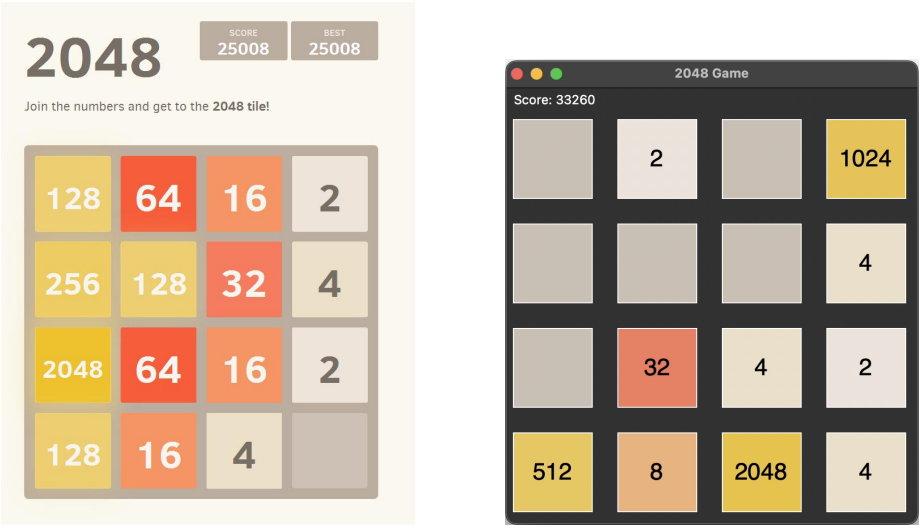
2048 AI: Probabilistic Monte Carlo Tree Search

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.



(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.

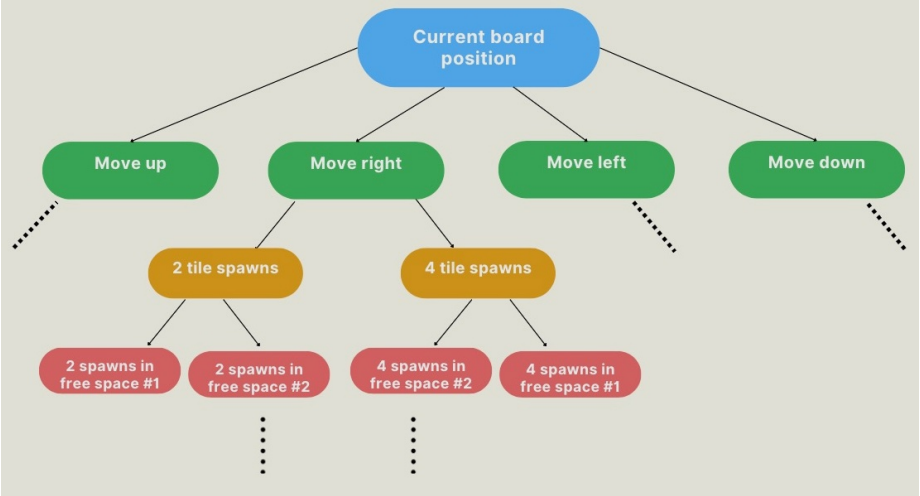


Figure 2: Decision tree representing possible board states after each move

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 Probabilistic Monte Carlo Tree Search

Algorithm 2 aims to address the concerns raised about Algorithm 1 in section 3.1. Where Algorithm 1 begins Monte Carlo simulations from the board position after a player's move (green nodes in Figure 2), Algorithm 2 begins Monte Carlo simulations from each possible tile spawn in response to a player's move (red nodes in Figure 2).

The Probabilistic Monte Carlo Tree Search algorithm maximizes the expected value of a move (Equations 2 and 3). m represents the number of empty tiles, R_{2i} and R_{4i} represent the average of the reported scores assuming a 2 or 4 tile spawn in empty location i, respectively.

$$E[move] = \frac{1}{m} \sum_{i=1}^{m} \left(\frac{9}{10} R_{2i} + \frac{1}{10} R_{4i} \right) \tag{2}$$

$$\text{Selected Move} = \underset{move}{\operatorname{argmax}}\, E[move] \tag{3}$$

Compared to the Pure Monte Carlo Tree Search, the probabilistic approach facilitates more sophisticated inference in two ways.

1. A portion of the games explored from a particular node with particularly poor performance can be discarded. This will increase the projected score associated with a node by reducing the impact of "unlucky" random moves, but will not discard simulations with low scores because of "unlucky" tiles spawning on the next turn. NEEDS TO BE REWORDED

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.

Algorithm 2 implements Equation 2 and Equation 3 in a manner which makes customizable the number of Monte Carlo searches per node (nodeSims) and proportion of top-performing simulated scores to keep (bestProportion). Both of these parameters impact the reported score associated with a direction, R_{ni}, in Equation 2. nodeSims is a dictionary which maps the number of empty tiles on the board to the number of Monte Carlo simulations to be performed, with different values for 2 spawns and 4 spawns. The expectedValue function referenced in Algorithm 2 is given by Equation 2.

Algorithm 2 Probabilistic Monte Carlo Tree Search

```
function MCTS2(currentPosition, nodeSims, bestProportion)
    directionScores ← list()
    for nextMove ∈ legalMoves(currentPosition) do
        node2Scores ← list()
        node4Scores ← list()
        for childBoard ∈ possibleTileSpawns(nextMove) do
            childBoardScores ← list()
            for gameNumber ← 1 to nodeSims[childBoard] do
                result ← playRandomChildGame(currentPosition, nextMove, childBoard)
                childBoardScores.append(result)
            end for
            if childBoard has 4 tile then
                node4Scores.append(childBoardScores)
            else
                node2Scores.append(childBoardScores)
            end if
        end for
        node2Scores ← sort(node2Scores, descending=True)
        node4Scores ← sort(node2Scores, descending=True)
        topNode2Scores ← node2Scores[0:len(node2Scores)*bestProportion]
        topNode4Scores ← node2Scores[0:len(node4Scores)*bestProportion]
        directionScore ← expectedValue(topNode2Scores, topNode4Scores)
        directionScores.append(topNextMoveScores)
    end for
    return directionScores
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.

Table 1: Results of Different Models

	2048 [%]	4096 [%]	8192 [%]	16384 [%]	Avg. Score
Pure MCTS					
Proba. MCTS (top 100%)					
Proba. MCTS (top 75%)					
Proba. MCTS (top 50%)					
Proba. MCTS (top 25%)					

5 Future Directions

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.