

PROFESSIONAL TRAINING REPORT
at
Sathyabama Institute of Science and Technology
(Deemed to be University)

Submitted in partial fulfillment of the requirements for the award
of Bachelor of Technology Degree in

Information Technology

By

GNANAMANI CT
(Reg. No.38120026)



DEPARTMENT OF INFORMATION TECHNOLOGY
SCHOOL OF COMPUTING

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited with Grade "A" by NAAC
JEPPIAAR NAGAR, RAJIV GANDHI SALAI, CHENNAI - 600 119

JANUARY 2021



SATHYABAMA INSTITUTE OF SCIENCE AND TECHNOLOGY

(Established under Section 3 of UGC Act, 1956)
Jeppiaar Nagar, Rajiv Gandhi Salai, Chennai -
600119 www.sathyabama.ac.in



SCHOOL OF COMPUTING

BONAFIDE CERTIFICATE

This is to certify that this Project Report is the bonafide work of
Name **GNANAMANI CT (Reg.no.38120026)** who carried out
the project entitled “**2048 GAME**” under our supervision from
Jan 2021 to April 2021.

Internal Guide

Dr.Bevis Jinila

Head of the Department

Dr.SUBHASHINI M.E., Ph.D. ,

Submitted for Viva voce Examination held on _____

Internal Examiner

External Examiner

DECLARATION

I **GNANAMANI CT (38120026)** hereby declare that the Project Report entitled **2048 GAME** done by me under the guidance of **Dr.Bevis Jinila** at **SATHYABAMA UNIVERSITY** is submitted in partial fulfillment of the requirements for the award of Bachelor of Technology degree in **INFORMATION TECHNOLOGY**

DATE:30/01/2021

PLACE:CHENNAI

SIGNATURE OF THE CANDIDATE

ACKNOWLEDGEMENT

I am pleased to acknowledge my sincere thanks to **Board of Management of SATHYABAMA INSTITUTE OF SCIENCE AND TECHNOLOGY** for their kind encouragement in doing this project and for completing it successfully. I am grateful to them.

I convey my thanks to **Dr. T.Sasikala M.E., Ph.D, Dean**, School of Computing , **Dr.SUBHASHINI M.E., Ph.D. ,** Head of the Department of Information and Technology for providing me necessary support and details at the right time during the progressive reviews.

I would like to express my sincere and deep sense of gratitude to my Project Guide **Dr.Bevis Jinila** for her valuable guidance, suggestions and constant encouragement paved way for the successful completion of my project work.

I wish to express my thanks to all Teaching and Non-teaching staff members of the **Department of Information Technology** who were helpful in many ways for the completion of the project.

TRAINING CERTIFICATE



RETECH
SOLUTIONS PVT. LTD.
An ISO 9001:2015 certified company

INT: RE20S067

TO WHOMSOEVER IT MAY CONCERN

This is to certify that **GNANAMANI CT** student of **SATHYABAMA INSTITUTE OF SCIENCE AND TECHNOLOGY, BTECH/IT** has undergone the Internship Training through online in our concern entitled **PYTHON** from **12th Aug 2020 to 26th Aug 2020** in relevant departments related to their academic studies.

During the above period, the performance was good & we wish great success in all your future endeavours.

J. meru

Technology drives the world. We are the people who are behind the technology.

Authorised Signature



No-31, 1st Floor, Alagesan Street, Tambaram West, Chennai-45.

Office : 044-42077204 | +91-8122266653

www.retechsolutions.in

COURSE CERTIFICATE



gnanamani chidambaram
is here by awarded the certificate of achievement for
the successful completion of
Game development using PyGame

A handwritten signature in blue ink, reading 'S.P. Balamurugan', positioned over a circular blue stamp that contains the GUVI logo.

S.P.Balamurugan
Co-founder, CEO

Certificate issued on: April 20 2020

Verified certificate ID: 539rA18J7588p41n48

Verify certificate at: www.guvi.in/certificate?id=539rA18J7588p41n48

CERTIFICATE OF COMPLETION

This is to certify that

GNANAMANI.CT

successfully completed

5 Studies in the

Python Course

This course covers the fundamentals of
Python for software development.

Certificate URL: pragate.com/course_certificate/87672907qf490a

August 15th 2020

Masa Kato

.....
Certification verified by Masa Kato
Founder & CEO of Pragate

 **pragate**

ABSTRACT

Python is a programming language with high range of applications. In this project, I have used Python with the Tkinter GUI along with Numpy and Matplotlib to create the 2048 game. This game is available in the play store as an app. I have made the web-version of this game using simple functions and classes in Python .

2048 is played on a 4×4 grid, with numbered tiles that slide smoothly when a player moves them using the four arrow keys. Every turn, a new tile will randomly appear in an empty spot on the board with a value of either 2 or 4. (Probability of 90%: 2, 10%: 4) Tiles slide as far as possible in the chosen direction until they are stopped by either another tile or the edge of the grid. If two tiles of the same number collide while moving, they will merge into a tile with the total value of the two tiles that collided. All the playing methods are the same, all you have to do is move numbers on a grid to merge them to form the number 2048; after all, you can keep on playing the game by creating larger numbers. In each move, a new number of displays (2 or 4). The player has to move the numbers using arrow keys and try to collide the same two numbers which create a new number with the total sum of the two numbers. The gameplay design is simple that the user won't find it difficult to use and navigate.

TABLE OF CONTENTS

CHAPTER No.	TITLE	PAGE No
	ABSTRACT	8
	LIST OF FIGURES	10
	LIST OF TABLES	11
1	INTRODUCTION	12
	1.1 PROJECT PREQUISTICS	12
2.	HOW TO PLAY 2048 GAME	12
3.	INSTRUCTION	14
4.	DISCUSSION	15
	4.1 TASK DEFINITION	16
	4.2 APPROACH	17
5	CONCLUSIONS & FUTURE WORK	23
	REFERENCES	24
	APPENDIX	
	A.SCREENSHOTS	25-29
	B. SOURCE CODE	

LIST OF FIGURES

FIGURE NO.	FIGURE NAME	PAGE NO.
1.1	2048 PLAY DIAGRAM	12-14
1.2	2048 FLOWCHART	15

LIST OF TABLES

TABLE NO.	TABLE NAME	PAGE NO.
1.1	REFLEX TABLE	14

INTRODUCTION:

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 expect lmax. 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.

Project Prerequisites

The prerequisites are as follows:

1. Basic concepts of Python
2. Tkinter

In this article we will look python code and logic to design a 2048 game you have played very often in your smartphone. If you are not familiar with the game, it is highly recommended to first play the game so that you can understand the basic functioning of it.

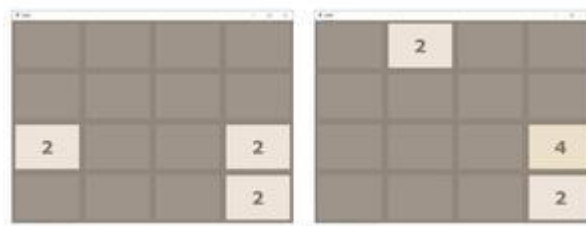
How to play 2048:

1. There is a 4*4 grid which can be filled with any number. Initially two random cells are filled with 2 in it. Rest cells are empty.



2. we have to press any one of four keys to move up, down, left, or right. When we press any key, the elements of the cell move in that direction such that if any two

identical numbers are contained in that particular row (in case of moving left or right) or column (in case of moving up and down) they get add up and extreme cell in that direction fill itself with that number and rest cells goes empty again.



3. After this grid compression any random empty cell gets itself filled with 2.



4. Following the above process we have to double the elements by adding up and make 2048 in any of the cell. If we are able to do that we wins.



5. But if during the game there is no empty cell left to be filled with a new \2, then the game goes over.

4	2	16	2
32	You	Lose!	16
8	128	32	4
2	16	4	2

In above process you can see the snapshots from graphical user interface of 2048 game. But all the logic lies in the main code. So to solely understand the logic behind it we can assume the above grid to be a 4*4 matrix (a list with four rows and four columns). You can see below the way to take input and output without GUI for the above game.

INSTRUCTIONS

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.

II. Task Definition

I. State-based Definition **Player input:** actions (up, down, left and right) based on the current state **Computer input:** adds a '2' tile on an empty grid randomly based on the current state **Output:** result state, reward (instant) **Terminal state (success):** One '2048' tile **Terminal state (failure):** All adjacent tiles occupied and have different values, i.e. no more moves available

ii. Evaluation Metrics The main goal of this project is to get a '2048' tile as fast as

possible. The success of the system is measured by the state variables, including the number of tiles occupied, tile values, number of moves taken, etc.

Infrastructure and Baseline

In this project, we have chosen an open-source simulator of 2048 written in C++ as our infrastructure

The codes are modified to suit our needs. Our baseline for this project is a random agent, where a random move of up/down/left/right is chosen at every action step. Comparisons with the baseline are made in the later parts.

Approach

i. Reflex Agent (by Chun Wai LEE)

a. Agent definition

The reflex agent acts as a baseline to demonstrate how Expectimax search outperforms simple reflex agents. It performs maximization over all possible moves. Given the current state, a score is calculated for each action based on several heuristics. The action that leads to the highest score is chosen. To be specific, for each direction ('a', 'w', 's' and 'd'), the program first simulates the state of the board by sliding to that way. It then gathers information about the state (e.g. values and positions of each tile of the board), which is used to compute some parameter values (e.g. number of empty tiles and possible merges). A final score is given to that direction by merging these values with different weights. The movement to the direction that will result in the best state is performed as the actual move. This is repeated until a terminating condition is met, i.e. no more moves available.

b. Heuristics & weightings

Three heuristics were chosen:

- Sum of the tile values
 - Number of empty tiles
 - Number of possible merges (i.e. orthogonally adjacent tiles with the same value)
- Each parameter contributes to a sub-score and the final score is determined by multiplying each sub-score by a pre-defined weight and summing them up together.

The heuristic weights were optimized by gradually adjusting each coefficient and selecting the combination that leads to the best results.

Eventually, we have selected the following settings:

- Sum of tile values: 300
- Number of empty tiles: 270
- Number of merges: 520

In other words,

the heuristic score is computed by $\text{sum_of_tile_values} * 300 + \text{number_of_empty_tile} * 270 + \text{number_of_merges} * 520$.

c. Results

The result of running the reflex agent for 10,000 times are summarized as follows:

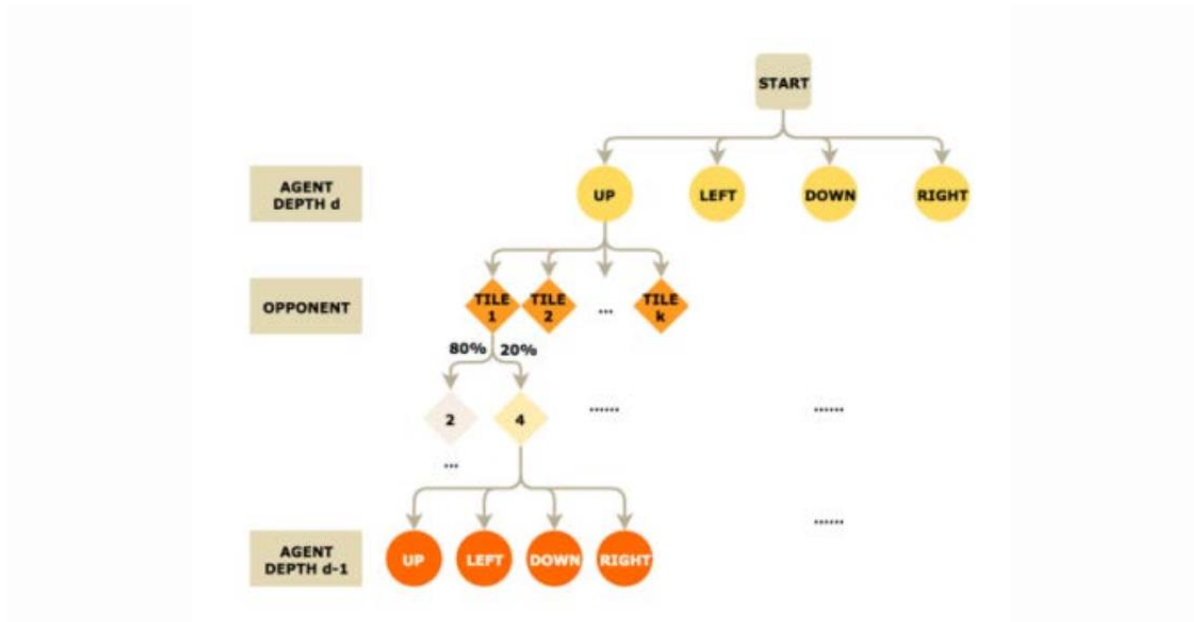
Max tile value	32	64	128	256
Frequency	01	118	5	173
Percentage	11.18%	51.73%	33.56%	3.53%

- Maximum score: 26,544
- Minimum score: 1,584
- Average score: 7,273
- SD of score: 352

Features

- Moves can be undone with the click of a button.
- Games can be saved in user-specified files to be resumed later.
- Grid can be resized to preference.
- In addition to the keyboard, there are two kinds of mouse interaction:
- Swipe: click and hold and then move in the direction desired
- Click: each section of the grid (top, bottom, left, right) when clicked will move the tiles in a different direction.
- The cursor will be an arrow pointing in the direction the tiles would go if you clicked.

- The colors in the grid can be rerandomized at any time (without messing up your game).



d. Discussion

The reflex agent only considers the current state and does not predict the future as deep as Expectimax. Therefore, as expected, it does not perform well. It reaches 128 and 256 mostly, 512 rarely and never gets to 1024. This agent, however, runs rather fast. We were able to execute 10,000 iterations in about 10 seconds, meaning accuracy is compromised for better efficiency. There is room for improvement for this type of agent. Firstly, the choice of heuristics could be further enhanced to favor certain states that are more likely to achieve the goal.

For example,

States with larger values on the edges of the board are favored to push tiles with huge values to the side and avoid them from blocking the small values to merge. Moreover, the proportion of heuristic weights could be improved by adopting different meta optimization algorithm. In this project, only a limited choice of heuristics was employed since having more parameters complicates the optimization of heuristic weights as more

weighting combinations will have to be tested

ii. Depth-limited Expect Imax (by Ming Hong LUI)

a. Expect Imax definition

For this search method, the game is broken down into a pseudo-two player game, with the player agent taking one of four actions 'w', 's', 'a' or 'd', which pushes the grid towards the chosen cardinal direction, and a computer 'adversary', who randomly spawns a new '2' tile where available. Given these behaviors, we can see that Expectimax search can comfortably describe the state transitions, as the goal is for the player to obtain the highest score possible. Based on this, two node classes were defined, maxNode and expNode, each storing the grid state, the action leading to the state, the parents and children nodes, and the score.

Respective functions, maxAgent and expAgent, were taken at the corresponding nodes to return a score value which propagates back to the parent.

b. Reward definition

The scores were calculated in two steps. If the node is a leaf node at the deepest level, it returns an expected value as the score, based on the composition of the state (see below). This is added to the instant rewards which is the merging of tiles following the actions taken. One nuance in this method is that since the expNode's action is simply to spawn a 2, all the instant reward for expNode is neglected as they cancel out due to being a constant, so the heuristic only takes merging into account to simplify the computation process. At the terminal state, the reward is simply defined as the sum of values across all tiles, which is also proportional to the number of moves made before the terminal is reached.

c. Expected value definition

Much similar to the heuristics for single player agents, the expected value at the

deepest leaves returns a sum of multiple intuitive parameters, including

1) **adjacency**, which refers to whether there are 'pairs' on the board ready to be combined in the next move, which is also value dependent;

2) **difference**, which refers to the numerical difference between adjacent occupied tiles, since we know that it is never ideal to have an '1024' next to a '2' as it clogs up future potential combinations;

3) **emptiness**, which is a measure of unoccupied grids in the game. This value is proportional to the number of empty grids, which fits its early-game importance, but diminishes in effect late-game;

4) **tile similarity**, which gives a comparison on how many tiles are not the same before the move. This is a crude estimate for stability of the board, as one of the optimal 'human' strategies is to stick to one corner as much as possible, since leaving the corner will expose the largest value to many chances of '2' spawning adjacent, which greatly disfavors tile merging.

The four parameters are varied with a few selected weights, and based on the test performances, the final weight distribution are shown here: {Adjacency, Difference, Emptiness, Similarity} = {2, -1, 100, -100}

d. Results

Due to limited computational power, the maximum traversable depth for this simulation is 3 (3 maxAgents and 3 expAgents), but the package is easily scalable 5 with no notable memory leak issues. Each depth from 1 to 3 have been run 50 times, and the score variances are shown below:

The idea would be to fix a V_0 value to all states (computable on the go). For example: $V_0(s) = a * (\# \text{ of free tiles on the board}) + b * (\text{number of tiles that can be merged}) + c * (\text{difference between adjacent tiles (more is bad)}) - \text{when } s \text{ is neither a losing or win state}$

Otherwise: $V_0(s) = W$ when s is a win state and $V_0(s) = L$ when s is a loose state.

e. Discussion From the results,

We can see that expectimax has a rather high fluctuation in performances. This is understandable given the many different possible states that are branching from each node, and the path taken will diverge quickly. Expectimax has demonstrated to be a capable tool to solve 2048, but its efficiency is still to be discussed. Notably during the simulations, its speed decreased drastically at higher depths. In depth 4, it took around 10 seconds to finish making one move. Therefore, it is worth looking into the memory and space complexities. Treating both maxAgent and expAgent into one depth level, the maximum branching factor for this search is 60 (4 cardinal directions for maxAgent and 15 possible empty spaces for expAgent). However, realistically speaking, the actual branching factor is much lower, and has a tendency to decrease later into the 6 game since, the grid will be filled with more clutter (lower value tiles before merging with high value tiles), together with more invalid moves in maxAgent. While it is hard to provide a reasonable approximate due to the highly random process, we can say that the upper bound for space complexity is $O(bd * \text{node size})$. Further pruning, after eliminating invalid moves, is also likely inefficient given the random nature of expAgent. In terms of realistic improvement in efficiency, the data structure should be one of the key aspects. Currently `std::vector[16]` is adopted for each grid state storage, which is fairly large in size and computationally expensive. To improve, a custom notation can be considered, as each tile value can be simplified into its $\log_2$ digits, where 2048 becomes a mere 11. With a dedicated class definition, one may even consider implementing bitwise manipulation to improve the computational speed. However, these improvements are beyond our capability for now for this one-month project.

iii. Markov Decision Process (by Adrien PROST)

In this discussion we will see if 2048 can be modelled as an MDP and if it can be solved efficiently using either value iteration or Q-value reinforcement learning. An intuitive way of modelling the game as an MDP would be: S = set of all possible states of the game until a 2048 tile is created: set of combination of all different tiles that are either blank or have values under 2048 = {empty, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024} For a 4x4 game board this gives us $|S| = 11^{16}$ different states (assuming all can be reached which is probably not the case) $A = \{A, S, W, D\}$ which describes the directions we choose to tilt the board, depending on the state, some of the actions are “illegal” because they would not change the position of the tiles currently on the board. $T(s, a, s')$: for each legal action a , there are a certain number of states s' that can be reached (equal to the number of empty tiles), with probability uniformly distributed (i.e. the board tilts and a new 2 tile spawns in any empty tile) $R(s, a, s')$: this depends on what kind of heuristics we want to set for a state. In this project we have chosen to give states values based on some heuristics, instead of using a reward function.

VALUE ITERATION

Let's first see how we would model an agent that could solve the MDP using value iteration

Now let's assume we have found convenient/coherent weights a, b, c, W, L for our heuristic. The idea would be then to apply value iteration on all states where $V(k+1)(s) = \max_a \sum_{s'} T(s, a, s') V(k)(s')$ The idea would be to iterate over k until the values converge.

The idea would be to fix a V_0 value to all states (computable on the go). For example: $V_0(s) = a * (\# \text{ of free tiles on the board}) + b * (\text{number of tiles that can be merged}) + c * (\text{difference between adjacent tiles (more is bad)})$ – when s is neither a losing or win state

Otherwise: $V_0(s) = W$ when s is a win state and $V_0(s) = L$ when s is a loose state

Now let's assume we have found convenient/coherent weights a, b, c, W, L for our heuristic.

However, this poses a problem since it requires us to store the $V(k)(s)$ values and $|S|$ has an upper bound of 11^{16} which is not feasible. There are probably ways of optimizing the storing of values but this is out of the scope of the course.

REINFORCEMENT LEARNING

Now let's see how we could solve the MDP using reinforcement learning. The idea would be to let the agent play the game (using random moves or with a fixed policy) and at every step we take we compute the q-value of that step in the following way:

- All the q values of the states are initialized at 0
- We use a reward function $R(a, s, s') = a * (\text{number of tiles merged}) + b * (\text{difference between adjacent tiles in state } s') + c * (\# \text{ of possible merges in the resulting state})$ (where s is the initial state, and s' the resulting state with the newly spawned random '2' tile)
- To compute the new q value every time we take a step, we use a learning rate A (that can vary over time), and compute the new q-value the following way: $Q(s, a) = (1 - A) * Q(s, a) + A * (R(a, s, s') + \arg\max_{a'} Q(s', a'))$ After some learning using random moves
- We can continue learning using with a probability P a policy that uses the action which has the highest Q-value. Again the same problem in the value iteration process arises, storing all of the $Q(s, a)$ would take too much memory and hence some other storing methods should be thought to solve this problem using MDP's.

Topology for optimality

To win the game, we must combine similar numbered tiles which combine to form a bigger numbered tile for example, two tiles numbered 2^n can be combined to form a tile

numbered $2n+1$. Since we can only move in the four cardinal directions, the two tiles 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 0 5 10 15 m-values Mathematical Analysis of 2048, The Game 5 must be adjacent to each other, either horizontally or vertically. This connects the game with topology. In an ideal configuration, the tiles must not only be adjacent to similar numbered tile but the higher numbered tile so formed must be adjacent to another tile of the same number and so on with the highest numbered tile being left in the end. It is preferable to have monotonicity in the grid along a line.

The position of head should be at a place where it can combine with other tiles easily without the new randomly generated tile breaking the chain midway. In other words, it would be better that the highest numbered tile is at the corner of the grid.

The resulting configuration would be a snake line (Image 1). Image 1: Snake line through the grid [1] However, practically such a configuration is hard to achieve due to non-deterministic nature of the game. The tiles will not land always exactly where we need them to be.

So this can be classified as a case for Best-Case Optimality. Since we are actually combining two adjacent tiles to form a new higher number, which is actually the sum of the two tiles, it is similar to Pascal Triangle (Image 2). The snake line (Hoang, 2014) in the above discussion, sounds good but is hard to replicate since it restricts the movement to only a single direction. But in Pascal's triangle, we need only restrict two directions and can even use third one although sparingly. Experimenting with this, we find the gameplay not only smoother but are also able to create the 2048 tile with ease and win the game (Image 3). This approach is closer to the stochastic optimality for the original game.

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 confidence that it can be used to solve the real problem in the real world. On the other hand, we found that finding a good, simple and effective heuristic is not that easy. It needs both the insight into 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 started the project.

Based on the above results, we can conclude that 2048 could be learned in multiple ways, from the simplest reflex agents to reinforcement learning. Simple algorithms may provide less satisfactory results, based on the terminal state reached, but are quick and time-efficient. Expectimax search was an improvement given the outcomes, yet high space and time complexity at larger depth, which supposedly further improves the result, may deter it from being a feasible agent. Therefore, given the discussion, it may be wise to turn towards feature-based learning methods, to design more efficient and generalized learning methods for better performance.

Future Work

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

Implementing a better heuristic function:

The heuristic we are using right now is complicated and computationally expensive, we can find more patterns 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 state space, 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.

REFERENCES:

- [1] 2048 - Wikipedia. [https://en.wikipedia.org/wiki/2048_\(video_game\)](https://en.wikipedia.org/wiki/2048_(video_game)).
- [2] 2048 AI - Using no hard-coded knowledge about the game. <https://github.com/ronzil/2048-ai-cpp>.
- [3] 2048 simulator with neural network. <https://raulmrebane.github.io/2048zero/>.
- [4] A simple AI for 2048. <https://github.com/ovolve/2048-AI>.
- [5] Keras.js. <https://github.com/transcranial/keras-js>.
- [6] Numba. <https://numba.pydata.org/>.
- [7] Gabriele Cirulli. 2048. <https://gabrielecirulli.github.io/2048/>.
- [8] David Foster. AlphaGo Zero Explained In One Diagram. <https://medium.com/applied-data-science/alphago-zero-explained-in-one-diagram-365f5abf67e0>.
- [9] Demis Hassabis and David Silver. AlphaGo Zero: Learning from scratch. <https://deepmind.com/blog/alphago-zero-learning-scratch/>, [10] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift.

SOURCE CODE:

```
import numpy as np

import matplotlib.pyplot as plt


NUMBER_OF_MOVES = 4

SAMPLE_COUNT = 50


SPM_SCALE_PARAM = 10

SL_SCALE_PARAM = 4

SEARCH_PARAM = 200


from game_functions import initialize_game, random_move,\
    move_down, move_left,\
    move_right, move_up,\
    check_for_win, add_new_tile


def get_search_params(move_number):
    searches_per_move = SPM_SCALE_PARAM * (1+(move_number // SEARCH_PARAM))

    search_length = SL_SCALE_PARAM * (1+(move_number // SEARCH_PARAM))

    return searches_per_move, search_length


def ai_move(board, searches_per_move, search_length):
    possible_first_moves = [move_left, move_up, move_down, move_right]
```

```

first_move_scores = np.zeros(NUMBER_OF_MOVES)

for first_move_index in range(NUMBER_OF_MOVES):

    first_move_function = possible_first_moves[first_move_index]

    board_with_first_move, first_move_made, first_move_score = first_move_function
(board)

    if first_move_made:

        board_with_first_move = add_new_tile(board_with_first_move)

        first_move_scores[first_move_index] += first_move_score

    else:

        continue

for _ in range(searches_per_move):

    move_number = 1

    search_board = np.copy(board_with_first_move)

    game_valid = True

    while game_valid and move_number < search_length:

        search_board, game_valid, score = random_move(search_board)

        if game_valid:

            search_board = add_new_tile(search_board)

            first_move_scores[first_move_index] += score

            move_number += 1

best_move_index = np.argmax(first_move_scores)

best_move = possible_first_moves[best_move_index]

search_board, game_valid, score = best_move(board)

return search_board, game_valid

```

```

def ai_play(board):

```

```

move_number = 0

valid_game = True

while valid_game:

    move_number += 1

    number_of_simulations, search_length = get_search_params(move_number)

    board, valid_game = ai_move(board, number_of_simulations, search_length)

    if valid_game:

        board = add_new_tile(board)

        if check_for_win(board):

            valid_game = False

        print(board)

        print(move_number)

    print(board)

    return np.amax(board)

```

```

def ai_plot(move_func):

    tick_locations = np.arange(1, 12)

    final_scores = []

    for _ in range(SAMPLE_COUNT):

        print('thing is ', _)

        board = initialize_game()

        game_is_win = ai_plot(board)

        final_scores.append(game_is_win)

    all_counts = np.zeros(11)

    unique, counts = np.unique(np.array(final_scores), return_counts=True)

    unique = np.log2(unique).astype(int)

```

```
index = 0
```

```
for tick in tick_locations:
```

```
    if tick in unique:
```

```
        all_counts[tick-1] = counts[index]
```

```
        index += 1
```

```
plt.bar(tick_locations, all_counts)
```

```
plt.xticks(tick_locations, np.power(2, tick_locations))
```

```
plt.xlabel("Score of Game", fontsize = 24)
```

```
plt.ylabel(f"Frequency per {SAMPLE_COUNT} runs", fontsize = 24)
```

```
plt.show()
```