2048 as a MDP

Hunter Mills
Institute for Computational and Mathematical Engineering
Stanford University
Email: hmills2@stanford.edu

Abstract—2048 is a simple computer game where the player takes an action, tiles move in a predicable manner and new tiles appear randomly with know probabilities, and the current state is completely observed. Thusly, 2048 can be modeled as a Markov Decision Process (MDP). Several online search methods will be considered including forward search, sparse sampling search, and Monte Carlo tree search. These methods will be compared to a greedy policy and a random action policy.

I. Introduction

The game 2048 is played on a 4x4 grid designed by Gabriele Cirulli [1]. Each grid point is either empty or contains a tile with a numerical value. All values are powers of two [Fig. 1]. additionally, there is a score counter. The player can choose from one of four actions: left, right, up, or down. When an action is chosen all tiles slide across the board in the selected direction until they hit the edge of the board or another tile. If two tiles of the same value collide, they merge and their values are added. The value of the merged tile is then added to the score. After each action and the tiles have moved, a random empty point is filled with a tile valued at 2 (90%) or 4 (10%). An action is forbidden if it causes no tiles to move or merge, and the game ends when no more moves can be made. The goal of the game is to obtain a tile valued at least 2048. It is also possible to go above and beyind the mark of 2048 and score 4096, 8192, 16384, and so on. Here success is measured by obtaining a tile valued at least 2048.

Fig. 1. 2048 game [1].

This game is difficult for several reason. It involves planning several moves ahead, and taking calculated risks to decide when to place the game in a precarious position to merge large tiles.

Since actions and transitions into successive states are well defined, and only depend of the previous state, 2048 can be modeled as a Markov Decision Process. Because of the number of possible game states and the possible transitions, online approximations will be considered. Online approximations compute a small number of steps into future instead of considering all paths. These approximations will include forward search, sparse sampling search, and Monte Carlo tree search.

II. Background

The goal of a MDP is to determine an optimal policy π^* that at state s returns an action a that transitions into state s' that maximizes

$$\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)$$

for reward $R(s, a)$ and learning parameter $\gamma = [0, 1]$. It can also be shown that an optimal policy π^* satisfies the Bellman equation [4]

$$U^*(s) = \max_a \left(R(s, a) + \gamma \sum_{s'} T(s'|s, a) U^*(s') \right). \quad (1)$$

Online methods approximate the Bellman equation as [4]

$$U_d(s) = \max_a \left(R(s, a) + \gamma \sum_{s'} T(s'|s, a) U_{d-1}(s') \right), \quad (2)$$

with

$$U_0(s) = R(s, a).$$

Here d is the depth of future states considered. This approximation will converge to the Bellman equation as $d \to \infty$ [4].

This approximation is considered because the explicit solution is computationally infeasible. Given the set of possible actions A and the set of possible outcomes S, the number of states to be considered at depth d is

$$O((|S| \cdot |A|)^d).$$

For 2048 $|A| = 4$ for up, down, left and right. $|S|$ is twice the number of open tiles because after every move, a 2 or 4 is inserted into any open tile.

A. Online search methods

Forward search explicitly calculates equation (2). The forward search algorithm is outlined in Algorithm 1 [4], where d and γ are hyperparameters that are supplied. As discussed earlier, the runtime for this type of algorithm is $O((|S| \cdot |A|)^d)$.

Algorithm 1 Forward search

```
function SelectActions(s, d)
    if d = 0 then return (NIL, 0)
    (a*, v*) ←(NIL,-∞)
    for a ∈ A(s) do
        v ← R(s, a)
        for s' ∈ S(s, a) do
            (a', v') ← SelectActions(s', d − 1)
            v ← v + γT(s'|s, a)v'
        if v > v* then (a*, v*) ← (a, v)
    return (a*, v*)
```

Sparse sampling further approximates equation (2). Instead of explicitly computing all children states s' from parent state s, it samples n possible children based on their likelihood for each possible action a. This reduces the computational complexity to $O((n \cdot |A|)^d)$. This reduction in computation allows the search depth d to be increased. Sparse sampling is outlined in Algorithm 2 [4]. Here d, γ and n are hyperparameters that are supplied.

Algorithm 2 Sparse sampling

```
function SelectActions(s, d, n)
    if d = 0 then return (NIL, 0)
    (a*, v*) ←(NIL,−∞)
    for a ∈ A(s) do
        v ← 0
        for i ← 1 : n do
            (s', r) ∼ G(s, a)
            (a', v') ← SelectActions(s', d − 1, n)
            v ← v + (r + γv')/n
        if v > v* then (a*, v*) ← (a, v)
    return (a*, v*)
```

Monte Carlo tree search can be thought of as a variation of sparse sampling. Instead of randomly sampling n children, Monte Carlo tree search weights samples based on how interesting they are. This allows for an unbalanced tree that ceases to explore areas that appear to perform poorly. Also, when new points are encountered, new actions are sampled directly using their direct reward R as weights. Monte Carlo tree sampling is outlined in Algorithm 3 [4] Here d, γ and Δt are hyperparameters that are supplied. Because of the structure of Monte Carlo tree search, runtime Δt can be capped which allows for graceful exit.

Algorithm 3 Monte Carlo tree search

```
function SelectActions(s, d, Δt)
    t_0 ← Time()
    while Time()− t_0 < Δt do
        Simulate(s, d, π_0)
    return arg max_a Q(s, a)
function Simulate(s, d, π_0)
    if d = 0 then return 0
    if s ∉ T then
        for a ∈ A(s) do
            (N(s, a), Q(s, a)) ← (1, R(s, a))
        T ← T ∪ {s}
        return Rollout(s, d, π_0)
    a ← arg max_a Q(s, a) + c sqrt(log Σ_a N(s,a) / N(s,a))
    (s', r) ∼ G(s, a)
    q ← r + γSimulate(s', d − 1, π_0)
    N(s, a) ← N(s, a) + 1
    Q(s, a) ← Q(s, a) + (q − Q(s, a))/N(s, a)
    return q
function Rollout(s, d, π_0)
    if d = 0 then return 0
    a ∼ π_0(s)
    (s', r) ∼ G(s, a)
    return r + γRollout(s', d − 1, π_0)
function π_0(s)
    a = sample(a_{1:i}, R(s, a_{1:i})/ Σ_i R(s, a_i))
    return a
```

B. Reward functions

In order to take actions given the formulation thus far, individual game states must be scored with a reward function R. These scores are based on heuristics designed with expert knowledge of the game [2]. Multiple heuristics can be used and combined to determine the quality of a given state. The following heuristics are considered for 2048:

- Maximum tile value
- Number of open points
- Sum square of tile value
- Monotonicity

Many of these heuristics are common for solving 2048 [3].

It is also important to consider the scaling and combination of these heuristics such that they do not dominate the value of a state unjustly [2], [4]. To satisfy scaling, the base two logarithm of tile values are considered. This reduces the range of values for tiles from 2-2048 to 1-11. Also, any values that are squared will eventually have the square root taken.

The max value heuristic promotes the merger of the largest tile. Formally for a state s with tile value $s_{i,j}$ at position i, j, the max tile value heuristic is formulated as

$$f_1(s) = \log_2 \left(\max_{i,j} (s_{i,j}) \right).$$

The open tile heuristic promotes tiles being empty. This is helpful because it allows more moves to be valid, which allows the game to continue. It is formed as

$$f_2(s) = \sum_{i,j} \mathbb{1}_{s_{i,j}=0}.$$

The sum square heuristic, like the max tile heuristic, promotes the merger of tiles, but not only the largest tile. Only non-zero tiles are considered.

$$f_3(s) = \sqrt{\sum_{\substack{i,j \\ s_{i,j}>0}} \left(\log_2 \left(s_{i,j}\right)\right)^2}.$$

The formulation of the monotonicity heuristic is more involved. Across each row and column, forward and back, an accumulating maximum is created. This max is squared and differenced by the square current i, j value. These differences are then summed. Here $s_{i,j:k}$ is the set $\{s_{i,j}, s_{i,j+1}, \ldots, s_{i,k}\}$, and $s_{i:k,j}$ is the set $\{s_{i,j}, s_{i+1,j}, \ldots, s_{k,j}\}$:

$$f_N(s) = \sum_{i,j} \left(\left(\log_2 \max_j \left(s_{i,1:j}\right) \right)^2 - \left(\log_2 s_{i,j}\right)^2 \right)$$

$$f_S(s) = \sum_{i,j} \left(\left(\log_2 \max_j \left(s_{i,j:4}\right) \right)^2 - \left(\log_2 s_{i,j}\right)^2 \right)$$

$$f_W(s) = \sum_{i,j} \left(\left(\log_2 \max_i \left(s_{1:i,j}\right) \right)^2 - \left(\log_2 s_{i,j}\right)^2 \right)$$

$$f_E(s) = \sum_{i,j} \left(\left(\log_2 \max_i \left(s_{i:4,j}\right) \right)^2 - \left(\log_2 s_{i,j}\right)^2 \right).$$

This creates a monotonicity measure for up, down, left and right. If a row or column is perfectly monotonic, this is valued as zero, and positive otherwise. These can then be combined to measure monotonicity in two directions as follows

$$f_{NW}(s) = f_N(s) + f_W$$

$$f_{NE}(s) = f_N(s) + f_E$$

$$f_{SW}(s) = f_S(s) + f_W$$

$$f_{SE}(s) = f_S(s) + f_E.$$

The best monotonicity score is then extracted (least positive), and the square root is taken for magnitude reasons discussed previously. This value is then reversed to promote monotonicity

$$f_4(s) = -\sqrt{\min\left\{f_{NW}(s), f_{NE}(s), f_{SW}(s), f_{SE}(s)\right\}}.$$

All of these functions are taken as a weighted combination, and the reward R of a state s is evaluated as

$$R(s) = w_1 f_1(s) + w_2 f_2(s) + w_3 f_3(s) + w_4 f_4(s).$$

Here w_1, w_2, w_3, w_4 are hyperparameters.

III. Implementation

The 2048 MDP solver was written in Julia [5]. The game mechanics (not solver) for efficient computation where inspired by Robert Xiao [6]. The grid was stored as a 64-bit unsigned integer were each 4-bit nibble represented a grid point. Two raised to the power of the nibble represents the grid point value. This allows the max tile value of $2^{16} = 32768$. Many move and score computations were precomputed across rows and columns allowing for fast lookup and simple recombination. This allowed for roughly 10^6 states to be analyzed per second.

Hyperparameters with similar runtimes in the worst case were considered. This included a forward search with $d = 2$ (FS2) and $d = 3$ (FS3). Included is sparse sampling with $d = 4, n = 4$ (SS4-4), $d = 5, n = 2$ (SS5-2) and $d = 6, n = 1$ (SS6-1), $d = 7, n = 1$ (SS7-1) and $d = 8, n = 1$ (SS8-1). And, also included is a Monte Carlo tree search with $d = 4, \Delta t = 0.2$ (MCTS4), $d = 5, \Delta t = 0.2$ (MCTS5) and $d = 6, \Delta t = 0.2$ (MCTS6). Runtimes for each of these settings are listed in Table 1.

TABLE I
Action selection runtime table in seconds

Policy	Parameters	Average	Deviation	Max
FS2	$d = 2$	0.00106	0.00089	0.00677
FS3	$d = 3$	0.017	0.028	0.351
SS4-4	$d = 4, n = 4$	0.242	0.068	0.442
SS5-2	$d = 5, n = 2$	0.107	0.035	0.217
SS6-1	$d = 6, n = 1$	0.0171	0.0056	0.0384
SS7-1	$d = 7, n = 1$	0.0583	0.0235	0.175
SS8-1	$d = 8, n = 1$	0.174	0.0723	0.499
MCTS4	$d = 4, \Delta t = 0.2$	0.201	0.001	0.204
MCTS5	$d = 5, \Delta t = 0.2$	0.201	0.002	0.207
MCTS6	$d = 6, \Delta t = 0.2$	0.201	0.005	0.213

For better comparison, all other hyperparameters were held constant. Heuristics weights were set as $w = [3.25, 4, 1, 1.75]$, the learning parameter $\gamma = 0.75$, and for Monte Carlo tree search, $c = 10$.

One hundred trials were collected for each search algorithm.

TABLE II
Success rates for each search policy

Policy	%Success	%2048	%4096	%8196
Random	0	0	0	0
Greedy	0	0	0	0
FS2	11	11	0	0
FS3	54	50	4	0
SS4-4	67	58	9	0
SS5-2	81	70	11	0
SS6-1	72	59	13	0
SS7-1	78	61	17	0
SS8-1	79	64	15	0
MCTS4	75	59	16	0
MCTS5	92	47	43	2
MCTS6	82	59	25	0

TABLE III
Scores for each search policy

Policy	Average	Deviation	Max
Random	1183	599	2864
Greedy	4340	1613	8412
FS2	14722	6726	32840
FS3	25452	11168	71836
SS4-4	31114	14630	78204
SS5-2	34603	13376	78136
SS6-1	31752	13894	72780
SS7-1	33854	13792	70680
SS8-1	34415	15181	80200
MCTS4	35585	16182	82744
MCTS5	49627	22540	155964
MCTS6	40388	16810	82760

IV. Results

All methods accounting for uncertainty performed better than the greedy policy and the random policy, and forward search of depth 3 performed better than depth 2 (Figure 2, 3, 4, 5 Table 2, 3).

All sparse sampling methods considered performed better than forward search of depth 3 despite only considering shockingly low values of n. Sparse sampling with $d = 5$ and $n = 2$ performed best (Figure 2, 3, 4, 5 Table 2, 3) despite having a lower runtime than both $d = 4, n = 4$ and $d = 8, n = 1$ (Table 1).

Monte Carlo tree search of depth 5 performed better than both 4 and 6. It also performed best overall. It had a 92% success rate, the highest game scores, the highest rate of achieving a max tile of 4096 (43%) and the only policy to achieve a 8196 tile (2%) (Figure 2, 3, 4, 5 Table 2, 3).

Overall, online MDP approximations considered were successful in playing and beating 2048.

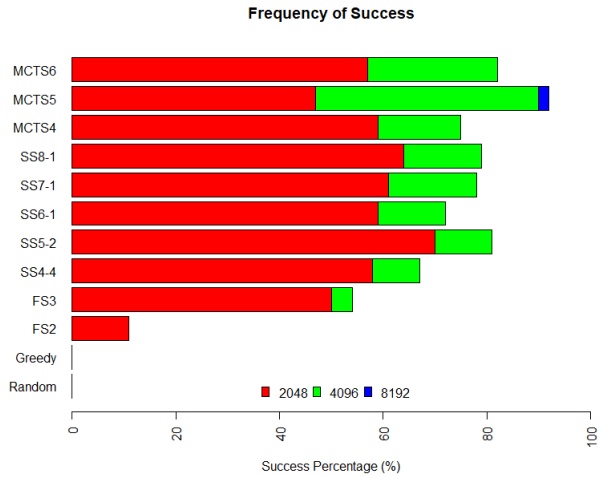

Fig. 2.

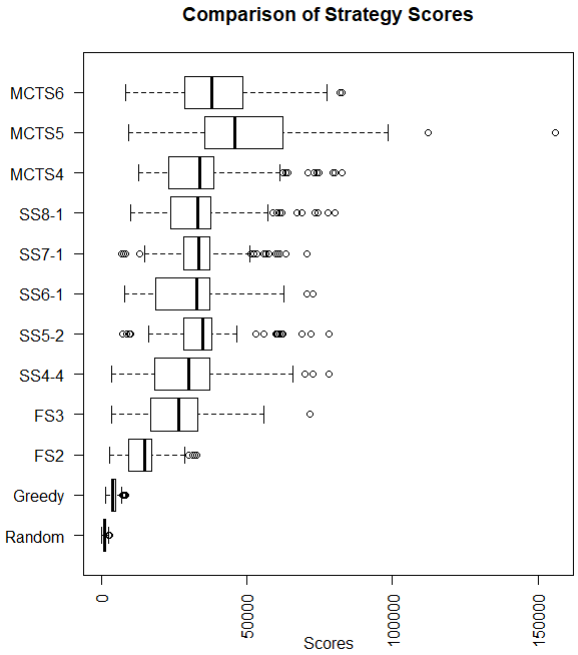

Fig. 3.

V. Further Work

Code optimization and parallelization could be performed to better improve time performance to allow deeper tree searches.

Parameter optimization could also be performed to better tune the heuristic weights w and learning parameters γ. This could include cross entropy methods, or evolutionary models [4].

Iterative deepening could be implemented to allow constant time computation in forward search or sparse sampling, by rerunning the computation one level d deeper or number of points sampled n each iteration until time expires [2]. Hybrid methods could also be considered that switch between forward search sparse sampling based on the current rate at which possible states expand. It would also be possible to implement multiple search methods with a voting scheme to choose actions.

References

[1] G. Cirulli. 2048. [Online]. Available: https://gabrielecirulli.github.io/2048/

[2] S. J. Russell and P. Norvig, Artificial Intelligence, A Modern Approach, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2010.

[3] What is the optimal algorithm for the game 2048. [Online]. Available: https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048

[4] M. J. Kochenderfer, Decision Making Under Uncertainty. Cambridge, MA: MIT Press, 2015.

[5] S. K. Jeff Bezanson, Alan Edelman and V. B. Shah, "Julia: A fresh approach to numerical computing," SIAM Review, 2017. [Online]. Available: http://julialang.org/publications/julia-fresh-approach-BEKS.pdf

[6] R. Xiao. 2048-ai. [Online]. Available: https://github.com/nneonneo/2048-ai

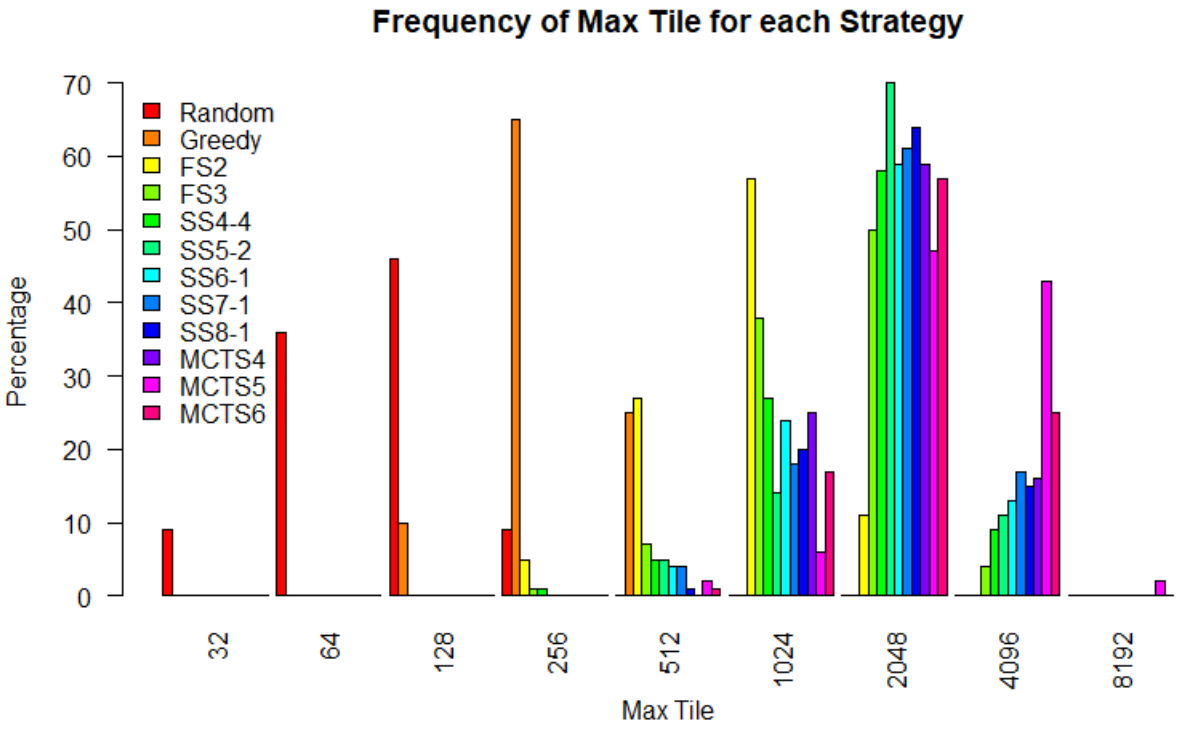
Frequency of Max Tile for each Strategy
Random
Greedy
FS2
FS3
SS4-4
SS5-2
SS6-1
SS7-1
SS8-1
MCTS4
MCTS5
MCTS6
Percentage
0
10
20
30
40
50
60
70
32
64
128
256
512
1024
2048
4096
8192
Max Tile

Fig. 4.

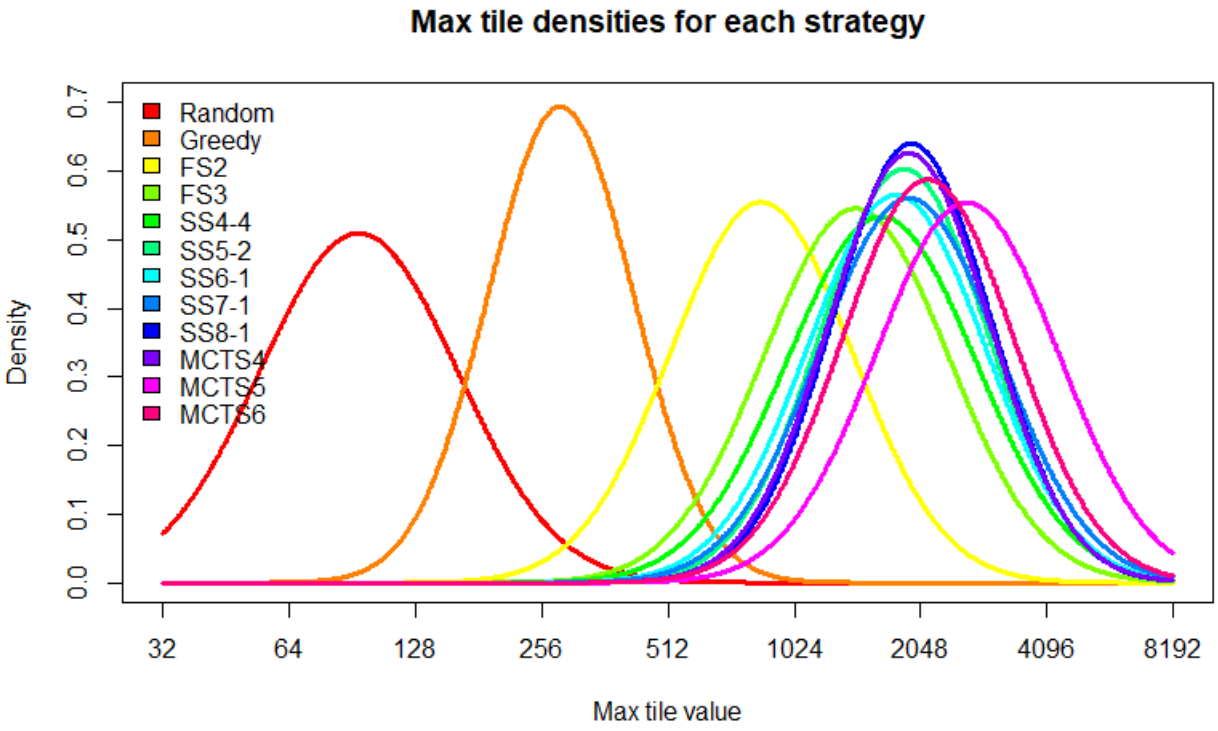
Max tile densities for each strategy
Random
Greedy
FS2
FS3
SS4-4
SS5-2
SS6-1
SS7-1
SS8-1
MCTS4
MCTS5
MCTS6
Density
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
32
64
128
256
512
1024
2048
4096
8192
Max tile value

Fig. 5.