From AlphaGo Zero to 2048

Yulin ZHOU[*]
zhouyk@shrewsbury.org.uk

ABSTRACT

The game 2048 has gained a huge popularity in recent years [6]. The game allows the players to move numbers (a power of 2 such as $2, 4, 8, 16$, etc.) on the screen to sum up to at least 2048. It is easy to learn to play since it has only 4 actions: up, down, left and right. However, it is very hard to obtain a number greater or equal to 2048 because each action you make at a current state will lead to hundreds of unpredictable results. In this paper, we present a similar algorithm used in AlphaGo Zero to the game 2048 with a unique rewards-and-punishments system in order to increase the self-study speed of Artificial Intelligence (AI) as well as to obtaining a higher score for the auto-play mode of AI. Furthermore, based on the results by the Minimax Algorithm with Alpha-Beta Pruning, the Expectiminimax Algorithm and the Monte Carlo Tree Search (MCTS), we conclude that the Monte Carlo Tree Search outperforms other algorithms when applied to the game 2048. Finally, we show that AI with reinforcement learning [9] can beat the highest score that humans have achieved in the Game 2048.

ACM Reference Format:
Yulin ZHOU. 2019. From AlphaGo Zero to 2048. In *The 1st Workshop on Deep Reinforcement Learning for Knowledge Discovery (DRL4KDD '19), August 5, 2019, Anchorage, AK, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

AlphaGo [10] is a computer program that plays the board game Go [12]. Moreover, it is the first computer program which has defeated a professional human Go player as well as the first program which has defeated a Go world champion. AlphaGo was developed by Alphabet Inc.'s Google DeepMind in London. It had three far more powerful successors, called AlphaGo Master, AlphaGo Zero and AlphaZero [12].The earliest version, AlphaGo, required thousands of human amateur and professional games to learn and to master the game Go, while AlphaGo Zero, was the first version that learned by playing Go games against itself, starting from completely random play. It had been shown that AlphaGo Zero is better than the previous version [11]. Ever since AlphaGo Zero has received a wide reputation, people have tried to study the algorithms/methods of AlphaGo Zero in order to implement them into other games. In this paper, we will present an AI that can play the game 2048 automatically and maximize the game score within a reasonable runtime.

Game 2048 is a single-player puzzle game. Gabriele Cirulli, an Italian user-interface designer and web developer, created the game and released it in March 2014 [13]. The game 2048 is simple: the player is given a 4×4 board, where each tile may contain a number which is a power of 2. At the beginning, there are only two numbered tiles, either numbering 2 or 4. The player controls and alters the board by pressing arrow keys, with the tiles in the board moving according to the players. For instance, if you press the up key, all of the tiles will go up. If the numbers on the adjacent cells match, they will merge; otherwise, they will remain in their positions (See Figure 1). Moreover, after each move, a new number, either 2 or 4, will appear on one of the empty cells uniformly with a ration that can be set. In this paper, we will set the ration of occurrence between 2 and 4 as $9 : 1$. Players aim to slide numbered tiles on a 4×4 grid to merge and create a tile with the number 2048 or greater (See Figure 1). When there is no empty cell and no more valid moves, the game ends. If the tile with number 2048 appears on the board before the game ends, the player wins.

We consider the following three methods as the most popular and effective strategies: the Minimax algorithm with the alpha-beta pruning, the Expectimax algorithm and the Monte Carlo Tree Search(MCTS). Then we set rewards and punishments which are based on the rules that we set for the game when training AI. In that way, 2048 can be derived from a single-player puzzle game to an auto AI game, which means no more human interactions are needed throughout the entire self-learning process. Training AI without datasets derived from human experts has significant implications for the development of AI with superhuman skills, because expert data is often expensive, unreliable or simply unavailable.

The paper is divided mainly into three parts:

[*]This project is mentored by Stella Dong who is a PhD. student in the applied mathematics program at University of California, Davis.

DRL4KDD '19, August 5, 2019, Anchorage, AK, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

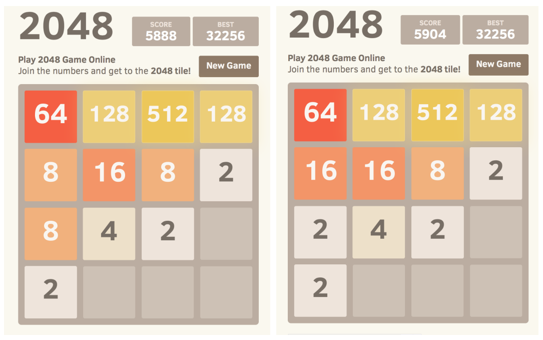

Figure 1: Example of a move in game 2048. Left graph shows the current state, and the right graph shows the next state after an "up" move. There are two 8's in the left. After you move the tiles upwards, the 8 at the bottom will move upwards and merge with the same number 8 above it, combining to create a 16.

(1) Two reward-and-punishment systems
(2) Three tree searching algorithms:
   (a) Minimax algorithm with alpha-beta-pruning
   (b) Expectimax algorithm
   (c) Monte Carlo Tree Search (MCTS)
(3) Comparison of results from three algorithms above.

2 TWO REWARD-AND-PUNISHMENT SYSTEMS

We set up two reward-and-punishment systems: one uses weight matrix α, and the other one uses weight matrix β. Weight matrix α is a matrix with the highest value in the top-right corner and its value decreasing diagonally (see Figure 2). Weight matrix β looks like a snack shape where player swipes the tiles along the path and merges them within the same top-right corner (see Figure 3).

According to our experiments, putting tiles with close values beside each other usually results in more possible merges afterwards. Obviously, a single swipe can merge two tiles with the same number while managing different tiles in an ascending or descending order, which can help tiles to merge consecutively later. From intuition, one can see that putting tiles with close values near each other usually leads to more possible merges. For tiles with the same number, a single swipe could make a merge; for tiles with different numbers, it is reasonable to place them in ascending or descending order, so that they could merge consecutively. Therefore, it seems to be attempting to use the matrix α that has the highest weight on one corner and is decreasing diagonally.

Since matrix α does not largely discriminate tiles: different weights of two tiles may be weakened by the large difference in numbers they bear, the weight matrix β was created to enlarge the range of weights. Weight matrices β are snake-shaped (see Figure **??**). The heuristic is that we expect swipes and merges along the snake to put a larger number in the corner. After trying symmetric and snake-shaped weight matrices with various weight values, we found the that the snake-shaped weight matrix gave the best performance.

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

Figure 2: Weight Matrix α

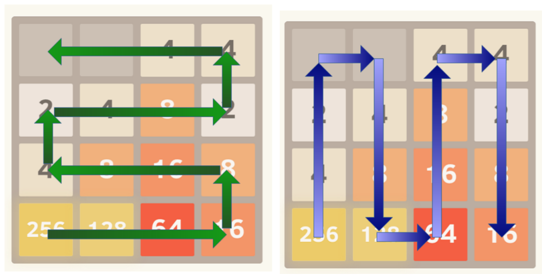

Figure 3: Left: weight matrix β in horizontal snack shape; Right: weight matrix β in horizontal snack shape vertical snack shape

3 THREE TREE SEARCHING ALGORITHMS: MINIMAX ALGORITHM, EXPECTIMAX ALGORITHM, AND MONTE CARLO TREE SEARCH (MCTS) WITH ALPHA-BETA-PRUNING

The game 2048 can be viewed as a two-player game, with the human player versus the computer. The human's turn is to move the board by choosing one of the four basic directions: up, down, left, or right; then, the computer plays the role of placing a number, 2, or, 4, at random within one of the empty cells. Therefore, the Minimax algorithm got our attention first since it is a decision rule used widely in games containing two players.

When applying the Minimax algorithm to the game 2048, the computer plays an opponent role which simply places new tiles of 2 or 4 with an 9: 1 probability ratio. First, we explain the notations in our algorithm:

(1) players:agent,opponent
(2) s: the grid board that reflects the current state
(3) a: the action taken
(4) $actions(s)$: possible actions at state s
(5) $result(s, a)$: 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
(10) d: the current depth of the search tree

The mathematical formula of the Minimax algorithm is presented below:

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

The player can choose from one of the four actions in every round, i.e., the action $a \in \{up, down, left, right\}$. The action of the computer, a will be randomly place 2 or 4 into the empty tiles of the current board. The evaluation function will be renewed every turn as the current score in each round when either reaches the maximum depth or result in a game over.

Next we derived the Expectimax algorithm from the Minimax algorithm above. The difference is that we added chance nodes between the agent nodes and opponent nodes. Therefore, the pseudocode for the algorithm, either when we add heuristics and domain knowledge into our evaluation function when AI reaches the last depth level, or when the game is over, is the following:

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

```
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
```

The Expectimax algorithm has some drawbacks, such as its number of moves to be analyzed quickly increases in depth and the computation power limits how deep the algorithm can go. Therefore, we tried the third method: the Monte Carlo Tree Search. The basic idea of the Monte Carlo Tree Search is a Bandit-Based Method. One player needs to choose movements between K actions and to maximize the cumulative rewards by continuously selecting the best move. Once the player chooses, the true rewards of this action will depend on subsequently possible moves.

The Monte Carlo Tree Search algorithm trains a neural network which takes state s of the game 2048 as an input and outputs both move probabilities and a value, (p, v). The scalar value v represents the "goodness" of state s. In AZG, v represented the probability of winning from the state, but since we only have one player, it is also a ratio between the expected current game result and the average result using current network. This way, values above 1 are good and below 1 are not so good (see Figure 4[1]).

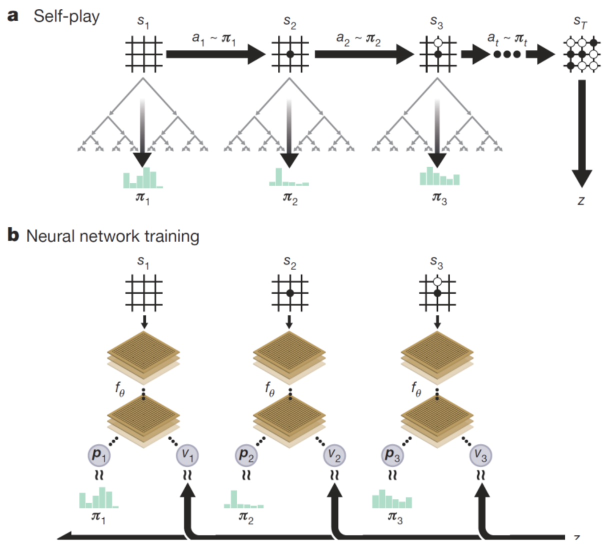

Figure 4: MCTS

In each state, the MCTS is executed, and a tree is built, with the root node being the initial state s_0. For each state s in the tree, there are four edges (s, a) which correspond to possible moves, with each edge containing statistics:

(1) $N(s, a)$ - number of times action has been taken from state s
(2) $W(s, a)$ - total value of move a from state s
(3) $Q(s, a)$ - mean value of move a from state s
(4) $P(s, a)$ - prior probability of selection move a

The Algorithm iterates over three phases (initially 1600 times):

(1) Select: Each iteration starts at s_0 and finishes when simulation reaches a leaf node s_L. At each time t, a move is selected according to the statistics in the search tree.

$$U(s,a) = P(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}. \tag{1}$$

This search-control strategy initially prefers actions with high prior probability and low visit count, but asymptotically prefers actions with high action value.

(2) Expand and evaluate: The leaf node s_L is evaluated using the current neural network. Leaf node is expanded and each edge (s_L, a) is initialised to $N = 0, W = 0, Q = 0, P = pa$. The value v is then backed up.
(3) Back-Propagation: Edge statistics are updated in a backward pass.

$$N(s,a) = N(s,a) + 1,$$
$$W(s,a) = W(s,a) + v$$
$$Q(s,a) = W(s,a)/N(s,a).$$

Once the iterations finish, probabilities are calculated. The next move is simply selected by the maximum. In its training phase, AGZ uses sampling from the distribution. We used a deterministic selection, since there is already so much randomness in 2048 already anyway.

We noticed that greater depths lead to a dimension explosion of search trees. In addition, the entire tree-search algorithm explores some unnecessary parts of the tree. Hence we implement Alpha-Beta Pruning in all three tree searching algorithms to prevent these problems. The Alpha-Beta Pruning estimates two values: the alpha (the lower-bound of the gain) and the beta (the upper-bound of the gain). In any case, where the beta is less than the alpha, the rest of the subtrees are pruned.

When leveraging the human heuristic of snake-shaped weight matrix β, we consider only some of the most important empty tiles which have higher weights on them. After several tiles with the minimal value of (depth, num of empty tiles, 4) or the minimal value of (depth, num of empty tiles, 6) these bounds typically get closer and closer together. This convergence is not a problem as long as there is some overlap in the ranges of α and β. At some point in evaluating a node, we may find that it has moved one of the bounds such that there is no longer any overlap between the ranges of α and β. At this point, we know that this node could never result in a solution path that we will consider, so we may stop processing this node. In other words, we stop generating its children, move back to its parent node and finish pruning this subtree.

```
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
```

4 IMPLEMENTATIONS AND PERFORMANCES

Minimax algorithm with Alpha-Beta Pruning

The game 2048 has a discrete state space which contains perfect information. It is also a turn-based game like chess and checkers. Thus we can apply the Minimax search with Alpha-Beta Pruning which have been proven to work on this type of games. The monotonicity heuristic tries to ensure that the values of the tiles are all either increasing or decreasing along both the left/right and up/down directions. This heuristic alone demonstrates the intuition many others have mentioned, that higher-valued tiles should be clustered in a corner. It will typically prevent smaller-valued tiles from getting orphaned, and it will keep the board very organized, with smaller tiles cascading in and filling up the larger tiles. The heuristic alone tends to create structures in which adjacent tiles are decreasing in value. Nevertheless, in order to merge, adjacent tiles need to obviously be of the same value. Therefore, the smoothness heuristic just measures the value difference between neighboring tiles in an attempt to minimize this count. Finally, there is a penalty for having too few free tiles, since options can quickly run out when the game board becomes too cramped. Table1 illustrates the result of the Minimax algorithm with Alpha-Beta Pruning.

Expectimax Optimization

The AI takes 10ms to 200ms to execute a move, depending on the complexity of the board position. There are four heuristics for this Expectimax algorithm:

(1) Granting "bonuses" for open squares
(2) Granting "bonuses" for having large values on the edge
(3) A penalty for having non-monotonic rows and columns which increased as the ranks increased, but non-monotonic rows of large numbers hurt the score substantially.
(4) The second heuristic counted the number of potential merges (adjacent equal values) in addition to open spaces.

Table 1: The results of Minimax Algorithm with Alpha Beta Pruning

Depth	2	3	4
Highest 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%

After running 100 times via remote control, the results of the Expectimax algorithm are shown as in Table 3.

Table 2: The results of Expectimax Algorithm

Depth	2	3	4
Highest score	31924	33940	59436
Average score	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%

Based the results in Table 3, we observed that the average scores for depth 2, 3 and 4 are 11279, 16766 and 59436 respectively, and the maximum scores achieved are 31924, 33940 and 59436 for each depth, respectively. The AI never failed to obtain the 2048 tile at least once. This game took 27830 moves over 96 minutes, or an average of 4.8 moves per second. Initially, there are two heuristics: granting "bonuses" for open squares and for having large values on the edge.

Monte Carlo Tree Search

If we start with a randomly initialized neural network, the scores that the simulators achieved are in the range from 500 to 4000. These scores are similar to the results from a random play. The training was carried out for 10 hours. When we used 100 simulations in the Monte Carlo Tree, 66 training cycles were performed in total. All 16 simulators combined were able to perform about 7 moves per second. Of all 66 cycles the new trained model was able to beat the neural network with the old weights four times. The best achieved score was 2638.

When using 50 simulations in the Monte Carlo Tree, 32 game simulators were able to perform approximately 40 moves per second. During 10 hours of gameplay and training, 334 training cycles were completed and a new version of the initial model achieved a better median score only two times.

During the course of development and testing, the parameters of the neutral network mostly stayed the same while only trying out a different learning rate and the amount of residual layers. Other parameters, like the amount of samples the network being is trained on each training cycle, the amount of games being played to evaluate the model were also tried out to achieve better results. Unfortunately, none of the tried combinations were able to improve the model from self-playing during the 10 hours of gameplay, and the results during the training process stayed similar to the results achieved by the randomly initialized network. The results of the Monte Carlo Tree Search have been shown in Figure

Table 3: The results of Monte Carlo Tree Search

Depth	2	3	4
Highest score	59924	69956	80728
Average score	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%

Comparison among three algorithms

We observed that as the depth level increased from depth 2 to depth 4, the score increased as well for all algorithms (see Figure 5). Nevertheless the Monte Carlo Tree Search received 59924 in depth 2, 69956 in depth 3, and 80728 in depth 4, which were all the highest among the three algorithms's performances.

In general, one would expect that the scores of three algorithms in the same depth would have a similar average score, and performances would be better when the depth level increases. Oddly, Figure 6 shows that the average values of Monte Carlo Tree Search is 20597 in depth 2 but 17770 in depth 3, which is a lot lower.

When we concentrated on depth level 2, the scores of three different algorithms had a huge difference. All of the algorithms reached 256 but the probability of reaching 512 for each game was different: 87% for Minimax algorithm, 94% for Expectimax algorithm and 94% for Monte Carlo Tree Search. Moreover, when the scores were setting higher, the probability of reaching it became much more difficult than the smaller scores. The Minimax algorithm had a probability of 46% to reach 1024, while the Expectimax algorithm had a probability

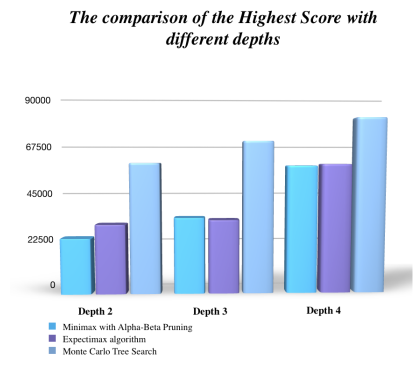

Figure 5: The comparison of the Highest Scores with different depths

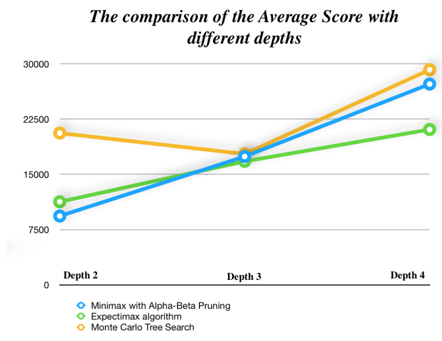

Figure 6: The comparison of the Average Scores with different depths

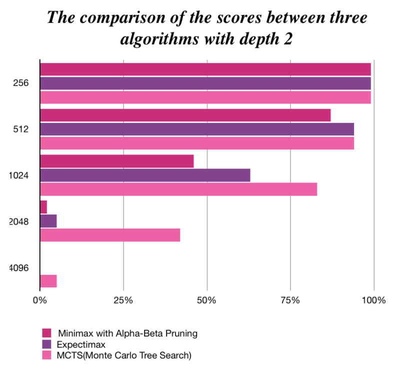

Figure 7: The comparison of the scores between three algorithms with depth 2

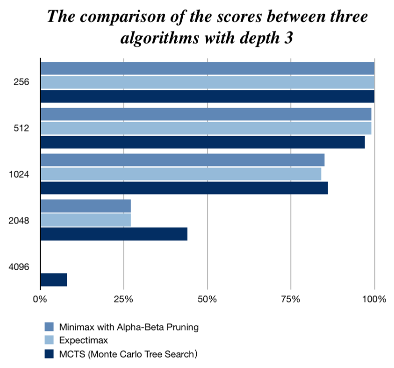

Figure 8: The comparison of the scores between three algorithms with depth 3

of 63%. Moreover, Monte Carlo Tree Search would reach 1024 at a percentage of 83%, which was the highest among all of them. However, reaching 2048 was not that easy for all the algorithms, as Minimax would probably reach 2048 with a probability of 2%, and Expectimax had a probability of 5%. At the same time, Monte Carlo Tree Search had a probability of 42% to reach 2048. Surprisingly, only Monte Carlo Tree Search had a probability of 5% of reaching 4096, while the others could not even reach it once in a depth level of 2 (see Figure 7).

Figure 8 shows that the chances of all three algorithms achieving higher scores decreased as the score requirements increased on the depth level 3. The probability of getting 2048 for both Minimax and Expectimax raised to 27% from 2% and 5%, respectively. Additionally, the Monte Carlo Tree Search had a probability of 44% of reaching 2048 and 8% of reaching 4096. In comparison, the Monte Carlo Tree Search did the best among the three algorithms.

It was not unexpected that all three algorithms improved their performances during the experiments of depth level 4. Scores shown in Figure 9 show the Minimax algorithm had a 71% ability to get 2048 and a 4% probability to get 4096, which was the first time a Minimax algorithm was able to reach a score of 4096. Although the Expectimax algorithm did not improve the performance of getting 1024 and acted worse than the Minimax algorithm the first time, it had a better probability of getting 2048. At the same time, the Expectimax algorithm had a 4% chance of achieving 4096. Additionally, the Monte Carlo Tree Search had a percentage of 64% of obtaining 2048 and a 16% chance of reaching 4096, which was the best one among the three algorithms. The performance of AI on game 2048 in these cases was quite exciting for us.

5 CONCLUSIONS

From this research paper, we have incorporated three basic algorithms: the Minimax algorithm, the Expectimax algorithm, and the Monte Carlo Tree Search. Based on the results among all three algorithms, we found that the Monte Carlo Tree Search is the best way of training AI for the game 2048, as it provides us a probability of 100% of achieving 256, 98%

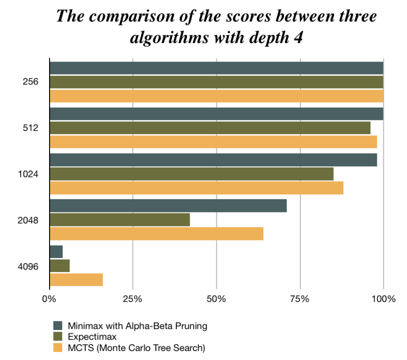

Figure 9: The comparison of the scores between three algorithms with depth 4

for 512, 88% for 1024, 64% for 2048, and 16% for 4096. When we connect theses results with AlphaGo Zero, the much simpler game 2048 reflects the same algorithm and explains how AlphaGo Zero works.

REFERENCES

[1] Karl Allik, Raul-Martin Rebane, Robert Sepp, Lembit Valgma, https://neuro.cs.ut.ee/wp-content/uploads/2018/02/alphago.pdf

[2] Jonathan Amar, Antoine Dedieu, http://www.mit.edu/ amarj/files/2048.pdf

[3] 2048 GUI, https://github.com/adichopra/2048

[4] Monte Carlo Tree Search explained, https://int8.io/monte-carlo-tree-search-beginners-guide/

[5] Yun Nie (yunn), Wenqi Hou (wenqihou), Yicheng An (yicheng), http://cs229.stanford.edu/proj2016/report/NieHouAn-AIPlays2048-report.pdf

[6] Dickey, Megan Rose (23 March 2014). "Puzzle Game 2048 Will Make You Forget Flappy Bird Ever Existed". Business Insider. Retrieved 27 March 2014.

[7] Graham, Jefferson (28 March 2014). "2048: The new app everyone's talking about". USA Today. Retrieved 30 March 2014.

[8] Pramono, Eric (8 April 2014). "2048: Game Strategy and Playing Tips". Retrieved 13 December 2014.

[9] Randy Olson(27 April 2015). "Artificial Intelligence has crushed all human records in 2048. Here's how the AI pulled it off". Retrieved 27 April 2014

[10] DeepMind(18 October 2017). "Mastering the game of Go without human knowledge". Retrieved 18 October 2017

[11] James Vincent(18 October 2017)."DeepMind's Go-playing AI doesn't need human help to beat us anymore". Retrieved 18 October 2017

[12] DeepMind. "The history of AlphaGo so far".

[13] 2048(video game) Wikipedia. "19-year-old Cirulli created the game in a single weekend as a test to see if he could program a game from scratch. Cirulli released a free app version of the game for iOS and Android in May 2014".

[14] Barto, Monte Carlo matrix inversion and reinforcement learning. Advances in Neural Information Processing Systems 687–694 (1994).

[15] Singh, Reinforcement learning with replacing eligibility traces. Machine learning 22, 123–158 (1996).

[16] Lagoudakis, Reinforcement learning as classification: Leveraging modern classifiers. In International Conference on Machine Learning, 424–431 (2003).

[17] Scherrer, Approximate modified policy iteration and its application to the game of Tetris. Journal of Machine Learning Research 16, 1629–1676 (2015).

[18] Littman, Markov games as a framework for multi-agent reinforcement learning. In International Conference on Machine Learning, 157–163 (1994).

[19] Enzenberger, Evaluation in Go by a neural network using soft segmentation. In Advances in Computer Games Conference, 97–108 (2003).

[20] Sutton, Learning to predict by the method of temporal differences. Machine Learning 3, 9–44 (1988).

[21] Schraudolph, Difference learning of position evaluation in the game of Go. Advances in Neural Information Processing Systems 817–824(1994).

[22] Silver, Difference search in computer Go. Machine Learning 87, 183–219 (2012).