AI Plays 2048

Yun Nie (yunn), Wenqi Hou (wenqihou), Yicheng An (yicheng)

***Abstract*—The strategy game 2048 gained great popularity quickly. Although it is easy to play, people cannot win the game easily, as they do not usually take as many as future possibilities into account. In this project, we came up with several search trees strategies (Minimax, Expectimax) and an attempt on reinforcement learning to achieve higher scores. Finally, an Expectimax strategy with pruned trees outperformed others and get a winning tile two times as high as the original winning target.**

I. Introduction

Game 2048 is a popular single-player video game released by Italian programmer Gabriele Cirulli. Specifically, the game is played on a $4 * 4$ grid board, with each tile holding an even number(Fig.1). Player can swipe on the board in four directions: up, down, left and right; then all tiles will move in that direction until stopped by another tile or the border of the grid. If two tiles of the same number collide while moving, they will merge into a new tile bearing their summation. The new tile cannot merge with another neighbor tile again during this move. After the move, a new tile with number 2 or 4 will randomly appear on one of the empty tiles, and then, player makes a new move.

When there is no empty cell and no more valid move, the game ends. If a tile with number 2048 appears in the board before the game ends, the player wins. A scoreboard keeps track of the user's score. The user's score starts at zero, and is incremented whenever two tiles combine, by the value of the new tile. As with many arcade games, the user's best score is shown alongside the current score.

Our goal for this project is to build up an AI that could play the game automatically and maximize the game score within a reasonable running time. Since the maximum score of each potential method is unpredictable, we decided to break the 2048 winning ceiling, and implement several methods simultaneously so that we could make a comparison between them and choose the best one.

II. Related Works

Since the launch of this game, a lot of game lovers share their AI commitments, with methods from simple Expectimax all the way down to complex reinforcement learning. The two most popular and effective strategies are:

1) Minimax with two main heuristics (monotonicity and smoothness), achieving a winning rate for 2048 tile around 90%.
2) Expectimax with heuristics (number of open squares and heavy weights for large values on the edge), achieving highest tile of 32768 for 36% of the trials under maximum search depth of 8. This result is amazing, but a max depth as large as 8 would increase running time dramatically.

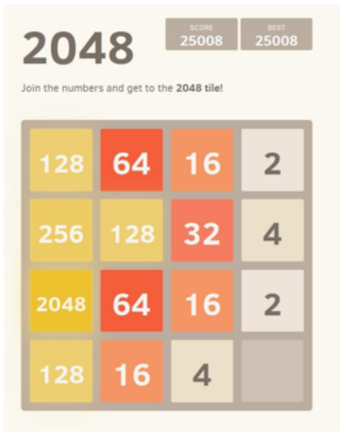


Fig. 1. Game Interface

III. Approaches

In the first stage, we implemented Minimax and Expectimax search trees and improved them with human heuristic. Later on, we pruned the trees because when the depth goes larger (beyond 3), the search trees would be exponentially expanded, and the running speed would be dragged down significantly. At last, we attempted reinforcement learning using discretization of states. However, since our discretization far from the actual number of possible states, which is computationally expensive for modern computers, the performance is obviously better than human players but not better than search trees.

A. Minimax

Minimax is a classic method to play a double-player game, players will take turns to play until the game ends. Here at 2048 game, the computer (opponent) side is simplified to a fixed policy: placing new tiles of 2 or 4 with an $8 : 2$ probability ratio. Before describing the specific math formulations of the algorithm, we have to clarify the following notations:

1) $players$: $\{agent, opponent\}$
2) s: the grid board that reflects the current state
3) a: action taken
4) $actions(s)$: possible actions at state s
5) $succ(s, a)$: resulting state if action a is chosen at state s
6) $isEnd(s)$: whether s is an end state (the game is over)
7) $eval(s)$: evaluation at state s
8) $player(s) \in players$: the player that controls state s
9) $totalScore(s)$: the score at the end of the game

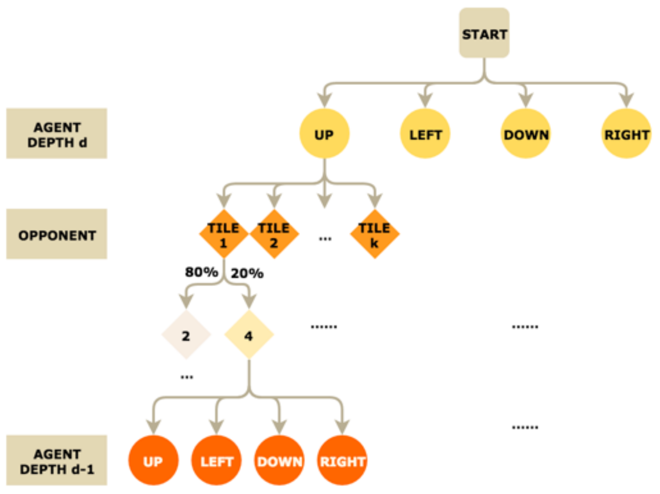


Fig. 2. Minimax Serach Tree

10) d: the current depth of the search tree

Then we could use these notations to represent the mathematical formula of Minimax, which is shown as below.

$$V_{max,min}(s,d) = \begin{cases} totalScore(s), isEnd(s) \\ eval(s), d = 0 \\ \max_{a \in actions(s)} V_{max,min}(succ(s,a),d), \\ (players(a) = agent) \\ \min_{a \in actions(s)} V_{max,min}(succ(s,a),d), \\ (players(a) = opponent) \end{cases}$$

During the game, the agent is our implemented AI, and the opponent is the computer. When the player is agent, the action $a \in \{up, down, left, right\}$, when the player is opponent, the action a will be randomly placing 2 or 4 into the empty tiles of the current board. In our implementation of the algorithm, the evaluation function is the current score when either reaching the maximum depth or coming to a game over. The search tree for the formula above is shown in Fig. 2.

B. Expectimax

The Expectimax algorithm is pretty the same as the minimax algorithm mentioned above, except that we add chance nodes between agent nodes and opponent nodes.

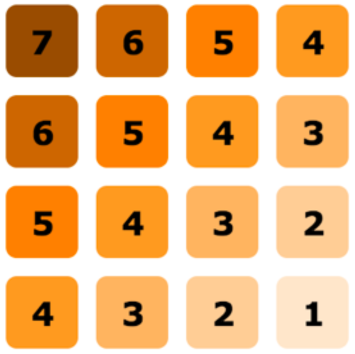
Then the pseudo code for the algorithm would be like this: Besides we added heuristics and domain knowledge into our evaluation function when AI reaches the last depth level, or when game is over:

$$eval(s) = \sum_{i=0}^{3}\sum_{j=0}^{3} weight[i][j] \times board[i][j]$$

From intuition, one can realize that putting tiles with close values near each other usually leads to more possible merges. For tiles with same number, a single swipe could make a merge; for tiles with different number, it is reasonable to place them in ascending or descending order, so that they could merge consecutively. Therefore, it seems attempting to use a weight matrix that has highest weight on one corner and is decreasing diagonally (Fig. 3). However, it did not largely discriminate tiles: difference weights of two tiles may be weakened by the large difference in numbers they bear. We then came up with weight matrix b that enlarges the range of weights. Another intuition would be to use a snake-shaped matrix which looks like Fig. 5, where the heuristic is that we expect swipes and merges along the snake to put larger number in the corner. After trying symmetric and snake-shaped weight matrices with various weight values, we found the following S-shaped weight matrix giving best performance.

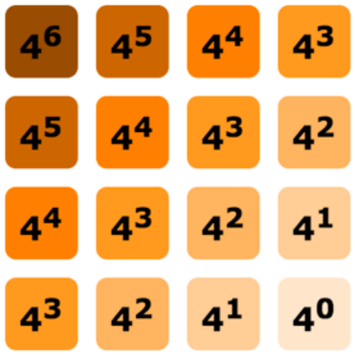
Algorithm 1 Expectiminimax

```
if state $s$ is a Max node then
    return the highest Expectiminimax-value of $succ(s)$
end if
if state $s$ is a Min node then
    return the lowest Expectiminimax-value of $succ(s)$
end if
if state $s$ is a Chance node then
    return the average Expectiminimax-value of $succ(s)$
end if
```



7	6	5	4
6	5	4	3
5	4	3	2
4	3	2	1

Fig. 3. Weight Matrix a



4^6	4^5	4^4	4^3
4^5	4^4	4^3	4^2
4^4	4^3	4^2	4^1
4^3	4^2	4^1	4^0

Fig. 4. Weight Matrix b

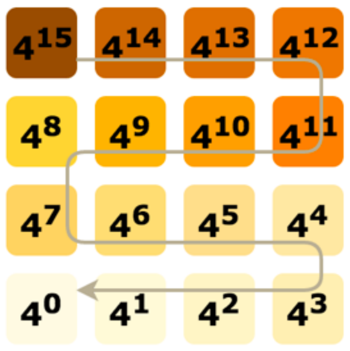


Fig. 5. Weight Matrix c (Snake-shaped)

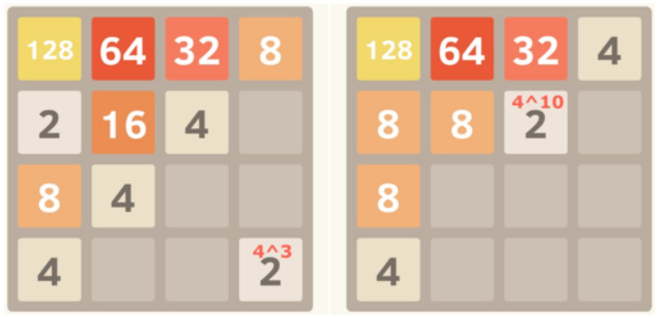


Fig. 6. Empty Tiles with Lower and Higher Weights

C. Pruning

As discussed earlier, greater depths lead to dimension explosion of search trees, and the need for pruning comes into being. Again leveraging the human heuristic of snake-shaped weight matrix, we consider only some most important empty tiles, where importance is proportional to the weight attached to a tile. Take the two situations in Fig. 4 as an example. A new generated tile with number 2 had higher importance in the second board as it occupied a tile with larger weight.

After several trials with the minimal value of $(depth, \texttt{num of empty tiles}, 4)$ or the minimal value of $(depth, \texttt{num of empty tiles}, 6)$, we discovered that to consider four or fewer empty tiles at each depth could better balance the need for both high score and running time. For example: for maximum depth of 4, the search tree considers no more than 4 empty tiles at first layer; then it considers no more than 3 empty tiles at one layer down; in the last but one layer it would consider at most one empty tile.

D. Reinforcement Learning

For the MDP in this game, we set policy $\pi^*(s) = argmaxP_{s,a}(s)V^*(s)$. With the winning tile of 2048, a board could have as many as possible states, let alone breaking the winning limit. Thus we tried to discretize the states by extracting important features of a given board:

1) Monotonicity of rows and columns
   Monotonicity could either be increasing or decreasing. To emphasize that we want to approximate a snake-shaped board, we check whether the first and third rows and columns are decreasing, and whether the second and last rows and columns are increasing. This results in 2^8 combinations.
2) number of empty tiles
   We deem it as an important feature, since it imposes a sort of reward or penalty on current board: the more freedom tiles, the more likely to last longer until death. The best situation is when game just begins and only two tiles are occupied and the other fourteen are free, while the worst is when the board is full and game is over. For other situation in between, we expect to see number of tiles in range $(0, 14)$.
3) number of possible merges
   This metric measures the efficiency of potential swipes from current to next state: the more merges, the more tiles to be freed. Even for boards whose monotonic characteristics and number of empty tiles are the same, their states could be largely different due to the number of merges available. A board that could merge tiles more than once in either direction is clearly more flexible and promising than a board that only moves but not merge tiles. Since each row or column allows at most merges, this metric should be an integer between 0 and 8, inclusive.

The above three features give $2^8 \times 15 \times 9 = 34560$ states, , which are less computationally expensive but also sacrifice some information of the board. We assigned the reward $R(s)$ according to the three board features as well, and implemented value iteration shown in following procedures until it consecutively converged for more than 15 times.

Algorithm 2 Reinforcement Learning

```
**for** each state $s$ **do**
  initialize $V(s) := 0$
**end for**
**while** no convergence yet **do**
  **for** each state $s$ **do**
    update $V(s) = R(s) + \max_{a \in A} \gamma \sum_{s'} P_{s,a}(s')V(s')$.
  **end for**
**end while**
```

The state transition probabilities in state s are estimated as:

$$P_{s,a}(s') = \frac{\texttt{\# times action } a \texttt{ leads to } s'}{\texttt{\# times action } a \texttt{ is taken}}$$

We use similar method for averaging R(s):

$$R(s) = \frac{\sum \texttt{reward of state } s' \texttt{ following } s}{\texttt{\# times of } s \texttt{ appearing}}$$

IV. Result Analysis

After 100 trials for each approach, we compared the above methods from multiple aspects (Table 1):

1) highest score achieved.
2) average score, which could be more meaningful because it reflects the overall performance of the algorithm in relatively large number of trials.

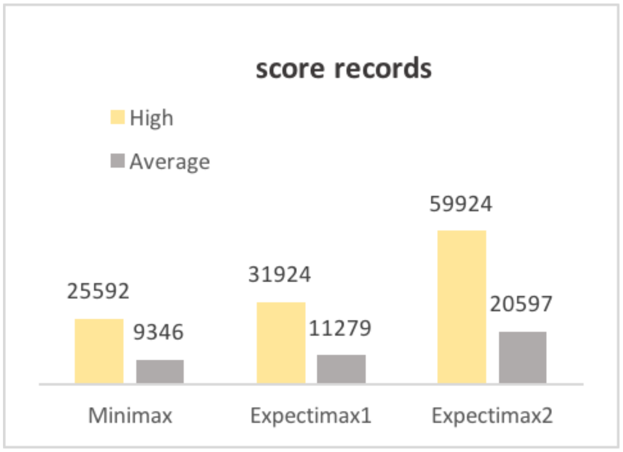


Fig. 7. Score Results

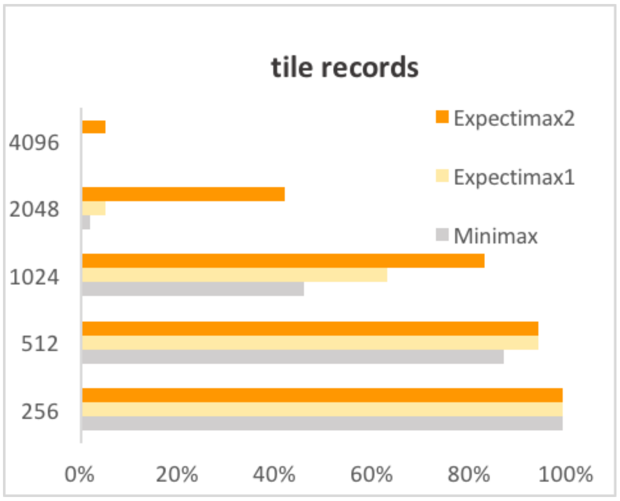


Fig. 8. High Tile Results

TABLE I
MINIMAX RESULTS

	Minimax		
Evaluation	score		
Depth	2	3	4
High Score	25592	34968	58732
Average Score	9346	17421	27250
Pruned	No	Yes	Yes
256	99%	100%	100%
512	87%	99%	100%
1024	46%	85%	98%
2048	2%	27%	71%
4096	-	-	4%

3) average runtime per move, as we all know generally the deeper depths we want to reach, the slower of the game would be, and it is unacceptable if one step would take too long.
4) statistics of each kind of number tile achieved, as it could reflect the stability of the algorithm: unstable algorithm may fail to ensure the lower tiles are reached before it happens to achieve high tile.

Larger depth, even with tree pruning, gave better performance than smaller depth, while pruning restrict running time per move in a reasonable level. Expectimax with snake-shaped heuristic weight matrix outperformed other strategies as it managed to keep larger numbers on the upper left part of the board. Overall, the percentages of various tiles achieved for each method were mono-decreasing, indicating that these strategies are quite stable in as they go deeper: they reach new high score occasionally but ensure to keep achieving lower tiles in rest of the time.

For reinforcement learning, current strategy of discretization and value iteration did not learn quickly and perform as well as search trees. One probable reason is the randomness embedded in the game: the game engine randomly chooses a 2 or 4 tile to appear at any empty spot on the game board.

TABLE II
EXPECTIMAX RESULTS

	Expectimax		
Evaluation	$\sum weight * value$		
Depth	2	3	4
High	31924	33940	59436
Average	11279	16766	21084
Pruned	No	Yes	Yes
256	99%	100%	100%
512	94%	99%	96%
1024	63%	84%	85%
2048	5%	27%	42%
4096	-	-	6%

TABLE III
EXPECTIMAX RESULTS 2

	Expectimax		
Evaluation	$score * \sum weight * value$		
Depth	2	3	4
High	59924	69956	**80728**
Average	20597	17770	29187
Pruned	No	Yes	Yes
256	99%	100%	100%
512	94%	97%	98%
1024	83%	86%	88%
2048	42%	44%	64%
4096	5%	8%	**16%**

TABLE IV
REINFORCEMENT LEARNING RESULTS

	High	Average	256	512	1024	2048
Rl	27744	5668	92.7%	59.6%	13.2%	0.7%

References

[1] https://github.com/adichopra/2048 (We used his 2048 GUI)