Strongly Solving $2048_{4\times3}$

Tomoyuki Kaneko*and Shuhei Yamashita
Graduate School of Arts and Sciences, the University of Tokyo, Japan

October 7, 2025 (preprint)

Abstract

2048 is a stochastic single-player game involving 16 cells on a 4 by 4 grid, where a player chooses a direction among up, down, left, and right to obtain a score by merging two tiles with the same number located in neighboring cells along the chosen direction. This paper presents that a variant $2048_{4\times3}$ with 12 cells on a 4 by 3 board, one row smaller than the original, has been strongly solved. In this variant, the expected score achieved by an optimal strategy is about 50724.26 for the most common initial states: ones with two tiles of number 2. The numbers of reachable states and *afterstates* are identified to be $1,152,817,492,752$ and $739,648,886,170$, respectively. The key technique is to partition state space by the sum of tile numbers on a board, which we call the *age* of a state. An age is invariant between a state and its successive afterstate after any valid action and is increased two or four by stochastic response from the environment. Therefore, we can partition state space by ages and enumerate all (after)states of an age depending only on states with the recent ages. Similarly, we can identify (after)state values by going along with ages in decreasing order.

2048, stochastic MDP, strongly solve

1 Introduction

2048 is a popular video game and also a challenging environment for reinforcement learning agents as investigated by, e.g., Szubert and Jaśkowski (2014); Antonoglou et al. (2022). While the original game uses a 4×4 board, this paper discusses a variant with a 4×3 board. Let $2048_{r\times c}$ be a variant with rows r and/or columns c. In 2022, $2048_{3\times3}$ (or Mini2048) was strongly solved by Yamashita et al. (2022). Since then, it has served as a tractable evaluation environment for reinforcement learning agents, for example, by Matsuzaki and Terauchi (2024).

This paper presents that a larger variant $2048_{4\times3}$ was strongly solved. The results tell us that the numbers of states and *afterstates* are $1,152,817,492,752$ and $739,648,886,170$, respectively, in this variant. The numbers are more than 10^4 times larger than those of $2048_{3\times3}$ where the number of states is merely $48,713,519$. The number of states in $2048_{4\times3}$ is huge enough to prohibit straightforward approaches from enumerating them all. The key idea that enabled the computation, in a few days on a personal computer is to partition state space by the sum of the numbers on a board, which we call the *age* of a state. An age is invariant between a state and its successive afterstate after any valid action and is increased two or four by stochastic response from the environment. Therefore, we can partition state space by ages and enumerate all (after)states of an age depending only on states with the recent ages. As shown in Fig. 1, the number of (after)states is found to be less than $3\cdot10^6$ for each age. Therefore, it is feasible to keep several consecutive ages in memory assuming gigabytes of memory that are now commonly available. Similarly, we can identify (after)state values by going along with ages in decreasing order. In addition to the partitioning, we introduce a compact representation adapting Elias-Fano codes to reduce disk usage. While a straightforward store requires $739,648,886,170\cdot48/8\approx4.4$ TiB, our representation did it in about 1.4 TiB in a search-efficient manner and in less than 300 GiB if specialized for optimal playing.

We also present several statistics on optimal value function identified verifying game players intuitions. Note that this paper is an extended version from a workshop proceeding written by the same

*kaneko@acm.org

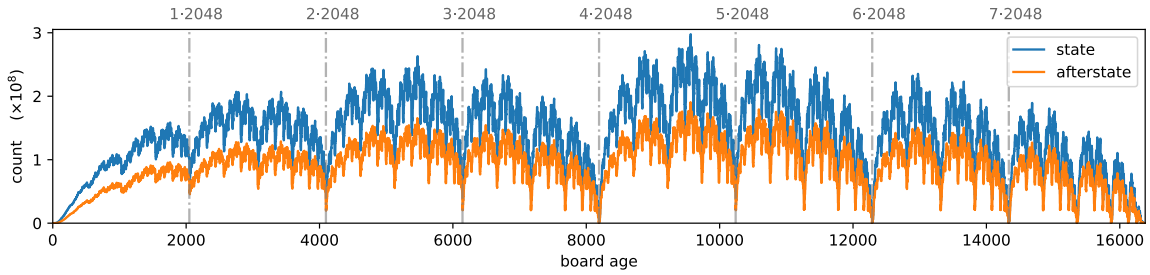


Figure 1: Number of states and afterstates per age. The number of states is at most $297,784,736 < 3{\cdot}10^6$ for each age, so we can keep several ages in memory. Multiple valleys were observed at ages near multiple of 2048, shown by vertical dashed lines. Apparently, they indicate the difficulty of making tile 2048, which requires careful arrangement of 10 tiles ($2, 4, 8, 16, 32, 64, 128, 256, 512$, and 1024) while there are only 12 squares.

authors published in Japanese.[1] Although the primary idea of age and basic statistics were presented in the earlier version, our method have been substantially improved including the compact state representation and the overall efficiency. Moreover, several experiments help understanding of the game have been augmented.

2 Backgrounds and related work

2.1 Markov decision process and 2048

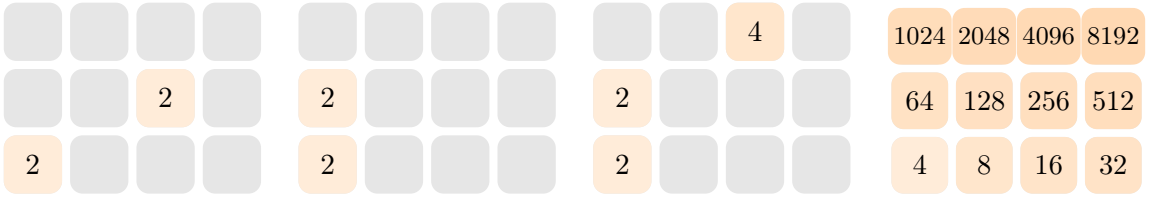
We briefly introduce the stochastic Markov decision process (MDP), and then define game of *2048* following the notation. An initial state s_0 is given among a set of initial states $\mathcal{S}_0$. For each time step $t \geq 0$, an agent chooses an action a_t from a set of valid actions $A(s_t)$ in the current state s_t. The state at the next time step s_{t+1} and reward r_{t+1} are sampled following the transition probability conditioned on s_t and a_t.

In the original *2048*, a state consists of 16 cells in a 4×4 square grid. In our variant $2048_{4\times3}$, a state consists of 12 cells. Each cell has a tile with number of power of two, i.e., $2, 4, 8, \cdots, 2^n, \cdots$, or is empty. Let c_i be the number of a tile in the i-th cell in a state or 0 if empty (the order in i is arbitrary as long as neighborhood and directions are properly defined). Then, we denote state s as a sequence of cells $s = [c_i]_{i\in[1,12]}$. Action space consists of four directions, $A(s) \subseteq \{\text{left, right, up, down}\}$, some of which can be invalid depending on the state. The transitions of *2048* are well understood by *afterstate* (Chapter 6.8 in the textbook by Sutton and Barto (2018)). In *2048*, after taking action a_t at s_t, the state deterministically transits to afterstate s'_t. The transition causes all tiles to slide towards direction a_t, merging exactly two adjacent tiles with the same number along the direction if such tiles exist. A merged tile has a new number that is the sum of two number tiles (i.e., double the old number) and gives an agent the reward of a new number. Then, the transition from afterstate s'_t to the next state s_{t+1} is stochastic; a new tile is placed at an empty cell with equal probability and its number is 2 with 90% or 4 (with 10%). The set of initial state $\mathcal{S}_0$ consists of a state with exactly two tiles placed by adopting the same procedure for the transition from afterstate to the next state. If action a_t does not involve change in any cell, i.e., afterstate s'_t and state s_t are equivalent, the action is invalid. Consequently, a valid afterstate must have at least one empty cell as a result of sliding and/or merging. A game terminates when there is no valid action.

Fig. 2 illustrates an example of initial state (2a), and its afterstate after *left* action (2b). After the transition, tiles were left aligned while there are no merged tiles in this example. Fig. (2c) shows its possible next state by spawning a new tile. The rightmost one (2d) is one of the best reachable terminal state. There are no valid action because there are no empty cells or tiles can be merged in any direction.

The goal of an agent is to maximize cumulative reward $\mathbb{E}_\pi\left[\sum_{t\geq0} r_t\right]$ where the expectation covers the stochastic transition of MDP and the agent policy $\pi(a \mid s)$, which is the probability distribution

[1] http://id.nii.ac.jp/1001/00229224/



(a) an initial state (age 4) (b) afterstate after *left* (age 4) (c) one of next states (age 8) (d) a terminal state at best (age 16380)

Figure 2: Example of states in $2048_{4\times3}$. *Age* is defined in Sect. 3.1

(including a deterministic one) over actions conditioned on state s.[2] To strongly solve the game, we want an optimal policy $\pi^*(a \mid s)$ and optimal state value function $v^*(s_t) = \mathbb{E}_{\pi^*}\left[\sum_{t' \geq t} r_{t'}\right]$ presenting total rewards received in the future when the agent visit s_t at time t. Similarly, we define the optimal afterstate value function $v^*(s'_t)$. Because the transition to afterstate is deterministic, denoting with T as $s'_t = T(s_t, a_t)$, an optimal policy is defined to give probability one for action(s) with maximum $\max_{a \in A(s)} v^*(T(s, a))$ (ties can be distributed arbitrary) and zero for others, for each state s.[3] For simplicity, we say the (after)state value of state s for $v^*(s)$ if it is clear in the context that the value is for an optimal policy not subotpimal ones. In short, we compute all afterstate values in $2048_{4\times3}$ and store them in a retrievable manner so that one can easily identify an optimal action at runtime when visiting state s.

2.2 Related work

The solutions of games have long been intensively studied as summarized by van den Herik et al. (2002). A game is called *strongly solved* when the values of all reachable states are identified and *weakly solved* when the value of the initial state is identified, as in checkers by Schaeffer et al. (2007) or in Othello by Takizawa (2023). Many interesting games remain unsolved, and the number of valid positions can be very difficult to identify or even estimate, as in e.g., Go by Tromp (2016) or chess[4].

In imperfect-information games, different from perfect-information games or puzzles, the goal is to obtain a set of strategies (policies) near Nash equilibrium. Although totally different techniques are needed for this purpose, there is a similar challenge in how to manage huge state space. In methods based on counterfactual regret minimization (see e.g., Moravčík et al. (2017); Brown and Sandholm (2017) for famous achievements), *counterfactual* values are maintained on each information state which is a set of worldstates composed considering hidden information, instead of state values in our case.

This study is in the line of adding a new single-player perfect-information game to strongly solved ones. In general, stochastic games including *2048* are more difficult than deterministic games because state values involve expectation over transition probabilities, disabling pruning techniques such as $\alpha\beta$ pruning effective in deterministic games.

3 Method

We introduce the property of *age* for states and afterstates, as a key idea. Then, efficient enumeration of valid states leveraging their age and compact representation of results are described.

3.1 Partition of states

We define the *age* of states and afterstates to partition them.

Definition 3.1. The *Age* of state s is the total of number tiles as $\text{age}(s) = \sum_{c_i \in s} c_i$, where we define state s as a set of cells, $s = [c_i]_{i \in [1,12]}$. The value of variable c_i is 0 if the corresponding cell is empty and the number of its tiles otherwise. The age of afterstate s' is defined similarly.

[2] We treat discount factor $\gamma = 1$ to be consistent with this game.
[3] There can be multiple optimal policies though all of them share the same optimal value function.
[4] https://github.com/tromp/ChessPositionRanking

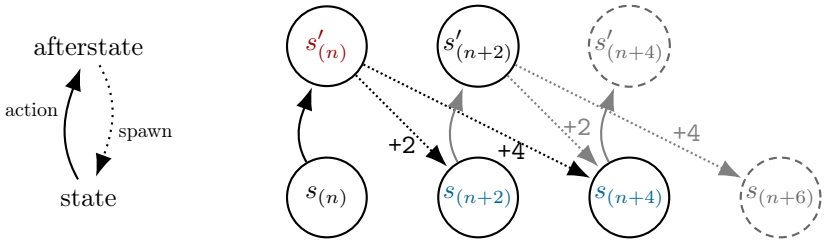


Figure 3: Transitions between state and afterstates by action and spawning of a new tile of 2 or 4, along with age n, i.e., each transition between consecutive states s_t and s_{t+1} increases age by 2 or 4.

```
1  def forward(initial_set):
2      '''enumerate all states and after-states reachable'''
3      age = age_of(initial_set)
4      # keep three state sets for (age, age+2, age+4)
5      set_0, set_2, set_4 = initial_set, [], []
6      # main loop
7      while any(set_0, set_2, set_4):
8          # walk through states in this age
9          afterstate_set = make_afterstate(set_0)
10         spawn(afterstate_set, set_2, set_4)
11         store(age, afterstate_set)
12         # increment age
13         set_0, set_2, set_4 = set_2, set_4, []
14         age += 2
15     # terminate when no more state in (set_0, set_2, set_4)
```

Figure 4: Pseudocode of `forward` procedure (borrowing Python syntax)

All valid states are partitioned by age, i.e., we can define a sequence of subsets of states, such that (1) all elements of each subset have the same age, (2) the intersection of any two subsets is empty, and (3) the union of all subsets includes all valid states. Similarly, we can partition all valid afterstates by age.

Fig. 3 illustrates changes of age along with transition between a state and afterstate. Let $s_{(n)}$ and $s'_{(n)}$ be a state and after state with age n, respectively. Whenever a valid action leads $s_{(n)}$ to $s'_{(m)}$, we can confirm that their age is the same, i.e, $n = m$. It is straightforward if there are no merging tiles and from the fact that the number of new tiles is equivalent to the sum of the tiles merged otherwise. For transition from an afterstate to a state, age is increased by a new tile being added (noted by spawn in the figure). Note that although age n monotonically increases along with time step t as $n \geq 2t$, they do not form one-to-one mapping due to randomness in the number of tiles newly spawned.

Now it is clear that

1. a set of valid afterstates with age n, denoted by $\mathcal{S}'_n$, is identified by a set of valid states with age n, denoted by $\mathcal{S}_n$, and
2. a set of valid states with age n, $\mathcal{S}_n$, is identified by set of valid afterstates with two previous ages, $\mathcal{S}'_{n-2}$ and $\mathcal{S}'_{n-4}$.

Therefore, we can enumerate all valid states and afterstates by going through age n in increasing order wihle keeping only three subsets of states in memory. Fig. 4 lists pseudocode (borrowing Python's syntax) of forward procedure for enumeration of (after)states, where the name is for increasing age. In the main `while` loop starting at line 7, we keep three state sets: `set_0` for age, `set_2` for age+2, and `set_4` for age+4. At the beginning of each loop `set_0` is completed, but `set_2` is partially enumerated, and `set_4` is empty. This is because the age of a state after spawn depends on of tile number 2 or 4. At line 10, procedure `spawn` iterates over each empty cell in each afterstate with the current age and adds a state to `set_2` by placing tile 2 and similarly to `set_4` by placing tile 4. More details of the spawn procedure will be discussed in Appendix. As listed in line 11, we suggest storing afterstates (without storing states) to save disk space.

```
1  def backward(terminal_age=16380, initial_age=4):
2      '''identify all state values from terminal to initial'''
3      age = terminal_age
4      # main loop
5      while age >= initial_age:
6          # (1) ensure future (after)states and values are available
7          # states_4: states of age + 4
8          # states_2: states of age + 2
9          # values_4: value of states_4
10         # afterstate: afterstates of age
11         # afterstate_2: afterstates of age + 2
12         # values_a_2: value of afterstate_2
13         # (2) identify state values
14         values_2 = accumulate_action(state_2, afterstate_2, values_a_2)
15         # (3) identify afterstate values
16         values_a = accumulate_spawn(afterstate, states_2, values_4, states_4, values_4)
17         # decrement age
18         age -= 2
19         states_4, values_4 = states_2, values_2
20         afterstate_2, values_a_2 = afterstate, values_a
21         # prepare states, afterstates for the next iteration
22         # load afterstates of age-2 to update (states, state_2) by spawn
```

Figure 5: Pseudocode of `backward` procedure (borrowing Python syntax)

To identify the value of each state or afterstate, we go backward in decreasing order of age. For this backward path, we observe

1. the value of each state with age n is defined as the maximum value of corresponding afterstates, in $\mathcal{S}'_n$, and
2. the value of each afterstate with age n is defined as the expectation of all corresponding states in $\mathcal{S}_{n+2}$ and $\mathcal{S}_{n+4}$.

Fig 5 illustrates pseudocode of the backward path. At line 14, procedure `accumulate_action` identifies values of states with age+2 by using that of afterstates with age+2. Then, at line 16, procedure `accumulate_spawn` identify values of afterstates with age by using those of states with age+2 and age+4. The actual implementation of the omitted part is straightforward, but has a slightly tedious part that involves recovering a set of states from stored afterstates.

3.2 Compact representation

We need two efficient representations of a state or afterstate: for keeping unique IDs as a set in memory, and the other for storing in disk keeping look-up efficiency as high as possible.

3.2.1 ID for in-memory computation

For in-memory computation, we adopted an unsigned 64-bit integer as (after)state identity, ID. As shown in Fig. 2d, the maximum number appearing on any tile is 2^{13}. Therefore, each cell c_i is presented in a 4-bit integer $v_{c,i} \in \{0, 1, \cdots, 13\}$, 0 for empty cell $c_i = 0$ or logarithm of its tile number $\log_2 c_i$ otherwise. Aggregating 12 cells in a state, we have a 48-bit unsigned integer. Because the number of states for each age is manageable, less than 300 million as shown in Fig. 1, we did not adopt further compressionc during in-memory computation.

With (after)state IDs, the forward and backward path require basically two functionalities; transition between (after)states and removing duplicates. The former is straightforward as one can easily recover the status of each cell from (after)state ID. Regarding the latter, there are multiple sources of duplicates; different states may lead to the same afterstate after sliding and merging, and different afterstates may result in the same state by adding tile 2 and 4. Also, equivalent states with symmetry (horizontal, vertical, and rotation of 180 degrees) should be unified. Moreover, parallel computation in each age may matter (implementation details will be discussed in Appendix). In our experience, the following

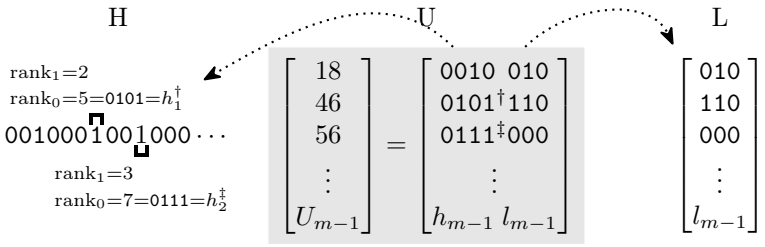


Figure 6: An illustrative example; set $U = (18, 46, 56, \ldots)$ whose values are less than 2^n ($n = 7$) are presented as a pair of higher part H with $q = 4$ bits and lower L with $n - q = 3$ bits. Binary digits are written in typewriter font.

straightforward way was most efficient; keep a sequence of 64-bit integers, generate new sequences ignoring duplicates, and then merge them after sorting of each vector (provided as `std::vector` and `std::set_union` in C++[5]).

3.2.2 Compact representation in storage

After enumeration of all (after)state IDs, we want to keep them in storage for the backward path as well as for further applications. A straightforward way is to save IDs as they are. As the numbers of states and afterstates were about 10^{12} and $7 \cdot 10^{11}$, they need 6.9 TiB and 4.4 TiB in total, respectively.[6] While these sizes might be acceptable for current storages, they should preferably be reduced for further applications, e.g., evaluation of gameplay. Here, we introduce a classical technique about compact representation of a very sparse bit-vector supported by Elias-Fano codes. Here, we briefly introduce core techniques. Readers can safely jump to the Results section because the results did not depend on the techniques described here except for storage efficiency, where all afterstate IDs were stored in about 1.4 TiB maintaining decent look-up efficiency. For further details of compact data structures, readers are referred to Chapter 4.4 of the textbook by Navarro (2016).

There are several equivalent understandings of the challenge on what to store; a sparse set of unsigned integers, a sequence of unsigned integers in strictly increasing order, a (conceptual) vary sparse bitset where the n-th bit is 1 if and only if number n is included in the set in the previous problem. We basically adopt the second interpretation. Let U be a sorted sequence of m unique IDs that we want to store, U_i be the i-th smallest element in U, and n be a minimum integer satisfying $0 \le U_i < 2^n$ for all $i \in [0, m-1]$ (note that we intentionally adopt the 0-index here). Therefore, each U_i is in n bit length, resulting in nm bits in total in a naive representation. The core idea is to separate n bits into a higher part h_i with q bits and a lower one l_i with $(n - q)$ bits, such that $U_i = h_i \cdot 2^{n-q} + l_i$, where width q is carefully chosen as $\lfloor \log_2 m \rfloor$. Hereafter, we omit floor symbols $\lfloor \rfloor$ for simplicity in notation. Then, the lower l_is are concatenated into single bit-vector L of length $(n - q)m$ bits and stored as is. The higher bits h_is are embedded into a well crafted bit-vector H. Important properties of H are:

- H is bit-vector of $(m + q)$ bit-length, containing m 1s and q 0s,
- bit 1 in H is set for each position $k + h_k$ for $0 \le k < m$,[7] and consequently,
- for any j and k such that the j-th bit in H is the k-th 1 from the beginning, i.e., $H[j] = 1$ and $\mathrm{rank}_1(H, j) = k$, then, value h_k is given by $\mathrm{rank}_0(H, j)$,

where functions $\mathrm{rank}_0(H, j)$ and $\mathrm{rank}_1(H, j)$ are defined as the number of 0s and 1s in the first j bits in bit-vector H, respectively. In other words, one can visit all h_ks by iterating each bit in H, from the beginning to the end, with counting of 0s and 1s. Note that, although items in the original set U are unique, their higher bits may have the same value and these properties are consistent with such cases. The reduction in size comes from data structure H because the lower part L is stored as it is. While

[5] `https://en.cppreference.com/w/cpp/algorithm/set_union.html`

[6] We assume 48-bit for each ID in estimation here.

[7] The positions are distinct because u_k and consequently h_k are sorted in increasing and non-decreasing order, respectively.

Table 1: Example of stored afterstates: #IDs of IDs are stored in $H + L$ bytes much smaller than in-memory size, which is 8byte/item.

age	#IDs (m)	base (b)	hi (q) (bits)	low (bits)	H (bytes)	L (bytes)	in memory size (bytes)	ratio (%)
2000	75,344,033	12	26	14	17,952,888	114,771,256	602,752,264	22.0
8000	78,982,989	14	26	14	17,384,728	125,945,520	631,863,912	22.7

naive storing of the higher part requires qm bits (m items and q bits each), we stored H in $m + q$ bits. With $q = \log_2 m$, the difference in size is $qm - (m + q) = (m - 1)\log_2 m - m > 0$ ($m \geq 3$). Note that the sparseness, $m \ll 2^n$, is crucial as $m \approx 2^n$ indicates $q \approx n$ where all bits of items will be stored in H in a redundant manner. Fig. 6 illustrates an example of set $U = \{18, 46, 56, \cdots\}$ where each element in U is presented in $n = 7$ binary digits. Assuming that $q = 4$, we separate each item into its higher 4 bits and the other 3 bits. The latter parts are concatenated into bit vector L. The higher bits are stored in H where each bit with value 1 has one-to-one mapping of each item in U, as shown in illustration of items $h_1 = \texttt{0101}^\dagger$ and $h_2 = \texttt{0111}^\ddagger$.

We adopted three additional tweaks to have smaller bit lengths n for IDs, while the number of IDs, m, is constant. To do so, we want to bound the maximum ID as small as possible. Simply choosing the smallest one among symmetries works. Also, we can change the base depending on the largest tile number. So far, we encode state ID as $\sum_{i\in[1,12]} v_{c,i} \cdot 16^{i-1}$ giving each cell 4-bit width. However, given that the maximum value of $v_{c,i}$ is 13, it is safe to encode ID with base $b = 14$ as $\sum_{i\in[1,12]} v_{c,i} \cdot b^{i-1}$. Base b can be decreased more in accordance with the maximum value for each age. The last tweak is to drop information about cell c_1 to formulate compressed ID as $\sum_{i\in[\mathbf{2},12]} v_{c,i} \cdot b^{i-2}$. Recall that we partitioned (after)states by their age, so, the sum $\sum_{i\in[1,12]} c_i$ is constrained for each partition. Therefore, we can safely forget exactly one cell and later recover all information.

Table 1 lists two examples of storage size for age 2,000 and 8,000. Column #IDs m is for the number of afterstates in each age, already shown in Fig. 1. For the width of the higher part, we used 26 as the floor of $\log_2 m$, though the results are similar if the ceiling is adopted instead of the floor. The actual storage size of H and L are slightly larger than the theoretical value due to 64-bit boundary with a sentinel. Compared to bytes in memory used in Sect. 3.2.1, the storage size of the sum of H and L is about 22%, and we believe that it is well compressed.

Overall, one can look up an afterstate by the following procedure:

1. identify its age as the sum of all tile numbers,
2. encode ID as 48-bit unsigned integer, normalize it as the minimum ID in the equivalent symmetries,
3. convert it to compressed ID u by dropping the least significant digit, and rebasing with base $[12, 14]$ depending on the age
4. separate u as pair of higher and lower bits (u_h, u_l), such that $u = u_h \cdot 2^{(n-q)} + u_l$ where width $(n - q)$ depends on the age
5. lookup storage H of the age to find the minimum index i_h satisfying $\mathrm{rank}_0(H, i_h - 1) = u_h$. If the index is more than its length then the result is not found. Otherwise,
6. let $i_l = \mathrm{rank}_1(H, i_h)$ and lookup storage L of the age to see whether the i_l-th value equals u_l. If true, return success. Otherwise, check if $(i_h + 1)$-th bit is 1 in H. If so, because it means multiple items in U share the same upper bits u_h, continue with incremented i_h and i_l.

For (after)state values, we simply store values following the order of afterstates, at each age. Therefore, it is straightforward to retrieve the (after)state value by using its age and rank i_l. It is known that by adding appropriate indices, rank and select operations on H becomes more efficient. However, we simply adopted a linear search with 64-bit pop-count operation equipped with modern CPUs because it is efficient enough compared to the cost of disk read for L or values.

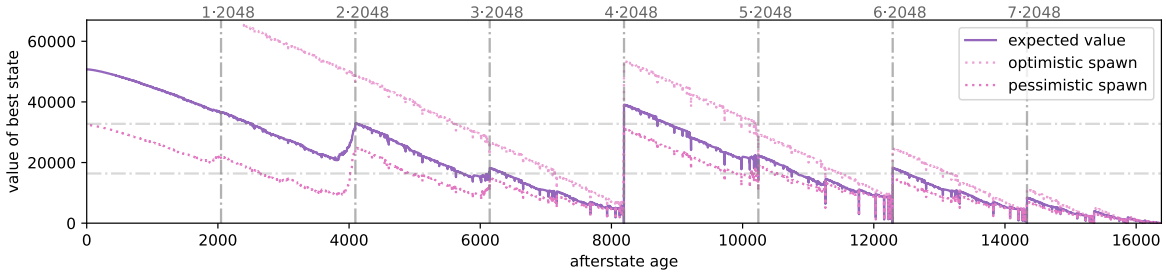


Figure 7: Value (expected returns in future) of best afterstates at each age (purple). Peaks at age near multiples of 2048 indicate difficulty of making a tile of larger numbers. Two dotted lines (pink) are for optimistic and pessimistic variation assuming each tile spawns at the most and least preferable cell in the future, respectively. The former starts 82705.6 at age 4 though y-axis is arranged to better focus on primary values.

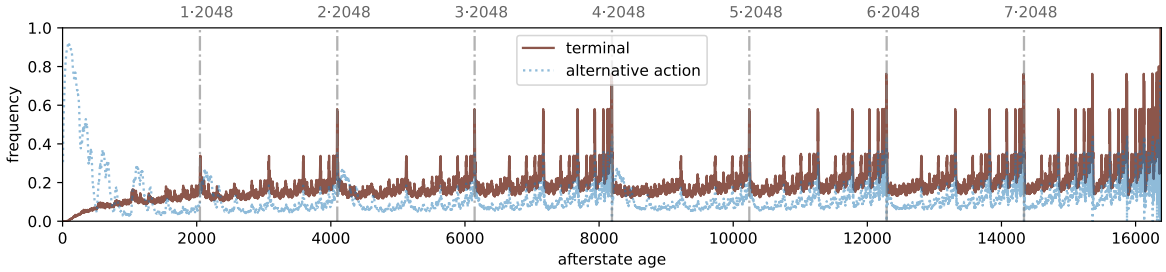


Figure 8: Relative frequency of afterstates being terminal at each age (brown). The ratio is around 0.2 for most ages, but there are occasional peaks, meaning that one needs careful arrangement of tiles to go beyond that age. Dotted line (blue) shows the frequency of states having an effective alternative action (see main text for details).

3.2.3 Representation for optimal playing

For optimal playing, one can compress more to have an optimal move for each state less than 275 GiB total. Recent advances enable us to represent a set of (after)states with 1.5 bits per item and a state-value mapping with $1.01 \cdot r$ bits per item where r is bit length of the value for each state (see, e.g., survey by Lehmann et al. (2025)), if one can assure a valid (after)state is given with each query. This is much more efficient than using more than ten bits as in our empirical data listed in Table 1. For optimal playing, we want to lookup an optimal action for a state, where $r = 2$ bits are enough for at most four valid actions. In fact, in our experiences, the BuRR algorithm, by Dillinger et al. (2022) with publicly available implementation,[8] successfully stored all optimal moves within about 1.9 bits per state. That is, the storage for states themselves is almost negligible while values available are kept for any valid state in a query.

A drawback of these techniques is that one cannot recover or enumerate (after)states from the database itself and needs to an (after)state give with each query. This prevents some applications requiring access to the entire (after)state space, e.g., study on exploring starts in reinforcement learning. Note that our backward procedure also requires entire afterstates at each age. Therefore, it is suggested to keep the original until its completion.

4 Results

Fig. 1 already shows the number of reachable states and afterstates (the vertical axis) for each age (the horizontal axis), which are identified by the forward path (listed in Fig. 4). Multiple valleys were observed at ages multiple of 2048, shown by vertical dashed lines. This is consistent with common sense:

[8] `https://github.com/lorenzhs/BuRR`, `https://github.com/ByteHamster/SimpleRibbon`

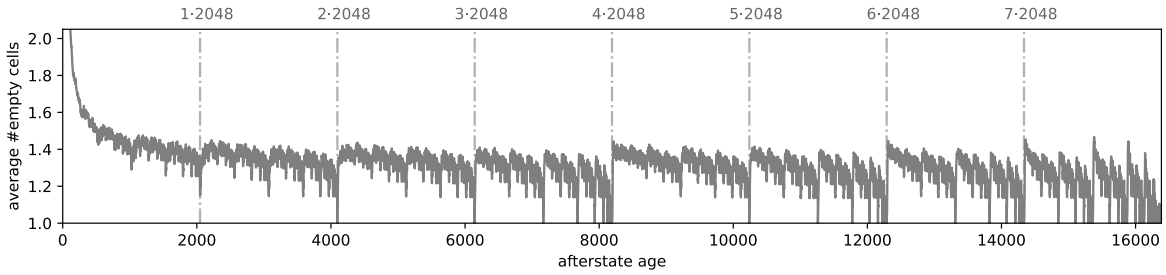


Figure 9: Average number of empty cells at each age. Note that a valid afterstate has at least one empty cell.

making tiles of 2048 or more is difficult because it requires careful arrangement of n tiles to make a tile of 2^n while the number of available cells is limited. Related to this topic, there are no valid afterstates at age $8190, 12286, 14334, 15358, 15870, 16126, 16254, 16318, 16350, 16366, 16374$, or 16378. This is because at least 12 tiles are needed to make the sum of any of these numbers while a valid afterstate can have at most 11 tiles. We further confirm the consistency in terms of state values.

The backward path (listed in Fig. 5) identified the value of each (after)state (that is, expected returns in the future), starting at the afterstate. Fig. 7 shows the values of the best afterstates at each age the in primary line (purple). The value takes about 50724.26 at age 4, initial states, and roughly decreases as age increases with occasional *jumps* at ages multiple of 2048, indicating that one needs some luck to make a large number of tiles. It is natural that the difficulty of obtaining more rewards avoiding termination increases when a game proceeds in general and just before making a large number of tiles. Recall that state value does not include past rewards obtained before reaching the state. The other two dotted lines (pink) in Fig. 7 show optimistic and pessimistic variants of values assuming that each tile spawns the most and least preferable cell, respectively. After its number is sampled with the original probability, i.e., 2 with 90% or 4 with 10%, the cell with the maximum and minimum values (of that variant) is chosen. The gaps between primary (purple) and the dotted (pink) lines are small near terminal or ages before making a large tile. Please note that there are other interesting analyses with different definitions because this is our interpretation of an optimistic or pessimistic situation and an example demonstrating that our method consistently works with slight changes in an environment. The difficulty of making a large tile can also be illustrated by relative frequency of afterstates being terminal, i.e., there are no more valid actions after spawning a new tile, as shown by the primary line (brown) in Fig. 8 for each age.

For initial states, there are three cases, age 4 (two tiles of 2), age 6 (one tile 2 and one tile 4), and age 8 (two tiles of 4). We found that all initial placement and actions are equivalent for each of these cases. For the first case, all afterstates with two 2 tiles have value 50724.26, and all afterstates with one 4 tile (by merging two 2 tiles) have value 50720.26, where by adding an immediate reward of 4 by merging tiles we have the same expected score. For the second case, all initial placement and actions are equivalent with value 50720.62. This means that there is an about 3.64-point disadvantage to picking a tile 4 in an initial position. For the last case, the value is 50716.99 without merges, or 50708.99 after obtaining reward 8 by merging two 4 tiles, for all states. In this case, disadvantage is roughly 7 points. Therefore, in summary, having tile 4 at an initial state has a small disadvantage in the expected score.

Fig. 9 shows the average number of empty cells at each age. Interestingly, the values are rather stable under 1.5 except for the very beginning of a game, e.g., about 2.01 at age 120. Note that there must be at least one empty cell according to the game rules.

As a potential application of the obtained values, we additionally investigated the difference between optimal and sub-optimal actions. Fig. 10 shows the mean difference for each age. There are clear peaks near age 2048 and 4096, indicating the importance of the first several actions after obtaining a tile with large numbers. It would support empirical understanding of why separation of evaluation functions for stages worked well in past studies, Wu et al. (2014); Jaskowski (2017). Relatively large differences in general indicate that an agent need to follow the best action in most cases to achieve a good score. This observation is consistent with the frequency of afterstates where the difference is less than 1.0 shown by the dotted line (blue) in Fig. 8.

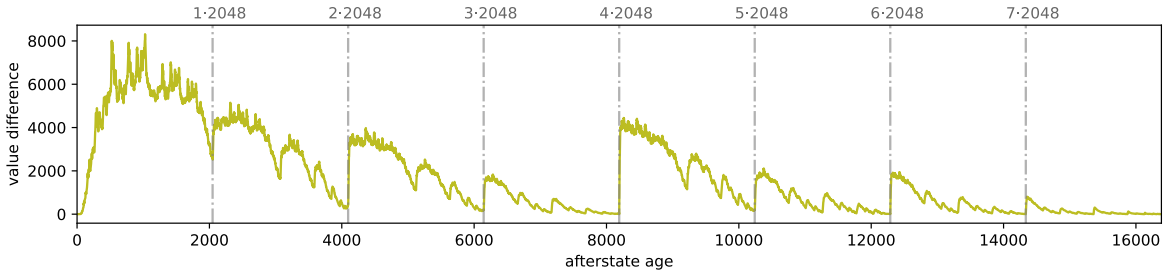


Figure 10: Mean difference in values between optimal action and suboptimal one.

5 Conclusion

This paper presented the solution for $2048_{4\times3}$, a 4×3 variant of *2048*, which is a stochastic single-player game. In this variant, the expected score achieved by an optimal strategy isabout 50724.26 for the most common initial states with two tiles of number 2 and few points lower for other initial states. The number of afterstates was $739,648,886,170$, and all IDs were stored in about 1.4 TiB. The key technique enabling $2048_{4\times3}$ to be solved is to leverage a game-specific property of *age*, the sum of the numbers on a board. Age is invariant between a state and its successive afterstate after any valid action and is increased two or four by stochastic response from the environment. State (and also afterstate) spaces are clearly partitioned by age, enabling a huge reduction in memory usage. We also adopted a compact representation of state ID set on the basis of Elias-Fano code, which will hopefully work well in holding state IDs in other games where ranking/unranking functions are not known. By using these techniques, the optimal value function of $2048_{4\times3}$ is now available for researchers with decent computational resources, about 4 TiB ssds and few days of computation with a modern CPU. Also, data specialized for optimal playing only requires about 300 GiB. The authors believe that the dataset in $2048_{4\times3}$ as well as core ideas will serve as a baseline of further studies.

References

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

Ioannis Antonoglou, Julian Schrittwieser, Sherjil Ozair, Thomas K Hubert, and David Silver. Planning in stochastic environments with a learned model. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=X6D9bAHhBQ1.

Shuhei Yamashita, Tomoyuki Kaneko, and Taichi Nakayashiki. Strongly solving 2048 on 3×3 board and performance evaluation of reinforcement learning agents. In *Game programming workshop*, pages 1–8. IPSJ, 2022. (in Japanese).

Kiminori Matsuzaki and Shunsuke Terauchi. Yet more optimistic temporal difference learning for game mini2048. In *2024 IEEE Conference on Games (CoG)*, pages 1–8, 2024. doi: 10.1109/CoG60054.2024.10645595.

Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, 2nd edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

H. Jaap van den Herik, Jos W. H. M. Uiterwijk, and Jack van Rijswijck. Games solved: now and in the future. *Artif. Intell.*, 134(1-2):277–311, 2002. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/S0004-3702(01)00152-7.

Jonathan Schaeffer, Neil Burch, Yngvi Bjornsson, Akihiro Kishimoto, Martin Muller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, pages 1144079+, 7 2007. doi: 10.1126/science.1144079. URL http://dx.doi.org/10.1126/science.1144079.

Hiroki Takizawa. Othello is solved. *CoRR*, 2310.19387, 2023. URL http://arxiv.org/abs/2310.19387.

John Tromp. The number of legal go positions. In Aske Plaat, Walter Kosters, and Jaap van den Herik, editors, *Computers and Games*, pages 183–190, Cham, 2016. Springer International Publishing. ISBN 978-3-319-50935-8.

Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 2017. ISSN 0036-8075. doi: 10.1126/science.aam6960.

Noam Brown and Tuomas Sandholm. Libratus: The superhuman ai for no-limit poker. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 5226–5228, 2017. doi: 10.24963/ijcai.2017/772. URL https://doi.org/10.24963/ijcai.2017/772.

Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016. ISBN 9781316792926.

Hans-Peter Lehmann, Thomas Mueller, Rasmus Pagh, Giulio Ermanno Pibiri, Peter Sanders, Sebastiano Vigna, and Stefan Walzer. Modern minimal perfect hashing: A survey. *CoRR*, abs/2506.06536, 2025. URL https://arxiv.org/abs/2506.06536.

Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. Fast Succinct Retrieval and Approximate Membership Using Ribbon. In Christian Schulz and Bora Uçar, editors, *20th International Symposium on Experimental Algorithms (SEA 2022)*, volume 233 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:20, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-251-8. doi: 10.4230/LIPIcs.SEA.2022.4. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SEA.2022.4.

I-Chen Wu, Kun-Hao Yeh, Chao-Chin Liang, Chia-Chuan Chang, and Han Chiang. Multi-stage temporal difference learning for 2048. In Shin-Ming Cheng and Min-Yuh Day, editors, *Technologies and Applications of Artificial Intelligence*, volume 8916 of *Lecture Notes in Computer Science*, pages 366–378. Springer International Publishing, 2014. ISBN 978-3-319-13986-9. doi: 10.1007/978-3-319-13987-6_34. URL http://dx.doi.org/10.1007/978-3-319-13987-6_34.

W. Jaskowski. Mastering 2048 with delayed temporal coherence learning, multi-stage weight promotion, redundant encoding and carousel shaping. *IEEE Transactions on Computational Intelligence and AI in Games*, PP(99):1–1, 2017. ISSN 1943-068X. doi: 10.1109/TCIAIG.2017.2651887.

A Implementation details

In the main experiments, the forward and backward computation require about 25 and 40 hours, respectively, implemented in C++ ran on a Linux server with AMD Ryzen 9 7950X 16-Core Processor. The memory usage was about 32 GiB for full (16 cores) parallelization. Binaries are stored via a `mmap` system call on Linux, so, they are machine dependent. For detecting bit errors, we used PicoSHA2 to obtain hash digests with SHA256 for state IDs. Also, the consistency of the results was confirmed by three independent runs of the whole forward and backward procedures.

Here are some details on our forward and backward functions listed in Figs 4 and 5 in the maintext. The first function in Fig 11 illustrates how `make_state_after_action` is implemented, borrowing Python syntax for simplicity. The function produces a sequence of afterstate IDs reachable from state IDs given in `state_seq`. The outer `for` loop in line 5 may run in parallel with changing output variable `after_state_seq` thread-local. The outputs from all threads are then merged into a single sequence and sorted while removing duplicates. To remove duplicates, IDs should be normalized to resolve symmetries in advance. Similarly, the second function `spawn` takes a sequence of afterstate IDs and outputs successor state IDs. Here, two sequences are separately maintained in the output for spawn of tile 2 and tile 4. The outer `for` loop in line 20 may run in parallel, similarly.

In the backward path, two functions (`accumulate_action` and `accumulate_spawn`) compute values of states and afterstates, respectively, one half-step from an older age to a younger one. Fig. 12

```python
def make_afterstate(state_seq):
    '''make set of afterstates reachable by single action'''
    afterstate_seq = []
    # run in parallel
    for state in state_seq:
        for action in legal_action(state):
            child = state.make_move(action)
            afterstate_seq.append(normalize(child))
    # join and merge outputs here
    return unique(afterstate_seq)

def spawn(afterstate_seq, state_2, state_4):
    '''spawn tile of 2 or 4 to store in state_2 or state_4
    params:
    - afterstate_seq: afterstates of current age
    - state_2: output variable for states with current age + 2
    - state_4: output variable for states with current age + 4
    '''
    # run in parallel
    for afterstate in afterstate_seq:
        for cell in afterstate.empty_cell():
            child2 = afterstate.place(cell, 2)
            child4 = afterstate.place(cell, 4)
            state_2.append(normalies(child2))
            state_4.append(normalies(child4))
    # join and merge outputs here
    state_2 = unique(state_2)
    state_4 = unique(state_4)
```

Figure 11: Miscellaneous functions used in `forward` (borrowing Python syntax)

gives pseudocode for `accumulate_action` that identifies all values of states taking input of successor afterstates and their values as sequences ordered consistently. For each state, action value `q` is identified as the sum of immediate reward `reward` by merging tiles (if any) and afterstate's value. Function `rank` of `after_state` in the inner loop is given by `std::lower_bound` leveraging increasing order of `afterstate_seq`. The outer `for` loop at line 5 may run in parallel similarly as functions above. Function `accumulate_spawn` is similarly defined except for taking the expectation instead of the maximum.

The disk space required for afterstates was about 1.4 TiB and depends on conditions for values. Given that the game involves probability 0.1 at each step and may reach more than 8000 steps, it is reasonable to introduce approximation by floating point numbers. Therefore, we used the standard 64bits for in-memory computation. In storing those values, we simply kept precision of 2^{-16} by scaling 2^{16} before storing as integer. Considering the integer part of all values presented in 15 bits or less for most ages as shown in Fig. 7, this simple approach required about 2.56 TiB total. The scaling factor can be adjusted to balance between precision and space. For example, with precision 2^{-8}, storage is about 1.89 TiB. Note that one needs only 300 GiB when specialized for optimal playing, as introduced in Sect. 3.2.3. The authors' implementation is available at `https://github.com/tkaneko/db2048-4x3`.

```python
def accumulate_action(state_seq, afterstate_seq, afterstate_values):
    '''make set of afterstates reachable by single action'''
    assert len(afterstate_seq) == len(afterstate_values)
    values = [0 for _ in range(len(state_seq))]
    # run in parallel
    for i, state in enumerate(state_seq):
        maxq = 0
        for action in legal_action(state):
            afterstate, reward = state.make_move(action)
            idx = rank(afterstate_seq, afterstate)
            q = reward + afterstate_values[idx]
            maxq = max(q, maxq)
        values[i] = maxq
    # join here
    return values
```

Figure 12: Miscellaneous functions used in `backward` (borrowing Python syntax)