Documentation for Eric Shi's 2048 algorithm

Mini User Manual:

The program you just downloaded is an algorithm for the popular game 2048, alongside with the game itself. You should see 3 separate python files if everything is installed correctly. Before running the program, make sure that you are running the latest version of python, and have the random module installed for python. Once everything is installed, simply run the "gameloop.py" file and enjoy!

Once you have ran the file, a console should appear, at that point you have to type "a" to watch the algorithm play the game, "m" to play the game yourself. You can also type in "speed" to activate turbo mode and put the algorithm into turbo time, or "c" to put in a custom board state and start from there.

The rules of the game are simple. It is played on a 4 by 4 board, and on each cell there may be a number. You can then input a direction (w for up, a for left, s for down, and d for right), and all the numbers will then be sent in that direction. If a number collides with another number that is the same as itself (for example, a 2 collides with a 2), then the numbers will merge and become the sum of those two numbers (4 in this case). The goal of the game is to get the biggest number you can on the board. The game is over when the board cannot move in any of the 4 directions (https://en.wikipedia.org/wiki/2048_(video_game)).

If you choose to have the algorithm play it, it'll play with a depth of 3. It will take around 10-20 seconds per move depending on your computer's hardware, and will take about 1gb of ram, and the entire game will last around 40-60min. If you choose a custom board, it is highly recommend to not exceed a depth of 3 as preliminary testing showed that it will take at least 8gb of ram for a depth of 4.

Design Guide:

The program is split up between 3 separate python files with distinct purposes. The Board2048.py file contains all necessary code to have the basic game ready to be played. The

main_algorithm.py file contains all the code necessary for the algorithm to run, and it relies on the board2048.py file to run properly. Finally the gameloop.py file is the only file meant for users to interact with, as it contains the code which validates the user's inputs.

Board2048.py

For the board2048 file, the only important code present is the board2048 class. The class consists of a board variable, which stores the board as an array of arrays, specifically 4 arrays each of length 4. The __init__() has two versions depending on if there was a premade board inputted. If there was not a premade board inputted, it'll generate an empty board and then place 2 random tiles using the gen_new_tile() (which works by randomly selecting a random tile, verifying that tile is free, and then randomly selecting either a 2 or 4 at a 90%/10% probability respectively). Else, it'll use the premade board.

The board2048 class has a few important methods which rely on helper methods. The method swipe(self, in_dir) is one of the most important and will be run often. It will take the inputted direction and attempt to shift the board in that way. It first runs the respective verify method to make sure that the swipe direction is valid, before actually swiping the board.

The check works by seeing if the board after a swipe is the same as the board is before the swipe, and if yes then the swipe direction is invalid. There is a swipe check method for each of the 4 directions.

The swiping is done by manipulating the individual row or column that is affected. This section will focus on the merging left, but the general idea is applicable to all 4 directions. When the merge_left() method is called, it runs through a for loop for each individual row. In each for loop, a pointer position 0 is created, and a second for loop is started for each value in the. Then as long as the value in the second for loop is not a 0, it attempts to move the number to the pointer. If the pointer has an empty number, the value is inserted at that position. If the pointer has a value that is not the same as

the for loop value, the pointer increases by 1 and the for loop value is placed in the new pointer position. If the pointer value has the same value as the for loop value, the pointer value is multiplied by 2 and the pointer position is increased by 1.

The points method return the total value of the board in terms of how many points are present.

The method get_future() returns an array of 4 arrays that contain each possible board state after a swipe has occured and a value has been randomly generated. The first array represents a swipe up, second array swipe left, third array swipe down, and forth array swipe right. If the swipe is not possible, an empty array will be added to its respective position. The get_future() method relies on find_future(), which generates all the possible board positions after a random tile generation. Each individual array is generated by firstly getting what a board would look like after the swipe direction, then running find_future() on that board, then using the list that find_future() returns.

The eval_self() method returns a numerical representation of how good a board state. It is calculated by assigning points to each individual tile on the board based on its value, position, and position in relation to other values. For any individual tile, it takes the value that is present on it multiplied by a small value, then multiplies it by a value if it is next to the side of the board, and then another multiplicative bonus if its in the corner of the board, then it runs find_local_mul() and it multiplies it by the value that returns. The method find_local_mul() checks the 4 adjacant tiles up, down, left, and right if applicable then if it find the same value as the one in the tile it returns a very high multiplier at higher numbers, but less than at lower numbers. If the same value is not found, it then finds the largest adjacent number then applies a bonus based on the largest adjacent number. This way of calculating a board's total point incentivies putting large numbers in the corner, and priorities merges as much as possible, as well as the conditions nesseary to facilitate merging.

The method find_highest_tile() finds the the highest value.

The method detect_loss() checks if a board cannot be played anymore. It is done by verifing each swipe direction, and if none are valid it returns true, else it returns false.

The method raw_print() prints all the values without formatting, so high values can also be displayed

The method player_start() allows a user to play the board. While the board is not unplayable, it asks for a direction, verifies it with swipe(), then performs the action; repeating until the board is in a lost position.

main_algorithm.py

This file is dedicated for running the algorithm, and all of its associated functions. It relies on the board2048 class to function properly. It contains the classes algor2048 and future_board_states

The algor2048 class when created can either be based on a premade board, or it can generate a new board if one is not given. The variable self.sight is also assigned in the init.

The method export_boardstate() returns the board2048 that the algor2048 class is using.

The method do_move() performs the move inputted.

The method make_best_move() performs the move that trouver_best_move() reports. Then it generates a new tile using board2048.gen_new_tile().

The method trouver_best_move() reports the best move from a future_board_state class generated from the algor2048's board2048

The class future_board_state() is a tree that is built out of future_board_state nodes, and it recursively builds itself when called. When initialized, the variables tpoints, wpoints, apoints, spoints, dpoints, and answer are created. Then it takes all the possible futures of the board2048 class, and assigns it into arrays called w, a, s, and d respectivly. Then, the class calculates the points from each one of the arrays and assigns it to apoints, wpoints, spoints, and dpoints. The calculation is done by using an expectimax algorithm (https://www.geeksforgeeks.org/expectimax-algorithm-in-game-theory/). Firstly, each board in any given array runs self.eval_self() to return the points of the board.

Then that value is either multiplied by either 0.9 or 0.1 if the tile generated was a 2 or a 4. This aligns the algorithm closer to reality, as it'll predict the number 2 appearing more often, finally it adds up all the points from apoints, wpoints, spoints, and dpoints into tpoints (total points). After that, while the amount of layers in the tree isn't the right amount, it'll generate the next level.

The next level is generated by running self.generate_next_level(), which uses the arrays w, a, s, and d. It takes each one of the arrays, and turns each entry into another future_board_state class, repeating the steps above until the last level is reached.

After all levels have been generated, starting from the bottom most level until it reaches the root, it runs calc_partial_points(), where it grabs all the all the points from the future_board_state classes below it and adds it into tpoints.

Finally, the root runs calc_gradient_decent(), where it reassignes apoints, wpoints, spoints, dpoints to the points in a, w, s, and d respectivly. Then the root node runs self.find_best_move(), where it finds which direction has the most points and assigns it to self.answer.

Gameloop.py

This file contains all the code that is meant to abstract away the hard parts of the program. It contains two functions, replay and mainloop. Replay just asks if the player wants to replay the game, and if yes reruns mainloop.

Once mainloop is called, it prints out all the possible options that the program has to offer, and then accepts and validates an input.

The manual option creates a board and runs board.game_start(). At the end of the game, it prints out the points that the player achieved.

The automatic option starts a timer, starts counting the moves taken. While the game isn't lost, the algorithm finds the next best move, runs it, then prints out the time taken so far and the amount of

moves taken so far. Finally, once the game is lost, it prints out the total score achieved, total time taken and total moves taken.

Speed mode makes no changes other than visually changing the time taken by a factor of 0.75.

If a the custom board option is chosen, the program asks for 4 different inputs, corresponding to each row of the board. The program performs a simple validation on each input to ensure that it’s 4 comma seperated numbers (note it is intentional that users can insert non base 2 numbers). Then the user is prompted with either the option to play manually or have the algorithm play it for the user. This part reuses older code.

Secret option d (for debug) being an algorithm with an already pre-loaded board. Useful for testing something specific, such as beginning in the middle of a game to avoid slow early game where algorithm is slow.