CSCI 3901 Assignment 3

Due date: Friday, March 8, 2024 at 11:59pm Halifax time. Submissions through your CS gitlab repository in https://git.cs.dal.ca/courses/2024-winter/csci-3901/assignment3/????.git where ???? is your CS id.

Problem 1

Goal

Work with inter-related information.

Background

We often gather user feedback on products. In our case, we want feedback on the difficulty level of wooden and wire puzzles. We could ask individuals to assign a difficulty level from 1 to 10 to each puzzle, but everyone might have different notions of what each ranking number meant. Instead, we can ask people to compare puzzles to say whether puzzle A is harder or easier than puzzle B. People often have an easier sense of comparing puzzles than of ranking a puzzle on its own.

Gathering an evaluation consensus from a crowd that is doing comparisons is more challenging than just averaging individual puzzle numeric difficulty levels. You might encounter cases where

- two puzzles aren't compared (a frequent case)
- one person seems inconsistent with puzzle A harder than B, B harder than C, and C reported harder than A
- one person thinks that puzzle A is harder than B and another person thinks that B is harder than A

The latter two cases arise because some comparisons are subjective and everyone can have their opinion. We generally resolve these cases with a concept of support. Support represents how what proportion of the votes favour one comparison outcome. We would say that if a comparison has at least x% of the overall votes then the comparison is valid with a support of x%. Then, in the possibility of contradictions, we would compute an outcome and ignore any comparisons that have less than some level of support, such as 5% support.

Once the low-support comparisons are removed, if two puzzles A and B still have comparisons that rank A both above and below B in difficulty then we would say that A and B have comparable or equivalent levels of difficulty.

You will develop a Java class that will help us understand the crowd consensus of puzzle difficulties, given this understanding of support and equivalence.

Problem

Implement a class called PuzzleLibrary that accepts comparisons on the relative difficulty of puzzles from the crowd and then answers questions on these comparisons.

Your PuzzleLibrary class must include the following methods, at a minimum:

- Constructor PuzzleLibrary(int support)
- boolean setSupport(int support)
- boolean comparePuzzles(BufferedReader streamOfComparisons)
- Set<String> equivalentPuzzles(String puzzle)
- Set<String> harderPuzzles(String puzzle)
- Set<String> hardestPuzzles( )
- Set<Set<String>> puzzleGroups( )

The constructor and setSupport methods will define a minimum support that any comparison must have to contribute to the reporting of equivalentPuzzles, harderPuzzles, hardestPuzzles, and puzzleGroups.

Your implementation of the methods can add exception throwing to the method signatures. You are responsible for making the code robust, including any methods or stubs provided to you as a start to the assignment.

PuzzleLibrary(int support) constructor

Create the PuzzleLibrary object. The "support" parameter identifies the level of support needed for comparisons between puzzles to take note of that comparison. The support is an integer to be divided by 100 to get the support percentage, so a 15% support level would have support = 15.

The support level is used in all other reporting methods of this class to base the work only on a given support level for comparisons.

boolean setSupport(int support)

Set a new level of support needed for this class in choosing which comparisons should be considered in reporting for the class. As in the constructor, the support is an integer to be divided by 100 to get the support percentage.

Return true if the new support level will be used. Return false if there is an error, in which case the old support level remains.

boolean comparePuzzles(BufferedReader streamOfComparisons)

Read in a set of comparisons between puzzles. These comparisons are added to the set of comparisons that we already know about. Each puzzle is represented by a string as its name.

Each line of the stream compares two puzzles; it consists of two strings separated by a tab character where the first string is the harder puzzle and the second string is the easier puzzle.

Return true if all the comparisons have been read into the library. Return false if some part of the comparison stream could not be read in. That latter case then has none of the input from the BufferedReader stream read in.

Set<String> equivalentPuzzles(String puzzle)

Return the set of all puzzles that would be deemed equivalent in difficulty level with the "puzzle" parameter. Consider equivalence to be transitive: if A is equivalent to B and B is equivalent to C then A is equivalent to C even if there aren't puzzle comparisons to draw the conclusion directly from A and C on their own.

Remember to report a puzzle as equivalent to itself.

Set<String> harderPuzzles(String puzzle)

Return the set of all puzzles that would be deemed harder in difficulty level with the "puzzle" parameter.

If puzzle A and B are equivalent in difficulty level and if puzzle C is harder than puzzle B in difficulty level, then equivalent puzzles A and B are not deemed harder than one another, but puzzle C is deemed harder than puzzle A by transitivity of being deemed harder than a puzzle equivalent to B.

Set<String> hardestPuzzles( )

Return the set of all puzzles that have no puzzle listed as harder than them, either by a direct comparison or by a comparison through an equivalent puzzle. Two equivalent puzzles are not deemed harder than one another, so two equivalent puzzles could both be listed as hardest puzzles.

Set<Set<String>> puzzleGroups( )

Our puzzle evaluators are only supposed to compare puzzles of the same type to one another. In other words, they should just compare the difficulty of string puzzles with other string puzzles, the difficulty of block puzzles with other block puzzles, and the difficulty of wire puzzles with other wire puzzles. Consequently, we can determine the types of puzzles by looking at which ones have some set of comparisons between them: if two puzzles are compared then we can conclude that they are of the same type of puzzle.

Of course, some newer puzzle evaluators make a few comparisons between puzzles of different types, but eliminating those comparisons as ones with low support then cleans up our groupings.

Using only comparisons with sufficient support, gather each the puzzles of one type into its own set. Then, return the set of all these sets of puzzles.

Assumptions

You may assume that

- All puzzle names are case-sensitive strings.
- Any string as a puzzle name given to you in comparePuzzles is a viable puzzle for you to track.

Constraints

- You may use any data structures from the Java Collection Framework.
- If you use a graph then you may not use a library package to for your graph or for the algorithm on your graph.
- If in doubt for testing, I will be running your program on timberlea.cs.dal.ca. Correct operation of your program shouldn't rely on any packages that aren't available on that system.

Marking scheme

- Documentation (internal and external), program organization, clarity, modularity, style – 4 marks
- Thorough (and non-overlapping) list of test cases for the problem – 3 marks
- Use of git as a proper version control system, meaning at least 4 meaningful, substantial, and spread-out (in time) commits – 1 mark
- Robustness of the code – 2 marks
- Marks for each of the requested methods:
    - comparePuzzles– 4 marks
    - equivalentPuzzles – 2 marks
    - harderPuzzles – 6 marks
    - hardestPuzzles – 4 marks
    - puzzleGroups – 4 marks

Marks for the constructor and setSupport are merged in with the rest of the methods through their test cases. You need the methods, but you don't get separate marks for creating what should be trivial setter methods.