Final Project: Creating an agent to play 2048

Omar Ceesay

May 2021

Abstract

In this paper, a Monte-Carlo tree search (MCTS) algorithm will be used to solve the popular mobile game 2048. The results of the game-play will be examined analytically and holistically. MCTS is typically an effective algorithm for game solving and in recent years have made headlines by being used to play against and beating the top players in Go, which has been historically a challenging game for AI to play when going against professionals.

1 Introduction

The game 2048 (https://play2048.co) is a single-player puzzle game that gained popularity in the 2010's. The game starts with a 4x4 grid with tiles in 2 of the grid spaces. The player then chooses to shift the tiles in one of the 4 directions. If 2 tiles of the same value are shifted into each other then those tiles merge into one tile with double the value. After each shift, a new tile with value 2 or 4 will spawn in an empty space randomly. When a human is playing the game 2048 for the first time they often try to anticipate which moves lead to future possible grid states that are better than their other options. Of course given how many possible number of states you could lead to in even 3 moves, the best players of the game 2048 often employ strategies that don't require you to think in detail about future moves. Often times for agents that don't use MCTS and instead use heuristic based algorithms to play 2048, they often use heuristics based on strategies humans have designed. However in my research, much of the success of these heuristics based algorithms come down to several heuristics taking into account factors no average 2048 player would intrinsically think about. A Monte-Carlo Tree Search of course isn't something an average player could replicate but it does have the benefit of not having to tell the program what are good board states.

For this problem, the agent would make one state change on the board, up, down, left or right, then it would randomly play moves for a certain amount of games until it would lose. This method ends up being very effect when the number of games played surpasses 250 or so. I hypothesize that between simulation increments of 50 I expect to see about a 10% increase in average score although I believe that most of that gain will be in the lower simulation numbers like from 50 to 100 to 150.

2 Literature Review

2.1 Introduction

Much of the differences between agents playing 2048 is the type of heuristics you use or whether you use an algorithm that uses them at all. Often times people will use variations of minimax algorithms. This strategy requires the use of several heuristics to determine what the optimal board state is. To understand why, we'll look at some of those papers and see the cost-benefit for heuristic based algorithms and one without heuristics.

2.2 Monte-Carlo Tree Search vs. Expectimax

One of my primary reasons for using MCTS was because of the work done in Pesquimat [2]. The work done in Pesquimat looks into comparing the MCTS and Expectimax algorithms against one another. For the Expectimax algorithm, their heuristic consisted of 5 different values. The evaluation consisted of getting the maximum tile value in a corner, maximizing empty cells, moving tiles along a snake-like path; monotonically, maximizing tiles values along an edge, and by looking at neighbor tiles with equal values to be merged. Many of these attributes intrinsically make sense but with the moving monotonicity along a snake-like path, is a not so clearly observed benefit. Even after applying these rigorous heuristics, they come to the conclusion that the MCTS is superior in games solved but at the expense of run-time [2].

Another paper also decided to compare MCTS with an Average Depth-Limited search that approximates Expectimax. In RodgersPhilip [3] they find that this use of Averaged Depth Limited Search gave results that were better than MCTS in both run-time and score. From my viewpoint, this paper didn't provide enough clarity for me to decide if their results were valid; often lacking primary sources and detailed explanations.

Because of the results from Pesquimat and that they were already using what I thought of as a robust set of heuristics, I didn't see a reason to attempt to improve on there implementation of Expectimax and instead went with implementing my own MCTS.

2.3 Monte-Carlo Tree Search adaptations and optimizations

Since I'd decided on using MCTS as the algorithm for my agent, I also looked into was that others have adapted it to suit their needs and what became of these changes. Some adaptations and optimizations came from changing things specific to there game. For example in Nanchang they used some ”simple” optimizations like pre-generating the distributions in the card game Bridge [1]. They also discuss more specific optimizations for the card game. After thinking about how I could apply that to my agent, I decided to attempt to reduce the amount of work my simulation would do. My main issue was that the simulation was running abysmally slow compared to want I saw in Pesquimat. So after writing the functions that would create hard-copies of a grid, I was able to improve my run-time by about 5 times the original speed.

As for adaptations to MCTS itself, in CCDC‘[4], they look at the different methods of selecting the best node as their primary way to improve performance. They use 2 methods, one is where they count merges and the number of spaces on the board, and the other is where the sum of the merged numbers is the largest. These differences in heuristics not only affect the results of the agent but also the agents run-time. After briefly looking at different heuristic values that I could apply in a similar way, I decided to not implement them into my final product since I found limited or not improvements to score and run-time.

2.4 Conclusion

When creating an agent for an game there are always multiple ways of going about it. After looking into a few different pieces of literature related to the game 2048 or Monte-Carlo Tree Search, it seemed to me that the ”simple” solution of your typical MCTS is a very effective method for playing 2048. When it might not break any records as it stands, it is quite a bit better than your average player in terms of score reached.

3 Approach

Based on previous research, I decided to implement Monte-Carlo Tree Search to play 2048 for a variety of reasons. First and foremost I was confident I could implement it to an adequate degree in a reasonable time. Although, some of my early attempts were actually using a minimax type algorithm. This first attempt was actually not bad getting an average score of around 7,000 and even reaching a maximum score of around 14,000. But after exhausting my options and reaching diminishing returns I decided to rewrite most of my logic with MCTS as the primary algorithm.

Firstly I needed a game that my agent could interact with, both reading the board state and inputting actions. For this I just cloned the work done by ”hczhcz” on GitHub who had a working standard 2048 game (https://github.com/hczhcz/2048). Since I decided to use a version of the game played in the browser, using JavaScript was the easiest way to implement an agent. Like I mentioned earlier, my first attempt was using minimax to play the game. To get this to work, I created functions that any agent would need to play the game; a function to create a hard-copy of a grid and functions to evaluate board states. After that I added an event listener to the main JavaScript that when you press a certain key the main loop would start. This loop contained the main function for the agent that would return the next best move. None of this functionally needed to change after moving from minimax to MCTS.

The implementation of MCTS for this project is very straight forward. First you have the Selection phase of the algorithm which selects the best node in the tree of grid states. Then you expand that state in the Expansion phase. Then simulate the amount of games first inputted into the function. During the Simulation phase you keep track of the total score earned from each move. Then finally in the backpropagation phase you select the best starting move and propagate back up the tree.

One somewhat major limitation my implementation has is that doing all of the work in the browser doesn't allow my algorithm to take advantage of all the resources on my computer; primarily the CPU. Modern web-browsers do very well at not using so much of your computers resources as to slow down other programs or even other tabs in the same browser. Unfortunately, I was unable to find exact information on the percentage one Google Chrome tab could use although it never seemed to more than around 11% of my total CPU utilization while it used 100% of the tab's CPU resources.

Because of the resource constraint in the browser, if I had to improve run-time I would use a local web-server that could take advantage of my all of my computer's resources. This could easily be done by sending the state to the web-server as a web request then waiting for the response. The main reason for not doing this was because of the changes in the code needed to go from a synchronous environment in the browser to the asynchronous environment in the server; this is assuming I would use some JavaScript based server-side language or library like NodeJS. The hassle of asynchronous JavaScript is too much for me to worry about especially since I'm satisfied by the run-time I get in the browser.

4 Experiment Design and Results

4.1 Design

As I mentioned earlier, the implementation of MCTS for this project is as follows. First you have the Selection phase of the algorithm which selects the best node in the tree of grid states. Then you expand that state in the Expansion phase. Then simulate the amount of games first inputted into the function. During the Simulation phase you keep track of the total score earned from each move. Then finally in the backpropagation phase you select the best starting move and propagate back up the tree.

For the selection phase of the algorithm, a tree containing the states of the board is looked through. The algorithm is looking for the maximum UCB value of a leaf. The UCB value is calculated using this equation:

$$UCB = \frac{wins}{times} + \frac{\sqrt{2 \times \log{(parent.times)}}}{times} \tag{1}$$

For the expansion phase of the algorithm, after finding the node with the highest UCB value in the selection phase, the algorithm then expands on that grid state by getting the next possible moves; up, down, left or right. This is then added into the tree as a child of the node found in the selection phase.

During the simulation phase the 4 possible moves are then simulated a certain amount of times depending on the value inputted into the function. This is somewhat different than other MCTS algorithms because instead of simulating from just one state, the algorithm simulates from up to 4.

During the backpropagation phase the starting move with the highest average final score is the winner so the results is propagated back up the tree.

4.2 Results

The main question I was looking at was how does the number of simulations run affect the performance of the agent. I'm only evaluating score and run-time in my observations since memory is somewhat difficult to measure on the browser. In order to run multiple simulations using various different numbers of simulations, I created a function in the client-side JavaScript that would run the game a specified amount of times. Then after all games are finished and the results have been tallied, a JSON file is downloaded containing the results of the experiment. I did this several times adding and removing different game statistics that I found helpful.

The agent was run for a few different simulation values. The final scores and run-times of the games were then recorded and analysed. You can see a breakdown of the different simulation values in Table 1. As you can see as the number of simulations increases the average score does increase. Early on this number is significant but as the number approaches 250, the increase tapers off until there is no meaningful increase in score. But one factor to also consider is win percentage. Looking at Table 2 we can see that even though the average score of simulation number 250 is higher then simulation number 300, it's overall win-rate (achieving a tile value of 2048) is lower. It seems like this result might have to do with some sort of risk vs. reward difference in play where playing out more simulations overall leads to more games won but when wanting to reach the highest scores more risk is necessary. Although this abnormality in average score and win percentage may just come down to too small a sample size.

Of course another aspect to consider when evaluating the differences in the number of simulations run is the average run-time across games. It's obvious that as more simulations are run then so to does the run-time of the game. This is because of course you simulate more games per move but also to reach higher scores you must play more moves. I feel as though the distribution of the average times are normal which is why not much analysis is required to look at those.

Number of Simulations	Sample Size	Average Score	Average run-time (seconds)
50	160	5637.15	5.71
100	130	7675.82	13.45
150	110	8411.96	21.74
200	235	9522.59	32.91
250	115	10461.29	42.03
300	130	10355.85	49.06

Table 1: Number of simulations and details

Number of Simulations	Sample Size	Win percentage	Number of wins
50	160	0	0
100	130	2.3%	3
150	110	1.8%	2
200	235	2.1%	5
250	115	4.3%	5
300	130	5.4%	6

Table 2: Win percentage

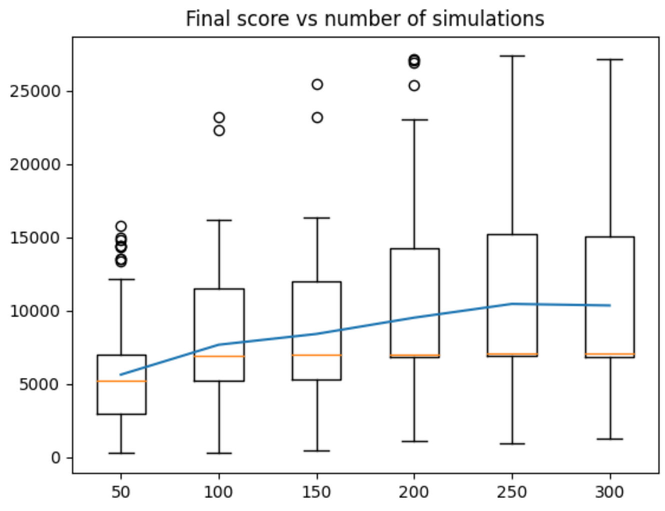

Figure 1: Final scores for different simulation numbers

5 Conclusion

Based on my experimental results, it seems the hypothesis that average score will change around 10% per 50 simulations is correct. The gains in average score between running the 50 and 100 simulations was 36%, between 100 and 150 were 9.6%, between 150 and 200 were 13%, between 200 and 250 were 9.9%, and between 250 and 300 were -1.0%. That averages out to 13.5% and by removing the outliers of 50-to-100 and 250-to-300 we get an average of 10.8%; of course if we were to extrapolate the data out further we'd see a different result. While we did see these gains in the low simulation numbers like 100 to 150, as the number of simulations grows we see diminishing returns until the jump from 250 to 300 where we even see negative gains. This behavior is consistent with the results found in Pesquimat [2] where running 300 simulations actually yielded worse results in both average score and run-time than running 250 simulations. Although one deviation from Pesquimat is that running 250 simulations yielded a better win-rate than running 300 simulations. In my tests, 300 simulations was better for win-rate. I believe that this difference in results is attributed to my smaller sample size although they don't provide their sample size so it's hard to compare.

You can find this project on my GitHub at https://github.com/Omar-Ceesay/2048-AI.

References

[1] C. Chen, H. Qiu, Y. Wang, Y. Chen, and Y. Xiao. Optimization and application of monte carlo sampling in bridge. In *2019 Chinese Control And Decision Conference (CCDC)*, pages 6276–6280, 2019.

[2] E. Noa Yarasca and D. M. K. Nguyen. Comparison of expectimax and monte carlo algorithms in solving the online 2048 game. *Pesquimat*, 21:1, 09 2018.

[3] P. Rodgers and J. Levine. An investigation into 2048 ai strategies. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–2. IEEE, 2014.

[4] J. Tao, G. Wu, Z. Yi, and P. Zeng. Optimization and improvement for the game 2048 based on the mcts algorithm. In *2020 Chinese Control And Decision Conference (CCDC)*, pages 235–239, 2020.