Alphadoku

1 Introduction

Alphadoku is a lot like Sudoku, but differs in two ways: first, it uses letters instead of numbers;[1] second, it doesn't necessarily have to be played on a 9×9 grid.

2 Helper functions

Before attempting to solve the challenge, there are several functions you must write. These functions will make your job easier, so getting them right is important.

`get_board(csvfile)` Reads a CSV file into a list-of-lists. For example, consider a CSV file with two lines; each line containing two elements. The function would thus return a list containing two sublists, each with two elements.

Note that the CSV file represents a partial puzzle. This means that all lines will contain the same number of elements, but not all of those elements will be letters. Elements that are letters represent a valid entry; elements that are not valid entries—values that your program must figure out—are represented as +, known as the *empty character*.

`in_row(board, row, column)` For a given `board`, returns a list of characters used within that `row`. The empty character should not be included in the return value.

`in_column(board, row, column)` For a given `board`, returns a list of characters used within that `column`. The empty character should not be included in the return value.

`in_sector(board, row, column)` For a given `board`, returns a list of characters used within the "sector," of which `row` and `column` are inclusive. A sector is the smaller square matrix within the larger overall game. In classical Sudoku, the board is made up of nine 3×3 sectors. In generalized Sudoku, and thus Alphadoku, there are $\sqrt{k}$ sectors in a $k \times k$ grid (or, equivalently, k sectors in a $k^2 \times k^2$ grid). The empty character should not be included in the return value.

`get_blank(board)` Returns the coordinates of any empty character within the board. Although there may be several empty cells within the board, this function only needs to return one of them. The coordinates should be returned as a list in which the first value denotes the row and second value the column. If there are no empty cells remaining, an empty list should be returned.

`print_board(board)` Prints a board (list-of-lists) in a way that is presentable. You can reuse code from the past using the ASCII decorators (+, |), or something else. The important thing is that it is more presentable than printing a list-of-lists by default, and the Python characters that come with the list are not present.

`valid_characters(board)` For a given board, returns a list of potential characters for this board. The set of valid characters are the sequence of characters that can go in a given row (and hence by definition a given column). In classical Sudoku, for example, that would be the integers one through nine, since there are nine rows/columns in the board. In our case, that would be the lower-case characters, starting at 'a'.[2] Remember the work we did with ASCII characters—that 'a' starts at 97—to help build this set.

3 Puzzle Solving

Su/Alpha-doku puzzles can be solved using backtracking. Remember the general idea: at each step we ask whether we should stop; are we in a position—at the goal, for instance—where we can no longer make progress? If that is the case, we are done. If that is not the case, we establish a guess and move forward. When solving a maze, "establishing a guess" meant leaving a breadcrumb.

Your function should be declared as follows:

solve(board) Given a board, returns `True` or `False` depending on whether it was able to find a solution. If it is able to find a solution, the current board—by now solved—should be printed.

3.1 Base case

The base case is when there are no empty cells in the board. This situation, you can assume, means that the puzzle has been solved.

3.2 Recursive step

For an empty cell, each potential value needs to be attempted. That is, given a set of potential values P, place a value $p \in P$ into the empty cell and attempt to solve the board again. The set of potential values can be thought of as the relative complement of the set of valid values in the union of values already in that positions row, column, and sector. Formally,

$$valid\_characters \setminus (in\_rows \cup in\_columns \cup in\_sector) \tag{1}$$

There are two parallels between what should be done here and how we solved the maze:

1. Breadcrumbs allowed us to keep track of where we had been. In this case, those breadcrumbs are the alphabetic guesses from the complement calculation (Equation 1). In the event that the recursive step did not solve the puzzle, we replaced the breadcrumb with the previous value at that cell. This was optional for the maze game, but is imperative for Alphadoku.
2. If a recursive step found the goal, we immediately stopped searching for solutions (by returning `True`). The same optimization can be made here.[3]

4 Execution

When run, your program should first print the partial board (that read in from the file). It should then run the `solve` function on that board, and print the amount of time it took to run. See the previous homework (*Pythonogram: Picross in Python*) detailing how to time a function.

This homework comes with two boards—one 4×4 and another 9×9—on which you can can test your program.

5 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.

Structure A valid implementation of all functions outlined in Section 2, and `solve` outlined in Section 3. Their parameters and return values must be as specified, and correct.

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.

If you decide to use a technique or special feature of Python that we have not discussed, you must add a comment explaining what the code is doing. Points will be deducted for undocumented significant alterations from class concepts.

[1] https://en.wikipedia.org/wiki/Sudoku#Alphabetical_Sudoku

[2] Because we are using single characters as board elements, there is a limit to the size of the board we can support—you do not have to worry about this. That is, build your program such that the number of single characters are infinite. Your code, however, will not be tested for boards of row/columns sizes greater-than 26.

[3] If you do not do this, you will go on to produce all potential solutions for the puzzle. While this is okay, it will increase the amount of time it takes for your program to run. In the file that you submit, please only produce a single solution.