CSCI 3901 Assignment 4

Due date: 4pm Halifax time, Tuesday, November 16, 2021 in git.cs.dal.ca at
https://git.cs.dal.ca/courses/2021-fall/csci3901/assignment4/xxxx.git
where xxxx is your CSID. This repository is being created for you.

Complete either problem 1 or problem 2. If you choose to do both then you must tell me which to mark. With no indication, we will mark problem 1 as a default.

The marks for the problems are different: problem 1 has fewer marks than problem 2, reflecting some added difficulties in problem 2. The assignment will be out of the marks for problem 1; any marks beyond that in problem 2 will be treated as a bonus.

Goal

Work with exploring state space. In both problems, my own solution uses recursion. Your solution does not need to use recursion, but it might help you.

Background

Games are a domain in which the player searches through a set of possible local solutions to find an answer that satisfies all of the puzzle's constraints. This assignment has you create a solver for one of two games.

Problem 1 – Boggle

Background

In the game of Boggle, you are given a grid of letters and are asked to report as many words as possible from a path among the letters. A path starts at on location in the grid and goes to a next letter in any direction (left, right, up, down, or diagonally) to a direct neighbour. The letters along the path should be a word from a known dictionary. You may not use a letter more than once in any word. An example of a game of Boggle can be found at https://www.puzzle-words.com/

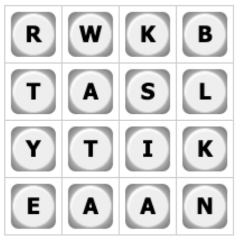

Figure 1 Sample Boggle game (from https://www.puzzle-words.com/)

Figure 1 shows a sample 4x4 Boggle grid in which you can find the English words rain, tail, silk, satin, nail, and yeti (to name a few).

Problem

Write a class called "Boggle" that accepts a dictionary of words and a puzzle grid (not necessarily square) and provides a sorted list of all the words found in the puzzle.

The class has at least 4 methods:

- boolean getDictionary(BufferedReader stream) – Read a set of words that are candidates for the solution. Each word will be on a line of its own. The end of the words is signalled either by the end-of-file marker on the stream or by a blank line in the stream. Return true if the words are all read and ready to use for puzzle-solving.
- boolean getPuzzle(BufferedReader stream) – Read a rectangular grid of letters that form the Boggle puzzle board. Every row of the puzzle should have the same number of letters. The end-of-file marker in the stream or a blank line in the stream marks the end of a puzzle. Return true if a puzzle is read and can be used for puzzle-solving.
- List<String> solve() – Do whatever you need to do to find all words of the dictionary that appear in the puzzle grid. Words can start at any entry of the grid and cannot re-use letters of the grid in the same word. Return the list of words found (in sorted order) as well as the path followed by the word (details on the format later).
- String print( ) – "print" the current puzzle as the returned string object.

You get to choose how you will represent the puzzle in your program and how you will proceed to solve the puzzle. You can also have any of these methods return an exception of your choice (that you document) in error conditions.

All words should be treated as case insensitive.

Inputs

The getDictionary( ) method will accept words with one word per line. A word must be at least 2 characters. A sample set of words (though with some 1 character words and some words with punctuation) can be found on timberlea.cs.dal.ca in /usr/share/dict/words

The getPuzzle( ) method will read the puzzle grid one line at a time. The grid of Figure 1 would then have the following input in the stream:

```
rwkb
tasl
ytik
eaan
```

ending with a blank line or an end-of-file marker.

Outputs

The print( ) method produces a String that can later be printed. That output should look like the puzzle's grid input.

The solve( ) method returns a list of strings found in the puzzle. The list has one string per unique word found and is sorted by the word. The format of each line is as follows:

<word> <starting X> <staring Y> <navigation sequence>

Where each component is separated by the others with a tab character.

<word> is the word found in the grid
<starting X> and <starting Y> are the grid coordinates of the start of the word, with the lower left corner having X and Y as both 1.
<navigation sequence> is a sequence of letters (no separation between them) to show how you navigate through the grid to form the desired word. The letters of the sequence (upper case) are

- L – go left
- R – go right
- U – go up
- D – go down
- N – go diagonally up and to the left
- E – go diagonally up and to the right
- S – go diagonally down and to the right
- W – go diagonally down and to the left

(the diagonal directions are the compass directions rotated 45 degrees counterclockwise).

For example, as we mention that Figure 1 contains the words rain, tail, silk, satin, nail, and yeti then those words would appear in the output of solve( ) as

```
“nail\t4\t1\tLUE”
“rain\t1\t4\tSSS”
“satin\t3\t3\tLDRS”
“silk\t3\t3\tDED”
“tail\t2\t2\gDEE”
“yeti\t1\t2\tDER”
```

If a word appears more than once in the grid then report the word that starts with the smallest X coordinate and, in the case of a tie, with the smallest Y coordinate.

Assumptions

You may assume that

- All input will be lower case.
- The dictionary of words is provided in sorted order.

Constraints

- You may use any data structures from the Java Collection Framework.
- You may not use an existing library that already solves this puzzle

- If in doubt for testing, I will be running your program on tiberlea.cs.dal.ca. Correct operation of your program shouldn't rely on any packages that aren't available on that system.

Notes

- Develop a strategy on how you will solve the puzzle before you finalize and start coding your data structure(s) for the puzzle.
- Work incrementally. First write the code to read in a dictionary. Test that. Next, write the code to read in the puzzle. Test that. Then, write the code to print the grid so you can verify that you read the grid properly. Test that. Last, write your code to solve the puzzle. Even then, write the code to solve the problem in just one direction of search to start with. Once that is working, add in another direction.
- Recall that you should first seek _a_ solution to solving the puzzle. That alone can be tricky in some instances. Even consider a brute-force version that tries all numbers in each cell.
- If you don't add some degree of cleverness to finding words then your solve( ) method will run for a very long time on larger grids. There are some marks for adding this efficiency so that we can work with non-trivial grids.

Marking scheme

- Documentation (internal and external), program organization, clarity, modularity, style – 4 marks
- List of test cases for the problem – 3 marks
- Clear explanation of how you are doing your solution (strategy and algorithm), why your strategy works, and what steps you have taken to provide some degree of efficiency (include in your external documentation) – 3 marks
- Reading in the inputs, creating the puzzle, and printing it – 3 marks
- Ability to list all words that appear in the puzzle with a correct path – 9 marks
- The effectiveness of your strategy to be efficient and to keep the running time of your program small (will be compared with timings from my solution on the same input data) – 3 marks

Problem 2 – Kakuro

Background

Kakuro is a puzzle that is similar to doing crosswords. You are given a rectangular grid with some blank cells. Single digits between 1 and 9 are placed in the empty cells. The puzzle is given an integer for each horizontal run and each vertical run of 2 or more empty cells; that integer is the sum of the digits placed in the run. No digit can be repeated within the run.

Figure 2 shows an example of a Kakuro game and its solution. For visual effect, the integer sum for each run is shown in the cell immediately to the left of a horizontal run or immediately above a vertical run. Although the grids are shown as 5x5, the playing dimensions are 4x4 and this assignment would treat this solution as a 4x4 grid.

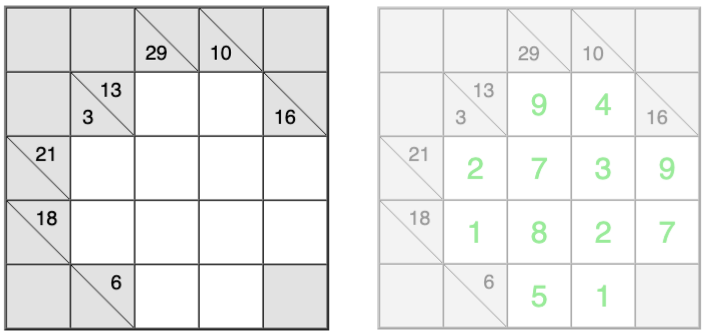

Figure 2 Sample Kakuro game (left) and solution (right) from https://www.kakuros.com/

Although there are a number of strategies used to solve this problem, you are not expected to become a master of Kakuro in this assignment nor do you need complex strategies to get the efficiency marks of this assignment.

Problem

Write a class called "Kakuro" that creates a puzzle grid, accepts horizontal and vertical runs for the puzzle along with the sum for the run, and provides a solution to the puzzle, if a solution exists.

The class has at least 5 methods:

- Constructor Kakuro(int width, int height) – create a grid with the given diemsions on which to place constraints.
- bollean addConstraint(int x, int, y, int length, int direction, int sum) – indicate that there should be a run of empty cells in the grid as given by the parameters and that the sum of those cells should be the given sum.

  The run starts at position (x, y) where the lower left corner of the grid is cell (1, 1). The run consists of "length" cells and their sum is "sum". The direction parameter tells you whether the run of cells goes horizontally (1) to the right or vertically downward (2).

  An example of the input for Figure 2 appears later.
- boolean readyToSolve( ) – returns true if you have all the information needed to solve the puzzle and false if there is information missing and the puzzle is not set to be solved.
- boolean solve( ) – place digits into the puzzle to provide a solution, if one exists. Return true if a solution exists and you've left the solution in the grid cell values. Return false if no solution exists to the puzzle.
- String toString( ) – Return a string that represents the puzzle. The method stores each row of the puzzle (top row first, bottom row last) in the string and separates the rows with a carriage return (\n). If we print the returned string then we should print the

picture of the puzzle.

Cells that are not part of any run are represented by a period (.). Cells that are empty are represented by an asterisk (*). Cells with a digit in them contain the digit itself.

Values for cells in one row appear side-by-side; there is no space between the cell values.

You get to choose how you will represent the puzzle in your program and how you will proceed to solve the puzzle. You can also have any of these methods return an exception of your choice (that you document) in error conditions.

Inputs

The addConstraint( ) method essentially makes spaces in a grid. Those spaces are linked to a sum constraint. The lower left corner of the grid is position (1, 1).

A sequence of the addConstraint() calls to create the puzzle of figure 2 are

```
/* Horizontal constraints */

addConstraint( 2, 4, 2, 1, 13 );
addConstraint( 1, 3, 4, 1, 21 );
addConstraint( 1, 2, 4, 1, 18 );
addConstraint( 2, 1, 4, 1, 6 );

/* Vertical constraints */

addConstraint( 1, 3, 2, 2, 3 );
addConstraint( 2, 4, 4, 2, 29 );
addConstraint( 3, 4, 4, 2, 10 );
addConstraint( 4, 3, 2, 2, 16 );
```

Outputs

The print( ) method produces a String that can later be printed. That output should look like the puzzle's grid input. If used on the unsolved puzzle of the sample input, you get the string

```
“.**.\n****\n****\n.**.\n”
```

That, when printed, would show on the screen as

```
.**.
****
****
```

```
.**.
```

After the puzzle is solved, the string from print( ) on the sample input puzzle would yield the output

“.94.\n2739\n1827\n.51.\n”

That, when printed, would show on the screen as

```
.94.
2739
1827
.51.
```

Assumptions

You may assume that

- Horizontal runs are only reported from left-to-right to addConstraint() and vertical runs are only reported from top-to-bottom to addConstraint().

Constraints

- A run in the puzzle (horizontal or vertical) must be separated from other runs by some background cell that will not take in a value.
- You may use any data structures from the Java Collection Framework.
- You may not use an existing library that already solves this puzzle
- If in doubt for testing, I will be running your program on tiberlea.cs.dal.ca. Correct operation of your program shouldn’t rely on any packages that aren’t available on that system.

Notes

- Develop a strategy on how you will solve the puzzle before you finalize and start coding your data structure(s) for the puzzle.
- Work incrementally. First write the code to create the puzzle grid. Test that. Next, write the code to read in constraints, maybe even starting only with one type of constraint. Test that. Then, write the code to print the grid so you can verify that you read the grid properly. Test that. You might even consider writing a method to test that you have all the constraints that you expect before proceeding to solve the puzzle; spending hours of debugging the solve( ) method only to find out that your input listed started a constraint at the wrong row or column is very frustrating. Last, write your code to solve the puzzle. Only after you have a solution working (on a tiny 2x2 puzzle) should you start implementing efficiency optimizations to your solution.
- Recall that you should first seek _a_ solution to solving the puzzle. That alone can be tricky in some instances. Even consider a brute-force version that tries all numbers in each cell.

- If you don’t add some degree of cleverness to finding words then your solve( ) method will run for a very long time on larger grids. There are some marks for adding this efficiency so that we can work with non-trivial grids.

Marking scheme

- Documentation (internal and external), program organization, clarity, modularity, style – 4 marks
- List of test cases for the problem – 3 marks
- Clear explanation of how you are doing your solution (strategy and algorithm), why your strategy works, and what steps you have taken to provide some degree of efficiency (include in your external documentation) – 4 marks
- Creating the puzzle grid and printing it – 3 marks
- Ability to solve small puzzles, up to 3x3 – 9 marks
- The effectiveness of your strategy to be efficient and to keep the running time of your program small (will be compared with timings from my solution on the same input data) – 6 marks