Solving 2048 as a Markov Decision Process

Tianyi Liu
Department of Chemistry
Stanford University
Stanford, CA 94305
`tliu26@stanford.edu`

Abstract

The game 2048 is a single-player board game in which in each move the player combines pairs of tiles of the same number into tiles of a bigger number, in order to reach the end goal of obtaining a tile of 2048. The game is a Markov Decision Process (MDP). In this paper I describe two algorithms for MDP to play the game, both of which have achieved 2048 or higher number. I then discuss and compare the performance between these two approaches.

1 Introduction

The game is played on a 4×4 board where on each cell there could be one or zero tile. There are four moves available for the player: up, down, left and right. When a certain direction is entered, all tiles on the board will be moved in that direction for as far as they can be. For example, if the player entered *right*, each tile will be moved horizontally to the right until it reaches the boundary or is blocked by another tile to the right of it. If two tiles of the same number are pushed to adjacent cells in a move, they will combine into a new tile with twice the number of the original tiles. After each move by the player, a new tile is spawned on an empty cell. The new tile has 0.9 probability of being 2, and 0.1 probability of being 4. The game ends when no more moves are allowed. This happens when all the cells are occupied by tiles, and no pairs of tiles with the same number are adjacent. The goal of the player is to move and combine tiles to get a tile with 2048, before the game ends. This game is difficult to play for two main reasons. First, it is impossible to predict future states, as the location of the new tile spawned is random, and so is the number on the tile. Second, the board will become increasingly more crowded as more tiles are added to the board, making it difficult to move tiles around.

Since the position to add a tile after each move is random, and the tile could be either 2 or 4 according to some probability, the state of the board evolves probabilistically. Therefore this game is a Markov Decision Process (MDP). There are a wide variety of methods for solving the optimal policy for MDPs, which can be categorized into either online or offline methods. Offline methods are not well suited for the game of 2048 due to the extremely large state space. Recall that the board is 4×4, and that each tile could be any number that is a power of 2: 2, 4, 8, etc. Even if we only restrict to states with only tiles less than or equal to 2048, there are 11 possible numbers on the tiles. Therefore a cell has 12 states (unoccupied or occupied by tiles with 11 possibilities). With 16 such cells, the total number of states of the board will then be 16^{12}. Offline methods for MDPs, such as value iterations, involve calculating the value functions of all the states. This is obviously infeasible for 2048, given the computing power that we could access with today's technology. However, even if the computing resources required for offline methods are available, it is not necessary to know about the entire state space to play the game. As will be discussed later on, the number of steps it takes to generate a tile of 2048 is about 1000, significantly smaller than the state space. Also consider the states where a tile of 2048 is present. This tile can be anywhere on the board, and all the other cells on the board can have

AA228: Decision Making Under Uncertainty, Stanford University (Fall 2019), Stanford, CA(LateX template borrowed from NeurIPS 2019.).

several possibilities of states. The number of such winning states is therefore also very large. In short, the number of "routes" (sequence of states) to a winning state and the number of possible winning states are both very big. One run of the game only involves a small portion of the state space, and we do not need all the information about all the other states to win the game.

After the above analysis, it is clear that online methods are more reasonable for this game, which do not requires computation of the entire state space, but only states that are reachable from the states visited in a game play. This report will focus on two types of online methods, both of which could win the game.

2 Related work

Several solvers for 2048 in the framework of MDPs have been developed, including both online and offline methods. Robert Xiao designed an expectimax algorithm [1]. The algorithm calculates the utilities of state/action pairs by adding the immediate reward (discussed in detail in the Methods section) and the maximum utilities of all the possible next states that are obtained by recursively calling the function, weighted by the transition probabilities. This is in fact the *Algorithm 4.6 Forward search* in the book *Decision Making under Uncertainty* (DMU) [2]. The algorithm achieved a 100% win rate. A similar approach called minimax search was also used. Instead of a weighted average of all possible next states, the utility of the worst possible next state is added to the immediate reward. This algorithm was used by Stack Overflow user ovolve [3], and achieved 90% win rate. There are also other computationally less expensive online methods such as Pure Monte Carlo Search by Stack Overflow user Ronenz [4], as well as tree search by Ho, et al. [5]. While the above algorithms are all online planning, where decision is being made by gathering information from nearby states, there are also algorithms that involves computing the state/action value functions. Szubert, et al. used several temporal difference learning algorithms to play 2048 [6]. In these algorithms, which included Q-learning, the agent learns the state/action value functions as it plays the game. As the game is being played, the value functions get updated according to the next states resulting from the actions taken, and hence the name temporal different learning. Due to the large state space, it is difficult to implement direct look-up tables as the value functions. The authors therefore used *N-tuple Network* as a function approximator. The algorithm achieved superior performance where the win rate is 100% for numbers that are much bigger than 2048.

3 Methods

We discuss two online algorithms for solving the 2048 game as an MDP.

3.1 Sparse sampling

This algorithm follows closely with *Algorithm 4.8* in DMU. The algorithm requires a generative model that outputs the next state and immediate reward given the current state and action. Since the transition model is known, namely the probability of spawning what number on which cell, we could implement the game and use that as the generative model. The immediate reward of a move is the sum of the numbers on tiles that are generated by combining tiles in this move, and is 0 if the move does not result in any merging. Starting with the initial state, the agent enumerates through all the allowed actions from that state and calculates the immediate rewards. It then uses the generative model to evolve the state with the available actions to get a new next state, and calls the function recursively to evaluate on these next states. Note that since the game is stochastic, the next state is not deterministic, and we can evolve several copies (n samples) of the original state, and take the average of the values of these different next states, in order to increase the accuracy. The value of a state/action pair would be the immediate reward plus the (averaged) optimal value of the next states corresponding to this state/action pair, calculated recursively. The recursion continues to some depth d, and returns 0 when it reaches the end. The agent then chooses the action that gives the best value for the current state. This process then continues until the game is over. The algorithm is outlined in Algorithm 1, taken from DMU [2].

Algorithm 1: Sparse sampling

```
Function SelectAction(s, d):
    if $d = 0$ then
        return ($\text{NIL}, -\infty$)
    end
    $(a^*, v^*) \leftarrow (\texttt{NIL}, -\infty)$
    for $a \in A(s)$ do
        $v \leftarrow 0$
        for $i \leftarrow 1$ to $n$ do
            $(s', r) \leftarrow G(a, s)$
            $(a', v') \leftarrow$ SelectAction(s', d-1)
            $v \leftarrow v + (r + \gamma v')/n$
        end
        if $v > v^*$ then
            $(a^*, v^*) \leftarrow (a, v)$
        end
    end
    return $(a^*, v^*)$
```

3.2 Pure Monte Carlo tree search

Different from sparse sampling, this algorithm does not calculate the immediate reward. Instead, it is only concerned with the end state. Given a state, the algorithm goes through all available actions. It uses the generative model to evolve the state with an action, and plays the rest of the game with a *random* policy. Once it reaches the end of the game, i.e. no moves are allowed, the end score is calculated. The score can be the sum of the numbers on all the merged tiles throughout the game play, as in sparse sampling, the largest number on the board, or some other quantity that reflects the quality of the end state. The agent then chooses the action that gives the best score of the end state. This algorithm has been used by Ronenz to achieve 2048 [4], though the performance is worse than other algorithm discussed above when it comes to even higher numbers. The algorithm is outline in Algorithm 2.

Algorithm 2: Pure Monte Carlo Tree Search

```
Function SelectAction(s, n):
    $(a^*, v^*) \leftarrow (\texttt{NIL}, -\infty)$
    for $a \in A(s)$ do
        $v \leftarrow 0$
        for $i \leftarrow 1$ to $n$ do
            $s' \leftarrow G(a, s)$
            $v \leftarrow v + \texttt{RollOut}(s')/n$
        end
        if $v > v^*$ then
            $v^* \leftarrow v$
            $a^* \leftarrow a$
        end
    end
    return $a$
Function RollOut (s):
    while CanMove(s) do
        $a \leftarrow$ RandomChoice(A(s))
        $s \leftarrow G(a, s)$
    end
    return CaclculateScore(s)
```

Table 1: Sparse sampling for various depths

Depth	Avg score	Avg max tile	% 1024	% 2048
1	2030.2	186.9	0	0
2	7065.2	563.2	20	0
3	11339.9	848.6	53	5
4	14637.1	1052.2	63	15

Table 2: Sparse sampling for various number of samples

Number of samples	Avg score	Avg max tile	% 1024	% 2048
1	7022.7	549.1	20	0
2	7280.2	559.4	19	0
3	7776.3	609.3	27	0
4	7590.3	586.2	23	0

4 Experiments

The performance of the algorithms depends on the parameters used. For sparse sampling, various values for the depth d and number of samples n are used to play the game and characterize the effectiveness of the algorithm. Similarly for pure Monte Carlo tree search, the number of samples n and the function used to calculate the score of the end states can be changed to see how they affect the performance.

5 Results and discussions

5.1 Sparse sampling

Table 1 shows the performance of sparse sampling for various depths with $n = 1$. The algorithm was run at each depth for 100 times. The metrics used for evaluating the performance are average final score, average maximum tile achieved, percent chance of getting 1024 and percent change of getting 2048. All of these quantities scale roughly linearly with depth. As mentioned in Introduction, a major difficulty of the game is that the board gets increasingly crowded as more tiles are added after each move. Increasing the depth effectively enables greater foresight of the game state, and helps the algorithm avoid making moves that results in low rewards in the future. At a small depth such as 1 or 2, the algorithm could only make about several hundred of moves before the game ends. When we go to a larger depth such as 3 or 4, the algorithm could take more than one thousand moves, which makes achieving 2048 possible.

Table 2 shows the performance for various number of samples n, with $d = 2$. The performance seems to increase by a little bit when the number of samples goes from 1 to 3, as the average score and maximum tile goes up. However, this does not help in winning 2048 in particular, as the number of times it achieved 2048 remained at zero. Interestingly, the performance worsened when number of samples was increased to 4. This shows that the dependence of performance on the number of samples is relatively week and unpredictable.

From these experiments it is clear that the search depth is an important parameter in sparse sampling algorithm for 2048, while number of samples is not.

5.2 Pure Monte Carlo tree search

Monte Carlo search requires large sample size (the n parameter in Algorithm 2) to be effective. The algorithm was run with several choices of n and the results are summarized in Table 3. Due to limitations in computational resources, only one simulation of each sample size was run. The 4th and 5th columns then correspond to whether 1024 and 2048 were achieved, instead of the percent chance. The table shows that increasing the number of samples greatly increases the performance, and can even achieve a 4096 tile. However, this method takes a long time to run.

Table 3: Pure Monte Carlo tree search for various numbers of samples

Number of Samples	Final score	Max tile	1024	2048	4096
10	12160	1024	Yes	No	No
20	23260	2048	Yes	Yes	No
30	27388	2048	Yes	Yes	No
40	35684	2048	Yes	Yes	No
50	69664	4096	Yes	Yes	Yes

Table 4: Pure Monte Carlo tree search for various choices of v

Choice of v	Final score	Max tile
End state score	12160	1024
End state maximum tile	2412	256
End state sum of tiles	16448	1024

In selecting the optimal action to take from a state, the algorithm needs to play the game starting from the next state using a roll-out policy (see Algorithm 2). The final score of that game play is used to determine the quality of the action (the v corresponding to the action in line 7 of Algorithm 2). However, the quantity need not be the score of the end state. It could be the maximum tile number of the final state, or the total number of all the tiles. Table 4 shows how the choice of the quantity can affect the performance of the algorithm. The choice of v does indeed affect the performance. Using just the maximum tile in the end state as the value for the state/action pair gives rise to far worse final score than other choices of the value function. This makes sense, because the quality of a state of the board does not only depend on the maximum tile, but also what and how many other tiles are present. This also shows that the usual definition of the value, i.e. the sum of merged tiles, is a good choice, and accurately reflects the quality of a state.

5.3 Comparison

Both sparse sampling and pure Monte Carlo tree search were able to achieve 2048. However, sparse sampling is far less computationally expensive than the latter. Implemented in Python, the run time for a sparse sampling with $d = 4$ is about 2 minutes, and it has 15% chance of winning. Monte Carlo, on the other hand, requires about 20 samples in each search to win the game, and the run time is about half an hour. Monte Carlo is more time consuming because for each state/action pair, it needs to play the game with random policy until the game is over, which resembles a large depth compared to $d = 4$ in sparse sampling.

6 Conclusion and future work

We used two methods to develop an agent to play the game of 2048: sparse sampling and pure Monte Carlo tree search, both of which succeeded in getting tiles with 2048. Sparse sampling, with a finite depth for search, is computationally less expensive than Monte Carlo. The depth of sparse sampling is important for its performance, while the number of samples is not. For pure Monte Carlo, the number of samples is crucial. Future work can be focused on exploring more algorithms, such as expectimax, temporal difference learning with n-tuple network, etc.

References

[1] Robert Xiao. 2048-ai, 2014. `https://github.com/nneonneo/2048-ai`.

[2] Mykel J Kochenderfer. *Decision making under uncertainty: theory and application*. MIT press, 2015.

[3] ovolve. 2048-ai, 2014. `https://github.com/ovolve/2048-AI`.

[4] Ronenz. 2048-ai, 2014. `https://github.com/ronzil/2048-AI`.

[5] Ho Shing Hin, Wong Ngo Yin, and Lam Ka Wing. Using artificial intelligent to solve the game of 2048, 2017. `https://home.cse.ust.hk/~yqsong/teaching/comp3211/projects/2017Fall/G11.pdf`.

[6] Marcin Szubert and Wojciech Jaśkowski. Temporal difference learning of n-tuple networks for the game 2048. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.