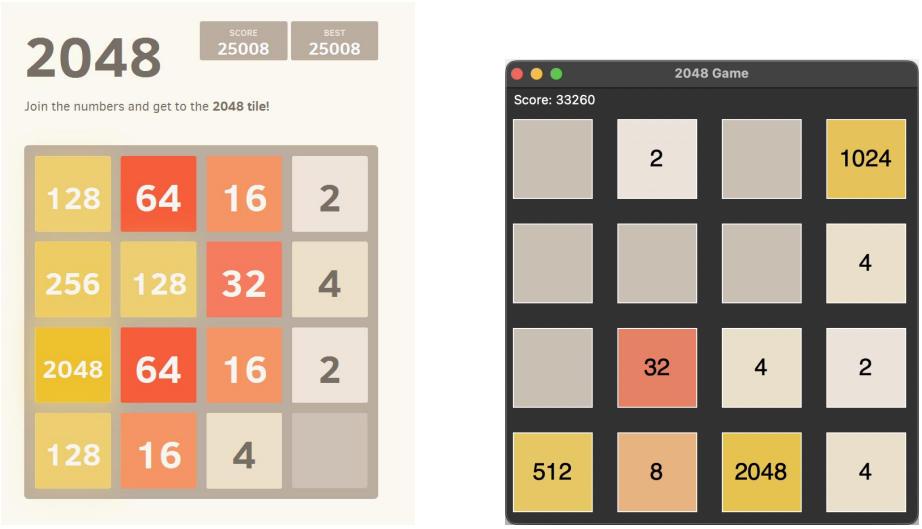
2048 AI: Probabilistic Monte Carlo Tree Search

Noah Ripstein

June 2023

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. TALK HERE ABOUT WHAT I DID BRIEFLY



(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 $2(l)(m)$ possible states after the player's move, where l is the number of legal moves, and m is the number of empty spaces on the board after tiles have been combined from the player's turn.

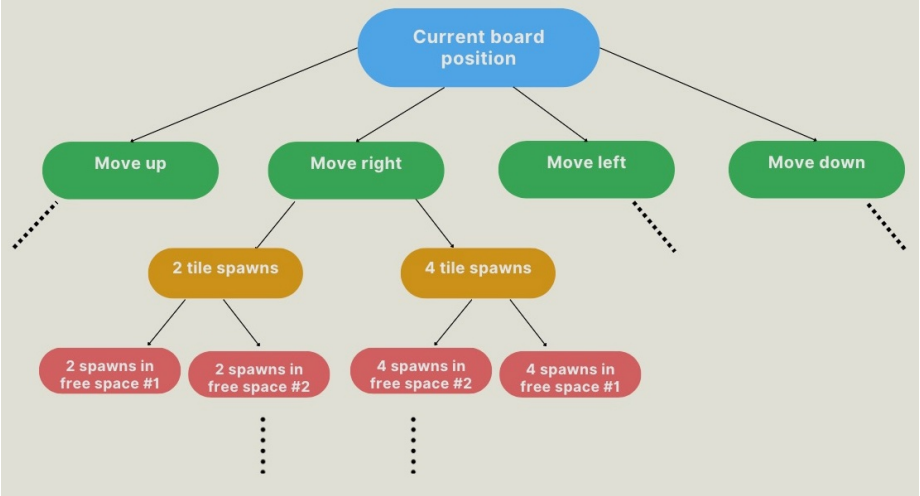


Figure 2: Decision tree representing possible board states after each move

3 Monte Carlo Tree Exploration: AI Designs

3.1 Pure Monte Carlo Tree Search

The initial algorithm I 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:

$$\text{Selected Move} = \underset{move}{\operatorname{argmax}}\, \text{MCTS1}(move) \tag{1}$$

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 yellow and 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 we 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

```
1:  function MCTS1(currentPosition, n)
2:      directionScores ← list()
3:      for nextMove ∈ legalMoves(currentPosition) do
4:          nextMoveScores ← list()
5:          for gameNumber ← 1 to n do
6:              result ← playRandomGame(currentPosition, nextMove)
7:              nextMoveScores.append(result)
8:          end for
9:          averageScore ← mean(nextMoveScores)
10:         directionScores.append(averageScore)
11:     end for
12:     return directionScores
13: 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). 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 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 return.

$\theta_1, \theta_2, \theta_3$ are scaled so they have the same total number of simulations, and θ_4 uses the same value for each of 10-15. $\vec{\theta}$ remains constant throughout a single play-through of the game. As will be discussed in SECTION NUMBER, 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 in Equation The policy function is given in 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 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
```

4 Results

Have chart of % reached 2048, 4096, etc, and avg score for each model. probs need each of them at 100 runs.

can have some sort of Bayesian BDF bc can model the proportion of time it reaches 2048 as berneulli so can do classic PDF.

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

function VALUE($s, \vec{\theta}$)
  $resultList \leftarrow$ empty list
  for $i \leftarrow 1$ to $\theta_{\text{num\_sims}}$ **do** ▷ $\theta_{\text{num\_sims}}$ depends on empty tiles
    $result \leftarrow$ RANDOMGAME(s) ▷ Call the RandomGame function
    add $result$ to $resultList$ ▷ Record the result
  end for
  sort $resultList$ in ascending order
  $proportion \leftarrow \theta_{\text{proportion}} \times \text{length}(resultList)$
  $topResults \leftarrow resultList[1 : \text{round}(proportion)]$ ▷ Select the top results
  return $topResults$
end function

function RANDOMGAME(s)
  while game not over **do**
    Pick random $a \in A$
    Make move a
  end while
  return game score
end function

Table 1: Results of Different Models

	2048	4096	8192	Avg. Score
Pure MCTS				
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 (not done)

MAIN IDEAS FOR FUTURE DIRECTIONS:
- optimize $\vec{\theta}$ using surrogate modelling:

1. Evaluate model with a series of parameters created by latin hypercube sampling
2. Design cost function which should be minimized to have "optimal performance". Will need to trade off speed and performance. probably ideally will have on average 2 seconds per move and reward higher scores.
3. EITHER Train some sort of machine learning model to approximate the underlying function, and then minimize that function's cost
4. OR use Bayesian optimization so I can get a better sense of uncertainty.

Use heuristics

Note that part of the challenge here was not to use heuristics https://arxiv.org/abs/2110.10374 These stanford math profs made a deep reinforcement learning model that doesnt use heuristics and mine is better than it I think (theirs seems to be for a class they taught rather than research really). Check performance to be sure.

dynamically switch to minimax or expectimax when there are few open tiles.

6 FORMALIZING AS MDP (notes to self)

The components of a Markov Decision process are:

1. States: A set of possible states (board positions)
2. Tansition model: $P(s'|s,a)$ The probability of reaching state s' given that we perform action a while in state s.
3. Action: $A(s)$ set of possible actions in a state (what's legal of up, down, left right)
4. Value function: $V(s)$ normally recursively defined with bellman equations, gives the value associated with a given state.
5. Policy: function takes in a state, returns an action a. can be given by

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

   MC12 uses: $\pi(s) = \underset{a}{\operatorname{argmax}} \left(\sum_{s' \in S} P(s'|s,a) V(s') \right)$

   $R(s,a)$ is the immediate score we get if we perform action a while in state s. γ is discount rate for future rewards (follows principle that certain reward now better than uncertain reward later) $V(s')$ is the value of reaching state s'. Done with monte carlo tree search.