2048 - AI Agent

CS7IS2 Project (2021/2022)

Deepayan Datta, Amith Nair, Muvazima Mansoor, and Sherin Miriam Cherian

ddatta@tcd.ie, nairam@tcd.ie, mansoorm@tcd.ie, scherian@tcd.ie

Abstract. In this project, different AI algorithms are trained to play 2048, the single player tile-based puzzle game, and their performances are compared. The algorithms implemented are - Greedy Search as the baseline, Expectimax, Monte Carlo Tree Search and Deep Q-Learning. The Performance of these algorithms were compared based on their execution time, highest score achieved and percentage of wins. Expectimax was found to be the best performing, achieving a high score of 8192.

Keywords: AI, 2048, Greedy Search, Expectimax, Monte Carlo Tree Search, Tuple Q learning.

1 Introduction

The tile based game, 2048 designed by Gabriele Cirulli [1] [3] consists of a 4x4 grid where each cell in the grid contains a number which is a power of 2 or remains empty. The initial state of the grid contains two randomly placed cells which can each either have a value of 2 or 4. The objective of the game is to combine the numbered cells on the grid to achieve a cell with a value of 2048. The cells can be combined by using the 'awsd' keys on the keyboard to move left, up, down and right. A new tile is added after the merge which can either have a value of 2 with a 90% probability or it can have a value of 4 with a 10% probability. The game can be continued even after 2048 is obtained to achieve cells with much larger values like 4096, 8192 and so on. The game ends when no more legal moves can be made. This state occurs when all cells in the grid are filled and no tiles can be combined. Achieving the goal state in this game is a difficult task since it involves taking calculated risks and planning the moves several steps ahead. It is quite a challenge for humans as well and therefore this motivated the research to train state of the art algorithms to solve the 2048 game and observe their performance. The goal of the project is to evaluate and analyze 3 radically different AI algorithms such as Expectimax, Monte Carlo Tree Search and Deep Q learning. Their performance with respect to execution time, highest score achieved and percentage of wins is compared against a baseline algorithm, Greedy Search. Finally, each algorithm is explained and the results are presented along with the conclusions.

2 Related Work

Robert Xiao in his stackoverflow[3] answer explains how he solved the 2048 problem using minimax algorithm with alpha-beta pruning. He was able to reach till the 16384 tile in some cases. Hunter mills[7] showed how using Monte Carlo Tree search helped him reach the 4096 tile. In a conference paper published in IEEE, Jun Tao, Gui Wu, Zhentong Yi and Peng Zeng[9] discussed optimisations that could be implemented in the UCT algorithm of Monte Carlo Tree Search for the 2048 problem. In 2016, Wojciech Jas kowi showed that various strategies to solve the 2048 game can be learnt by embedding the game into a discrete-state Markov decision processes framework. By employing temporal difference learning with systematic n-tuple networks, he was able to build one of the best known 2048 playing program till date.

For this project, we are going to use Expectimax, Monte Carlo Tree Search and Tuple-Q learning. We use Expectimax algorithm to explore the idea of winning the game by giving rewards for every step based on certain pattern formations. Monte Carlo Tree Search algorithms are known to be able to solve problems with no prior knowledge of the domain. We have used Monte Carlo Tree Search algorithm to observe if the algorithm comes up with unconventional strategies to solve the problem. Q-learning is a Reinforcement Learning algorithm that learns the optimal policy required to reach a goal state by actively exploring the environment. It is a model-free RL algorithm designed to handle stochasticity in the environment .

3 Problem Definition and Algorithm

We build and test 4 algorithms to play the 2048 game i.e. our goal is to reach the 2048 tile during game play. We compare the algorithms in terms of execution time and success rate.

3.1 Greedy Best First Search

We have used Greedy Best First Search as a baseline algorithm to make comparisons and evaluate the performance of different algorithms. Greedy Best First Search selects the path which appears to the best at any step. Two different heuristics functions based on specific patterns have been implemented to obtain the reward involved for every action.

The Heuristic functions considered for the problem is as follows:

1. **Weighted Average of the following Heuristics**
   (a) **Monotonicity Heuristic:** Any seasoned player of 2048 knows that in order to maximise the possibility of getting a tile greater than or equal to 2048, the tile with the highest value needs to be in a corner, with the other bigger number tiles along the edges adjoining the corner in a decreasing manner. This heuristic tends to capture this strategy. Along

the horizontal direction, the number of cells greater or smaller than the previous cells are calculated. The greater of the two values is the heuristic along the horizontal direction. The same thing is done in the vertical direction as well. Initially we tried adding the difference between successive cells. However this did not produce the desired results as the algorithm tried to bring smaller cells close to the bigger cells to maximise the difference. The heuristic is maximised when the highest number tile is located at a corner and the cells adjoining to it are arranged in a decreasing fashion.

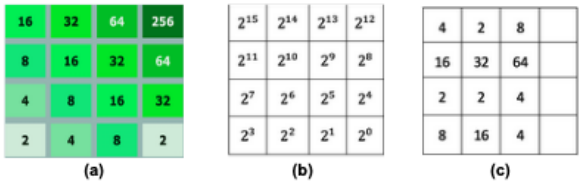

Fig. 1: (a) Monotonic Grid (b) Weights for Snake Pattern Heuristics (c) Example game state

Figure 1(a) shows the output obtained by running the algorithm using just this heuristic function

(b) **Smoothness Heuristic:** This heuristic tends to bring equal valued tiles closer to each other by penalising unequal numbered tiles close to each other with their magnitude as the difference between the two numbers.

(c) **Max Value Heuristic:** This heuristic returns a value proportional to the highest value in the grid.

(d) **Empty Cells Heuristic:** The chances of winning the game reduces if there are less number of empty tiles in the grid. The empty cell heuristic provides a value proportional to the number of empty cells in the grid.

2. **Snake Pattern Heuristic:** Another popular strategy to play the 2048 game is to arrange the numbers descendingly in a snake format. To achieve this, weights are assigned to each tile in the format shown in Figure 1(b).

Comparing the results obtained for both these heuristics, we observed that the Snake heuristic performs much better for both Greedy best first search and Expectimax algorithm.

3.2 Expectimax

2048 is a stochastic game and has the probabilistic element of the numbers 2 or 4 popping in any of the empty spaces. The Expectimax algorithm is a variation of the minimax algorithm which is used to maximise the expected utility and is best applied in case of a stochastic problem. Hence this algorithm is most suitable for our use case.

The expectimax game tree consists of Max nodes and chance nodes. The Max nodes represents the maximum value of the child nodes. The values of the chance nodes represent the expected utility value of all the random states.

In figure 1(c), if we consider only the merge scores(i.e. the score obtained by merging two tiles. The magnitude of the score is equal to the value of the merged tile), the action down will maximise the utility value. Hence this action will be selected by the max function. By doing action down, there will be 5 empty tiles in the grid and a new numbered tile can appear in place of any of the 5 empty tiles. Furthermore, the new numbered tile can be a tile with value 2 or 4. The chance node represents the expected utility value for all these states.

Since the number of states that can be obtained in a game is extremely high there is a depth limit set for calculating the Expectimax value.

As we increase the depth, Expectimax is able to consider more levels in it calculation and hence becomes more efficient. Thus with a depth of 4, Expectimax is easily able to reach till the 4096 tile and in some cases even 8192. However the number of calculations involved also increases considerably and therefore the time taken for each step also increases.

Algorithm 1 Expectimax

```
1: function EXPECTIMAX(board, depth)
2:     if game ended or depth = 0 then
3:         return heuristic value of board
4:     if agent is max then
5:         let α := −∞
6:         for a ∈ A(s) do
7:             new_board = execute_action(a, board)
8:             α := max(α, EXPECTIMAX(new_board, depth − 1))
9:     else
10:        if agent is chance then
11:            let α := 0
12:            for tile ∈ get_empty_tiles(board) do
13:                board_with_2 = generate_number_2(tile)
14:                board_with_4 = generate_number_4(tile)
15:                α :=α + 0.9 ∗ EXPECTIMAX(board_with_2, depth − 1)/num_empty_tiles
16:                α :=α + 0.1 ∗ EXPECTIMAX(board_with_4, depth − 1))/num_empty_tiles
```

3.3 Monte Carlo Tree Search

The possible actions and states in the 2048 game are defined well and are dependent only on the previous state. This feature enables the game to be modeled as a Markov Decision Process. The Monte Carlo Tree Search (MCTS) algorithm is a heuristic and online search algorithm that finds the solution to a problem by

running random simulations (called monte carlo simulations) on a given state and picks the best action. MCTS consists of 4 steps which can be visualized in Fig. 2:

1. **Selection**- From a given state, an action is chosen that maximizes the value, $Q(s,a)+c\sqrt{\frac{\log \sum_a N(s,a)}{N(s,a)}}$. Where $Q(s,a)$ is the action value of a state s where action a is taken. The parameter c is the exploration parameter where higher values of c are assigned to infrequent actions. $N(s)$ is the amount of times a particular state s is reached and $N(s,a)$ is the amount of times an action a is taken from a state s.
2. **Expansion** - A child node is added to the Monte Carlo tree which was obtained from the selection step.
3. **Simulation** - Until the terminal state is reached or the specified *depth* of tree is traversed, a simulation is performed by repeatedly choosing random actions.
4. **Backpropagation**- once the value in the new node is updated, backpropagation from the new node to the root node is performed to update the remaining tree. This process leads to an increase in the number of simulations at each node.

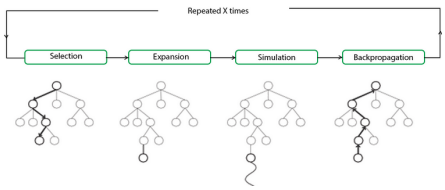

Fig. 2: One iteration of MCTS [2]

As we increase *rollout*, the number of simulations increase and the number of random moves considered increases. This increases the exploration and increases the chances of finding the best action. However, this increases the execution time. Algorithm 2 outlines the Monte Carlo tree sampling. The hyperparameter *depth* is the number of future states considered and it is used to limit the depth of the Monte Carlo tree since there are a large number of states possible. The reward function is used to score the actions taken which determines the values of different nodes in the search tree. In this case, the reward function is considered as the merge score after a move is made. This is chosen due to the fact that the objective of the game is to obtain high scores after merging and the merge score is a heuristic that quantifies this action.

Algorithm 2 Monte Carlo Tree Search

function MAKE_MOVE(s, a)
  if $a \in A(s)$ **then**
    $new\_s, merge\_score \leftarrow G(s, a)$
    fill empty cell with either a 2 (90% probability) or 4 (10% probability)
    return $new\_s, merge\_score$
  return $s, 0$
function SIMULATE($s, depth, Q, T, N$)
  if $depth = 0$ or game ended **then return** 0
  if $s \notin T$ **then**
    for $a \in A(s)$ **do**
      $N(s, a) \leftarrow 1$
      $Q(s, a) \leftarrow G(s, a)$
    $T \leftarrow T \cup s$
    return Rollout $(s, depth)$
  else
    $a \leftarrow \arg\max Q(s, a) + c\sqrt{\frac{\log \sum_a N(s,a)}{N(s,a)}}$
  $new\_s, merge\_score \leftarrow MAKE\_MOVE(s, a)$
  $N(s, a) \leftarrow 1 + N(s, a)$
  $q \leftarrow merge\_score + SIMULATE(new\_s, depth - 1, Q, T, N)$
  $Q(s, a) \leftarrow (q - Q(s, a))/N(s, a) + Q(s, a)$
  return q
function ROLLOUT($s, depth$)
  if $depth = 0$ or game ended **then return** 0
  for $a \in A(s)$ **do**
    $new\_s, merge\_score \leftarrow G(s, a)$
    if $merge\_score > best\_merge\_score$ **then**
      $best\_merge\_score = merge\_score$
      $best\_a = a$
  $new\_s, merge\_score \leftarrow MAKE\_MOVE(s, best\_a)$
  return $merge\_score + 0.75 * ROLLOUT(new\_s, depth - 1)$
  =0

3.4 Q learning

Feature Slicing/Tuple Slicing Q learning: Three feature-slicing sampling techniques have been used to create overlapping patterns between the features. This is to ensure that the Q-learning agent learns the similarities between various parts of the board (collaborative learning) instead of concentrating on separate disjointed portions. Now, two horizontal pairs of 4x3 and two vertical pairs of 3x4 sub-matrices have been used to create the features_pair list.

For the features_triplets list,the following triplets have been defined: verticals (three 4x2 matrices), Horizontals(three 2x4 matrices), X_except_top_left(all 3x3 sub-matrices except the one starting from the top left corner), X_except_top_left(all 3x3 sub-matrices except the at the top right corner), X_except_top_left(all 3x3 sub-matrices except the one at the bottom left corner), X_except_top_left(all 3x3

sub-matrices except the one at the bottom right corner) Similarly, for the features_quartets list, combinations of (1. four rows 2. four columns, 3. four 3x3 squares, from the 4x4 grid) have been considered. We define the heuristic as the sum of the weighted features for the pairs, triplet and quartet tuples.

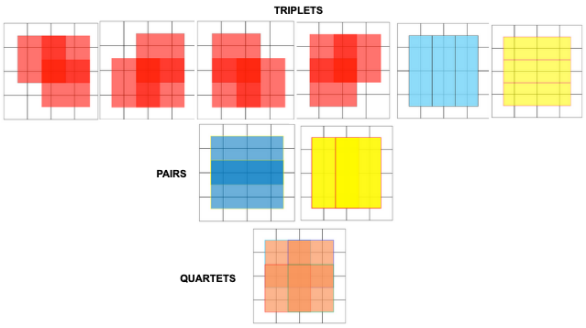

Fig. 3: Tuple configuration for features [5] [8]

Each episode starts with the initialisation of a game. Until the game reaches its terminal state, we keep evaluating the above defined heuristic to get the optimal values and actions for a state. If the value of Epsilon is not 0, the agent also takes a set of random decisions to explore the environment.At each of the steps we check which possible actions leads to a state with the highest value. We then move to that state, and use the Bellman Equation to make a back-propagation update for the previous state. We used the usual parameters of Q-Learning namely discount and reward to control the flow of agent in the parametric environment space.

$$NewQ(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma max Q'(s',a') - Q(s,a)]$$

Fig. 4: Bellman Equation

Since the 2048 game involves millions of possible steps, if we introduce a discount after every step, we risk slowing down the learning. We observed this behaviour when the learning became extremely slow after 5000 episodes, when we used a discount of 0.999. Hence we opted for a discount of 1 for all future trials. Similarly, we tried varying the value of Epsilon to introduce stochasticity to enable the Q-Learning agent to learn from the environment as much as possible. This deteriorated the learning and we saw that the agent started to slow down after around 200 episodes. We believe that since 2048 already has

Algorithm 3 R Learning - N-Tuple based Q-Learning

function GENERATE_FEATURES(s)
    generate feature pairs with orders of 16 {1,0}
    generate feature triplets with orders of 16 {2,1,0}
    generate features quartets with orders of 16 {3,2,1,0}
    return concatenated arrays for pairs, triplets, quartets
function BASIC_REWARD(s, a)
    $s' \leftarrow$ take action a on s.copy
    return $s'.score - s.score$
function EVALUATE_HEURISTIC
    return sum of the features from GENERATE_FEATURES(s)
function UPDATE_FEATURES_WITH_WEIGHTS($s, \delta weights$)
    apply the δweights to the features of the state s
function EPISODE($\alpha, discount, reward = BASIC\_REWARD$)
    Initialise Game with an instance
    while not game ended **do**
        $action, best\_value \leftarrow 0, -\infty$
        if ϵ aided learning **then**
            $test \leftarrow$ take random action on s.copy
            $val \leftarrow$ EVALUATE_HEURISTIC(test)
            if $val > best\_value$ **then**
                $best\_val \leftarrow$ val
        else
            for $action \in (possibleactions)$ **do**
                $test \leftarrow$ take action on s.copy
                if val > best_value **then**
                    update best_action, best_value
        if not initial state **then**
            $reward \leftarrow R(s, a)$
            $dw \leftarrow \alpha*$(reward+discount*best_value - prev_best_value)/no.of features
            UPDATE_FEATURES_WITH_WEIGHTS(s,a)
            take best_action on state s and set previous_best_value with current best_value
            add new tile to Game instance
    dw$\leftarrow -\alpha*$(reward + previous_best$_value$)$/numberoffeatures$
    UPDATE_FEATURES_WITH_WEIGHTS(s,a)
function TRAINING($number\_of\_episodes$)
    for $episode \in (number\_of\_episodes)$ **do**
        EPISODE($\alpha, discount, reward = BASIC\_REWARD$)
        if new best score recorded **then**
            save agent state in a file
    return stats and results
function TEST($number\_of\_gameplays\_requested$)
    load agent state from file
    for $gameplay \in (number\_of\_gameplays\_requested)$ **do**
        instantiate Game and play the game with the agent state
    Display the grid for the best game play recorded

very high stochasticity, (as random tiles are introduced at random places at the beginning of each move) introducing more randomness into the system would only destabilise the solution to a point of no return (very slow learning).

4 Experimental Results

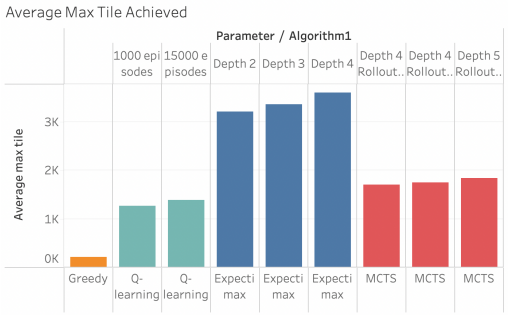

Fig. 5: Average Max Tile Achieved in 15 runs

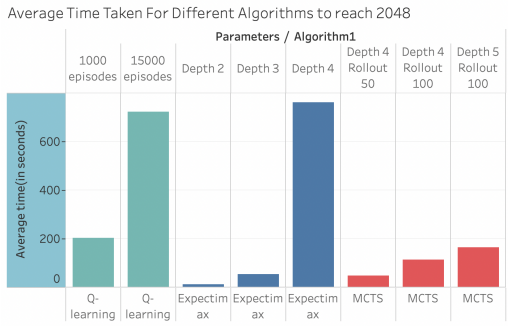

Fig. 6: Average Time Taken to Reach 2048 Tile

We ran the above mentioned algorithms for 15 runs using different parameters and observed the following results:

4.1 Methodology

Our evaluation of the 3 algorithms is performed with the following criteria against the Greedy Best First Search baseline 1. Average Max Tile Achieved in 15 runs 2. Average Time Taken to reach 2048 tile.

4.2 Results

Fig. 5 and Fig.6 are plots that compare the average max tile achieved and average time taken to reach the 2048 tile by different algorithms. For Q -Learning, we took the sum of the training and test run times.

4.3 Discussion

We can see from the above plots that the Expectimax algorithm performs much better than MCTS, Q-Learning and the baseline, even with lower values of depth. Greedy Best First Search did not achieve 2048 and its max tile achieved was 512 amongst all the runs while all other algorithms were able to reach the 2048 tile. We can deduce from the average max tile values from Fig.5 that MCTS reaches the 2048 tile more number of times than Q-Learning.

We observed that in some game plays of MCTS, in order to achieve a higher merge score, the algorithm tried to combine some of the higher valued tiles without considering the possibility that the game will end. Hence the algorithm sometimes lacks foresight and cannot always recognise a move that is advantageous in the long term.

The time taken for lower depth values is much lower in Expectimax as compared to other algorithms, and it still achieves better scores. With increasing depth, the average time taken increases considerably, without much of an effect on the max tile achieved. The weakness of Expectimax is that it requires us to specifically define the heuristics particular to a problem. Without pre-existing domain knowledge of the problem, this would not be possible.

The training time for Q-Learning is considerably high and the statistics reported in the graph are for 1000, 15000 episodes. It is possible for Q-learning to produce much better results when run for more episodes, but it would take over 6 hours to run 100000 episodes, owing to lower computational limits. However, once the training is complete, it takes less than 1 second to reach the 2048 tile, which is considerably lower than any of the other algorithms.

5 Conclusions

Expectimax is observed to be the perfect candidate for this use case beating the MCTS and Q-Learning algorithms. All the tested algorithms (except Greedy Best First Search) were able to solve the game and reach 2048. But Expectimax was able to achieve a high score of 8192 with a depth of 5 using the Snake Heuristic. The next best algorithm was observed to be MCTS, which was able to reach 4096 with depth 5 and roll out 100, using the merge score as the heuristic. The N-Tuple based Q-learning was also able to achieve 2048 and 4096 but exhibited larger training time. Future improvements include running Q-learning for more episodes, observing the results achieved and harnessing GPU support to test out more complex heuristics.

References

1. G. Cirulli. 2048. [Online]. Available: https://gabrielecirulli.github.io/2048/
2. GeeksforGeeks. 2022. ML — Monte Carlo Tree Search (MCTS) - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/
3. StackOverFlow Discussion https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048
4. Multistage Temporal Difference Learning for 2048-Like Games https://ieeexplore-ieee-org.elib.tcd.ie/stamp/stamp.jsp?tp=&arnumber=7518633&tag=1
5. A Very Simple and Fast Reinforcement Learning Agent for the 2048 Game, Alex Bachurin https://github.com/abachurin/2048
6. UI for the game https://github.com/kiteco/python-youtube-code/tree/master/AI-plays-2048
7. 2048 MDP Solver https://github.com/huntermills707/2048MDPsolver
8. Jaśkowski, W. (2017). Mastering 2048 with delayed temporal coherence learning, multistage weight promotion, redundant encoding, and carousel shaping. IEEE Transactions on Games, 10(1), 3-14.
9. Tao, J., Wu, G., Yi, Z., Zeng, P. (2020, August). Optimization and Improvement for the Game 2048 Based on the MCTS Algorithm. In 2020 Chinese Control And Decision Conference (CCDC) (pp. 235-239). IEEE