AI playing 2048

January 2023

Introduction to Artificial Intelligence

Phạm Tuấn Kiệt
20214909
Nguyễn Quang Tri
20210860
Nguyễn Thanh Bình
20210106
Hoàng Tú Quyên
20214929

Table of Contents

I Introduction

2048 game is a well-known 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; afterwards, all tiles will move in that direction until stopped by another tile or the border of the grid. When two moving tiles of the same number meet, they will combine to form a new tile that contains their total. During this movement, the new tile cannot combine with a neighboring tile once again. A new tile with the numbers 2 or 4 will at random emerge on one of the empty tiles after the move, and the player will then make a new 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. Before the game is over, if a tile with the number 2048 appears on the board, the player wins.

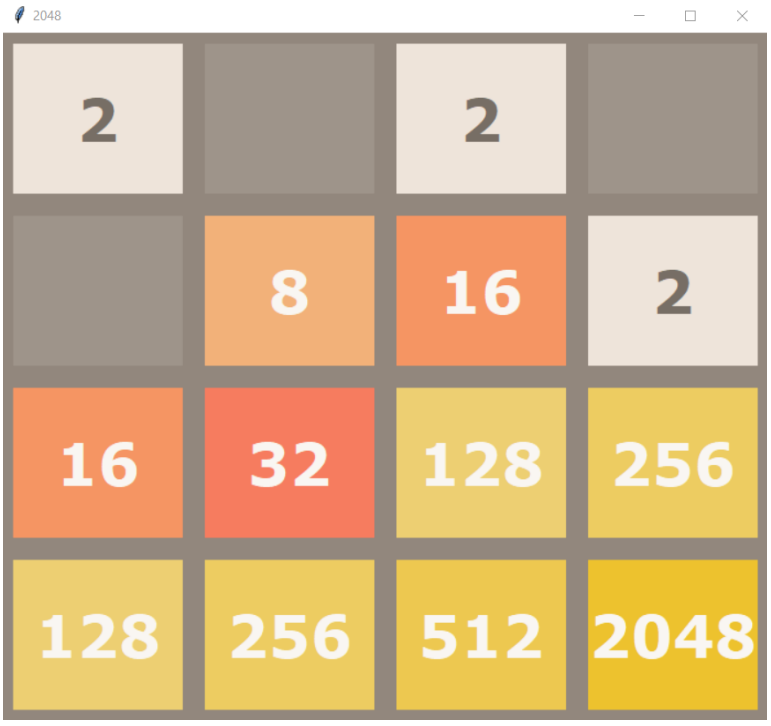

Fig. 1: 2048 Game Interface

The objective of this project is to develop an AI that can play the game automatically while maximizing scores in a manageable amount of time. Since the 2048 winning ceiling is quite easy to achieve, we decided to break the rule, to see the maximum score we can get. When there is no empty cell and no more valid move, the game ends. At the end of the game, the top 4 biggest tiles will be shown as the figure below(Fig.2).

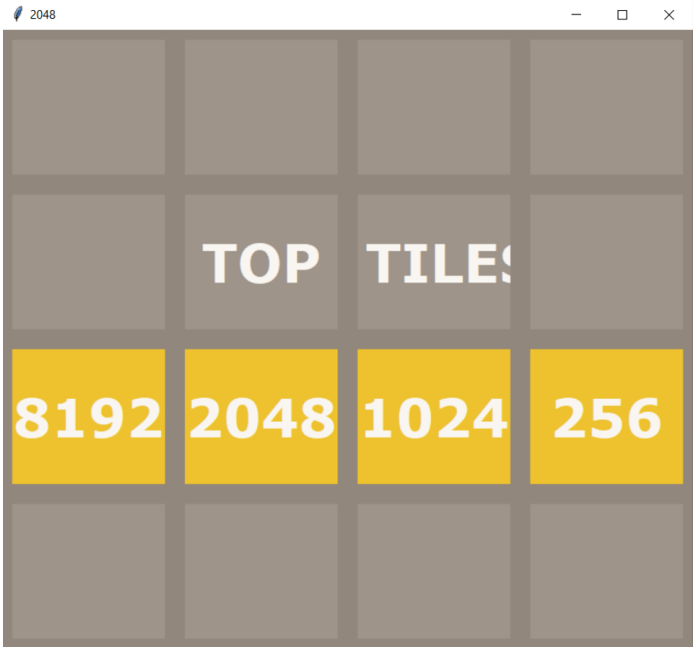

Fig. 2: 2048 Ending Display

Strategy

Maintaining the highest tile in a certain corner, keeping that tile in that corner, and filling the specified row with the highest numbers are all strategies used in the game. Some key rules could be used to win the 2048 game:

1. Use only two directions (as much as possible)
2. Never move your tiles up
3. Keep your tiles tidy
4. Focus on your goal
5. Aggressively combine downward and horizontally as needed

II Algorithm used

Expectimax

Why use this algorithm ?

The Expectimax search algorithm is a game theory algorithm used to maximize the expected utility. It is a variation of the Minimax algorithm. While Minimax assumes that the adversary (the minimizer) plays optimally, the Expectimax doesn't. This is useful for modelling environments where adversary agents are not optimal, or their actions are based on chance. Unlike deterministic games, stochastic ones have at least one probabilistic element. In our problem, the probability of 2 and 4 will be placed are 90% and 10%, respectively, so MIN is an imperfect player. Therefore, Expectimax is a potential search algorithm to be applied.

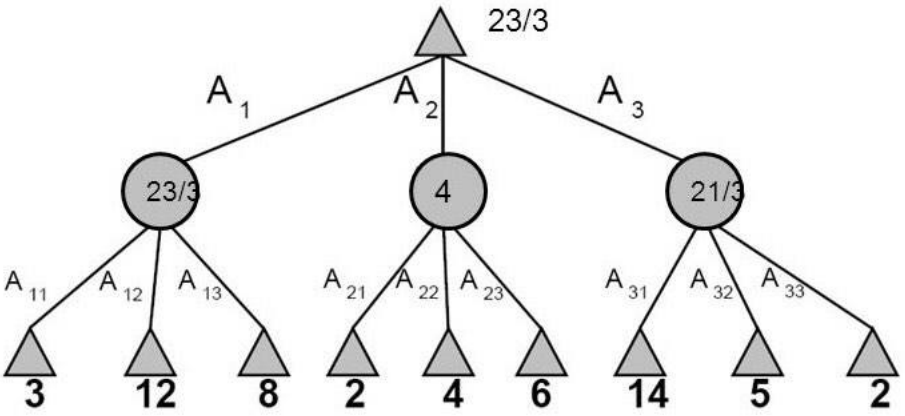

Fig. 3: Expectimax Diagram

Definition

- Chance nodes, like min nodes, except the outcome is uncertain.
- Calculate expected utilities (using EVAL).
- Max nodes as in minimax search.
- Chance nodes take the average (expectation) of value of children.

Properties

- In expectimax search, we have a probabilistic model of how the opponent (or environment) will behave in any state.
- The model might say that adversarial actions are likely!
- Expectimax algorithm helps take advantage of non-optimal opponents.
- But Expectimax is not optimal. It may lead to the agent losing (ending up in a state with lesser utility)
- Running time: Same as Minimax since we need to search all states

How it works

The pseudo-code for the Expectimax algorithm solving 2048 would be like this:

```
if state s is a Max node then
    return the highest Expectiminimax-value of Successors(s)

if state s is a Chance node then
    return the average Expectiminimax-value of Successors(s)
```

III Algorithm implementation

III.1 How to implement

We apply the Expectimax algorithm to determine the best direction for the agent to proceed in order to achieve the best state subsequently. The idea is to compute the board's score using an evaluate function. The backup value of a state will be determined by Max nodes(AI) and Chance nodes(computer). In terms of Max nodes, they choose the maximum value of Chance nodes below them. Other with Max nodes, the actions at Chance nodes are to place a possible tile of "2" or "4" in the tile that is still empty on the board, while the actions at Max nodes are 4 that correspond to 4 directions for each turn. For instance, if there are 4 empty tiles on the board, chance nodes have 8 possible actions or 4 possibilities to place tiles "2" and "4". The value of Chance nodes is the expected value of their child nodes. At each state, the probability to place a tile to an empty tile is likely to equal, so we divide the sum of all scores including tile 2 with a rate of 0.9 and 4 with a rate of 0.1 by the number of current empty spaces. We then iterate over every node in the graph with an altering depth (since if there are too many empty tiles on the board, there will be an enormous number of states to take into

consideration, which will take the agent too much time to compute. So, for the states with a lot of empty tiles, the depth limit is small. The inverse is true for those with few empty tiles. For the states that have more than or equal to 6 empty tiles, the depth is 2 and for those that have less than 6, the depth is 5).

The problem here is that whenever the Max or Chance nodes proceed, the original board's state could change. To solve this, you must first copy the board before letting the Max or Chance nodes make a move, and then you must return that board until you find the best move, which is the action that will benefit the agent the most.

III.2 Three functions of Evaluation

III.2.1 Overview

Evaluation (or Heuristic) is the task where the score for the board state is calculated. This score is different from the score the game uses; this score can be considered to be the signal that makes the AI know how good the board state is so that it can make the decision for the next move.

III.2.2 First version evaluation function: Weighted matrix

One of the most intuitive strategies when it comes to 2048 is to corner the large tiles. The larger the value is, the more difficult it becomes to perform the merging action. This action, on the one hand, reduces the possible moves to big-valued tiles. On the other hand, it increases the probability of merging cells when clustering large valued tiles at one corner. Therefore, it seems reasonable to assign different weights to each position. The arrangement of weights will be represented by a matrix. The evaluation function in this situation is written as

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

where S is the matrix representation of the grid at state s, M is the chosen weighted matrix.

In order to perform the clustering action, the weights have to be arranged such that the higher will be concentrated in one corner (assumingly the top-left corner) of the table. Intuitively, one such matrix can be given below:

$$\begin{bmatrix} 7 & 6 & 5 & 4 \\ 6 & 5 & 4 & 3 \\ 5 & 4 & 3 & 2 \\ 4 & 3 & 2 & 1 \end{bmatrix}$$

This matrix has values that decrease diagonally and higher values will be concentrated in the top-left corner. It is expected to corner the large tiles, but the collected data shows that this weighted matrix does not make any significant influences when it comes to practical tests. The large values may not be cornered and stay tuned at the center of the board. This unexpected result may come from the small differences between weights (always smaller than 6). The small distance between positions can not surpass the influence of large differences in the numbers they bear. Hence the weight matrix b that enlarges the range of weights appears to be recommendable.

$$\begin{bmatrix} 4^6 & 4^5 & 4^4 & 4^3 \\ 4^5 & 4^4 & 4^3 & 4^2 \\ 4^4 & 4^3 & 4^2 & 4^1 \\ 4^3 & 4^2 & 4^1 & 4^0 \end{bmatrix}$$

This performed pretty well when compared to a(shown in Result Analyst). Some other proposed arrangements can be given below

- Example of a snake-like matrix:

$$\begin{bmatrix} 4^{15} & 4^{14} & 4^{13} & 4^{12} \\ 4^8 & 4^9 & 4^{10} & 4^{11} \\ 4^7 & 4^6 & 4^5 & 4^4 \\ 4^0 & 4^1 & 4^2 & 4^3 \end{bmatrix}$$

- Example of an edge evaluation matrix:

$$\begin{bmatrix} 4^6 & 4^5 & 4^4 & 4^3 \\ 4^5 & 0 & 0 & 4^2 \\ 4^4 & 0 & 0 & 4^1 \\ 4^3 & 4^2 & 4^1 & 4^0 \end{bmatrix}$$

- Example of a spiral matrix:

$$\begin{bmatrix} 4^{15} & 4^{14} & 4^{13} & 4^{12} \\ 4^4 & 4^3 & 4^2 & 4^{11} \\ 4^5 & 4^0 & 4^1 & 4^{10} \\ 4^6 & 4^7 & 4^8 & 4^9 \end{bmatrix}$$

After trying each kind of matrix with different weight values, the collected results showed a more positive tendency toward the snake-like matrix. This matrix yielded more appreciable results in comparison to other kinds, in both the max value of the tiles and the probability to achieve it(shown in Result Analyst).

III.2.3 Second version evaluation function: Big numbers

In this approach, we expect to achieve a state in which the number of empty grid cells, the number of large numbers are maximized while the balance of the table is maintained. A table is considered to have a high balance if there are no substantial differences between any pair of its numbers. The second version of the evaluation function can be expressed as follows:

$$eval(s) = empty_w \times empty(s) - unbalance_w \times unbalance(s) \\ + bignumber_w \times bignumber(s)$$

In the above formula:

- $empty_w$: the weight assigned for the function $empty(s)$
- $unbalance_w$: the weight assigned for the function $unbalance(s)$
- $bignumber_w$: the weight assigned for the function $bignumber(s)$
- $empty(s)$: the number of empty cells in the grid at state s
- $unbalance(s)$: a number that represents the unbalance of the grid at state s.

  $unbalance(s)$ is the sum of two components:

  - The unbalance with respect to rows: is the sum of differences between every pair of consecutive rows

  - The unbalance with respect to columns:is the sum of differences between every pair of consecutive columns

  The illustration is given by the example below:

  Assume that we need to calculate the $unbalance(s)$, where s is the following state:

$$\begin{bmatrix} 16 & 15 & 14 & 13 \\ 12 & 11 & 10 & 9 \\ 8 & 7 & 6 & 5 \\ 4 & 3 & 2 & 1 \end{bmatrix}$$

  The unbalance with respect to rows of the above grid is:

$$|16-12| + |15-11| + |14-10| + |13-9| \\ |12-8| + |11-7| + |10-6| + |9-5| \\ |8-4| + |7-3| + |6-2| + |5-1| = 48$$

The unbalance with respect to columns of the above grid is:

$$\begin{aligned}&|16-15|+|15-14|+|14-13|\\&|12-11|+|11-10|+|10-9|\\&|8-7|+|7-6|+|6-5|\\&|4-3|+|3-2|+|2-1|=12\end{aligned}$$

Hence, $unbalance(s) = 12 + 48 = 60$

- $bignumber(s)$: is a number that represents the dominance of big numbers in the table. It is the sum of squares of numbers on the grid at state s. Assume that we need to calculate the $bignumber(s)$. The unbalance with respect to rows of the above grid is:

$$\sum_{i=0}^{3}\sum_{j=0}^{3}(grid[i][j])^2 = 1496$$

III.2.4 Third version evaluation function: Potential merges and monotonic rules

This version of the evaluation function is an improvement of the previous function at a and b. Apart from inheriting the idea of the weighted matrix and empty cells, it also takes into account 3 additional criteria:

- The number of potential mergers $pmerge(s)$
- The number of monotonic rows $mrow(s)$
- The number of monotonic columns $mcol(s)$

Then, the formula of the evaluation function can be given as

$$\begin{aligned}eval(s) = empty_w \times empty(s) + snake_w \times snake(s) + pmerge_w \times pmerge(s)\\ +mrow_w \times mrow(s) + mcol_w \times mcol(s)\end{aligned}$$

In the above formula:

- $empty_w$: is the weight assigned for the function $empty(s)$
- $snake_w$: is the weight assigned for the function $snake(s)$
- $pmerge_w$: is the weight assigned for the function $pmerge(s)$

- $mrow_w$: is the weight assigned for the function $mrow(s)$
- $mcol_w$: is the weight assigned for the function $mcol(s)$
- $empty(s)$: is the number of empty cells in the grid at state s
- $snake(s)$: is the trace of the matrix $M_{snake}^T \times S$, in which S is the matrix representation of the grid at state s and M_{snake} if the snake-like weighted matrix.
- $pmerge(s)$: is the number of potential merges at the state s, which is the number of pairs of equal consecutive grid cells. An example of this is given as follows: Assuming that we have to calculate the $pmerge(s)$, where s has the matrix form:

$$\begin{bmatrix} 1 & 1 & 0 & 2 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 2 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

  Then, $pmerge(s) = (1 + 1 + 1) + (1) = 4$
- $mrow(s)$: is the number of monotonic rows (rows on which the numbers appear in increasing or decreasing order)
- $mcol(s)$: is the number of monotonic columns (columns on which the numbers appear in increasing or decreasing order). Consider the state having the matrix representation as:

$$\begin{bmatrix} 16 & 15 & 14 & 13 \\ 12 & 11 & 10 & 9 \\ 8 & 7 & 6 & 5 \\ 4 & 3 & 2 & 1 \end{bmatrix}$$

  Then, we have $mrow(s) = 4$ and $mcol(s) = 4$.

IV Result Analyst

We have run every Evaluation Function 100 times to see which Function has the best result. Units are measured in percentage. The result is shown below:

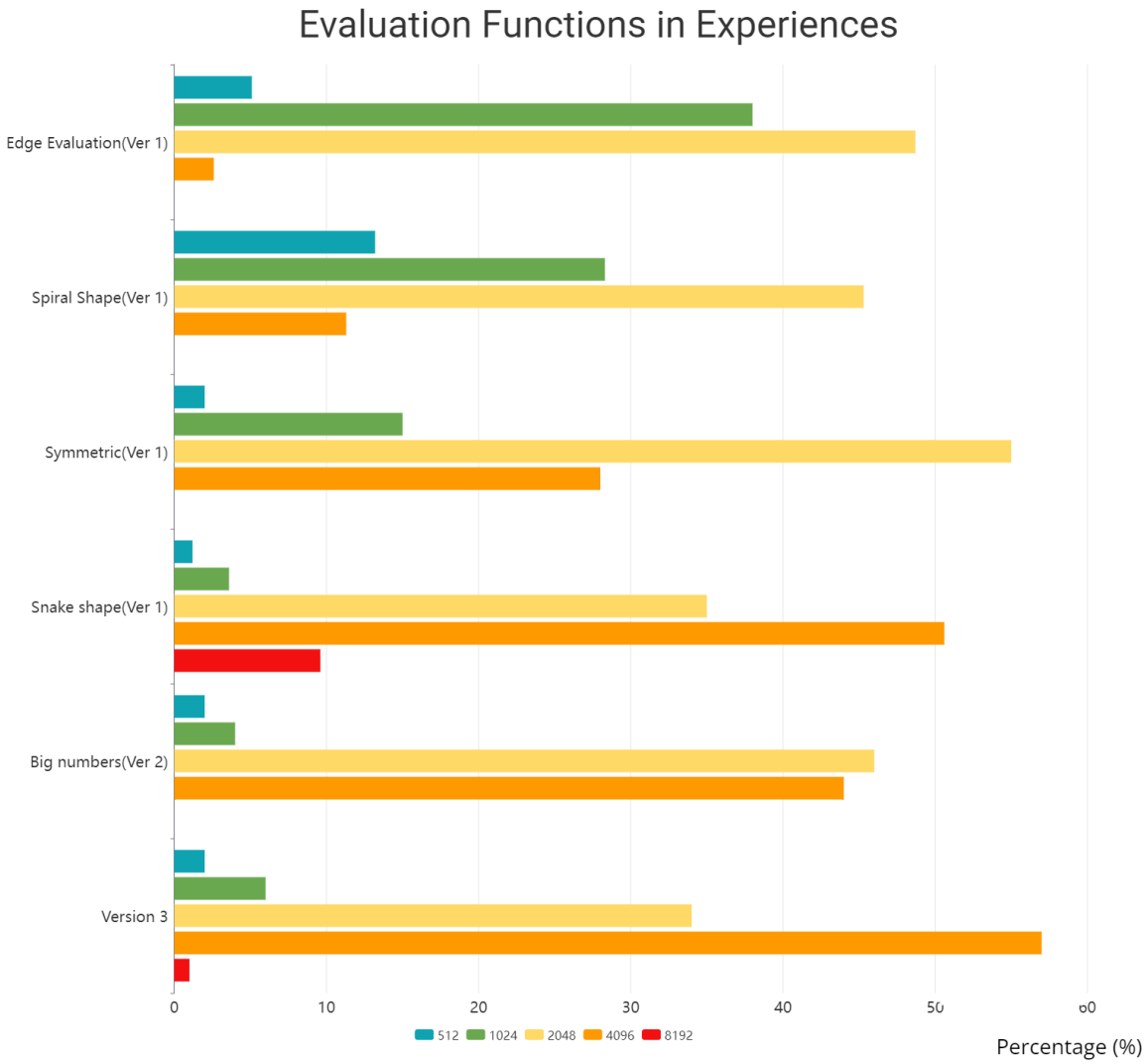

Fig. 4: Evaluation Functions in Experience - Grouped Bar Chart

Overall, as can be seen from the bar chart, **snake shape** is clearly **the best algorithm** for playing 2048 game with 10% of achieving 8192 tile and the rate of 51% for getting a 4096 tile. In spite of having the shortest code among all

functions, Snake Shape seems to be the most efficient algorithm. What a strategy to beat 2048 puzzle game! **The second best** is **the third version evaluation** with potential merges and monotonic rules whose rate of getting a 8192, 4096 and 2048 is 1%, 57% and 34% respectively.

Symmetric weight and Big numbers evaluation function is quite a powerful way to beat 2048 when their rate of getting at least a 2048 tile is 83% and 90%, but if we want to score much more from 2048, we should use the two functions above. On the other hand, Edge evaluation and Spiral Shape is the two worst function to play 2048 when their rate of getting at least a 2048 is just 51% and 57%.

V Difficulties during execution

Algorithm and the maximum depth to choose

In the beginning, the search algorithm that we were about to implement was Minimax. But for a stochastic and imperfect player game like 2048, it is obvious that Minimax was not the optimal algorithm for our project, which then lead us to the decision of rechoosing Expectimax algorithm.

Moreover, the maximum depth is also a factor that we have to consider as it affects much of the efficiency and running time of each move. So we have to consecutively change the depth to test for the most efficient depth(in our opinion).

Parameters of evaluate functions

Similar to the maximum depth, each parameter of evaluate function also need optimized, which take us much time to consider the best parameters for each evaluate function.

VI Conclusion and Future work

Conclusion

Expectimax is an excellent search algorithm for 2048 as we can see, for any evaluate function proposed, there is a high percentage for the agent to win the game.

Future work

- Optimize parameters in each evaluate function and the maximum depth

  We are going to optimize those factor by applying CMA-ES, a particular kind of strategy for numerical optimization, which we consider not an easy problem to deal with.

- Try some other search algorithms

  There are some search algorithms we consider worthy to try such as Minimax, Monte Carlo Tree Search and so on. These algorithms will definitely be worthy to try as it will be more objective when we can compare the efficient of those to Expectimax.

VII Reference

https://en.wikipedia.org/wiki/Expectiminimax
http://cs229.stanford.edu/proj2016/report/NieHouAn-AIPlays2048-report.pdf
https://github.com/Lesaun/2048-expectimax-ai

Library used:

(Click on the library name for the link to the library)

1. **Numpy**: NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

2. **Tkinter**: Tkinter is a Python binding to the Tk GUI toolkit. It is the standard Python interface to the Tk GUI toolkit, and is Python's de facto standard GUI. Tkinter is included with standard Linux, Microsoft Windows and macOS installs of Python. The name Tkinter comes from Tk interface.