AI algorithms for the game 2048

Hadi Pouransari, Saman Ghili

1 Task definition

The purpose of this project is to write an AI solver for the game 2048, with a reasonable runtime (i.e., less than about a second per each move).

2 Game description

Here we briefly describe the game and its rules:

The game consists of 16 squares (4 rows and 4 columns). Each square is either empty or has a numbered tile with value 2^k for some $k \in \mathbb{N}$.

Initially, the board is randomly loaded with one or two tiles with values 2 or 4.

At each turn, the player can move all the tiles on the board in one direction (up, down, left, or right). As a result, tiles slide as far as possible in the chosen direction until they are stopped by either another tile or the edge of the grid. If two tiles with the same value collide while moving, they will merge into a tile whose value will be the sum of the two individual tiles. However, each during each turn each tile can merge only once. For example, consider a row with values [2 2 2 2], and the action `left`. This results in a row with tiles [4 4 0 0].

The score is initialized to zero. At each turn the score is incremented by the sum of the values of the merged tiles. For instance if two tiles with values 8 merge, and create a new tile with value 16, then score will be implemented by 16.

After each move, a tile with the value 2 or 4 (chosen randomly) will appear in one of the empty squares.

The game ends when the board is full with tiles, and there are no more possible merges. In other words, when the board cannot be changed by taking any of the four actions.

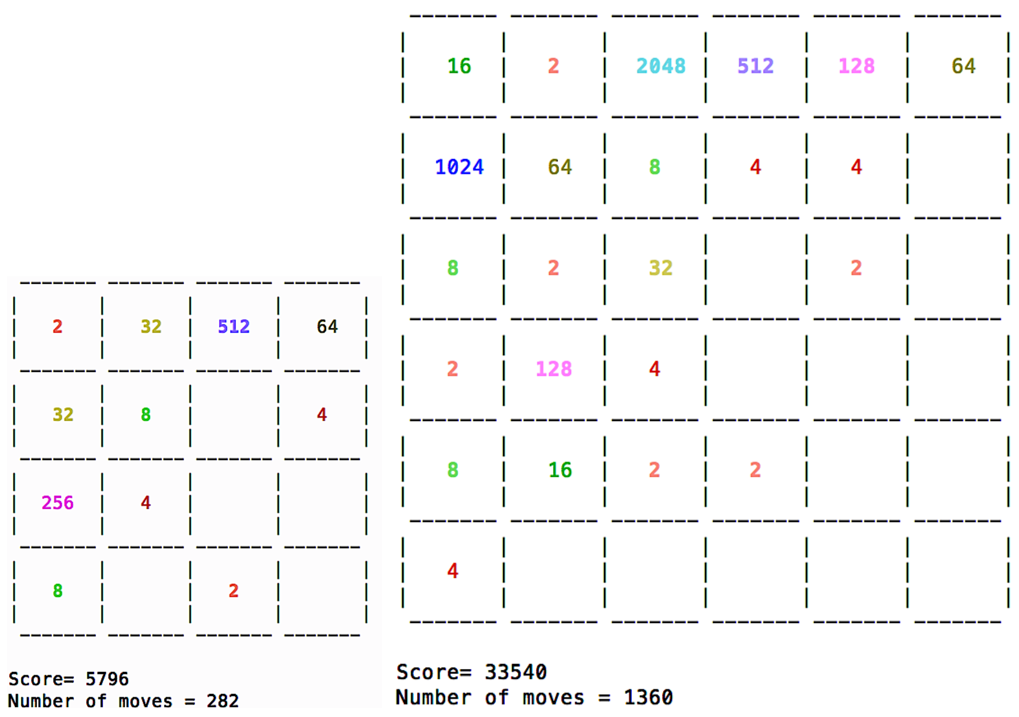
3 Modeling approaches

We have modeled this game as a state-based two-player AI problem. Each state is a particular configuration of the tiles on the board (i.e., a tuple s with length 16 consisting of values of the tiles on the board). There are two agents playing against each other: the human agent, and the computer agent. Possible actions for the human agent at each state s are `actions`$(s) =$ {`up` , `down`, `left`, `right`}. A legal action for the Computer agent is to choose an empty square and put a tile with value either 2 or 4 on that square. The successor state after application of an action obtains based on the above definition of the game. Naturally, the reward of each (s, a, s') is the amount by which the score is incremented, where s is the previous state, a is the taken action by the human agent, and s' is the resulting state (after tiles slide in the direction of a, colliding tiles with the same value merged, and a random tile was added on the board). Note that the actions of the computer agent have zero reward. In the original game the computer's policy is random. This leads to an *expectimax* problem.

4 Implementation

Based on the above modeling, we have implemented the game from scratch in Python. Here, we briefly explain the implementation.

- `class State`: This class stores one state of the game, that is the values of the tiles on the board (i.e., n^2 numbers for a general $n \times n$ board), and the score. In the following we list the important member functions of this class that provide the required framework for our AI algorithm implementation.
  - function `isGameOver()`: This function returns `True` when the player has no more actions, i.e., the game is over.
  - function `generateSuccessor( agentIndex, action )`: This function gets an agent index and its action, and generates the state resulted from this action. When the input agent is human, this function uses another function `pushLeft()` to generate the new state (and compute the resulting reward, which will be added to the score). Basically, if we assume the given action is `left`, we only need to write the update rule for one row, and repeat it n times. For other directions (i.e., `up, right,` and `down`), we first rotate the board accordingly, then apply `pushLeft()`, and finally rotate back.
  - function `getLegalActions( agentIndex )`: This function gets an agent index and returns a list of legal actions of the agent based on the current state. The legal actions for the human agent is any action from the set { `up , down, left, right` }, which changes the game state. The legal actions for the computer agent is a tuple of the form `( (i,j), v )`, where, `(i,j)` is an empty square on the board, and `v` is either 2 or 4 In general the agent can be the `human` agent which has actions { `up , down, left, right` }, or the computer agent with actions of the form { `((i,j), v)` }
- `class Agent`: This is a virtual class which provides a common framework for various agents. An agent represents either a human with index 0, or a computer with index 1. Each class of agent has a member function `getAction( state )`, which takes a game state as input, and based on the agent index chooses one of the actions from the list of legal actions.
- `class Game`: This class keeps track of the game, including the current state and the agents (a human and computer agents). It runs the game until the game is over.
- `class ColorPrint`: This is a simple text based graphic interface to show the board at each turn. In Figure 1 a snapshot of the output of the code is illustrated for two different board sizes.



```
 -------- -------- -------- --------
|        |        |        |        |
|   2    |   32   |  512   |   64   |
|        |        |        |        |
 -------- -------- -------- --------
|        |        |        |        |
|   32   |   8    |        |   4    |
|        |        |        |        |
 -------- -------- -------- --------
|        |        |        |        |
|  256   |   4    |        |        |
|        |        |        |        |
 -------- -------- -------- --------
|        |        |        |        |
|   8    |        |   2    |        |
|        |        |        |        |
 -------- -------- -------- --------

Score= 5796
Number of moves = 282
```

```
 -------- -------- -------- -------- -------- --------
|        |        |        |        |        |        |
|   16   |   2    |  2048  |  512   |  128   |   64   |
|        |        |        |        |        |        |
 -------- -------- -------- -------- -------- --------
|        |        |        |        |        |        |
|  1024  |   64   |   8    |   4    |   4    |        |
|        |        |        |        |        |        |
 -------- -------- -------- -------- -------- --------
|        |        |        |        |        |        |
|   8    |   2    |   32   |        |   2    |        |
|        |        |        |        |        |        |
 -------- -------- -------- -------- -------- --------
|        |        |        |        |        |        |
|   2    |  128   |   4    |        |        |        |
|        |        |        |        |        |        |
 -------- -------- -------- -------- -------- --------
|        |        |        |        |        |        |
|   8    |   16   |   2    |   2    |        |        |
|        |        |        |        |        |        |
 -------- -------- -------- -------- -------- --------
|        |        |        |        |        |        |
|   4    |        |        |        |        |        |
|        |        |        |        |        |        |
 -------- -------- -------- -------- -------- --------

Score= 33540
Number of moves = 1360
```

Figure 1: Snapshots of the game board for $n = 4$ (left), and $n = 6$ (right).

Results

In this section, we discuss the result of our game simulations for different agents. The computer agent in all cases is assumed to be a random agent that picks an empty tile (with uniform random distribution), and put the value 2 (with probability 90%), or 4 (with probability 10%) on the tile.

4.1 Random agent

The human random agent, at each step of the game picks a legal action randomly (with uniform distribution). In Figure 2 the probability distribution functions of the logarithm of the maximum tile value on the board, at the end of the game, are plotted. The other statistics of the random agent is summarized in table 1.

4.2 Reflex agent

The simplest improvement to the random agent is to implement the reflex agent. Reflex human agent at each turn considers all of the legal actions, and computes the corresponding score for each action. It chooses the action with maximum score afterwards. In Figure 3 the probability distribution functions for the reflex agent are plotted. Evidently, the reflex agent has a better performance compared to the random agent; however, still not promising.

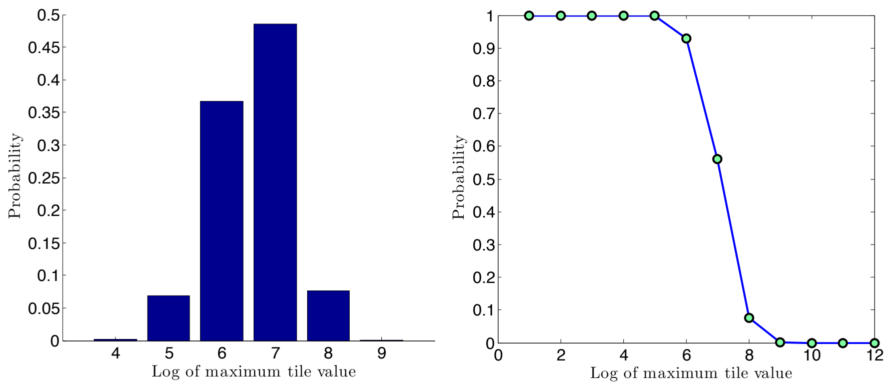


Figure 2: Probability density function (PDF: left) and complementary cumulative distribution function (CCDF: right) for the logarithm of the maximum tile value on the board at the end state, with **random** human agent.

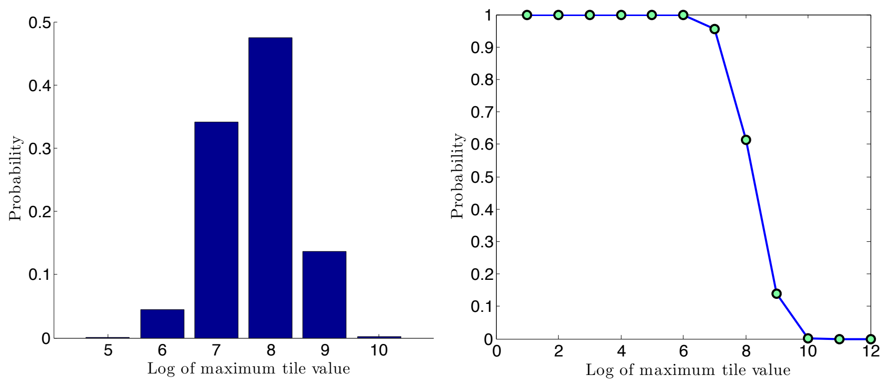


Figure 3: Probability density function (PDF: left) and complementary cumulative distribution function (CCDF: right) for the logarithm of the maximum tile value on the board at the end state, with **reflex** human agent.

4.3 Expectimax agent

Perhaps the most natural agent for this game is an expectimax agent. A human expectimax agent recursively considers all legal actions, followed by every possible action for the computer agent. This results in a search tree. In this problem, at each turn the human agent has at most 4 legal actions. The number of legal actions for the computer agent is 2 times the number of empty cells on the board. Therefore, the expectimax search tree grows very rapidly, and the computation becomes very expensive. We considered a limited depth search tree that starts with a human action, and also ends after a human action. This allows us to consider more actions for the human agent for a given depth. For instance, Figure 4 shows a search tree with depth two. A limited depth expectimax search tree requires an evaluation

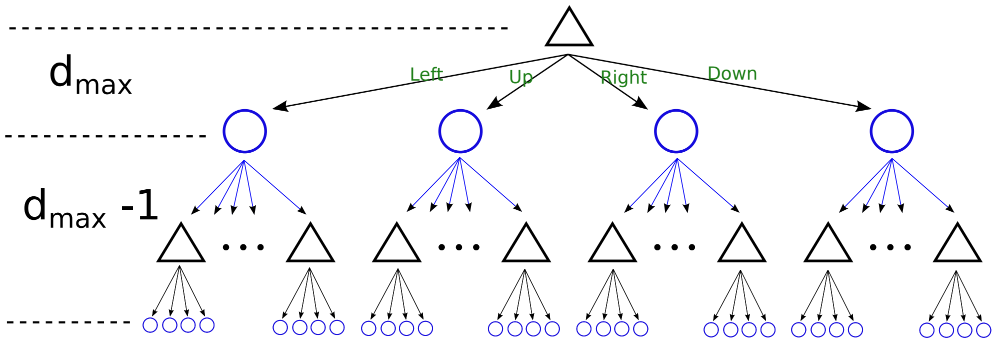


Figure 4: The expectimax search tree.

function to score the leaf node states. The simplest evaluation function for a given state is the score. The results of the human expectimax agent with search depth 2, and game score evaluation function is demonstrated in Figure 5. As mentioned in table 1, despite better performance, the expectimax algorithm has a much longer running time (about 100 times slower compared to the reflex agent).

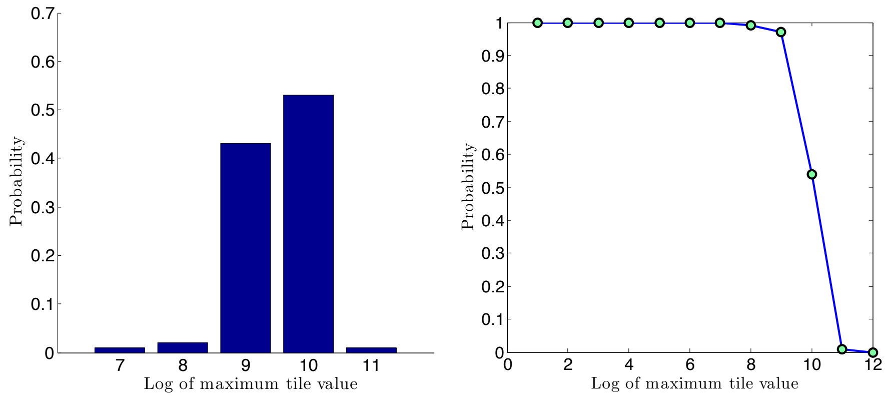


Figure 5: Probability density function (PDF: left) and complementary cumulative distribution function (CCDF: right) for the logarithm of the maximum tile value on the board at the end state, with **expectimax** human agent with **depth 2** and **score** evaluation function.

4.4 Expectimax agent with feature

Using game score as the state evaluation function is reasonable; however, is not using any domain specific knowledge of the game. Using game score as the evaluation function needs a very deep search in order to be effective. In this section we are going to define a specific evaluation function that helps the human agent to follow a particular game strategy.

Paying attention to the game rules, one can realize that it is more efficient to keep tiles with close values adjacent to each other. We want to define an evaluation function that enforces the human agent to keep tiles on the board with an S-shaped order. To achieve this, we defined a 4×4 weight matrix as shown in Figure 6. To evaluate the score of each state we compute the *inner product* of the board values and the weight matrix. For instance, the score of the board shown in Figure 1 (left) is:

$$2\times\mathbf{4^{15}}+32\times\mathbf{4^{14}}+512\times\mathbf{4^{13}}+64\times\mathbf{4^{12}}+4\times\mathbf{4^{11}}+8\times\mathbf{4^{9}}+32\times\mathbf{4^{8}}+256\times\mathbf{4^{7}}+4\times\mathbf{4^{6}}+4\times\mathbf{4^{2}}+8\times\mathbf{4^{0}}$$

This weight enforces the agent to keep the tiles with S-shaped order (based on their values) on the board. Note that we could choose any value greater than 2 as the basis in the definition of our weight matrix. The higher the basis is the less flexibility the agent has in terms of the order of tiles on the board. We investigated the performance of the expectimax

4^{15}	4^{14}	4^{13}	4^{12}
4^{8}	4^{9}	4^{10}	4^{11}
4^{7}	4^{6}	4^{5}	4^{4}
4^{0}	4^{1}	4^{2}	4^{3}

Figure 6: The weight matrix used in the definition of the evaluation function.

human agent, with depth 2, and using the S-shaped feature vector rather than the game score. The probability distribution functions for this agent are shown in Figure 7. Clearly, an agent with the S-shaped feature evaluation function outperforms the agent with score game evaluation function. It has 79% chance to reach to a tile with value 2048, while similar agent with score game evaluation function only had 1% chance. Also, note that using the S-shaped evaluation function does not increase the run time of the program (see table 1).

4.5 Pruned expectimax agent with feature

The performance of the expectimax agent can be improved simply by increasing the depth of the search. However, the runtime of the algorithm increases exponentially with the depth of the search tree. We would like to have our algorithm running in a reasonable time (i.e., less than about a second per move). One way to reduce the runtime of the expectimax algorithm is to prune some of the actions. At each turn the computer agent has too many legal actions. For a given maximum runtime, it is not worth it to consider all of the actions. For instance, when there are many empty cells on the board, we do not need to consider all of them.

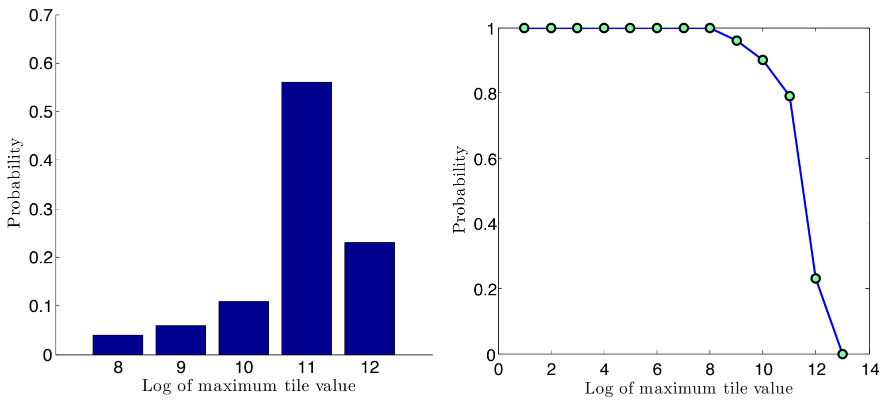


Figure 7: Probability density function (PDF: left) and complementary cumulative distribution function (CCDF: right) for the logarithm of the maximum tile value on the board at the end state, with **expectimax** human agent with **depth 2** and **S-shaped** evaluation function.

Instead we can only consider the computer actions that are worse. Since we are using the S-shaped feature as our evaluation function, those actions of the computer agent that results in a new tile on the cells with larger weights are worse. Also, we can only consider new tiles with value 2 (rather than both 2 and 4).

Based on the above explanation we proposed the following pruning: At the first level of the search tree consider only first 4 worst actions of the computer action. In the next level only consider first 2 worst actions of the computer action. After the second level only consider 1 (the worst) action for the computer agent. Note that we consider all of the human legal actions at every level.

Having the above pruning idea implemented, we can consider a deeper search tree, while the runtime of the algorithm remains reasonable. In Figure 8 the probability distribution functions for the pruned expectimax human agent with depth 3, and S-shaped evaluation function are shown. Performance is greatly increased compared to the previous agent, whereas the runtime is about only twice (see table 1). We also extended the search depth to $d_{\max} = 4$ using the pruning idea. The corresponding results are shown in Figure 9 and table 1.

Agent	$d_{\max}$	**Eval. func.**	**# runs**	**mSec./step**	**Top max**	**Top score**	$\mathbb{P}(2048)$
random	0	(-)	10,000	1.2	512	4.6k	0%
reflex	1	score	10,000	1.6	1024	12.7k	0%
expectiMax	2	score	100	145	2048	22.6k	1%
expectiMax	2	S-shape	100	142	4096	76k	79%
prunExpectiMax	3	S-shaped	100	370	8192	102k	89%
prunExpectiMax	4	S-shaped	100	1492	8192	152k	94%

Table 1: Summary of performance of different agents.

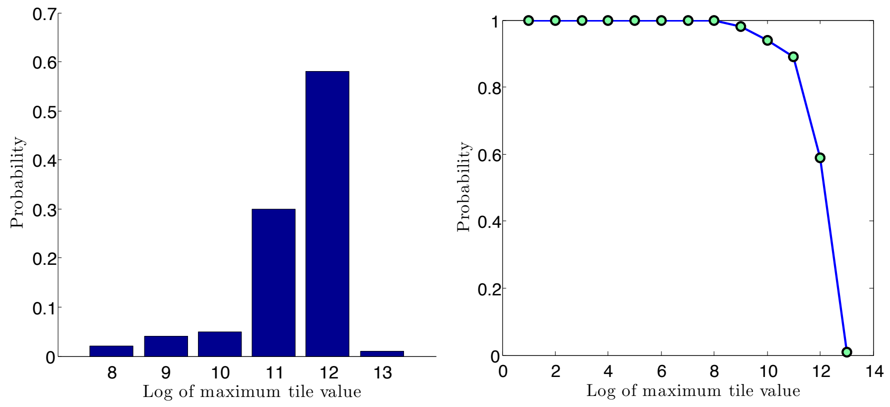


Figure 8: Probability density function (PDF: left) and complementary cumulative distribution function (CCDF: right) for the logarithm of the maximum tile value on the board at the end state, with **pruned expectimax** human agent with **depth 3** and **S-shaped** evaluation function.

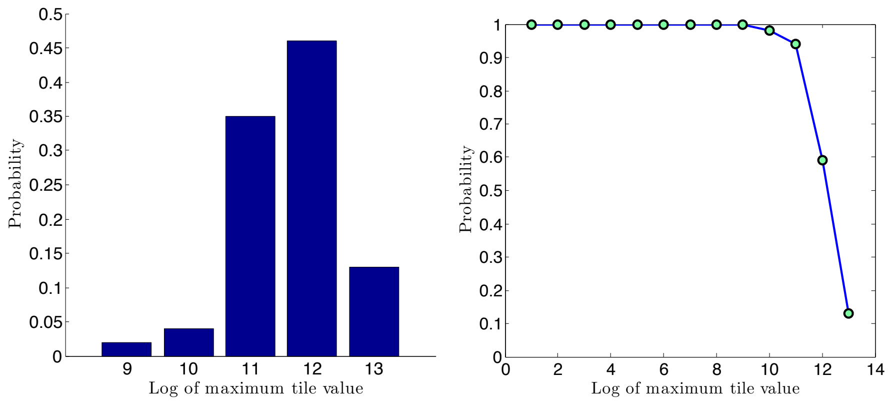


Figure 9: Probability density function (PDF: left) and complementary cumulative distribution function (CCDF: right) for the logarithm of the maximum tile value on the board at the end state, with **pruned expectimax** human agent with **depth 4** and **S-shaped** evaluation function.

5 Conclusion and future work

In this project we implemented the game 2048 in python, and applied various AI analysis to that. We investigated a series of agents, and improved the performance of the human agent gradually. We found that pruning some of the computer actions can accelerate the calculation, while the performance is not reducing a lot. In addition, we introduced an evaluation function based on our domain specific knowledge of the game. This improved the performance of the expectimax agent greatly. Our best agent is able to reach a tile with value 2048 with 94% probability. In fact, in most of the cases that it reaches 2048, it has reached 4096 as well (see Fig. 9). Note that, our code is general for any agent. Increasing the depth of the search results in even better results (but the run time increases).

There are couple of possible extensions for this work.

In order to be able to use higher depth search we can optimize the implementation: use C++, hash a table of the merged columns (so we do not need to compute the resulted merged state after every human action), present the state with one 64-bit number (4 bit per cell, representing the log of the tile value on the cell) rather than 16 integer numbers (current implementation).

Also, since our code is general ($n \times n$ board), one can study the similar statistics for different values of n. Here, we only considered the original 4×4 case.

As a variation, one can consider the scenario where the computer is trying to minimize the player's score. This leads to a *minimax* problem.

One interesting variation of the problem is to change the goal state to any state with a tile with value 2048. In this variation, the objective is to reach a goal state (i.e., the first time that a tile with value 2048 appears on the board) with as few actions as possible. This variation can be modeled as a non-deterministic search problem (assuming the location and the value of the new tile added at each step are random). One way to deal with this non-deterministic problem with a large state-space is to look for a stochastic variation of the A^* algorithm.