Marianopolis College

Final Project Report

Oliver Stappas, Xu Zhe Zhang, Maxwell Zhixing Lee

Programming Techniques and Applications 420-LCW-MS

Professor Robert Vincent

May 6th 2019

Mini-manual:

What the program does

The program that we created allows us to, in a 3 by 3 matrix with up to 9 unknowns, calculate the value of unknowns if possible and create a possible solution if the matrix is not solvable given current information. The matrix has to satisfy the following condition: each column and row must be a arithmetic sequence, like 1,2,3 or 4,10,16 where 3 integers have the same gap, an algebraic illustration would be: a, a + d, a + 2d where a is the first value and d is the gap and a + 2d is the final element of the arithmetic sequence.

How to use the program

Open and run the "solver.py" file. The program will prompt the user to enter the name of a .in input file containing the 3 x 3 grid with integers and Xs. The program will then print the arithmetic square solution if there's only one solution and the program will print one version of the solution if there are are more than one solution to the matrix.

Design guide:

The "solver.py" file is divided into two sections. First there is the matrix class which contains multiple methods that are necessary for the program in the solver file to operate. The second part is the implementation which contains a function for computing the solution the input and additional code.

The matrix class

The first method is __init__(self), which is the internal method that allows the user to create a list which is the original matrix that you will perform all the operations on.

The second method is __repr__(self), which is the internal method that allows the user to print the matrix object in the format of readable, actual matrix in the following form:

X X X

X X X

X X X

The third method is construct_matrix(self, file), which is the method that transforms the data in the test file into a 2D list. The list will contain sublists which are the rows of the matrix.

The fourth method is x_counter(self), which allows the user to count the number of X in the matrix, this method is useful when there are exactly 9 X in the matrix,which means if you put any number to all the X, the returned solution will be a valid one.

The fifth method is matrix_(self), which returns the matrix object.

The sixth method is zero_Matrix(self), which allows the user to turn all the element of a matrix to 0.

The seventh method is construct_transpose(self, other) ,which returns the transpose version of the matrix object.

The eighth method is convert_int(self). Considering that the 2D list that the user will create will strictly contain string object even for the numbers. This method transforms the numbers in the form of string to integer, which allows the user to perform arithmetic operation.

The ninth method is basic_op(self, row), which is the building block of many methods that we will discuss afterwards. This method is the basic operation that allows us to calculate and return the integer value of X when it is possible, which means when there is only one X in the corresponding column or row. If X is the first element of a row or column we deduce the following: Since the difference between two consecutive values in a row or column in an arithmetic square is the same, the difference between the third and second values and the difference between the second and first values in a row or column will be the same. So, we can find the difference between the third and second element in a row or column and subtract the second element by that difference to find that value for X:

For example, in this row:

X 4 6

X = 2nd element - (3rd element - 2nd element)

X = 4 - (6 - 4) = 4 - 2

X = 2

Secondly, if X is the second element in the row or column, we deduce that, since the differences between X and the first and third elements are the same, X must be the middle value between the first and third elements, or the average between them:

For example, in this row:

5 X 15

X = (3rd element - 2nd element) / 2

X = (15 - 5) / 2 = 10 / 2

X = 5

Thirdly, if X is the third element in the row or column, using the same reasoning as if X is the first element in the row, we deduce that the difference between the third and second values and the difference between the second and first values in a row or column will be the same. So, we can find the difference between the first and second element in a row or column and we can add double this difference to the first element to find the value for X:

For example, in this row:

7 8 X

X = (2nd element - 1st element) * 2 + 1st element

X = (8 - 7) * 2 + 7 = 1 * 2 + 7 = 2 + 7

X = 9

The tenth method is value_replace(self, other, updated).The method allows the user to replace X with a value that will make the row to form an arithmetic sequence and the X in the transpose and the original matrix will be replaced. The variable “updated”,which is boolean ,will be turned to true once the X has been replaced,this will serve us in future operations.

The eleventh method is contains_X(self, contains_x),it returns a boolean value which indicates whether there is an X in the row or not.

The twelfth method is add_val(self, other, change),which is the method that allows the user add an arbitrary value (random value) in a position of X if no value_replace can be possibly performed. The way it works is it searches for the first row that contains two X’s such that it can replace an X in the row with a random value based on the already existing value in the row. If the existing value is not in the final position of the row, it replaces the X immediately after the existing value with the random value. If the existing value is in the last position of the row, it will replace the X immediately before the existing value with the random value. It must also be brought to attention that the matrix and its corresponding transpose will both be updated when the value is added. The variable “change” is only meant to indicate to the main code whether it managed to add a random value to the rows. If it didn’t, it means that the random value has to be added to the columns.

The thirteenth method is the set_matrix(self, matrix),which is the method that allows us to essentially help creating the copy of a matrix. This method will take a given matrix and object and then set the object's matrix to be the one passed to the method.

The fourteenth method is the verifier(self), which is the method that allows us to check whether the row is an arithmetic square or not, if not it will print false.

The implementation

The implementation basically asks the user for an input file then, using the methods defined the matrix class, finds and prints the solution to the incomplete arithmetic square.

A while loop is used in order to allow the user to continuously input test files. Then, a matrix and its transpose matrix will be initialized into the matrix class. The user is then asked to enter a file name for a .in file containing test data, and this name will be stored. The program then opens the file, then, using the construct_matrix(self, file) method, the program uses the input file to create the corresponding 2D list. The program then checks for the exception where every element in the input matrix is X. In this case, the program always prints the solution where all the X's are zeros.

Otherwise, the program continues by using the construct_transpose(self, other) method in order to create a 2D list corresponding the transpose of the original inputted matrix. The program then converts all numbers in the matrix and the transpose matrix from strings into integers. The original matrix is saved to the variable "grid_backup" which is initialized into the matrix class. It is deepcopied into listy. The same procedure is done to the transpose matrix. Now the solving process begins.

Then, this is where the function main_code(grid, grid_2) comes into play: Updated is initialized to "False". The program then introduces a while loop in order to allow the program to continuously perform operations on the inputted matrix. Then, using, “updated” the value is set to “True” if the matrix and its transpose can be updated using the value_replace method. Then, “contains_x” is initialized to “False”, and is then updated to “True” if the matrix contains any X’s with the grid.contains_X method. If the matrix is incomplete and the remaining values could not be calculated, then “change” is initialized to “False” and a random X is replaced with a random value with the add_val method. “change” is then updated. If “change” is still “False”, then, with with the add_val method, one of the X’s in a column will be replaced with a random value. If the matrix is finally complete, the while loop is broken, updated is set to “False”, and the function returns the changed grid and its transpose.

However, a second while loop is used for the situation if it could not find the correct solution to the method. The program introduces a brute force method in that it increments the random value in add_val by 1 every time it could not find the correct solution,this way of attacking the problem is not included in the main_code.s

Other possibility to consider:

What if it the input was an n x n (> 3 x 3) matrix?

This is done under the assumption that there is a possible solution. As long as there is at least two values in single row/column, the rest of the values can be calculated within that row/column. This can simply be done by using a function similar to basic_op that finds the difference between the two values and calculates ‘d’ by dividing the difference by the number of gaps between the two values (Recall that in an arithmetic square each row/column is a, a + d, a + 2d,..., a + nd). In the case that it reaches a point where it can no longer calculate values because there are too many gaps (X’s), it will insert a “random” value into the matrix based on the existing value within the row/column using a function either similar or identical to the add_val function. It will then continue to either calculate the values or add “random” values until the matrix is filled. To ensure that it produces whole numbers, it will increment the “random” value by one and pass the matrix through the function again until it finds a solution with whole numbers. In the case that the entire matrix is X’s, it will simply fill the entire matrix with 0’s. As the size of the matrix increases, it will take longer to solve since it fills each position one at a time rather than in groups or chunks. The matrix grows according to a_n = (n+1)^2, for n = 1,2,3,... Since the number of elements is filled one at a time and the number of elements increases exponentially with the increase in size of the matrix, it can be expected that the time to solve the matrix may also increase exponentially.

Task distribution for the project:

Contribution by all:

- the process of debugging
- ideas of algorithms and problem solving

Xu Zhe Zhang did:

- coding of the matrix class (all the methods inside of it)
- coding of the function of basic_op
- coding of the initialization of the program (input of the file, creation of the function of the 2D list (original matrix and the transpose version of it))
- part of the writing of the design guide
- the mini-manual of the report (the general function part)
- the writing of task distribution for the project
- improvement of interface(the representation of result)

Oliver Stappas did:

- writing of design guide and part of the mini-manual (the way to use the program part)
- the improvement and annotation of the program
- the general structural design improvement of code (creating functions and methods)
- ensured that code conforms to PEP 8 Style Guide for Python (improved readability)

Maxwell Zhixing Lee did:

- the implementation of main algorithms used to solve matrices with more than one solution
- the coding of value_replace function
- the coding of add_val function
- the final conversion of multiple functions into methods of matrix class
- partial writing of the design guide
- writing of other possiblities to consider