 5 and shows the relationship between the total points and total moves made for Monte Carlo Tree Search. In turn, Figure 6 depicts the relationship between

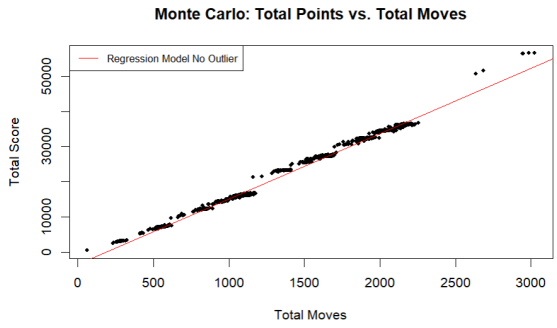

Figure 5

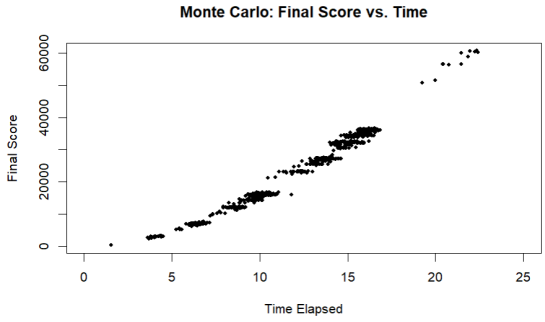

Figure 6

the final score and seconds taken for each trial.

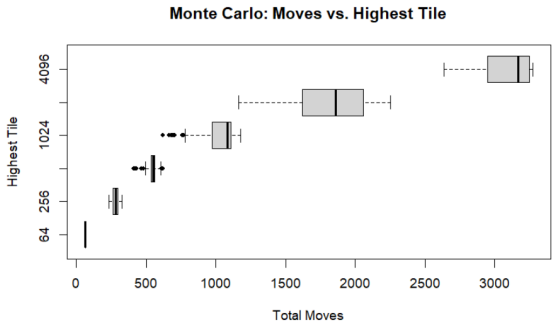

Figure 7

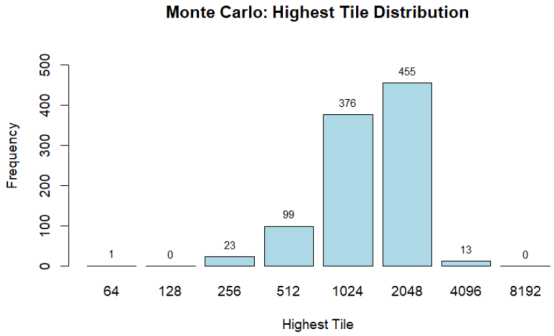

Figure 8

The box plot in Figure 7 gives a visualization for the varying distributions of total moves for the highest value tiles generated. The histogram in Figure 8 plots the frequency of the highest tiles for the Monte Carlo Tree Search simulations.

Expectimax

For Expectimax, 983 simulations were successfully completed. Of the simulations, the highest score of 34,620 was achieved in 2,139 moves over 3.650 seconds. Table 2 has additional statistics with respect to the Expectimax trials' scores, moves, and time.

Figure 9 is a scatter plot that compares the total score in a trial with the moves completed in that trial. This graph excludes an outlier that highly skews the rest of the data found. Figure 10 in turn compares the final score of a trial and the duration

Means			Medians		
Score	Moves	Time	Score	Moves	Time
10921.70	1645.798	1.876	11444	796.0	1.863

Table 2: Expectimax Data

of the trial. The following graph, Figure 11 shows how the number of moves varies with respect to the highest tile generated. The last graph for Expectimax is Figure 12, which illustrates the frequency of values for each highest tile.

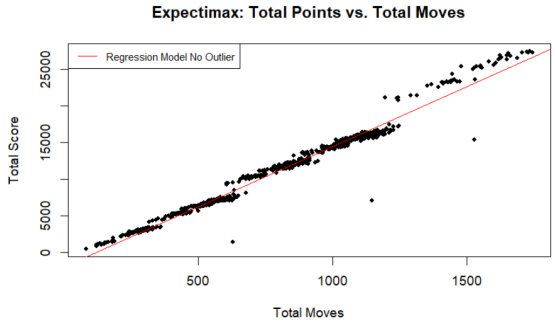

Figure 9

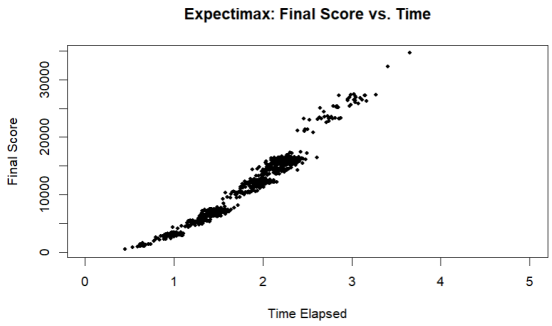

Figure 10

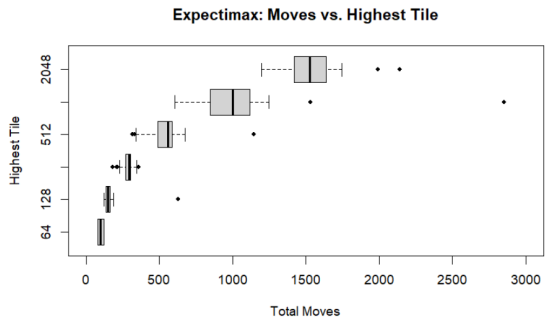

Figure 11

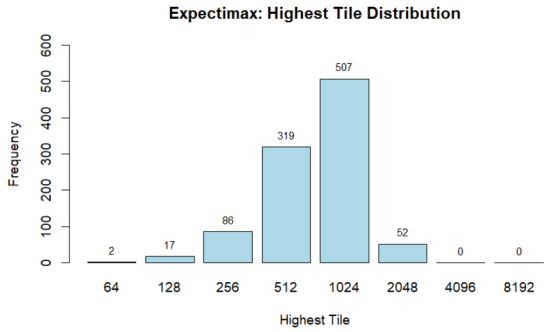

Figure 12

Minimax

998 simulations were successfully run using the Minimax algorithm. The highest score of the Minimax trials was 78,536 after 3,604 moves in 14.580 seconds. Table 3

Means			Medians		
Score	Moves	Time	Score	Moves	Time
22332.38	1201.484	7.322180	18048	1068.5	5.025

Table 3: Minimax Data

The graph in Figure 13 plots each trial's total score and the total moves taken to get there. In addition, Figure 14 shows the relationship between the final score and the time taken in a trial. Figure 15 aids to visualize the distribution in total moves depending on the value for a trial's highest tile. Figure 16 shows the frequency for each value for the highest tile.

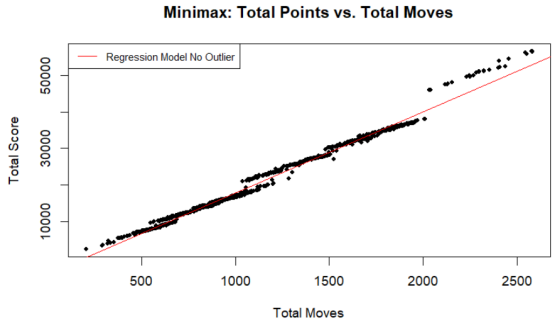

Figure 13

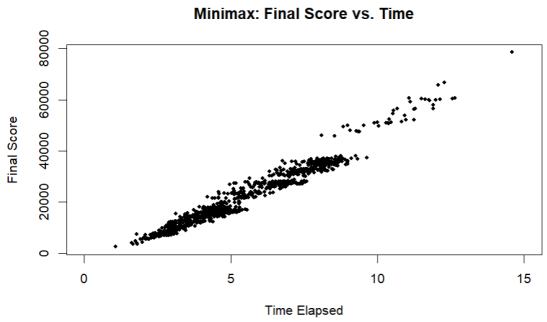

Figure 14

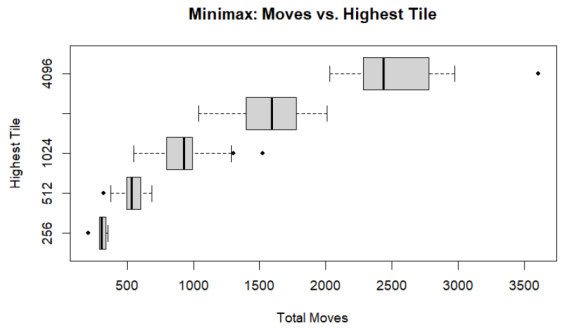

Figure 15

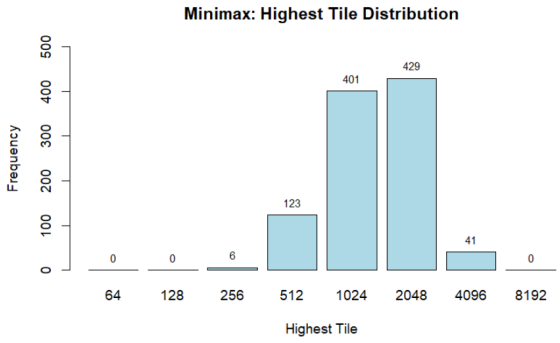

Figure 16

Analysis

When examining the graphs that compare the total points with total moves executed by each algorithm, it is evident that there is a linear correlation between the two. Thus, we can fit a linear regression model to each. This can be seen in Table 4.

	Monte Carlo	Expectimax	Minimax
Slope	18.54	16.39	22.26
R^2	0.9926	0.9785	0.9893

Table 4: Total Points vs. Moves Models

The linear regression models were fit excluding extreme outliers. A data point was considered an outlier if it varied from the upper and lower quartiles more than 1.5 times the range between the upper and lower quartiles (IQR). The slope of these regression models can be interpreted as the change in points achieved per additional move accomplished. Although this model would allow for a prediction of total points in a trial based on the number of moves made, this application is not particularly useful in the context of the algorithms' performance since the vast possible combination of actions and states makes it very difficult to predict the result of a trial. Instead, it can be used to examine how much each action picked by a specific algorithm improved the score. As the slope found increases, it indicates more favorable moves chosen by an algorithm. For Monte Carlo Tree Search, the slope was 18.54, and on average, the score increased by 18.54 points for each move chosen by MCTS. Expectimax had the smallest slope with a change of 16.39 points per move. Minimax had the greatest slope of 22.26, meaning it tended to pick better moves than both MCTS and Expectimax. Thus, Minimax selects the highest-scoring moves in general.

In turn, an R^2 value was calculated for each of these regression models. This is $1-\frac{RSS}{TSS}$, where RSS is the residual sum of squares and TSS is the total sum of squares. R^2 is a measure of how well the regression model fits the data. The R^2 value for the relationship between total points and total moves for MCTS is 0.9926, which means 99.26% of the variability in the total score can be explained by its relationship with the total moves made. Similarly for Expectimax and Minimax respectively, 97.85% and 98.93% of variability in the trials' of the total score can be explained by its relationship with the number of moves made.

Trial Times

Both mean and median runtime for each algorithm were calculated. Since several outliers were present in the data, the median can be more telling for the overall performance of these samples. The median time was 10.654 seconds for MCTS, 1.863 seconds for Expectimax, and 5.025 seconds for Minimax. While Expectimax performed the worst in terms of highest score achieved, median score, and the slope of its regression model, it was substantially faster than the two other algorithms. The scatter plots of final score and time all show direct relationships. However, a linear regression model cannot be fitted to these since they "fan" out. As time increased, variation in the final score increased.

Highest Tiles found

A box plot and a histogram were created for MCTS, Expectimax, and Minimax. The first gives insight on the distribution of moves taken with respect to the highest tile found by that trial. It can be seen in the histograms that the highest tile found by MCTS was 4096, 13 times. The highest tile for Expectimax was 2048, 52 times. The highest tile found by Minimax was 4096, 41 times. According to the rules of the game, and obvious in its name, a "win" occurs when the 2048 tile is generated. MCTS won the highest proportion of its trials with 48.40% resulting in wins. Expectimax trials had the lowest proportion of wins at only 5.29%. Minimax had a win rate of 47%. In terms of this, MCTS and Minimax had comparable proportions of wins, while Expectimax underperformed significantly.

Outliers

In the Expectimax data, several outliers could have skewed our trend line. While our calculations ignored outliers, the original data contained several. However, it does point to some oddities as the other data sets did not contain as many outliers. Without seeing the exact game states we cannot say for sure what happened. However, it could have been a software issue, such as the agent getting stuck in a non-terminal state while still merging any new tiles. Regardless, these data points had a low impact on the final analysis.

Conclusion

The goal of these experiments was to compare how three different algorithms fared at solving the well-known 2048 puzzle game. Monte Carlo Tree Search, Expectimax, and Minimax are widely used algorithms in the artificial intelligence field—especially in the context of solving puzzles and games. Thus, it seemed likely that these three approaches would be effective in solving the game. Through nearly 3000 discrete trials, Minimax performed the best in terms of attaining the highest score, highest median score, and choosing moves that produced greater increases in score. These results were marginally better than those for Monte Carlo Tree Search, yet had a median run time less than 50% of MCTS. Expectimax had the lowest high score, median score, and win percentage. However, it did have the fastest execution time. An area that can be expanded on is the search depth of Expectimax and Minimax. Due to machine resource limitations, the search depths needed to be limited such that trial runtimes weren't unreasonably long. This could lead to improvements in solutions found by Expectimax and Minimax. In addition, alternative weight matrices used in the evaluation function could be refined to produce better over-all simulations.

Contributions

- Eren: Background research, 2048 code base refactoring, Multi-process C script, Simulations, Final Paper
- Samantha: Background research, Multi-process C script, Data Analysis, Final Paper

References

[1] What is the optimal algorithm for the game 2048?, Mar 2014.

[2] Abdelkader, A., Acharya, A., and Dasler, P. On the complexity of slide-and-merge games, 2015.

[3] Demaine, E. D., and Hearn, R. A. Playing games with algorithms: Algorithmic combinatorial game theory, 2008.

[4] Goel, B. Mathematical analysis of 2048, the game. *Advances in Applied Mathematical Analysis 12*, 1 (2017), 1–7.

[5] Knuth, D. E., and Moore, R. W. An analysis of alpha-beta pruning. *Artificial Intelligence 6*, 4 (1975), 293–326.

[6] Kohler, I., Migler, T., and Khosmood, F. Composition of basic heuristics for the game 2048. In *Proceedings of the 14th International Conference on the Foundations of Digital Games* (New York, NY, USA, 2019), FDG '19, Association for Computing Machinery.

[7] Langerman, S., and Uno, Y. Threes!, fives, 1024!, and 2048 are hard. *Theoretical Computer Science 748* (2018), 17–27. FUN with Algorithms.

[8] Lees-Miller, J. The mathematics of 2048: Minimum moves to win with markov chains.

[9] Matsuzaki, K. Developing value networks for game 2048 with reinforcement learning. *Journal of Information Processing 29* (2021), 336–346.

[10] Melkó, E., and Nagy, B. Optimal strategy in games with chance nodes. *Acta Cybernetica 18*, 2 (2007), 171–192.

[11] MICHIE, D. Chapter 8 - game-playing and game-learning automata. In *Advances in Programming and Non-Numerical Computation*, L. FOX, Ed. Pergamon, 1966, pp. 183–200.

[12] Russell, S. J., Norvig, P., and Davis, E. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[13] Schadd, M. P. D., Winands, M. H. M., van den Herik, H. J., Chaslot, G. M. J. B., and Uiterwijk, J. W. H. M. Single-player monte-carlo tree search. In *Computers and Games* (Berlin, Heidelberg, 2008), H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds., Springer Berlin Heidelberg, pp. 1–12.

[14] Shannon, C. E. Programming a computer for playing chess. *Philosophical Magazine 41*, 314 (March 1950), 256–275.

[15] Slizkov, A. Computational bounds for the 2048 game, 2023.

[16] Świechowski, M., Godlewski, K., Sawicki, B., and Mańdziuk, J. Monte carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review 56*, 3 (2023), 2497–2562.

[17] Watanabe, S., and Matsuzaki, K. Enhancement of cnn-based 2048 player with monte-carlo tree search. In *2022 International Conference on Technologies and Applications of Artificial Intelligence (TAAI)* (2022), pp. 48–53.

[18] Zaky, A. Minimax and expectimax algorithm to solve 2048.