# MCTS Algorithm and Heuristic in 2048

May 8th, 2023

Jeffrey Chen
University of Minnesota
Minneapolis, Minnesota, USA
chen6377@umn.edu

Andy Thao
University of Minnesota
Minneapolis, Minnesota, USA
thao0726@umn.edu

Mohamed Bashir
University of Minnesota
Minneapolis, Minnesota, USA
bashi052@umn.edu

## 1 Abstract

The 2048 game involves the player moving and combining numbered tiles in a 4x4 grid to get a number that achieves 2048 or higher. In order for the player to lose the game, all tiles in the grid must be occupied where no action can no longer change the state of the game. This imposes a challenge in seeking an agent to be able to consistently beat the game and maybe achieve a number even greater than 2048. There are many algorithms that have been made for the game already, but our group has chosen a Monte Carlo Tree Search (MCTS) to resolve this issue. Using the MCTS algorithm, we applied a heuristic that calculates the score based on several factors, including the highest value tile on the board, the highest value tile in the four corners, the difference between them, and the values of tiles in specific rows of the grid that consist of the four corners.

It turns out after applying our heuristic, that it achieves 2048 all the time. Compare this to a simpler heuristic, this shows ours to have a considerably better performance, even with it being a bit flawed. Overall, our heuristic managed to achieve what we as a group wanted from an agent playing the 2048 game, as it provides a reasonable strategy for achieving high scores and beating the game more often than not.

## 2 Introduction

When playing a new particular game for the first time, it is often the case a player will make many sub-optimal plays to achieve their given objective. Eventually, if played long enough, the player will develop some sort of plan or strategy in order to succeed. Coming around to AI, our intentions are to give an AI agent a strategy to achieve a higher chance of success, or greater performance, in a game of 2048. In modifying an MCTS algorithm with our own heuristics, we strive to improve this agent at playing the game.

### 2.1 The 2048 Game

There are many websites that host the game and explain how the game is played[3]. The original game is a 4x4 tile grid, where each tile is either occupied (by a number) or unoccupied. The player is able to partake in the game by moving all tiles in a direction- up, down, left, or right- until stopped by the boundaries of the grid or by another tile. Whenever the player moves any of the tiles, some random unoccupied space will become occupied with a new number, either two or four. This becomes an obstacle for the player, as completely running out of space will result in a loss.

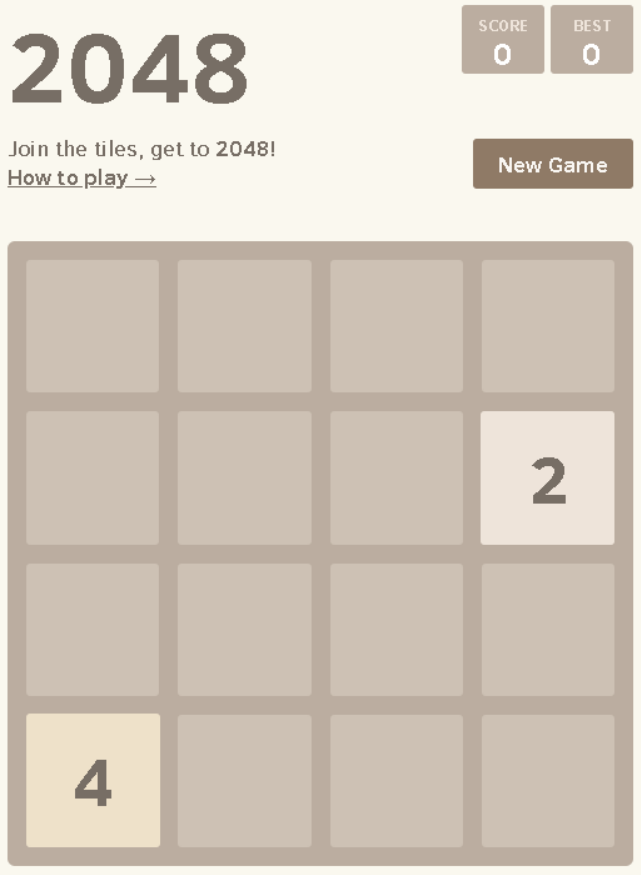


**Figure 1.** Visual of the 2048 Game

In order to keep the grid from filling up, two occupied tiles can be merged into one if and only if the two tiles are the same number. The new occupied tile's number will thus be the sum of the previous two numbers. This is also how you win the game of 2048, by merging enough tiles to reach that number the game is titled after. Although, you can still continue to play to boost up your score, which awards points based on the number of a newly merged tile.

You can see what the game looks like in Figure 1.

### 2.2 Interests of the Problem

The interests of this problem comes from the environment the agent will be in. As one may of observed, there is a small amount of indeterminable possibilities at play when an action is taken. To elaborate further on what was said previously in Section 2.1; whenever an action is taken from the player (in this case moving tiles in a direction), an unoccupied space will randomly be replaced by a new tile. This new tile also has some randomness applied to it, since there is a 90% chance for the tile to be the number 2, and a 10% for the tile to be the number 4. Therefore, the environment we will be working with will be nondeterministic, unlike games such as Checkers or Chess.

## 3 Preliminary Work

The preliminary work and results indicate the usefulness of a hybrid method that combines the Monte Carlo Tree Search algorithm with heuristics to produce an automated player for the 2048 game. It demonstrate that their method outperforms previous automated players and achieves results comparable to human players. There are experiments conducted to examine the influence of heuristics on algorithm performance, demonstrating that fine-tuning the weights of the heuristic scores increases algorithm performance even more. The findings give helpful insights into the tactics and algorithms that may be utilized for comparable games and can be applied in a variety of applications; including educational games, cognitive training, and entertainment.

### 3.1 Deep Reinforcement Learning for 2048

In a paper proposed by two MIT students[4], it made use of a hybrid algorithm that combines the MCTS algorithm with heuristics to create an automated player for the 2048 game. This will be the approach our group will also be doing. The MCTS algorithm is a widely used algorithm for game tree search that is also known to be effective in games with a large search space, such as chess and Go. The MCTS algorithm generates a search tree by simulating the game forward multiple times and selecting the move that leads to the best outcome. In contrast, heuristics are evaluation functions that assign scores to possible moves based on the game state's features. The heuristics assess the game state based on the number of vacant tiles, the maximum tile value, and the number of tiles with the same value.

To increase the approach's performance even more, it is recommended to modify the MCTS algorithm to integrate heuristics into the selection process. To pick the move, the modified MCTS algorithm employs a weighted sum of the MCTS and heuristic scores. This demonstrates that this adjustment enhances the program's performance over the original MCTS algorithm. To boost the algorithm's performance even further, the authors present a way for fine-tuning the weights of the heuristic scores.

### 3.2 AI 2048 Game Player

There is a GitHub project[1] made for 2048 that explores the Monte-Carlo Tree Search (MCTS) and reinforcement learning techniques. Looking through the files, you can find that an iteration of the MCTS algorithm is implemented in the program along with multiple evaluation and rollout strategies. This means it does 1000 roll-outs before each move to find the most promising. Each roll-out is done using a greedy strategy, which combines various heuristics, by default optimizing for smoothness and free squares. The eventual node is evaluated using the SumMeasure, which means taking the sum of all tiles.

## 4 The Approach

Inspired from the researched preliminary work, we've chosen to approach the problem using the Monte Carlo Tree Search (MCTS) algorithm and integrating it with a heuristic. The best way we found to successfully and consistently get a high score in the 2048 game is by positioning the highest numbered tile in one of the four corners of the grid. Knowing this, we can develop a heuristic that would encourage this behavior.

### 4.1 Monte Carlo Tree Search Algorithm

MCTS is based on the concept of "rollouts" or simulations, where a game is played out randomly from the current state to a terminal state to estimate the outcome of different actions. It is able to estimate the outcomes because it is a tree-based search algorithm that maintains a tree of game states and actions.

There are four stages to MCTS, which are continuously repeated in order until a termination condition, or goal, is met:

- **Selection:** Selects a child node from the tree to expand.
- **Expansion:** Generates all possible child nodes (actions) from the selected node and adds them to the tree.
- **Simulation:** Performs a simulation (rollout) on one of the new child nodes to get to a terminal state.
- **Backpropagation:** The algorithm updates the statistics of all nodes along the path from the selected node to the root based on the outcome of the simulation.

An example of these stages can be seen in Figure 2.

In other words, at each iteration, the algorithm selects a node in the tree, expands it by generating its child nodes, simulates a rollout from one of the child nodes, and updates the statistics of the nodes in the tree accordingly. The algorithm balances exploration and exploitation by choosing nodes to expand based on their current estimates of the expected value of taking an action from that state, adjusted by an exploration term.

One of the advantages of MCTS is that it can be applied to games and decision-making problems with complex and

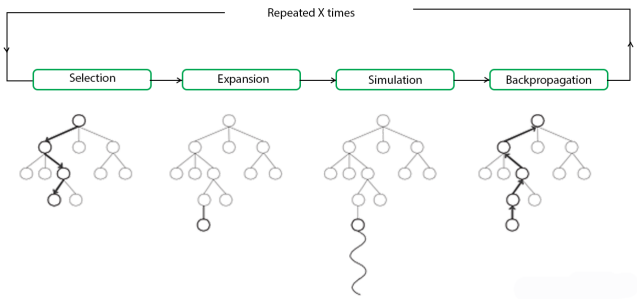


**Figure 2.** The four stages of MCTS. Starting from selection, to expansion, simulation, and backpropagation. There are some versions of this that do some of the stages differently.

stochastic dynamics, where traditional search algorithms like minimax and alpha-beta pruning may not be feasible or effective.

For our problem, 2048 is a game with a large search space and a stochastic element (the appearance of new tiles), making it difficult to determine the best move at each state using traditional search algorithms.

MCTS can handle such problems by generating random simulations from the current game state to estimate the value of different moves. This allows the algorithm to explore a large portion of the game tree and evaluate moves based on their expected outcomes, taking into account both the current state of the game and the potential future states.

### 4.2 Strategy and Game Plan for 2048

The strategy for 2048 is to place the biggest number in one of the corners of the grid. This approach allows for more potential merges and reduces the chance of getting stuck with unmergeable tiles. The justification for this is because the higher the value of the tile is, the more difficult it is for it to be merged since it requires an additional tile that is just as large. As a result, the amount of space in the grid will be significantly reduced if the highest tile is not positioned in the corner. In order for the player to keep the highest tile in any of the four corners, it highly depends on where the corner of the tile is. This is because the player must do a set of two actions that are commonly repeated to keep that tile from moving elsewhere. These commonly repeated actions (left, right, up, and down) are "adjacent", meaning both of the actions are not opposites of one another. For instance, if placed at the top-left corner, the player following this strategy will either choose the actions up-left or left-up to keep the highest tile from moving. Once a pair of actions is chosen, it must be repeatedly done in order to keep all other higher tiles from being in multiple rows (if up/down was done first) or columns (if left/right was done first). This is to ensure that only one row or column has the biggest tiles

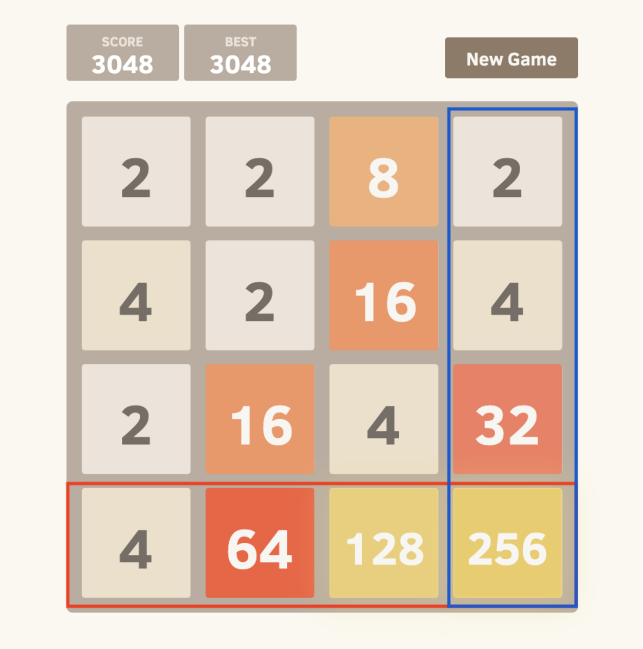


**Figure 3.** The 2048 Strategy. The largest tile in this example, 256, is in the bottom right. The second largest, 128, is to the left of it. In this example, the player has chosen the row in red to be where the tiles build up.

within it and to further increase the amount of space on the grid. And example of this is shown in Figure 3.

We can create a heuristics that can be used in the program to evaluate the game state and award points to probable moves. The heuristic algorithms is going to be similar to those suggested in the article that was tied to Deep Reinforcement Learning for 2048.[4] It evaluates the game state based on the amount of vacant tiles, the maximum tile value, and the number of tiles with the same value. We want to be able to keep the biggest tile in one corner using monotonicity. This means that through a series of consistent movement, the largest tile stays in the same place.

## 5 Related Works

### 5.1 Multiple Uses of Monte-Carlo Tree Search

[6] is a research article that provides an overview of the Monte-Carlo Tree Search (MCTS) algorithm and its applications in different fields. The authors discuss the principles of MCTS and the various improvements made to the algorithm to enhance its performance. They also review other techniques used in game-playing, including traditional methods such as minimax and alpha-beta pruning. The article goes on to discuss the potential applications of MCTS in fields beyond game-playing, including robotics and decision-making. These case studies help to demonstrate the practical utility

of MCTS beyond just game-playing. We will talk about some of these studies.

**5.1.1 Autonomous vehicles.** The first case study describes how MCTS can be used to optimize the behavior of autonomous vehicles, specifically for path planning and collision avoidance. They demonstrate how MCTS can be used to enable a self-driving car to navigate a complex urban environment with multiple intersections and other vehicles. The algorithm uses MCTS to search for the best path while taking into account various constraints, such as vehicle dynamics, traffic rules, and safety requirements.

**5.1.2 Scheduling of surgical procedures.** In this second case study, the authors describe how MCTS can be used to optimize the scheduling of surgical procedures in a hospital. The algorithm uses MCTS to search for the optimal sequence of surgeries while considering various factors such as patient priority, surgical complexity, and resource utilization. The authors report that the MCTS algorithm was able to generate schedules that were more efficient and cost-effective than those generated using traditional scheduling methods.

**5.1.3 Robotics exploration.** This third case study describes how MCTS can be used to enable a robot to explore an unknown environment efficiently. They demonstrate how MCTS can be used to plan the optimal path for a robot to explore a new environment while taking into account various factors such as uncertainty, obstacle avoidance, and resource consumption. The authors report that the MCTS algorithm was able to generate paths that were more efficient and robust than those generated using traditional path planning methods.

**5.1.4 Conclusion.** While Monte-Carlo Tree Search (MCTS) is a powerful algorithm with many potential applications, there are several challenges that researchers face when working with it. Here are challenges discussed in the article.

Fine-tuning for specific tasks: MCTS involves many parameters and hyperparameters that need to be optimized for the particular problem at hand. This process can be time-consuming and computationally expensive, and it requires a deep understanding of the problem domain.

High computational requirements: MCTS involves simulating many random games or scenarios, and this can require a significant amount of computing power. In some cases, MCTS may be too computationally expensive to use in real-time applications.

Limited scalability: MCTS can struggle to scale to large problem spaces. This is because the algorithm requires a search tree to be built, and as the problem space grows, so does the size of the search tree. This can make the algorithm impractical or even impossible to use for very large problems.

Despite these challenges, MCTS remains a valuable tool for researchers in a variety of fields. As technology advances and computing power increases, it is likely that many of these challenges will become less significant, making MCTS an even more powerful algorithm for solving complex problems.

### 5.2 Modifications and applications

**5.2.1 Modifications.** [7] is a review article that provides an overview of the recent developments in the Monte Carlo Tree Search (MCTS) algorithm and its applications in various fields. The article then discusses recent modifications to the MCTS algorithm, including techniques for optimizing the search process, improving the accuracy of the value estimates, and incorporating domain-specific knowledge into the search. It describes various applications of the MCTS algorithm, such as playing games, robotics, resource allocation, and planning. Here are some of the recent modification and applications.

**5.2.2 Optimizing the search process.** Several modifications have been proposed to optimize the search process in MCTS. One example is the use of parallelization to speed up the search. Another example is the use of progressive widening, which involves dynamically expanding the search tree to explore new actions.

**5.2.3 Improving the accuracy of the value estimates.** In traditional MCTS, the value estimates are based on the results of simulations, which can be noisy and inaccurate. Several modifications have been proposed to improve the accuracy of the value estimates, such as using regression models or neural networks to learn from the results of past simulations.

**5.2.4 Incorporating domain-specific knowledge.** MCTS can be modified to incorporate domain-specific knowledge, such as heuristics or expert knowledge, into the search process. One example is the use of domain-specific evaluation functions to guide the search towards promising actions.

**5.2.5 Applications.** [5] explains how it is successfully applied to a variety of games, including games of hidden information such as Poker, Goofspiel, and Settlers of Catan. In these types of games, players have private knowledge that is not visible to their opponents. As a result, the outcome of the game depends on the ability to infer the hidden information of the opponents.

One of the challenges in using MCTS in games of hidden information is how to capture and reuse information effectively. To address this challenge, researchers have proposed various strategies, including:

Opponent modeling: The algorithm builds a model of the opponent's behavior based on their past actions and uses it to predict their future moves.

Information reuse: The algorithm reuses the information obtained from previous searches to improve the efficiency of subsequent searches.


**Figure 4**

Exploration-exploitation trade-off: The algorithm balances the exploration of new moves with the exploitation of known good moves to find the optimal strategy.

**5.2.6 Conclusion.** The authors discuss the future directions of research in the field of MCTS, including the need for more efficient search algorithms, better integration of domain-specific knowledge, and the development of hybrid methods that combine MCTS with other algorithms. They conclude by stating that the future of MCTS is promising and that the algorithm will continue to be an active area of research in artificial intelligence and game theory.

### 5.3 MCTS experiments on games

[4] is another article that talks about MCTS and the experiments that examines its performance and effectiveness in various contexts. These studies have explored different variants of MCTS, such as UCT and MCTS with domain knowledge, as well as different ways of applying MCTS to specific problems. The results of these studies have generally been positive, showing that MCTS can achieve strong performance in a wide range of domains. Here are the experiments they did.

**5.3.1 Go.** The first experiment involved playing games of Go between the MCTS program and a human player. The goal was to test whether the MCTS program could learn to play Go at a level comparable to that of a human player. The results showed that the MCTS program was able to learn to play Go at a very high level, and was even able to beat some professional players.

**5.3.2 Chess and Checkers.** The second experiment involved comparing the performance of the MCTS program to other game-playing algorithms on a variety of games, including chess and checkers. The results showed that the MCTS program performed better than many of the other algorithms, and was able to learn to play new games quickly and effectively.

**5.3.3 Poker.** The third experiment involved testing the performance of the MCTS program on games with imperfect information, such as poker. The results showed that the MCTS program was able to learn to play these games effectively, and was even able to beat some professional poker players.

**5.3.4 Hex.** The fourth experiment involved testing the performance of the MCTS program on games with large branching factors, such as Hex. The results showed that the MCTS program was able to handle these large branching factors effectively, and was able to learn to play Hex at a very high level.

**5.3.5 Conclusion.** MCTS has been shown to be effective in finding optimal decisions in various games and decision-making problems. The article also discusses some of the challenges in implementing MCTS, such as the need for a good evaluation function and the computational cost of simulating game outcomes. However, researchers have found ways to optimize MCTS, such as using neural networks to improve the evaluation function or using parallel processing to speed up simulations.

### 5.4 Monte Carlo Tree Search Methods: Focusing on the Core Algorithm

[2] provides a comprehensive examination of the Monte Carlo Tree Search (MCTS) algorithm. The authors cover the algorithm's fundamentals and its successful application across various domains, such as computer Go, while recognizing its advantages in other areas. The paper aims to provide a snapshot of the state of the art in MCTS research after the first five years, discussing the core algorithm's derivation, numerous proposed variations and enhancements, and key applications in both game and non-game domains.

We will focus specifically on Section III of the paper, which dives into the Monte Carlo Tree Search (MCTS) algorithm. This section provides an in-depth examination of the core algorithm, its development, Upper Confidence Bound for Trees (UCT), characteristics, comparison with other algorithms, and terminology. We will not be discussing the other sections of the paper, which cover background information, variations, tree policy enhancements, other enhancements, applications, and summary.

Section III of the paper discusses the following topics:

**5.4.1 Algorithm.** The MCTS (Monte Carlo Tree Search) algorithm involves building a search tree iteratively until a predefined computational budget is reached. Each node in the tree represents a state, and child nodes represent actions leading to subsequent states. The algorithm has four steps: selection, expansion, simulation, and backpropagation. The tree policy selects or creates a leaf node from the nodes already in the search tree, while the default policy plays out the domain from a given nonterminal state to produce a value estimate. The simulation refers to playing out the task to completion according to the default policy. The best performing root action is returned as the result of the search. The algorithm can be used in multiagent domains, and the reward value may be discrete or continuous. The winning action can be selected based on four criteria: max child, robust child, max-robust child, and secure child.

**5.4.2 Development.** MCTS is a popular algorithm used in games with randomness and partial observability. It has been applied to games of perfect information as well. Coulom and Kocsis/Szepesvári proposed different approaches that combined Monte Carlo simulations with tree search and selectivity mechanisms to improve the accuracy of the simulations. Multiarmed bandit problems can be used to capture the exploitation-exploration dilemma in MCTS, with UCB1 being a popular choice for node selection. The development of MCTS is the result of the integration of different research fields in AI.

**5.4.3 UCT.** The Upper Confidence Bounds for Trees (UCT) algorithm is the most popular algorithm in the MCTS family. UCT approximates the game-theoretic value of actions by iteratively building a partial search tree. The tree is built based on how nodes are selected, and the success of MCTS is primarily due to the tree policy, which uses UCB1. The balance between exploitation and exploration in UCB1 is essential, as each node is visited and the exploration term decreases, but the numerator increases if another child of the parent node is visited. The constant in the exploration term can be adjusted to increase or decrease the amount of exploration. The rest of the algorithm proceeds as described in Section III-A, and the algorithm terminates and returns the best move found once some computational budget has been reached.

**5.4.4 Characteristics.** This subsection highlights three essential characteristics of MCTS that contribute to its popularity and success across various domains:

Domain Independence: MCTS does not rely on domain-specific knowledge, making it applicable to any tree-based domain. Although incorporating domain-specific knowledge can enhance performance, a balance between performance, generality, and speed is necessary [77].

Anytime Algorithm: MCTS immediately backpropagates game outcomes, allowing the algorithm to return an action at any time. Additional iterations can improve results, while minimax with iterative deepening has coarser progress granularity.

Asymmetric Tree Growth: MCTS tree selection favors promising nodes, leading to an asymmetric tree over time. This tree shape provides insights into the game itself. In summary, MCTS's domain independence, anytime behavior, and asymmetric tree growth make it a versatile and successful algorithm across diverse domains, often outperforming traditional depth-limited minimax algorithms in complex cases.

**5.4.5 Comparison with other algorithm.** This section compares MCTS with other algorithms and provides insights on when to choose between MCTS and minimax. If the game tree is large and there is no reliable heuristic, MCTS is suitable. If domain-specific knowledge is available, both algorithms may be viable. However, MCTS performs better in games like Go, while iterative deepening minimax performs better in games like Chess that have many trap states. Additionally, in a class of synthetic spaces, UCT outperforms minimax with a fixed additive cost penalty for suboptimal choices, and the performance gap increases with tree depth.

**5.4.6 Conclusion.** This review focused on Section III of [2] examining the core MCTS algorithm, its development, UCT, characteristics, and comparisons with other algorithms. MCTS has proven to be versatile and successful across various domains due to its domain independence, anytime behavior, and asymmetric tree growth.

MCTS outperforms traditional depth-limited minimax algorithms in complex cases with large game trees and no reliable heuristic. While both MCTS and minimax can be viable with domain-specific knowledge, MCTS excels in games like Go, while iterative deepening minimax is better for games like Chess.

In summary, MCTS's adaptability, domain independence, and capability to handle complex problems make it a valuable tool in AI research and application, with potential for continued success and innovation.

## 6 Design of the Experiments and Results

With the Java program provided by the GitHub[1], we are given a simplified recreation of the 2048 game. The files within include a board to setup the game, and various measures and strategies for our agent in this environment.

Our group was only concerned with making a new heuristic for the MCTS that would improve upon the agent in beating the game. We were not concerned with the numerous strategies given to us, so we stuck to the default strategy used: UCT. UCT stands for Upper Confidence Bounds Applied for Trees and is one of the more popular algorithms in the MCTS family. It basically selects the most promising child node by looking at which nodes have higher rewards and which nodes have been visited infrequently, striking a balance between the two.

One thing to note about the strategies however are the number of rollouts. For each action or iteration the agent was going to make, it would make 1000 rollouts, or evaluations, to find the most promising move.

Running the program in a terminal with your set strategies and measures gives you an end result that looks something similar to this:

| | | | |
|---|---|---|---|
| 2 | 3 | 5 | 8 |
| 1 | 4 | 6 | 10 |
| 2 | 3 | 7 | 9 |
| 1 | 2 | 3 | 8 |

As you can see, instead of displaying the possible numbers you would usually get in a vanilla 2048 game, we are given

the exponents of 2 that would correlate with the actual number. For example, looking at the first row, we would have 4, 8, 32, and 256 in that order from left to right.

Once we had gotten a grasp of the program, we created a new measure class called MyMeasure.java. The measure class is used for evaluating a given game state, and is basically our heuristics we plan on using when running the simulation. This class has a score method that takes in a Board object, which is the 2048 game grid, and returns an integer value that represents the score of the given board state. By modifying the function to adjust what score the simulation receives for various actions, we change the behavior of the agent in the environment.

Since we emphasized the importance of the corners in our approach, we decided to define an array that held the locations of the grid's four corners. By grabbing all four corners, we are able to determine the best spot for the largest tile to be positioned, since there is a possibility that one of the unoccupied spaces would not be reachable. We also kept track of the rows at the top and bottom of the grid, keeping them in their own array variables. This is because when a corner is picked, it is either at the top or bottom of the grid. By keeping track of this, we can increase the score if all the bigger numbers are placed alongside each other in a row. We could have involved the columns to the left or right of the grid to further improve on this heuristic, but we figured this would do for now. The only concerns we would have is making sure to reduce the amount of high valued tiles in multiple rows or columns. In other words, making sure that if a row has all the higher numbers that it isn't intent on having a column with higher numbers as well; or vise versa.

In our experiment, we were given other kinds of measures. The one that we were only comparing against was SumMeasure, which summed up a score based on the entire board and the values of each tile. In terms of the code, it would iterate through an array that represented the 4x4 grid. Every time it found a tile with a value, it would greatly improve the score based on how big the value was. Figure 3 shows the majority of the code that calculates the score.

SumMeasure was a good heuristic, but one that did not actually win a whole lot. Most of the time, it wouldn't get far; getting to a score around 1024 only to fall short because it had its highest value tile someplace in the center. The grid shown a few paragraphs back is an example taken with SumMeasure. However, it was still better than MyMeasure during the early stages of building it.

When starting our new heuristic, we built off another measure called BestMeasure. This measure was very simplistic and took in a score corresponding to the highest value tile. We intended on getting the highest value tile from the entirety of the grid, and with it, compare its location on the grid to one of the corners. That way, we would attempt to persuade the agent to lock its highest valued tile in a corner; just like how we wanted in our approach strategy. Unfortunately, as we tested our heuristic in the simulation, it resulted in a lower score compared to SumMeasure- though we did keep the highest valued tile in a corner most of the time. It was significantly worse, as our agent's maximum highest valued tile would be usually around 256. Even accounting for the rows at the top and bottom of the grid, as said earlier, did not help as much in improving the agent at beating the game.

As a result, we decided to add the SumMeasure heuristic into our own. The reasoning for this was that when observing its simulation, we saw that it continuously merged its tiles. This was most likely due to the fact that SumMeasure's heuristic continuously accounted for the rest of the board, whereas MyMeasure only accounted for the four corners. Because it accounted for the rest of the board, it would continuously combine same numbered tiles, unlike MyMeasure which seemed to do it occasionally. Though our first implementation of SumMeasure did not do well as we had hope, readjusting the score it gave fixed that problem. So, by including SumMeasure, it significantly changed the outcome by allowing tiles to merge and therefore allowing more space for bigger valued tiles.

In the end with all three measures combined into MyMeasure, we tested our heuristic and found that it won a lot more than SumMeasure since the agent did not want to place higher valued tiles in the center.

**Algorithm 1** SumMeasure.java

```
0: function SCORE(board)
0:     Initialize score ← 0
0:     for all tiles s in board.grid() do
0:         if s > 0 then
0:             score ← score + 2^s
0:         end if
0:     end for
0:     return score
0: end function=0
```

**Algorithm 2** BestMeasure.java

```
0: function SCORE(board)
0:     Initialize best ← 0
0:     for all positions p in Board.all do
0:         best ← max(best, board.grid()[p])
0:     end for
0:     if best = 0 then
0:         return 0
0:     else
0:         return 2^best
0:     end if
0: end function=0
```

**Algorithm 3** MyMeasure.java

```
0: function SCORE(board)
0:     Initialize score ← 0
0:     Find best (the highest tile value on the board)
0:     Find bestCornerValue (the highest tile value in the corners)
0:     if best is in the same position as bestCornerValue then
0:         score ← score + best × 10
0:     end if
0:     if bestCornerValue = best then
0:         score ← score + 100
0:     else
0:         score ← score + 10 × (best − bestCornerValue)
0:     end if
0:     for all tiles s on the board do
0:         if s > 0 then
0:             score ← score + 2^s
0:         end if
0:     end for
0:     return score
0: end function=0
```

## 7 Analysis of the Results

As stated in the previous section, we compared our results obtained by the AI agent using the default UCTStrategy and the custom MyMeasure heuristic with one that used the default heuristic, SumMeasure. The performance of the AI agent in the 2048 game was evaluated based on several key metrics. These metrics are the win percentage, average score, standard deviation, minimum/maximum scores, and average time per move.

- **Win Percentage:** This metric represents the percentage of game won by the AI agent using the measure.
- **Average Score:** This represent the mean score of all the game played.
- **Standard Deviation:** This represent the variability of the score.
- **Minimum and Maximum Score:** These metric represent the lowest and highest score obtained in the game played.
- **Average Time per Move(ms/m):** This metric represents the mean time taken by the AI agent to make a move, measured in milliseconds.

Thanks to the provided information received from every test conducted, enabled us to have an understanding of the effectiveness and efficiency of the MyMeasure heuristic in comparison to the default strategy.

### 7.1 Comparisons

In all of the games tested with MyMeasure, all of them achieved 2048 and thus a 100% win percentage. Compared this to SumMeasure, about four-fifth of the games got that far. This means that MyMeasure is a lot more consistent than SumMeasure. Logically, this makes sense, as MyMeasure is attempting to follow a plan that requires a way of forming a specific game state. For SumMeasure, it only looks at the grid as a whole - not caring where specific tiles are located.

Using the MyMeasure heuristic also led to a *decrease* in the average score achieved by the AI agent in the game of 2048. For our tests, SumMeasure had an average score of around 6800.4, about a 600 difference to MyMeasure's 6284.2. So though MyMeasure is a lot more consistent, SumMeasure is capable of going further beyond MyMeasure's range. This might mean that MyMeasure might lead to a more messier game state, and our original plan isn't working as intended. Perhaps the way we determined the score of a number in our corner and rows is incorrect, or maybe SumMeasure just has a more commonly reached higher ceiling its values can obtain.

For the standard deviation of the two, MyMeasure had the value 2266.2221 as compared to SumMeasure having the value of 2506.0516. The standard deviation measures the amount of variation of data values from the average score. This means the bigger the value is, the more spread out the data is. As we said previously in regards to the win percentage, MyMeasure is a lot more consistent than SumMeasure. Therefore, its lower standard deviation is to be expected compared to SumMeasure's.

The most interesting piece of information gathered from the tests involved with the two measures are the Min and Max values. For MyMeasure's part, its minimum is 4178.0, basically doubling SumMeasure's 2048.0. Not only that, its maximum is 11839.0, which is higher than SumMeasure's 8192.0 by around 3000. This further proves the consistency of MyMeasure's heuristics and shows that it will often win more so than SumMeasure's. In addition to that, it can exceed a score past in the 10000's. SumMeasure's maximum doesn't mean that it can not however, but it still shows potential in MyMeasure.

The SumMeasure had an average time per move of around 103.6579 milliseconds, while MyMeasure reduced it to around 74.199 milliseconds. This indicates that the MyMeasure heuristic not only improves the AI agent's performance in the game but also makes it more time-efficient. Seeing this result is very surprising for us, as we expected that our numerous loops of iterating through the grid in our code would result in the performance slowing down. We are constantly checking for the four corners of the grid as well as the top and bottom rows. It is possible that the MyMeasure heuristic implemented in the MCTS algorithm reduces the number of unnecessary iterations through the grid. By focusing on the positions of the highest valued tiles, we remove some actions that would stray farther from those positions.

### 7.2 Reflection

In our approach, we wanted to make the agent succeed by aiming to position the highest value tile in the corners of the grid to facilitate more effective merging of tiles and ultimately achieve higher scores. Though the agent succeeded in winning the objective of the 2048 game, our heuristic wasn't perfect as we anticipated. Retrieving an outcome from a MyMeasure test, we have:

| | | | |
|---|---|---|---|
| 2 | 12 | 11 | 2 |
| 7 | 10 | 8 | 7 |
| 3 | 6 | 5 | 3 |
| 1 | 2 | 1 | 2 |

We can observe that our strategy did not always succeed in placing the highest value tile in the corners. As you can see, the highest tile (12) is placed near the top-left corner, but not in the actual desired position. This isn't the only particular instance. Often it is the case that the agent prefers placing its highest valued tiles between two corners. This is most likely due to our implementation of the top and bottom rows we stored and compared to with the corner that held the biggest value. If not that, then it could of been, or the addition of, the score value multiplier that we wrote.

There are definitely room for further improvement in our heuristic. By refining our approach and accounting for situations where the highest value tile is not ideally positioned, we may be able to further enhance the AI agent's performance in the 2048 game and more consistently achieve our intended strategy. This in turn might also push our heuristics to have a better average score than SumMeasure.

In summary, the MyMeasure heuristic significantly outperforms the default UCTStrategy in terms of win percentage, average score, and time efficiency. Incorporating the custom heuristic in the MCTS algorithm has led to a more effective and efficient AI agent for playing the 2048 game. Though that may be the case, more adjustments can be made to improve this heuristic.

## 8 Conclusion

In this study, we addressed the problem of creating an AI agent that can consistently beat the 2048 game and potentially achieve a number higher than 2048. To do so, we employed a Monte Carlo Tree Search (MCTS) algorithm, combined with our own heuristics with a bit of support from ones given to us by the GitHub project. Our heuristics were designed to evaluate game states based on the highest tile values on the board and in the corners, the difference between these values, and the tile values in specific positions on the board.

Through numerous testing, our results showed that the MCTS algorithm, when equipped with our own heuristics, was able to achieve a higher chance of success and improved performance in the game of 2048. By adapting the AI agent's strategy and incorporating heuristics that take into account various aspects of the game state, the agent was able to make more informed decisions and achieve better results.

Despite the promising outcomes of our approach, there is still room for further improvement and future work. Some possible directions for future research include:

- Refining the heuristics: The current heuristics could be further optimized by exploring different weighting schemes or incorporating additional factors that influence the game's outcome. For example, considering the columns of the grid as well, not just the rows. The heuristics for keeping the highest value in one of the four corner also requires more fine-tuning.
- Investigating alternative algorithms: While MCTS has proven to be effective, it is worth exploring other algorithms, such as deep reinforcement learning, to compare their performance and possibly combine them with our current approach. Deep reinforcement learning is a type of machine learning that combines reinforcement learning and deep neural networks. In this approach, the AI agent learns to take actions in an environment by maximizing a reward signal. The agent receives feedback in the form of positive or negative rewards based on its actions, and it adjusts its behavior accordingly to maximize the rewards over time. This approach has shown promising results in various domains, including game playing, robotics, and natural language processing.
- Expanding the search space: Our current implementation is limited to the standard 4x4 grid of the 2048 game. However, extending the AI agent's capability to handle larger grids or alternative game modes would increase its versatility and applicability.
- Adapting the AI agent for real-time play: An interesting direction would be to modify the AI agent to play against human opponents in real-time, allowing it to learn from its opponents' strategies and further improve its performance.

In conclusion, our work demonstrates that the combination of the MCTS algorithm and carefully designed heuristics can lead to a robust AI agent capable of achieving high performance in the 2048 game. As seen in our results, a good addition to the heuristics can drastically improve on its design, and thus, meet our group's objective. By building on our results and exploring future directions, we hope to further improve the agent's performance and contribute to the development of AI solutions for a wide range of gaming applications.

## References


[1] Thomas Dybdahl Ahle. 2020. mcts-2048. https://github.com/thomasahle/mcts-2048.

[2] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 1 (2012), 5–11. https://doi.org/10.1109/TCIAIG.2012.2186810

[3] Gabriele Cirulli. 2014. 2048. https://play2048.co.

[4] Antoine Dedieu and Jonathan Amar. 2017. Cooperative Pathfinding. In *Deep Reinforcement Learning for 2048*. Conference on Neural Information Processing Systems (NIPS), Long Beach, California, 1–10.

[5] Edward J. Powley, Peter I. Cowling, and Daniel Whitehouse. 2014. Information capture and reuse strategies in Monte Carlo Tree Search, with applications to games of hidden information. Artificial Intelligence, York, United Kingdom, 92–116.

[6] Richard Senington. 2022. The Multiple Uses of Monte-Carlo Tree Search. IOS Press, Skövde, Sweden, 713–723.

[7] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. 2022. Monte Carlo Tree Search: a review of recent modifications and applications. Artificial Intelligence Review, Poland, 2497–2562.