Using Artificial intelligent to solve the game of 2048

Ho Shing Hin (20343288) WONG, Ngo Yin (20355097) Lam Ka Wing (20280151)

Abstract

The report presents the solver of the game 2048 base on artificial intelligent techniques. The solver is based on two different approaches, including tree search and expectimax algorithm. It discusses the differences in these two approaches and heuristics functions used to improve the performance of the expectimax algorithm.

Introduction

Creating effective artificial intelligent for games is one of the most interesting challenges as an undergraduate student. In this report, our group are trying to develop a solver to the game 2048, which is a popular game on mobile devices in the recent years, not only to investigate the potential of different algorithm and techniques we learned, but also to keep it interesting and exciting in the process of implementing these algorithms.

In this report, we are going to develop the solver using two techniques, tree search and expectimax. The former one is relatively simple, but give an unsatisfactory performance. The later gives and better result, with the help of a number of different heuristics function that we would introduce later.

The problem

Figure 1: The game of 2048

The game 2048 is a single player puzzle game, developed by Gabriele Cirulli. Player needs to slide the puzzles using four actions, including moving left, right, up and down, which operates on all of the puzzles on the board. When two tiles have the same value and are slid to the same direction, two tiles will merge and the value of the tile will be the sum of the values of the original tiles. When each slide happens, a new tile will appear randomly on one of the empty grid with value of two at 90% chance or with value of for at 10% chance. The goal of this game is to get a 2048 tile on the board, though the game can continues even after a 2048 tile is achieved. If the board is stuck with all 16 tiles with no possibility to move or merge, the player lose the game.

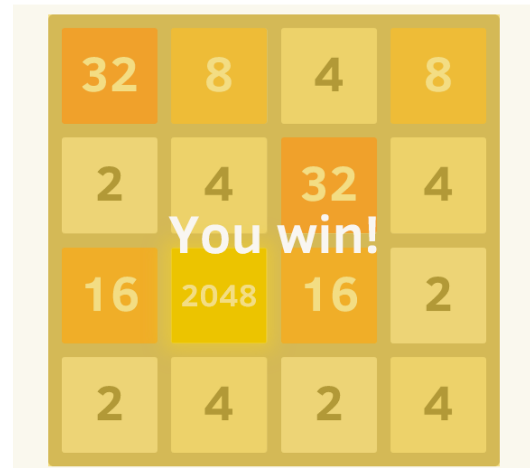

Figure 2: The wining state

Score is given at each merge with the value equal to value of the tile formed by merging. By calculation, getting a score higher than 20000 is equivalent to achieving a 2048 tile on the game board.

Problem Definition and Environment

Our artificial intelligent agent take all 16 numbers on the board (empty grid indicated as 0 value) as the input, and should output an optimal action one can do to maximize the possibility to win the game.

For our project, we are using a terminal version of 2048 named “term2048” written in python [1]. The original version of “term2048” project can be found on GitHub, with links provided in the references. We chose this version of 2048 because we believe a terminal version speed up the computation, and using python to develop the agent is more sample and efficient to understand the algorithms just from reading the source code.

Challenges and Baseline

The game may seems easy to play with at the first glimpse, with only 16 numbers on the board only, but the limit of space is what makes the game difficult. By simple calculation, we know that a 2048 tile is merged from 1024 tiles with value 2, while each of those tiles is generated when we play one move. To win the game, we need to merge those 1024 tiles in only 16 grids space, with about 1024 move. This requires the agent to make almost no mistake in the whole process, making the development of the agent more difficult.

Making matters worse, the game is including some randomness, as one tile will be generated at one of the empty grid after each move. As we will see later, one of the method to take advantages in this game is to put tiles with larger value at the corner and edges, but randomness make this a risky strategy. These large value tiles sometimes are forced to move to the center when no other moves are possible and a newly generated tile may block the large value tile to go back to edges.

Moreover, the size of the state space of the game is huge. There are around 10^16 of states possible before getting a 2048 tile, and there is even more if we continue the game after winning. This prevent us using Q-learning techniques, which require a Q table, unless we use deep learning network to simulate the Q function.

We take random moves as the baseline of our object.

Performance of random movement in 100 episode

	Random moves
Highest score	3124
Average score	1095
Winning percentage	0%

Random moves did not give a good result in the game, in did not even get a tile with value 512. This indicate that we need to find out move powerful methods to develop our solver.

First Approach: Tree search

The Algorithm

Tree search is one of the basic method in artificial intelligent. It try to search for the possible future states and find out the one with the highest possibility to win the game. In our implementation, we use depth first search using recursion with a given depth to search for all possible state in a given steps, and record the score taken by the agent at each path, and select the path that receive the highest score. The next move of this agent will be the first move of the best possible path the agent choose.

Result

Tree search is possible to solve this game, but with a low success rate.

Depth	1	2	3	4	5
Highest score	5216	16132	23676	35268	32112
Average score	1826	7319	10934	18092	16976
Winning rate	0%	0%	2.9%	28%	25%

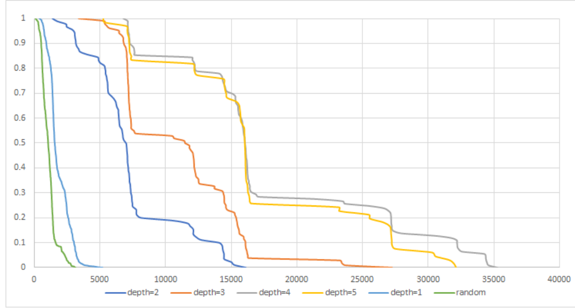

Figure 3: possibility of a game to reach certain score

At depth equal to four, simple tree search reach its best performance, with successful rate around 28%, and we are seeing decline in performance in depth of five. Obviously, we need a better algorithm to solve the 2048 problem.

Second Approach: Expectimax Search

The Algorithm

Expectimax search is a typical search algorithm to develop artificial intelligent for zero sum two player games. It is a specialized variation of minimax search. Minimax search tree classifies two player as maximizer and minimizer, where maximizer trying two maximize the overall utility of the game, while minimizer doing the opposite. A value is associated in each state of the game. The difference in expectimx search is that instead of the minimizer trying to minimize the overall utility of the game, the “min” node take action by chance, and the value associated to the “min” node is the expected value of the utility of the states it may take. The algorithm search all possible states of the game given depth, and evaluate how good the state is using heuristic functions in the leave nodes. After getting the value, the algorithm carry the value up through the branches, and evaluate the value of each “min” node and “max” node, and finally the agent, which is represented as a “max” node, will choose the action that takes it to the “min” node with the highest value.

In our 2048 problem, we define the “max” node as the artificial intelligent agent, while the “min” node as the game itself, which randomly place a newly generated tile on the empty grid on the board after each move. The agent first simulate four action that it could take, and pass the states to the “min” node, the “min” node evaluate its value by listing out all possible states that the new tile will possibly be, and take the expected value of all those states. Running this search tree recursively, and now we get a working expectimax search tree for this single player game.

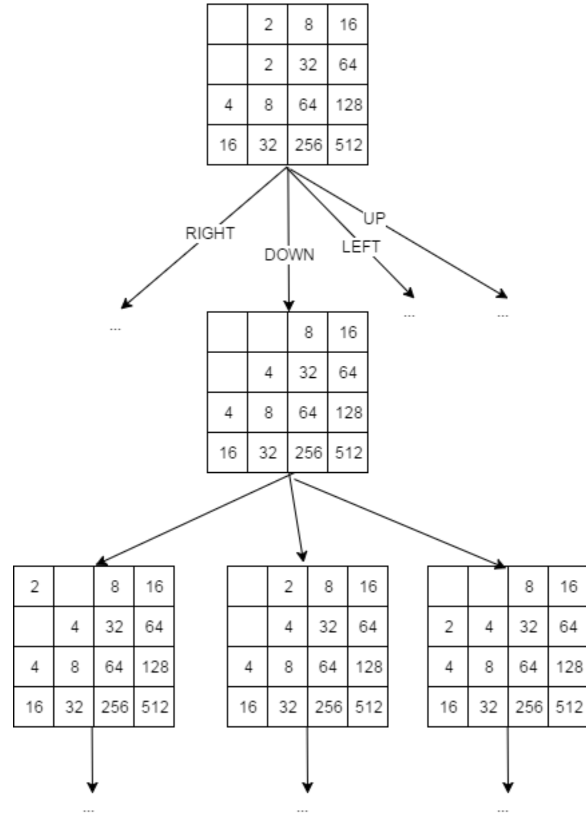

Figure 4: expectimax search tree for 2048

In our expectimax search, we only consider the case that a new tile with value 2 is generated, we ignore the case of value 4 tiles, because it occurs at a low frequency. This assumption reduce the out branch of each “min” node by half, dramatically improved the searching speed.

Evaluating the states

To evaluate how good a state is at the leave node of the expectimax search tree, we used three different heuristic function. The overall heuristic value of each state is a weighted sum of these three heuristic functions.

Pattern heuristics

By carefully looking at how the tree search achieve high score in some of the game, and try it out ourselves, we observe some pattern in order to get high score. First, we find that the tile with the highest value should be at the corner of the board, and second the tile with the second highest and the third highest tile should be at the edge of the board and right next to the highest one. These two observations are easy to explain; the tile with the highest value has the least probability to merge with the others, so it should be at the corner, otherwise it would become an obstacle for other smaller tile to merge. The second and third largest tile are at the edge for the same reason, and they are closed together because it is easy for them to merge once they are allow to so do. As a result, we design our first heuristic as follows.

The value of the heuristics is obtained by element-wise multiplication of the value on the game board and the table below.

0	0	1	3
0	1	3	5
1	3	5	15
3	5	15	30

Cluster heuristics

After some trials, we found out that the previous heuristics is not enough to keep the tiles with large value all together. Although three tiles with the largest value has a high probability to stay at one corner, some of the other tiles are spread to two edges, at an opposite corner of the board. In order to tile with similar tile keep close together to ease merging, we introduce a penalty to tiles that stay close together but have very different value:

```
penalty = 0
for each cell on the board as i:
        for each neighbour of i as j:
                penalty = penalty + absolute(value[i] - value [j])
        end loop
end loop
```

In order for the agent to minimize the penalty, tiles with large value should come close together to form a cluster, while tiles with small values only introduce a smaller penalty because they are small in absolute value compare to the larger one, so this heuristics should have minimal effect on the merging of smaller tiles.

Monotonic heuristics

We use monotonic heuristics to ensure the tiles are aligned in increasing or decreasing order in their value. This make sure the game is mainly played in two direction, reducing the complexity of the game.

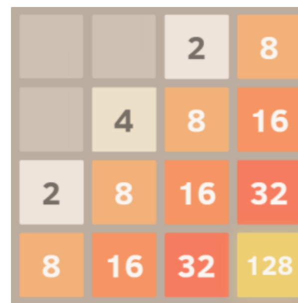

Figure 5: Value of the tile align monotonically

To implement this heuristics, we give reward to the states where the tile in the lower edge and right edge that align monotonically. We keep the other nine cells away from this calculation of this heuristics to make the merging of the tiles with smaller values remain unaffected.

Result

The expectimax search algorithm together with the above heuristics gives better result than the simple tree search in our first approach.

	expectimax
Highest score	49068
Average score	24670
Winning rate	79%
Rate of getting 4096	2%

Our expectimax agent on average can reach a score higher than 20000, with 79% of chance winning a game. It even can form a 4096 tile in some rare cases.

Conclusion

The result of the expectimax search algorithm, is generally a success, giving a better result than just simply looking in to all future states to maximize its score only. This experience gave us confident that it can be used to solve the real problem in the real world.

On the other head, we found that finding a good, simple and effective heuristic is not that easy. It need both the insight in to the game, great mathematical ability and a clever mind. Our complicated heuristic just barely works, with a successful rate way less than what we expected when we first start the project.

Future Work

Here are several things than can be done to improve the 2048 solver:

Implementing a better heuristic function: The heuristic we are using right now is complicated and computational expensive, we can find more pattern in the game to find a simpler and more effective heuristic.

Using deep-Q-learning: while using simple Q learning using a table is impossible as this game due to its extremely large space states, deep reinforcement learning can use a deep neural network to approximate the Q function, making Q learning possible. This is original in our plan of this project, but our self-study plan slightly lagged behind.

Acknowledgments

We would like to thank bfontaine and other open source contributors to make the infrastructure of or project, the terminal version of 2048 in python “term2048”, possible for everyone on GitHub. Inspired by them, we are going to make our project available for everyone on GitHub.

Our project available at:
https://github.com/comp3211finalprojectgroup/2048_ai

References

[1] term2048 project on GitHub
https://github.com/bfontaine/term2048