CSCI 2600 — Principles of Software

Homework 5: Professors Network

Due: Friday, Mar. 29, 2024, 11:59:59 pm

Submission Instructions

- This assignment uses the same repository as Homework assignment 4, so when you are ready to start working on Homework 5, pull Homework 5 files from the repository by right-clicking on your Homework 4 project in Eclipse and selecting `Team` → `Pull...` Make sure that `When pulling` is set to `Merge`, then click `Finish`.
- Be sure to commit and push the files to Submitty. Follow the directions in the version control handout for adding and committing files.
- Be sure to include any additional files in your repo using **Team** → **Add to Index**.
- **Important:** You must press the **Grade My Repository** button, or your answers will not be graded.

IMPORTANT NOTES:

You should have package `hw5` with the usual directory structure. Write your code under `src/main/java/hw5` and your tests under `src/test/java/hw5` (shows as `hw5` under `src/test/java` in Package Explorer). If your directory structure is incorrect autograding will fail resulting in a grade of 0!

Read this document **entirely** before starting to work on your homework assignment.

Introduction

In this homework, you will put the graph you designed in Homework 4 to use by modeling the courses which professors teach at some university. It may be RPI or some other university. So, your solution should be general enough to work with data from any school, not just RPI. By trying out your ADT in a client application, you will be able to test the usability of your design as well as the correctness, efficiency, and scalability of your implementation. You may have to overhaul, or just tweak your implementation, once you try it with a larger load.

The application builds a graph potentially containing hundreds of thousands nodes and edges. At this size, you may discover performance issues that weren't revealed by your unit tests on smaller graphs. With a well-designed implementation, your program will run in a matter of seconds. Bugs or less ideal choices of data structures can increase the runtime to anywhere from several minutes to 30 minutes or more. If this is the case you may want to go back and revisit your graph implementation from Homework 4. Remember that different graph representations have widely varying time complexities for various operations and this, not a coding bug, may explain the slowness.

The ProfessorPaths Application

In this application, your graph models a network of course instructors. Each node in the graph represents a professor, and an edge ⟨`Prof1, Prof2`⟩ indicates that `Prof1` has taught a course which `Prof2` has also taught. There should be a separate edge for every course, labeled with the course's catalog code. For example, there are six courses which both Professors Kuzmin and Goldschmidt have taught, so there should be six edges between them.

Your graph should not store reflexive edges from professors to themselves.

You will write a class `hw5.ProfessorPaths` (in file `ProfessorPaths.java` in package `hw5`) that reads the course data from a file, builds a graph, and finds paths between professors in the graph. We supply just one dataset with RPI Course data in a file called courses.csv but you should build additional datasets that follow the same format as the provided example in order to test your ADT implementation thoroughly. You are not required to write a `main()` method as a driver for your application; nevertheless, we encourage you to do so, both for your own convenience in testing and for the satisfaction that comes with creating a complete, standalone application. Do not implement the graph in `ProfessorPaths`. `ProfessorPaths` should use your graph class from `hw4` by composition.

As you complete this assignment, you may need to modify the implementation and perhaps the public interface of your Graph ADT from Homework 4. Briefly document any changes you made and why in `answers/hw5_changes.pdf` (no more than 1-2 sentences per change). If you made no changes, state that explicitly. You don't need to track and document cosmetic and other minor changes, such as renaming a variable; we are interested in substantial changes to your API or implementation, such as adding a public method or using a different data structure. Describe logical changes rather than precisely how each line of your code was altered. For example, "I switched the data structure for storing all the nodes from a `___` to a `___` because `___`" is more helpful than "I changed line 27 from `nodes = new ___();` to `nodes = new ____();`".

Leave your graph in the `hw4` package where it was originally written, even if you modify it for this assignment. There is no need to copy files or duplicate code! You can just `import hw4` and use it in Homework 5. If you do modify your `hw4` code, be sure to commit your changes to your repository.

Note that Submitty will be compiling your code in the order of package names, i.e., everything in `hw4` package will be compiled before compiling the `hw5` package. This way, it is guaranteed that your `hw5` code is able to access anything from the `hw4` package. However, it also means that forward references would not work. In other words, if you attempt to access any `hw5` contents from `hw4` code (although you should never need this!), it is not going to compile.

Do not make `ProfessorPaths` a subclass of your `Graph` class. Instead, use `Graph` in `ProfessorPaths` by composition.

Problem 1: Getting the RPI Course Data

Before you get started, obtain the RPI Courses dataset. We have not added this file to your Git repositories because it is fairly large. Instead, download the file from the course Web site. Store the file in `data/courses.csv`. (You might have to add the data directory under the project root.)

IMPORTANT: Do not commit `courses.csv` into your repository! There is a limit on each repository and committing such a large file may break this limit. This is easily taken care of in Eclipse Team Provider for Git, as you can simply exclude `courses.csv`. For those not using Eclipse, make sure to add `courses.csv` to your `.gitignore` file.

Take a moment to inspect the file. A CSV ("comma-separated value") file consists of human-readable data delineated by commas, and can be opened in any advanced text editor, like Notepad++. We do not recommend using basic text editors like standard Microsoft Windows Notepad, a spreadsheet like Microsoft Excel, or Eclipse to view or edit CSV files because such tools may not show the contents of CSV files correctly or may make undesired changes to the files causing parsing errors when your code attempts to read the data. Each line in `courses.csv` is of the form:

```
"professor","course"
```

where professor is the name of a professor, course is the catalog course code of a course which the professor has taught, and the two fields are separated by a comma. Note that while there exist several different variations of the CSV format, the one used in this assignment assumes that every value must be enclosed in double quotes (the `"` character). These double quotes only serve as delimiters, i.e., they are not part of the actual professor's name or course code.

Problem 2: Building the Graph

The first step in your program is to construct your graph of the Professors Network from a data file. We have written a class `ProfessorParser` that may help. `ProfessorParser` has one class method (i.e., a method with the `static` modifier), `readData()`, which reads data from `courses.csv`, or any file structured in the same format. `readData()` creates in-memory data structures: a `Set` of all professors and a `Map` from each course to a `Set` of professors who have taught that course. These are not the data structures you want, however; you want a `Graph`.

We have also included a `main()` method which takes the file name as a command-line argument and then calls `readData()`. To add `courses.csv` as a command-line argument in Eclipse, go to `Run` → `Run Configurations`. In the `Run Configurations` window, choose the "Arguments" tab. In "Program Arguments", type `data/courses.csv`.

Later on, when measuring coverage, you can comment out the `main()` method. Since it is never called from the tests, it will decrease your percent coverage. If you choose not to use our parser code, you can get rid of the file altogether. In this case though, you would need to write your own parser code.

You may modify `ProfessorParser` however you wish to fit your implementation. You may change the method signature (parameters and return value) of `readData()`, or you may leave `readData()` as is and write code that processes its output. The only constraint is that your code needs to take a filename as a parameter, so that the parser can be reused with any input file.

At this point, it's a good idea to test the parsing and graph-building operations in isolation. Verify that your program builds the graph correctly before you go on. The assignment formally requires this in Problem 4.

Problem 3: Finding Paths

The real meat (or tofu) of `ProfessorPaths` is the ability to find paths between two professors in the graph. Given the name of two professors, `ProfessorPaths` searches for and returns a path that connects them through the graph. How the path is subsequently used, or the format in which it is printed out, depends on the requirements of the particular application using `ProfessorPaths`.

Your program should return the shortest path found via breadth-first search (BFS). A BFS from node u to node v visits all of u's neighbors first, then all of u's neighbors' neighbors, then all of u's neighbors' neighbors' neighbors, and so on until v is found or all nodes with a path from u have been visited. Below is a general BFS pseudocode algorithm to find the shortest path between two nodes in a graph G. For readability, you should use more descriptive variable names in your actual code than are needed in the pseudocode:

```
start = starting node
dest = destination node
Q = queue, or "worklist", of nodes to visit: initially empty
M = map from nodes to paths: initially empty.
    // Each key in M is a visited node.
    // Each value is a path from start to that node.
    // A path is a list; you decide whether it is a list of nodes, or edges,
    // or node data, or edge data, or nodes and edges, or something else.

Add start to Q
Add start->[] to M (start mapped to an empty list)
while Q is not empty:
    dequeue next node n
    if n is dest
        return the path associated with n in M
    for each edge e=⟨n, m⟩:
        if m is not in M, i.e., m has not been visited:
            let p be the path n maps to in M
            let p' be the path formed by appending e to p
            add m->p' to M
            add m to Q

If the loop terminates, then no path from start to dest exists.
The implementation should indicate this to the client.
```

Here are some facts about the algorithm.

- It is a loop invariant that every element of Q is a key in M
- If the graph were not a multigraph, the `for` loop could have been equivalently expressed as "for each neighbor m of n"
- If a path exists from *start* to *dest*, then the algorithm returns *a* shortest path. Note that there may exist more than one shortest path.

Many professor pairs will have multiple paths. **For grading purposes, your program should return the lexicographically (alphabetically) least path.** More precisely, it should pick the lexicographically first professor at each next step in the path, and if those professors have both taught several common courses, it should print the lexicographically lowest catalog code of a course which they both have taught. The BFS algorithm above can be easily modified to support this ordering: in the for-each loop, visit edges in increasing order of m's professor name, with edges to the same professor visited in increasing order of catalog code. This is not meant to imply that your graph should store data in this order; it is merely a convenience for grading.

Because of this application-specific behavior, **you should implement your BFS algorithm in `ProfessorPaths`** rather than directly in your graph, as other hypothetical applications that might need BFS probably would not need this special ordering. Further, other applications using the graph ADT might need to use a different search algorithm, so we don't want to integrate a particular search algorithm in the graph ADT.

Using the full RPI Courses dataset, your program must be able to construct the graph and find a path in just a fraction of a second on your PC/laptop and on Submitty. Submitty autograder will be using different datasets to assess the quality of your code. Some of those datasets might be significantly larger than RPI Course data. When running tests, we will set a 10-second timeout for each test suite. Note that if your solution exceeds this limit, your process is terminated and the output file is truncated. So, you may see an error message about the "Kill signal" or receive a strange message about incorrect formatting of the output file or no error message at all.

Similarly to Homework 4, add an instance field that stores a graph in `ProfessorPaths`. For testing purposes, we require that you implement the following public methods in `ProfessorPaths`. Otherwise, design class `ProfessorPaths` as you wish and add operations as you wish.

```
public void createNewGraph(String filename)
```

This method creates a brand new graph in the instance field of `ProfessorPaths` and populates the graph from *filename*, where *filename* is the name and path to a data file of the format defined in Problem 1. While all data files should be located in the `data/` directory of your project, do not hard-code `data/` or any other part of the path in your `createNewGraph()`. This method should be designed to work with any valid path (whether relative or absolute) and filename.

```
public String findPath(String node1, String node2)
```

Find the shortest path from *node1* to *node2* in the graph using your breadth-first search algorithm.

Paths should be chosen using the lexicographic ordering described above. If a path is found, the returned String should contain the path in the format below. That is,

`System.out.print(pp.findPath("Prof`$_1$`", "Prof`$_N$`"))` where `pp` refers to an instance of `ProfessorPaths`, prints the following:

`path from Prof`$_1$` to Prof`$_N$`:`
`Prof`$_1$` to Prof`$_2$` via Course`$_1$
`Prof`$_2$` to Prof`$_3$` via Course`$_2$
...
`Prof`$_{N-1}$` to Prof`$_N$` via Course`$_{N-1}$

where $Prof_1$ is the first node listed in the arguments to `findPath()`, $Prof_N$ is the second node listed in the arguments of `findPath()`, and $Course_K$ is the title of a course which both Prof_K and $\text{Prof}_K + 1$ have taught.

For example, `pp.findPath("Aram Chung", "Fudong Han")` will construct and return the String:

`"path from Aram Chung to Fudong Han:\nAram Chung to Amir Hirsa via ENGR-2250\nAmir Hirsa to Fudong Han via MANE-9990\n"`

(We are just making this up to illustrate the placement of newline characters, MANE-9990 is not listed in the RPI Course Network.)

Not all professors may have a path between them. If the user gives two valid node arguments that have no path in the specified graph, output the following as String:

`path from Prof`$_1$` to Prof`$_N$`:`
`no path found`

If a professor with name *Prof* was not in the original dataset, simply output:

`unknown professor Prof`

If neither of the two professors is in the original dataset, output the line twice: first time for the first node, then for the second one. These should be the only lines your program produces in this case, i.e., do not output the regular `path from ...` or `path not found` output.

What if the user asks for the path from a professor in the dataset to themselves? A trivially empty path is different from no path at all, so the `no path found` output is not appropriate here. But there are no edges to print either. So, you should output the header line:

`path from Prof`$_1$` to Prof`$_N$`:`

but nothing else. Hint: a well-written solution requires no special handling of this case.

This only applies to professors in the dataset. A request for a path from a professor that is not in the dataset to itself should have the usual `unknown professor` *`Prof`* output.

In all cases the string should end with \n (newline), just like in the first case.

The following examples illustrate the required format of the output. Among multiple paths of length 4 between "Mohammed J. Zaki" and "Wilfredo Colon", `findPath()` needs to select the lexicographically least path which would be the first of the three paths listed below:

```
path from Mohammed J. Zaki to Wilfredo Colon:
Mohammed J. Zaki to David Eric Goldschmidt via CSCI-2300
David Eric Goldschmidt to Michael Joseph Conroy via CSCI-4430
Michael Joseph Conroy to Alan R Cutler via CHEM-1200
Alan R Cutler to Wilfredo Colon via CHEM-1100

path from Mohammed J. Zaki to Wilfredo Colon:
Mohammed J. Zaki to David L Spooner via CSCI-4380
David L Spooner to Peter A Fox via ITWS-4370
Peter A Fox to Wayne G Roberge via ISCI-4510
Wayne G Roberge to Wilfredo Colon via BCBP-2900

path from Mohammed J. Zaki to Wilfredo Colon:
Mohammed J. Zaki to Shianne M. Hulbert via CSCI-2300
Shianne M. Hulbert to Michael Joseph Conroy via CSCI-4430
Michael Joseph Conroy to James Crivello via CHEM-1200
James Crivello to Wilfredo Colon via CHEM-1100
```

To help you with formatting your output correctly, we provide several sample files described below:

Description	**An example of the call to `findPath()`**	**Sample file**
A path is found. The lexicographically (alphabetically) least path is returned.	`System.out.print (pp.findPath("Mohammed J. Zaki", "Wilfredo Colon"));`	sample_hw5_output_00.txt
No path exists.	`System.out.print (mp.findPath("David Eric Goldschmidt", "Hugh Johnson"));`	sample_hw5_output_01.txt
Professor not found.	`System.out.print (mp.findPath("Donald Knuth", "Malik Magdon-Ismail"));`	sample_hw5_output_02.txt
Both professors not found.	`System.out.print (mp.findPath("Donald Knuth", "Brian Kernighan"));`	sample_hw5_output_03.txt
A path to the professor themselves.	`System.out.print (mp.findPath("Barbara Cutler", "Barbara Cutler"));`	sample_hw5_output_04.txt
A path to themselves for an unknown professor.	`System.out.print (mp.findPath("Donald Knuth", "Donald Knuth"));`	sample_hw5_output_05.txt

Problem 4: Testing Your Solution

Since realistic professors graphs may contain literally thousands of nodes and hundreds of thousands of edges or more, using them for correctness testing is probably a bad idea. By contrast, using them for scalability testing is a great idea, but should come after correctness testing has been completed using much smaller graphs. In addition, it is important to be able to test your parsing/graph-building and BFS operations in isolation, separately from each other.

You should first write `*.csv` files in the format defined for `courses.csv` to test your `ProfessorPaths`. All these files will go in the `data/` directory.

After you are done testing correctness, evaluate the performance of your code by running it on a large realistic graph with at least several thousand (and maybe even tens of thousands) nodes and hundreds of thousands edges. You will need to create a CSV file for such a graph and place this file in the `data/` directory. Note that we do not require that data is real, but it should be realistic. You do not need to submit the code that you used to generate this CSV file, only the CSV file itself. Feel free to explore what tools and libraries (not even necessarily written in Java) for generating fake but realistic testing/mock data are available out there.

Write tests in the regular JUnit test class in folder `src/test/java/hw5` (use `hw5` package). You might have to create the folder `src/test/java/hw5` and specify that you want to put your test class in the `hw5` package. Make sure that you handle the edge cases. You will have to specify data files to load in your implementation tests, **so make sure you read the File Paths section for information about specifying filenames very carefully.**

As in Homework 4, run EclEmma and measure coverage of your tests. We will be measuring coverage too. You must achieve 80% or higher instruction (statement) coverage for all classes except for `*Test` classes to receive full credit on this part. We measure the coverage of your Java code as executed by your test suite.

Reflection [0.5 points]

Please answer the following questions in a file named `hw5_reflection.pdf` in your `answers/` directory. Answer briefly, but in enough detail to help you improve your own practice via introspection and to enable the course staff to improve Principles of Software in the future.

(1) In retrospect, what could you have done better to reduce the time you spent solving this assignment?

(2) What could the Principles of Software staff have done better to improve your learning experience in this assignment?

(3) What do you know now that you wish you had known before beginning the assignment?

We will be awarding up to 1 extra credit point (at the discretion of the grader) for particularly insightful, constructive, and helpful reflection statements.

Collaboration[0.5 points]

Please answer the following questions in a file named `hw5_collaboration.pdf` in your `answers/` directory.

The standard integrity policy applies to this assignment.

State whether you collaborated with other students. If you did collaborate with other students, state their names and a brief description of how you collaborated.

Grade Breakdown

- Quality of test suite, percent of your tests passed: 6 pts. (auto-graded)
- Quality of test suite, percent coverage: 6 pts. (auto-graded)
- Instructor ProfessorPaths small tests: 7 pts. (auto-graded)
- Instructor ProfessorPaths large tests: 8 pts. (auto-graded)
- Changes (`answers/hw5_changes.pdf`): 5 pts.
- Code quality (`hw5/*.java`, specs, rep invariants, AFs, etc.): 12 pts.
- Quality of small CSV datasets (number of datasets, their variety, covering edge cases, following testing heuristics reviewed in class, etc.): 2 pts.
- Quality of the large CSV dataset (size, presence of paths of different lengths, how realistic, etc.): 3 pts.
- Collaboration and reflection (`answers/hw5_reflection.pdf` and `answers/hw5_collaboration.pdf`): 1 pt., up to 1 extra credit point (at the discretion of the grader) for particularly insightful, constructive, and helpful reflection statements.

Paths to Files

When you use test files in `data/`, hard-code the relative path in your tests (but not in `ProfessorParser.readData()` or `ProfessorPaths.createNewGraph()`). For example, if you are using file `testfile1.csv` in directory `data/`, you can load this file using `BufferedReader reader = new BufferedReader(new FileReader("data/testfile1.csv"))`.

Behavior may vary from one version of Eclipse to another and from one IDE to another. As long as you hard-code your relative paths starting at `data/`, you will be fine on Submitty. Note that if you are using an IDE or an editor (notably, VSCode) which by default runs your code from a directory that is different from the project root (i.e., the directory one level above `data/`), you would either have to adjust the settings of your IDE/editor to run your code from the project root or you would have to edit the paths in your tests before submitting your code to Submitty.

Hints

Performance

If your program takes an excessively long time to construct the graph for the large Professors Network, first make sure that it parses the data and builds the graph correctly for a very small dataset. If it does, here are a few questions to consider:

- What data structures are you using in your graph? What is their "big-O" runtime? Are there others that are better suited for the purpose?
- Did you remember to correctly override `hashCode()` if you overrode `equals()`?
- What is the "big-O" runtime of your `checkRep()` function? Does performance improve if you comment it out? You are allowed to comment out your `checkRep()`, if this is required for performance reasons but do not simply remove `checkRep()` from your code.

Miscellaneous

As always, remember to:

- Use descriptive variables names (especially in the BFS algorithm) and inline comments as appropriate.
- Include an abstraction function, representation invariant, and `checkRep()` in all classes that represent an ADT. If a class does not represent an ADT, place a comment that explicitly says so where the AF and RI would normally go. For example, classes that contain only class methods and are never constructed usually do not represent an ADT. Please come to office hours if you feel unsure about what counts as an ADT and what doesn't.

What to Submit

You should add and commit the following files to your hw04 Git repository:

- `src/main/java/hw5/ProfessorPaths.java`
- `src/main/java/hw5/*.java` *[Other classes you create, if any (there may be none!)]*
- `answers/hw5_reflection.pdf`
- `answers/hw5_collaboration.pdf`
- `data/*.csv` (excluding `courses.csv`) [Your .csv test files. Don't commit courses.csv. Right click this file in your `Git Staging` panel of Eclipse and select `Ignore`.]
- `src/test/java/hw5/*Test.java` *[JUnit test classes you create.]*

Additionally, be sure to commit any updates you may have made to the following files, so the staff has the correct version for this assignment:

- `src/main/java/hw4/*.java` *[Your graph ADT.]*
- `src/main/java/hw5/ProfessorParser.java`

Notes

Parts of this homework were copied from the University of Washington Software Design and Implementation class by Michael Ernst.

RPI Course Data has been collected and generously provided for use in this homework assignment by William Powe and the Quatalog project.

Errata

None yet. Check the Submitty Discussion forum regularly.