CS221N Progress: 2048 Challenge

Rui Qiu(ruiq), Yanpei Tian(yanpeit), Yulian Zhou (zhouyl)

November 16, 2019

1 Introduction

2048 is a single player puzzle game, which is played on a 4×4 grid, with numbered tiles slide one step upon pressing 4 arrow keys [2]. In the computer's turn, a new tile of value 2 will randomly appear in an empty grid. In the player's turn, the tiles will slide towards the direction of the pressed arrow key until it is blocked by other tiles or the edge of the grid. Upon colliding of the tiles with the same value, they will merge into a single tile with the value summing up. At each step, the score is won by the newly merged tiles' value, and the game ends when there is no legal moves (no empty grid and no merge can be performed).

The game can be viewed as a two-agent opponent game, where the opponent (computer) makes random moves, while the agent (human) tries to identify the best policy. A obvious approach is to adopt expectimax search to maximize the agent's rewards, so that the recurrent for $V_{\text{exptmax}}(s, d)$ is expressed as:

$$V_{\text{exptmax}}(s,d) = \begin{cases} \text{Utility}(s) & \text{isEnd}(s) \\ \text{Eval}(s) & d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\text{exptmax}}(\text{Succ}(s,a), d) & \text{Player}(s) = \text{“human”} \\ \sum_{a \in \text{Actions}(s)} V_{\text{exptmax}}(\text{Succ}(s,a), d-1)/|\text{Actions}(s)| & \text{Player}(s) = \text{“computer”} \end{cases}$$

2 Algorithm

2.1 Game setup

We used the starter code from `https://github.com/yangshun/2048-python`, which provides a GUI interface to play the actual 2048 game. To build our model based on it, we have made two major modifications. Firstly, we replaced the game input from the keyboard with our AI agent. Secondly, we introduced a scoring system, which evaluates the current game state. As the game score is calculated by adding up the scores of the merged tiles, along with the fact that the computer player only introduces tile of value 2, we can pre-calculate the accumulative scores to generate tile values in all ranges (Table 1). For example, for a tile of value 64, it is generated from 2 tiles of 32, and 4 tiles of 16 and so on, and the total scores are $1 \times 64 + 2 \times 32 + 4 \times 16 + 8 \times 8 + 16 \times 4 = 320$. Moreover, we developed a new version of script, which is detached from the GUI to analyze the statistics with faster computation.

Table 1: Accumulative scores for generating a tile with the given value

Tile value	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}
Accumulative scores	0	4	16	48	128	320	768	1792	4096	9216	20480	45056	98304

2.2 Expectimax search

We implemented the expectimax algorithm as follows:

- IsEnd(s): Given a 4×4 grid, tell whether the game is over or not. The game ends when there is no legal move exists (a legal move is the move that results in a change to the grid). Figure 1d shows an example of the end state.
- Utility(s): We give the Utility(s) of any end state a very low score. The rational is that we want the game to play as long as possible, and we penalize any move that could lead to the end state. As a preliminary exploration, we have tried a uniform utility function that gives a value of either negative infinity or 0, and it gives similar results in both cases. In the following experiments, we will also try non-uniform utility functions, which take the end-state board information into account.
- Eval(s): As mentioned above, we use the accumulative scores as our evaluation function Eval(s) = $\sum_{i,j} \text{score}(i,j)$, and it can be calculate efficiently from the prior calculations (Table 1). Moreover, based on our game experience, we also experimented with a heuristic, that is we prefer to place the tiles with large value close to a corner. Particularly, the tile with the maximal value stays at a corner whenever possible. The intuition is that we want to form a ladder of values like [16, 8, 4, 2], so that once the value 2 is merged with another 2 to form a value of 4, we can continue to merge it to 4, 8, and then 16. One way to achieve it is to assign some weight to each grid. For example, if we want to keep the largest value at the upper left corner, then the weights in the first row would be of decreasing order like [4.0, 3.0, 2.0, 1.0]. For evaluation, we multiply the accumulative score to the grid weight and sum them up as Eval(s) = $\sum_{i,j} \text{weight}(i,j) \times \text{score}(i,j)$
- Player(s): There are two agents / players in the game: an AI agent player who moves the tiles on board, and a computer agent player who places a tile with value 2 at a random empty grid. In the computer's turn, it plays randomly, so we calculate the average value $V_{\text{exptmax}}(s,d)$ from all possible moves. In the AI agent's turn, it plays optimally to get the highest score, and we take the action that leads to the maximal value $V_{\text{exptmax}}(s,d)$. In both steps, the ties are broken at random.
- Actions(s): A movement is valid for the AI agent player if the grid contains at least one empty cell so some tiles can slide towards the empty cell(s), or at any row or column, there is at least one adjacent pair so that they can collide upon move.
- Succ(s, a): A successor state of the board after taking an action a at a certain state s.

Figure 1 shows a few example states during a game.

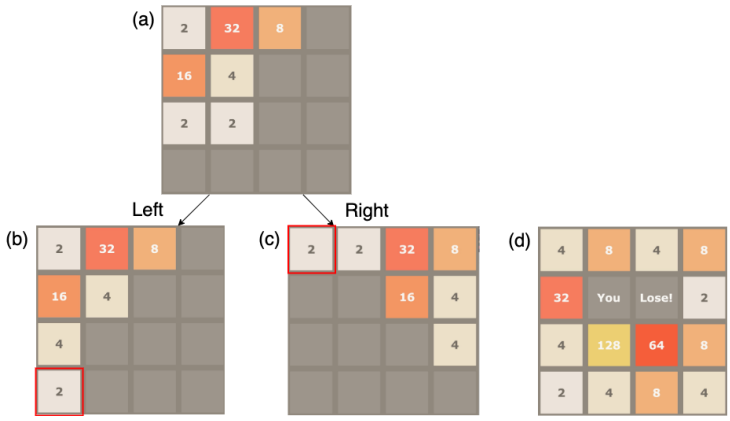

Figure 1: Different states in one 2048 game. (a) A prior state at the human player's turn; (b) a post state after human player takes a left move from (a); (c) a post state after human player takes a right move from (a); (d) an end state where no legal move exists. The red square in (b) and (c) highlights the new tile randomly introduced by the computer agent.

2.3 Minimax search

For the expectimax search, we have experimented with a depth of 2 and 3, while it become extremely slow when using a depth of 4. This is expected since there is an exponential growth in the search space with increasing depth. On the other hand, we noticed that the minimax search can be facilitated by alpha-beta pruning while the expectimax cannot. Therefore, we also experimented with minimax agent + alpha-beta pruning. The setup stays largely the same as the expectimax except that instead of averaging the values at the computer's step, we minimize the value, and that we prune a game node if its interval doesn't have non-trivial overlap with every ancestor. Indeed, we observed a remarkable increase in the search speed with $d = 2$ or 3, yet it is still very slow for $d = 4$. As expected, the overall performance of the minimax agent is lower than the expectimax agent.

3 Preliminary Result

We compared our expectimax agent ($d = 2$) with the baseline (random agent and greedy agent) by the cumulative distribution function of scores, and we noticed that the expectimax agent has a much higher probability of achieving scores more than 10,000, whereas the other two agents are generally scored lower than 5,000 (Figure 2).

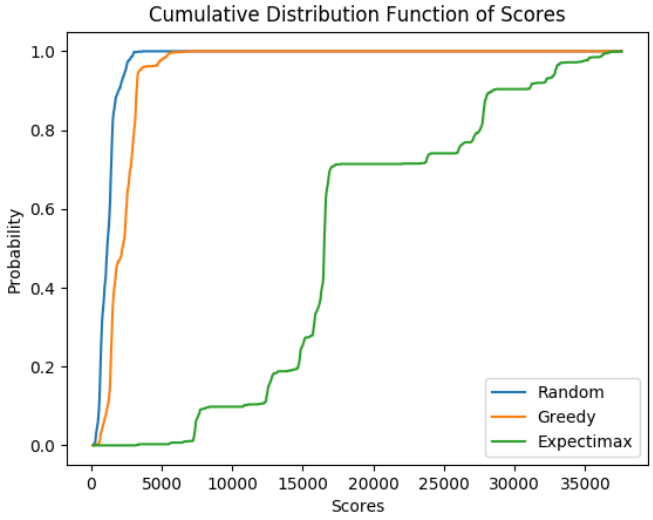

Figure 2: Cumulative distribution plots of the game scores by random agent, greedy agent and expectimax agent (1000 games)

.

We also compared the expectimax and minimax agents with the baselines, as well as the deep reinforcement learning oracle [1] on the maximal tile value that can be reached in 100 trials. With search depth $d = 3$, we noticed that the expectimax agent with weighted grid heuristics (denoted as "Expectimax (h)") exceeds other algorithms in these experiments (Table 2).

4 Conclusion

Overall, we have implemented the expectimax AI agent that achieves better performance than the deep reinforcement learning oracle. We also showed that minimax + alpha beta pruning is not very helpful for

Table 2: Success rate out of 100 games (Rounded to the nearest 5%)

Max tile value	Random	Greedy	Oracle (DRL)	Expectimax	Expectimax (h)	Minimax
4096	0%	0%	0%	0%	10%	0%
2048	0%	0%	10%	35%	70%	10%
1024	0%	0%	60%	85%	95%	70%
512	0%	5%	95%	100%	100%	95%
256	10%	50%	100%	100%	100%	100%
128	60%	90%	100%	100%	100%	100%
64	95%	100%	100%	100%	100%	100%
32	100%	100%	100%	100%	100%	100%

performance improvement. We plan to improve our algorithm from the two aspects: 1. we will experiment with other heuristics to boost the final scores, 2. our game uses the starter code from a real GUI game, while we can simplify the game board representation in bytes. In this way, it will increase the computation efficiency, which may further enable larger search depth. Additionally, we plan to explore one or more algorithms, such as Monte Carlo tree search and Q-learning, and compare their performance for the 2048 game.

References

[1] Navjinder Virdee. Trained a neural network to play 2048 using deep-reinforcement learning @ONLINE, 2018.

[2] Wikipedia. 2048 (video game) @ONLINE, 2019.