Learning to play the 2048 puzzle

Bakthavatsalam

Abstract

2048 is a popular sliding-tile puzzle played on mobile devices and web browsers. This paper discusses strategies for an AI to solve the puzzle. These strategies are compared and evaluated.

INTRODUCTION

2048 is a sliding-block puzzle where the objective is to combine tiles by sliding and form the 2048 tile. The game can be played further to form larger numbers. Due to the stochastic nature of the game, a strategy must be developed that maximizes the average score and an optimal path to goal cannot be calculated [1].

STRATEGIES FOR PLAYING GAMES

Minimax

The minimax algorithm is common for two player games where the strategy is as follows: choose a move to a position with the highest minimax value. The minimax algorithm is extended to alpha-beta pruning to reduce the number of nodes explored.

Expectimax

Expectimax is different from minimax in that chance nodes are interleaved in the search tree and decisions are based on the expected value of child nodes rather than the minimum or maximum values.

Monte Carlo methods

Monte-Carlo methods run many simulations of the game and use the average of these results in making a deciosn.

METHODOLOGY

In our project, we implement the pure Monte-Carlo game search combined with an evaluation function. The evaluation function is a weighted linear combination of selected board features. The weights for the function are learned using an optimization technique after several game sessions. We also implement a depth limited Monte Carlo search where the simulations are run only upto a certain depth.

Copyright © 2018, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Pure Monte Carlo search

The evaluation function is used for the decision at each state. The procedure is shown in Algorithm 1:

Algorithm 1 Return action using evaluation function

```
1: procedure BESTACTION(s)
2:     score ← 0
3:     action ← NULL
4:     for each a in ACTIONS(s) do
5:         s' ← MOVE(s, a)
6:         if EVALUATE(s') > score then
7:             score ← EVALUATE(s')
8:             action ← a
9:     return action
```

The action at state 's' is the depth 1 search using the evaluation function. This strategy guides the simulations in the Monte-Carlo search. The results of the simulation are the number of moves survived by the AI (Algorithm 2).

Algorithm 2 Simulation Results

```
1: procedure SIM(s)
2:     score ← 0
3:     while s not lose state do
4:         action ← BESTACTION(s)
5:         s ← MOVE(s, action)
6:         score ← score + 1
7:     return score
```

Finally, to determine the action at a state, the procedure in Algorithm 3 is used:

Depth limited Monte Carlo tree search

In this search method, we simulate the game only up to a limited depth and the results for the simulation are obtained from the evaluation at that state. The only difference in this search procedure is Algorithm 2. The new algorithm is described in Algorithm 4. This procedure is expected to match the expectimax search and is similar to the Averaged Depth-Limited Search in [1]. It provides the results of expectimax even when the probability of the chance nodes are not known.

Algorithm 3 Return action from state after simulation

```
procedure BESTACTIONSIM(s)
    score ← 0
    action ← NULL
    for each a in ACTIONS(s) do
        ActionScore ← 0
        s' ← MOVE(s, a)
        for N iterations do
            ActionScore ← ActionScore + SIM(s')
        if ActionScore > score then
            score ← ActionScore
            action ← a
    return action
```

Algorithm 4 Simulation Results

```
procedure SIM(s)
    score ← 0
    while s not at depth L AND not lost do
        action ← RANDOM(ACTIONS(s))
        s ← MOVE(s, action)
    score ← EVALUATE(s)
    return score
```

Evaluation function

Board features: To develop the evaluation function, few features from the game state were selected and the score was based on a linear combination of these features scaled by appropriate weights. These features are 1) the number of empty cells 2) the highest tile 3) the number of available moves 4) the sum of terms in the hadamard product of the board with a weight matrix. The last feature is to enforce the monotonic arrangement of tiles which is also the strategy employed by human players. This is in fact the topology for optimality as discussed in [2];

64.0	32.0	16.0	8.0
0.5	1.0	2.0	4.0
0.25	0.125	0.0625	0.03125
0.001953125	0.00390625	0.0078125	0.015625

Table 1: Weight matrix to enforce monotonicity

Objective function For the purpose of optimization, an objective function must be defined. We have used the duration of a game session in terms of the number of moves made in the objective function. It is a measure of how long the AI is able to "survive". Due to the random nature of the game, the duration can vary significantly for the same set of parameters. Hence, the objective function is the average duration of 'N' games played. For this project, we set N to 50.

Weight tuning Optimizing the weights for this is non-trivial as the search space is vast. As mentioned in the previous section, the objective function is also not deterministic for a set of parameters. Also, the derivatives for the function cannot be computed. For these reasons, a derivative free optimization technique has been used. A coarse estimate for a good set of candidate parameters was carried out using a grid search with a reasonable resolution. After a set of candidate parameters were found, a downhill-simplex algorithm with the starting point as the candidate solution was carried out. The results were used as the final weights.

RESULTS

The results for different search strategies are displayed on the table. The results were obtained after running 50 games for each strategy and averaging the results to the closest integer. With random moves as policy, the AI could only achieve a max tile of 512 and a duration of about 118 moves. An agent with a depth 1 search as the policy was able to achieve a 2048 tile 8 percent of the time and on average of about 331 moves. It is clear that the optimization algorithm is improving performance.

Strategy	no of moves	max tile	% of 2048 formed
Random policy	118	512	0
Depth 1 search with evaluation	331	2048	8
Depth 4 Minmax search with evaluation	742	2048	31
Pure monte carlo search	1512	4096	61
Pure monte carlo search with evaluation	1744	4096	68
Depth 4 Monte carlo search	932	4096	48

CONCLUSION

As seen from the results of "Pure Monte Carlo search" and "Pure Monte Carlo search with evaluation", the evaluation function does NOT significantly improve performance using the selected features. A game simulation guided by a random policy is just as good as a simulation guided by the evaluation function. The use of an optimized evaluation function with other search policies, however, significantly improves performance. Of these, the depth 4 Monte Carlo search provided the best results. This method could not match the results from Pure Monte Carlo search methods. However, the depth limited Monte Carlo search is less expensive computationally compared to the pure search methods.

FUTURE WORK

This optimization method is only expected to provide good results if the features selected were good. Feature selection is again a non-trivial task and usually requires human expertise. Other advanced features can be included to significantly improve the performance of the AI when combined

with a search strategy. However, the training takes longer as the number of features increase.

ACKNOWLEDGEMENT

We would like to thank the GitHub user chandruscm [3] for the C++ implementation of the game.

REFERENCES

[1] Philip Rodgers and John Levine, an investigation to 2048 ai strategies
[2] Bhargavi Goel, Mathematical analysis of 2048
[3] C++ 2048 "https://gist.github.com/chandruscm/2481133c6f110ced6dd7"