Pythonogram: Picross in Python

1 Introduction

Pythonogram is an interactive nonogram puzzle solver. It is essentially just like Picross developed for the Nintendo DS, except written in Python. Pythonogram consists of two grids: a master grid in which certain cells are filled and others are blank, and an empty grid in which all cells are blank. Players are presented with the empty grid and required to makes guesses. These guesses correspond to cell locations and are "good" guesses if the corresponding location is filled in the master; guesses are "bad" otherwise. The objective of the game is to find all of the filled cells in the master. Overall score is based on how many guesses were required to do so, and the amount of time it took.

So that players do not have to guess randomly, they are presented with clues. Specifically, for each row and column, there is a list of numbers corresponding to the number of filled cells in that row or column, along with their distribution. This effectively makes the game a logic puzzle known as a *nonogram*. From Wikipedia:

> "...the numbers are a form of discrete tomography that measures how many unbroken lines of filled-in squares there are in any given row or column. For example, a clue of "4 8 3" would mean there are sets of four, eight, and three filled squares, in that order, with at least one blank square between successive groups."

It is your job to not only create such a puzzle, but to develop the interaction in which users can make guesses against that puzzle.

2 Development

Development consists of first generating the puzzle, along with its annotations. An interaction model must then be created such that a user can make guesses and those guesses provide corresponding feedback.

2.1 Puzzle generation

Two puzzles must be generated: a master puzzle containing filled and empty cells, and the scratch puzzle on which correct user guesses will go. Both puzzles should be represented as lists of lists. In the master, each cell should be filled with the integer zero or one, selected randomly using Python's random number library. The scratch puzzle should be filled with spaces (strings).

The dimensions of the puzzles can be whatever you like, but those dimensions must be specified using a variable. Specifically, set two variables at the top of your program, `rows` and `columns`, and use these variables to create your table. Throughout your code, the programming logic should rely on these variables, rather than assumed dimensions of the puzzle.

2.2 Tomography calculation

For each row and column in the master, a numeric representation of the filled cells (1's in this case) along that dimension must be developed. Thus, for a 3×3 matrix, there will be nine distinct representations. This representation is actually a set of integers denoting the number of consecutive filled cells that are present. Each number, in other words, represents a group of filled cells. Although groups can identified, where the groups start cannot. See Figure 1 for a visual explanation.

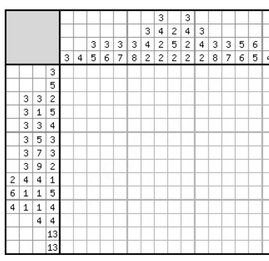

Figure 1: An example unfilled nonogram puzzle. Numbers along the left and top represent groups of filled cells along the respective dimension. The zeroth row has only the number three, meaning there are three consecutive filled cells in that row; the remaining cells are blank. The four values atop the seventh column mean that there are four respective filled groupings: one three cells long, another four cells long, and two that are two cells long.

Given a master puzzle, there are several ways in which the tomography can be generated. It is recommended that you first develop your algorithm using pencil and paper. Before getting into long rows, first consider how this is done for smaller rows—three or four cells, for example. Once you have an algorithm in mind, create small programs to test it out: hard code small one-dimensional lists to see if your thinking is correct. Fortunately, each row and column are independent, so once you understand how to do it in one dimension, you understand how to do it for all.

After your algorithm is developed, it must be applied to all rows and columns of the master. For a given row, you should store the tomography in a list. These lists should be aggregated into a larger list, containing all other rows in the puzzle. This should also be done for columns. Figure 2 provides an example for three calculated rows.

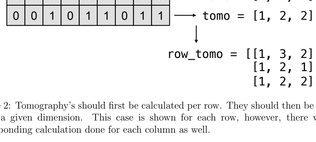

Figure 2: Tomography's should first be calculated per row. They should then be aggregated along a given dimension. This case is shown for each row, however, there would be a corresponding calculation done for each column as well.

2.3 User interaction

After the board and fill patterns are calculated, it is time to play the game. A round consists of a user being presented with the scratch board—empty initially—and asked for two coordinates:

```
Please enter your guess (row, col):
```

Once the coordinates are provided, your program must check to see if the corresponding cell in the master puzzle is filled. If so, that cell in scratch should be changed from a space to an 'x'. If the guessed cell in the master is empty, no action should be taken.

Boards should be printed using the same "pretty printing" style that was developed in class: each element of the matrix is separated by vertical bars; each row separated by alternating + and – signs. The printed board should also include filled-cell encodings along the right and bottom (a deviation from the way the encodings are presented in Figure 1). An unfilled 4×4 scratch board, for example, might look like this:

```
+-+-+-+-+
| | | | | 1 2
+-+-+-+-+
| | | | | 1
+-+-+-+-+
| | | | | 2 1
+-+-+-+-+
| | | | | 1
+-+-+-+-+
 1 2 2 1
 1     1
```

An example interaction should first print the board, then ask for a user guess. The underlying board should then be updated, and the loop started again. The master board is used only for verification and is never displayed.

2.4 Game over

The game should end once the user has found all filled cells. Upon ending, a report should be generated listing guess accuracy and game duration. Guess accuracy is the number of correct guesses divided by the total number of guesses taken. Remember, guesses to previously revealed filled cells should not be counted as correct.

You can calculate difference in time by using Python's `datetime` library:

```
>>> import datetime
>>> x = datetime.datetime.now()
>>> # ... wait for 5 minutes, 32.65 seconds
>>> y = datetime.datetime.now()
>>> y - x
0:05:32.650000
```

3 Deliverable

You should submit a single Python file containing your entire implementation. To receive full credit, you must implement this specification. Deviations or alterations will not be noticed, except in cases where such changes make it difficult to evaluate whether the specification has been met. Such situations could hinder your grade. If your alteration comes from an ambiguity in this write-up, please get clarification from the instructor in writing, and document the clarification in your code.

With respect to grading, this is what is expected:

Running code What you submit should run. Even if you have not implemented all parts of the assignment, the subset of parts that are implemented should be free of Python errors.

Programming structures You should use the programming concepts discussed in class. Specifically, demonstrate your understanding of the various structures we have covered: data structures, control structures, and decision structures. If implemented correctly, this homework will force you to use every concept discussed thus far.

Generality The dimensions of your board should be specified using variables. Your program should run regardless of the value of these variables. Marks will be deducted if there are places in the code that assume a given dimensionality.

Comments There should be comments at the top of the program specifying your name, and the purpose of the program. Additionally, there should be comments throughout the code announcing which sub-problem this section of code is implementing.