2048 AI: Monte Carlo-Powered Markov Decision Process

Noah Ripstein

August 10, 2023

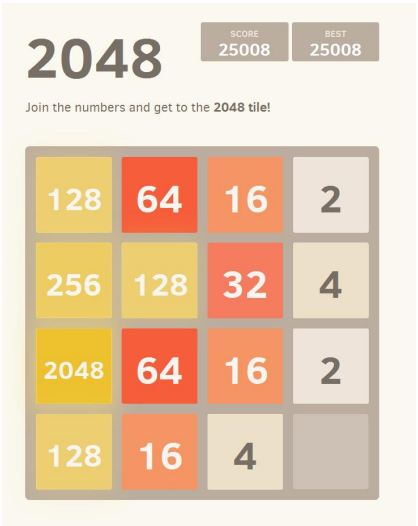
Abstract

A 2048 Python clone was developed and was used to test a series of AI strategies. A strategy which formalizes the game as a Markov Decision Process powered by Monte Carlo simulations significantly outperformed a Pure Monte Carlo Tree Search algorithm. Model search parameters were then optimized to find the pareto efficient solution set which maximises performance and minimizes runtime.

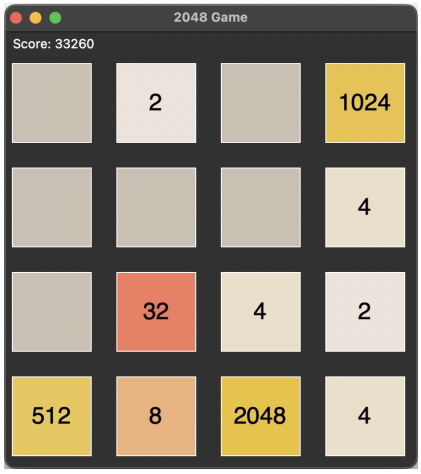
1 Introduction

2048 is an addictive game which was originally released in 2014. The game begins with two randomly placed tiles, each having a value of either 2 or 4, randomly placed on a 4x4 grid. The player can move the tiles in four directions: up, down, left, or right. When a direction is chosen, all tiles on the grid slide as far as possible in that direction, merging with any adjacent tiles of the same value to form a new tile with double the value. The value of the new tile is added to the score. After the player's turn, a new tile spawns in a random location; this new tile has a 90% chance of being a 2, and a 10% chance of being a 4. The game ends when the board is filled with tiles and the player has no legal moves. The goal of the game is to combine tiles until the 2048 tile is reached (although it is possible to continue playing after winning).

A Pure Monte Carlo Tree Search was first implemented, and then improved upon by formalizing part of the game's decision tree as a Markov Decision Process. The Markov Decision Processs achieves the 2048 tile 97% of the time and the 4096 tile 58% of the time.



(a) Original game



(b) My implementation

Figure 1: Screenshots from original game and my implementation

2 The Decision Tree

One of the challenges of creating an AI which plays 2048 is the sheer number of possible games. Figure 1 represents the possible board positions after the player makes only one move. If there are 8 free spaces on the board, for example, then there are 64 possible game states after the player's move (assuming each of left, right, up and down are legal moves which do not combine any tiles). In general, there are $2(|S|)(m) + c$ possible states after the player's move, where $|S|$ is the number of legal moves, m is the number of empty spaces on the board after tiles have been combined from the player's turn, and c is the number of tiles which get merged as a result of the player's move.

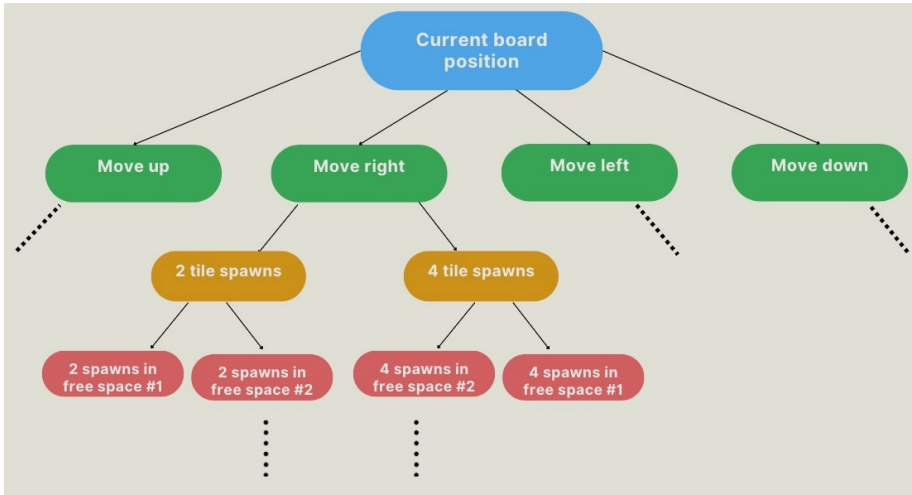


Figure 2: Decision tree representing possible board states after each move

3 AI Designs

3.1 Pure Monte Carlo Tree Search

The initial algorithm employed is a Pure Monte Carlo Tree Search (Algorithm 1). This algorithm takes the current game position and the desired number of simulations per direction (n) as inputs. It explores all legal next moves from the current position by simulating n games for each potential move. The scores from the end of these simulated games are then averaged to determine the desirability of each move. The direction with the highest average score is selected:

$$\text{Selected Move} = \underset{move}{\operatorname{argmax}}\, \text{PMCTS}(move) \tag{1}$$

This approach initiates from the green nodes in the game tree diagram (Fig. 2). From there, the algorithm proceeds through random exploration to reach the red child nodes, representing the spawning of a 2 or 4 tile in each possible location.

While this approach provides a comprehensive exploration of the game tree, it has significant limitations. The primary concern lies in the random nature of the search process. As a consequence, some of the simulated games performed during the Monte Carlo simulations may yield exceptionally poor results that are highly unlikely to occur in actual gameplay. This lead me to want to discard a portion of those simulated games with particularly poor scores from consideration.

Simply modifying Algorithm 1 to calculate the average score for a given move using only top-performing of simulated games would not adequately address this

source of randomness, however. There are two sources of randomness inherent in the Pure Monte Carlo Tree Search: randomness associated with game-play (which I aim to reduce), and randomness of tile spawns. Discarding randomly played games with low scores in an attempt to address the former source of randomness might prevent the AI from evaluating branches of the tree which involve unlucky tiles spawning after the next turn.

Algorithm 1 Pure Monte Carlo Tree Search

```
function PMCTS(currentPosition, n)
    directionScores ← list()
    for nextMove ∈ legalMoves(currentPosition) do
        nextMoveScores ← list()
        for gameNumber ← 1 to n do
            result ← playRandomGame(currentPosition, nextMove)
            nextMoveScores.append(result)
        end for
        averageScore ← mean(nextMoveScores)
        directionScores.append(averageScore)
    end for
    return directionScores
end function
```

3.2 Markov Decision Process

The pitfalls of the Pure Monte Carlo Tree Search raised in section 3.1 can be circumvented by formalizing the game structure as a Markov Decision Process (MDP). The core functionality of the MCTS is used as part of the MDP, but the evaluations made with MCTS are made more beneficial by explicitly maximizing expected value. Where the PMCTS algorithm begins Monte Carlo simulations from the board state after a player's move (green nodes in Figure 2), The MDP begins Monte Carlo simulations from each possible tile spawn in response to a player's move (red nodes in Figure 2). It is possible to model the game as a Markov decision process because the game satifies the Markov property: the probability of reaching a future state depends only on the current state, not on previous states:

$$P(s|s_{n-1}, s_{n-2}, ..., s_0) = P(s|s_{n-1})$$

In general, an is MDP is characterized as follows:

1. S: The set of states (board position and score) which could arise after the AI's next move.

2. A: The set of actions which the AI could legally take from the current position.

3. $P(s'|s, a)$: Transition probability of reaching state s' given current state $s \in S$ after taking action $a \in A$.

4. $V(s)$: The value function which determines the expected future reward associated with entering a state $s \in S$

5. $\pi(s)$: The policy function which uses the other functions to strategically pick an action $a \in A$ given any board state.

6. $R(s)$: The immediate reward associated with taking action $a \in A$ which leads to state $s \in S$ is a part of many MDP designs. Justification for why this function was excluded can be found in section 3.3.

A can be constructed by checking which of left, right, up or down are legal moves. S can be constructed by placing a 2 and then a 4 on each empty tile for each $a \in A$ (Algorithm 2). The value function, $V(s, \vec{\theta})$, (Algorithm 3) performs a Monte Carlo tree search, with the number of simulations determined by the parameter vector $\vec{\theta}$. $\vec{\theta}$ contains the following parameters:

1. Number of random searches for 2 spawning with 1-3 empty tiles.

2. Number of random searches for 2 spawning with 4-6 empty tiles.

3. Number of random searches for 2 spawning with 7-9 empty tiles.

4. Number of random searches for 2 spawning with 10-15 empty tiles.

5. How many times more searches 2 spawns should get compared to 4 spawns.

6. Top proportion of best performing moves to include for score evaluation.

$\theta_1, \theta_2, \theta_3$ are scaled so their respective empty tile ranges have the same total number of simulations. θ_4 uses the same number of simulations for each of 10-15 empty tiles. If $\theta_1 = 500$, for example, the Monte Carlo simulation will run 500 times when there is one tile, 250 times when there are two, and 125 when there are three; if $\theta_4 = 20$, then the Monte Carlo simulation will run 20 times for any number of empty tiles $\in [10, 15s]$.

$\vec{\theta}$ remains constant throughout a single play-through of the game. As will be discussed in Section 5.1, optimizing $\vec{\theta}$ became a primary direction of inquiry. The policy function, which determines what action $a \in A$ to take, $\pi(s)$, is given by Equation 2:

$$\pi(s) = \underset{a}{\operatorname{argmax}} \left(\sum_{s' \in S} P(s'|s, a) V(s', \vec{\theta}) \right) \tag{2}$$

Notice that $\pi(s)$ picks the action with the highest expected value, where the value associated with reaching a state is given by a Monte Carlo tree search. Compared to the Pure Monte Carlo Tree Search, the MDP facilitates more sophisticated inference in two ways.

1. By discarding a portion of explored games with poor results at each node, the impact of the Monte Carlo search playing out extremely poor moves due to chance can be mitigated. Crucially, simulations with lower scores

due to "unlucky" tile spawns in the subsequent turn are not eliminated, ensuring a more comprehensive exploration of potential game outcomes.

2. Unlike the Pure Monte Carlo Tree Search, where game states in which a 2 spawns after the players turn are explored 9x as frequently as those in which a 4 spawns, the number of Monte Carlo searches on each node type can be made independent. This can guarantee that all nodes are explored at least 3 times, which gives far more information than a node being explored 1 time, but does not significantly increase runtime.

The MDP is implemented in a manner which makes customizable the number of Monte Carlo searches per node ($\theta_1 - \theta_5$) and proportion of top-performing simulated scores to keep (θ_6). Each of these parameters impact the reported score associated with a direction, and therefore the move selected by $\pi(s)$.

Algorithm 2 Generation of possible next states S

```
function getStates(A)
    S = ∅
    for a ∈ A do
        c_s ← score associated with taking action a
        for tileNum ∈ {2, 4} do
            for tile in emptyTiles do
                place tileNum on tile
                Add s ← (board, c_s) to S
                remove tileNum from tile
            end for
        end for
    end for
    Return S
end function
```

3.3 The Problem with Bellman's Equation

A common approach in MDP applications is to use Bellman's Equation to evaluate the utility of taking action a. Rather than the utility of entering state s' being given by $V(s')$, Bellman's equation would hold that the utility would be given by $R(s,a)+\gamma V(s')$. Here, $R(s,a)$ is the immediate guaranteed value associated with taking action a when in state s, and $\gamma \in [0,1]$ is a discount rate for expected future rewards. In principle, and in many other applications of MDP, this approach outperforms simply using $V(s')$, because it adheres to the principle that immediate, certain rewards are more favourable than uncertain future rewards. Equation 3 shows how Bellman's Equation could be incorporated into the policy function, $\pi(s)$.

Despite the theoretical backing, using Equation 3 as the policy function decreased performance compared to Equation 2 (tested with $\gamma = 0.9$) and brought

Algorithm 3 Value Function: $V(s, \vec{\theta})$

```
function V(s, θ⃗)
    resultList ← empty list
    for i ← 1 to θ_num_sims do                ▷ θ_num_sims is a model parameter
        result ← RandomGame(s)                ▷ Call the RandomGame function
        add result to resultList              ▷ Record the result
    end for
    sort resultList in ascending order
    proportion ← θ_proportion × length(resultList)
    topResults ← resultList[1 : round(proportion)]   ▷ Select the top results
    return topResults
end function

function RandomGame(s)
    while game not over do
        Pick random a ∈ A
        Make move a
    end while
    return game score
end function
```

performance roughly in line with the less computationally intense Pure Monte Carlo Tree Search. This is likely because the random games explored using the Monte Carlo value function tend to end with a score only marginally higher than the current score. This leads immediate score gain, $R(s, a)$, to have a much larger impact on the policy than future scores in Equation 3. This is problematic because it makes the algorithm more "greedy," and prioritizes immediate rewards too much over future rewards (even with high values of γ).

$$\pi(s) = \underset{a}{\operatorname{argmax}} \left(\sum_{s' \in S} P(s'|s, a)(R(s, a) + \gamma V(s', \vec{\theta})) \right) \tag{3}$$

4 Results (figures to be added soon)

Table 1: Results of Different Models

	2048	4096	8192	Avg. Score
Pure MCTS	77%	10%	0%	31,011
MDP (top 100%)	97%	58%	0%	53,232
MDP (top 75%)	94%	58%	1%	53,565
MDP (top 50%)	94%	57%	2%	53,282
MDP (top 25%)	93%	58%	5%	55,966

5 Future Directions

5.1 Parameter Optimization

This is an ongoing project, and the next thing I hope to implement is the optimization of $\vec{\theta}$, the parameter vector which controls how many random Monte Carlo searches to perform for a given number of empty tiles. The goal is to find the optimal tradeoff of time vs score. Below is a rough outline of the plan:

1. Use a Quasi-Monte Carlo sampling technique with low discrepancy (such as Latin Hypercube Sampling or Sobol sampling) to draw a series of samples within the parameter space.

2. Repeatedly use Bayesian Optimization to determine which sets of parameters to evaluate next. (Implementation details such as Acquisition function not yet sorted out)

I intend to optimize for both move speed and score, and find the pareto efficient solution set which lets me choose the optimal speed and average score of the final model. Bayesian optimization is likely a strong framework for parameter optimization because it calls the target function (which can be non-differentiable) fewer times than other methods like evolutionary algorithms.

Model score is most strongly influenced by the proportion of time the 4096 tile is reached. Many trials (with a wide range of parameters) which do not reach the 4096 tile achieve a score around 36,000, when they are near the 4096 tile. AI strategies are likely to fail shortly before reaching the next milestone tile because this is when there are the most large tiles on the board which can not yet be joined. Once reaching the 4096 tile, it is uncommon for a score below 60,000 to be reached, although there is a meaningful distribution between 60,000 and 80,000, where 80,000 represents the point near achieving a 8192 tile. This requires that a given set of parameters is evaluated multiple times on the model before reporting performance. In order to illustrate this, consider the following example:

Suppose that a set of parameters $\vec{\theta_1}$ is stronger than the pre-optimized parameters, and has a true probability of reaching the 4096 tile 60% of the time, with the model scoring 35,000 when it doesn't reach 4096, and 70,000 when it does. This means that a score of $56,000 = 0.6(70,000) + 0.4(35,000)$ should be reported as the long term expected performance of $\vec{\theta_1}$. If $\vec{\theta_1}$ is tested 3 times, the probability of 0.375 that the AI would report a score of 47,333 by reaching 4096 only once in the three runs. With 9 repeats, the probability of 47,333 or a worse score being reported (reaching 4096 3 times or fewer of 9) is 0.099. Running the parameters 9 times drastically reduces the noisiness of the data, which enables the Bayesian optimization model to make more accurate predictions about how future sets of parameters will perform.

Contingency plan for parameter optimization:

If Bayesian optimization does not produce satisfactory results, I will use parameter performance data it generates to train a neural network, which will serve as a surrogate model (by Universal approximation theorem). The surrogate model will serve as a cheap-to-run approximation of the performance metrics obtained by a set of parameters. I can find the minimum of the surrogate model using an evolutionary algorithm, and then evaluate those parameters on the true model. Obtaining new data this way and retraining the model would ideally converge to a global minimum after repeating the process.

5.2 Dynamic Policy Switching

It is possible that when there are few tiles on the board, and explicit exploration of greater depth would be beneficial. Dynamically switching to an Expectimax algorithm seems like it may have strong results here. Expectimax uses Equation 3 with a recursive value function given by Equation 4.

$$V(s, \vec{\theta}) = \sum_{s' \in S} P(s'|s, a)(R(s, s') + \gamma V(s', \vec{\theta})) \tag{4}$$

Equation 4 would evaluate to a certain recursive depth, and then call Algorithm 3 as the final evaulation of $V(s, \vec{\theta})$ to estimate the value at the final search state. This strategy would be particularly useful when there are few open tiles on the board because the low number of states to search increases the maximum search depth in a given amount of time.