CSE 331 Software Design and Implementation

Homework 5: ***Modeling the Marvel Comics Universe***
Due: Tuesday, May 14 @ **11pm**

Handout H5

Contents:

Introduction

In this assignment you will put the graph you designed in Homework 4 to use by modeling the Marvel Comics universe. 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.

This application builds a graph containing thousands of 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 compelxities for various operations and this, not a coding bug, may explain the slowness.

This Marvel dataset was also used by researchers at Cornell University, who published a research paper showing that the graph is strikingly similar to "real-life" social networks.

The MarvelPaths Application

In this application, your graph models a social network among characters in Marvel comic books. Each node in the graph represents a character, and an edge `?Char1,Char2?` indicates that Char1 appeared in a comic book that Char2 also appeared in. There should be a separate edge for every comic book, labelled with the name of the book. For example, if Zeus and Hercules appeared in (say) five issues of a given series, then Zeus would have five edges to Hercules, and Hercules would have five edges to Zeus.

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

You will write a class `hw5.MarvelPaths` (in file `MarvelPaths.java`) that reads the Marvel data from a file (`marvel.tsv`), builds a graph, and finds paths between characters in the graph.

As you complete this assignment, you may need to modify the implementation and perhaps the public interface of your ADT. Briefly document any changes you made and why in `hw5/answers/changes.txt` (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 nor duplicate code! You can just `import` and use it in HW5. If you do modify your `hw5` code, be sure to commit your changes to your repository.

Problem 0: Getting the Marvel Universe Data

Before you get started, obtain the Marvel Universe dataset. For legal reasons, we have not added the file to your repositories. Instead, download the data from Infochimps, unzip the download, and copy and rename the `labeled_edges.tsv` file inside to `hw5/data/marvel.tsv`. (Note that although the download page encourages you to create an account, it does allow you to bypass that step.)

Take a moment to inspect the file. A TSV ("tab-separated value") file consists of human-readable data delineated by tabs, and can be opened in your favorite text editor or Eclipse. (If you don't have a favorite text editor, Notepad++ is a simple, free, easy-to-use option for Windows.) Each line in `marvel.tsv` is of the form

```
"character"	"book"
```

where *character* is the name of a character, *book* is the title of a comic book that the character appeared in, and the two fields are separated by a tab.

Problem 1: Building the Graph [60 points with Problem 2]

The first step in your program is to construct your graph of the Marvel universe from a data file. We provide you with a class `MarvelParser` to help you. `MarvelParser` has one static method, `parseData()`, which reads data from `marvel.tsv`, or any file structured in the same format. `parseData()` creates in-memory data structures: a `Set` of all characters and a `Map` from each book to a `List` of the characters in that book. These are not the data structures you want, however; you want a Graph.

You may modify `MarvelParser` however you wish to fit your implementation. You might change the method signature (parameters and return value) of `parseData()`, or you might leave `parseData()` 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 the parser can be reused with any file.

At this point, it's a good idea to test the parsing and graph-building operation in isolation. Verify that your program builds the graph correctly before you go on. The assignment formally requires this in Problem 3.

Problem 2: Finding Paths [60 points with Problem 1]

The real meat (or tofu) of `MarvelPaths` is the ability to find paths between two characters in the graph. Given the name of two characters, `MarvelPaths` searches for and returns a path through the graph connecting them. How the path is subsequently used, or the format in which it is printed out, depends on the requirements of the particular application using `MarvelPaths`, such as your test driver.

Your program should return the shortest path found via breadth-first search (BFS). A BFS from node u to node v visits all of n's neighbors first, then all of n's neighbors' neighbors', then all of n's 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 exists from start to dest.
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.

Many character 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 character at each next step in the path, and if those characters appear in several comic books together, it should print the lexicographically lowest title of a comic book that they both appear in. 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 character name, with edges to the same character visited in increasing order of comic book title. 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 `MarvelPaths` 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 hard-code a particular search algorithm in the graph ADT.

Using the full Marvel dataset, your program must be able to construct the graph and find a path in less than 30 seconds on attu. `ScriptFileTests`, the provided class used to run your specification tests, is set to put a 30000 ms (30 second) timeout for each test. (As a point of reference, the staff solution took around 5 seconds to run on attu).

Problem 3: Testing Your Solution [13 points]

Because the Marvel graph contains literally thousands of nodes and hundreds of thousands of edges, using it for correctness testing is probably a bad idea. By contrast, using it 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. For these reasons, you will use a test driver to verify the correctness of both your parser and BFS algorithm on much smaller graphs, as you did in Homework 4.

Specification Tests

You will write specification tests using test scripts similar to those you wrote in Homework 4. The format is defined in the test file format section below. In addition to writing `*.test` and `*.expected` files as before, you will write `*.tsv` files in the format defined for `marvel.tsv` to test your parser. All these files will go in the `hw5/data` directory.

You should create a class named `HW5TestDriver` that runs tests in the same way that `HW4TestDriver` did. We have provided a starter file in `test/` with method stubs and a small amount of starter code, but you can replace it. You have two choices as to your approach:

a. As one option, you may copy your `hw4/test/HW4TestDriver.java` source file to `hw5/test/HW5TestDriver.java`, change it to package hw5.test, and revise it to match this assignment's file format specification. This will cause duplicated code between the two problem sets, but avoids some of the tricky issues with subclassing. This is probably the easier option.
b. Alternatively, you may choose to write a new `HW5TestDriver` class that extends the `HW4TestDriver` and reuses some of its code. This approach may or may not be easier, but it is more elegant. You will need to go back and edit your `HW4TestDriver` to make it more extensible - in particular, by making some of its methods and/or fields protected instead of private. Also, static methods may not have the inheritance properties you want. You may find it easiest to just copy the `main` method from `HW4TestDriver` and change the references from HW4 to HW5. However you decide to

change `HW4TestDriver`, it must continue to work correctly for Homework 4.

Implementation Tests

Your specification tests will most likely cover the bulk of your testing. You may need to test additional behaviors specific to your implementation, such as handling of edge cases. If so, write these tests in a regular JUnit test class or classes and add the class name(s) to `ImplementationTests.java`. If you specify data files to load in your implementation tests, see the File Paths section for information about specifying filenames. Be sure to run your tests on attu with `ant validate` to verify that the data files are still found successfully under the configuration in which we will run your tests.

Problem 4: Running Your Solution From Command-Line [3 points]

Add a main() method to `MarvelPaths` that allows a user to interact with your program from the command line. The user should be able to input names of two characters from STDIN, and then your program outputs to STDOUT the path between those two characters if any, otherwise a suitable message. There is no rigid specificationhere for input or output formats, but especially creative ones may receive a small amount of extra credit.

Reflection [1 point]

Please answer the following questions in a file named `reflection.txt` 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 CSE 331 in the future.

a. In retrospect, what could you have done better to reduce the time you spent solving this assignment?
b. What could the CSE 331 staff have done better to improve your learning experience in this assignment?
c. What do you know now that you wish you had known before beginning the assignment?

Collaboration [1 point]

Please answer the following questions in a file named `collaboration.txt` in your `answers/` directory.

The standard collaboration policy applies to this assignment.

State whether or not you collaborated with other students. If you did collaborate with other students, state their names and a brief description of how you collaborated.

Time Spent

Tell us how long you spent on this homework via this catalyst survey: https://catalyst.uw.edu/webq/survey/mernst/198075.

Returnin [25 points]

After you receive your corrected assignment back from your TA, you will need to resubmit a corrected version of the code. You will have until Thursday, May 23 at 11pm to returnin your code. The returnin script will be enabled a week earlier, at 11pm Thursday, May 16.

You will only receive credit if you pass all the tests, and you have a fixed number of trials without penalty. See the returnin331 documentation for details.

Test script file format

The format of the test file is similar to the format described in Homework 4. As before, the test driver manages a collection of named graphs and accepts commands for creating and operating on those graphs.

Each input file has one command per line, and each command consists of whitespace-separated words, the first of which is the command name and the remainder of which are arguments. Lines starting with # are considered comment lines and should be echoed to the output when running the test script. Lines that are blank should cause a blank line to be printed to the output.

The behavior of the testing driver on malformed input command files is not defined; you may assume the input files are well-formed.

In addition to all the same commands (and the same output) as in Homework 4, the test driver accepts the following new commands:

LoadGraph *graphName file.tsv*

Creates a new graph named *graphName* from *file.tsv*, where *file.tsv* is a data file of the format defined for `marvel.tsv` and is located in the `src/hw5/data` directory of your project. The command's output is

```
loaded graph graphName
```

You may assume *file.tsv* is well-formed; the behavior for malformed input files is not defined.

Filenames supplied to the LoadGraph command should be simple, meaning they do not include the directory in which they are located. For example, `marvel.tsv` is a simple filename; `src/hw5/data/marvel.tsv` is not. Then, when your program actually loads the file, it must use precisely the relative pathname `src/hw5/data/file.tsv` in order to work with the autograder. (If you are running your program from the command line rather than from Eclipse, see the File Paths section for help configuring your program to find the file at this path.)

FindPath *graphName node_1 node_n*

Find the shortest path from *node_1* to *node_n* in the graph using your breadth-first search algorithm. For this command *only*, underscores in node names should be converted to spaces before being passed into any methods external to the test driver. For example, "node_1" would become "node 1". This is to allow the test scripts to work with the full Marvel dataset, where many character names contain whitespace (but none contain underscores). Anywhere a node is printed in the output for this command, the underscores should be replaced by spaces, as shown below.

Paths should be chosen using the lexicographic ordering described in Problem 2. If a path is found, the command prints the path in the format:

```
path from CHAR 1 to CHAR N:
CHAR 1 to CHAR 2 via BOOK 1
CHAR 2 to CHAR 3 via BOOK 2
...
CHAR N-1 to CHAR N via BOOK N-1
```

where *CHAR 1* is the first node listed in the arguments to FindPath, *CHAR N* is the second node listed in the arguments, and *BOOK K* is the title of a book that *CHAR K-1* and *CHAR K* appeared in.

Not all characters may have a path between them. If the user gives two valid node arguments *CHAR_1* and *CHAR_2* that have no path in the specified graph, print:

```
path from CHAR 1 to CHAR N:
no path found
```

If a character name *CHAR* was not in the original dataset, simply print:

```
unknown character CHAR
```

where, as before, any underscores in the name are replaced by spaces. If neither character is in the original dataset, print the line twice: first for *CHAR 1*, then for *CHAR N.* These should be the only lines your program prints in this case - i.e., do not print the regular "path from ..." or "path not found" output too.

What if the user asks for the path from a character in the dataset to itself? A trivially empty path is different from no path at all, so the "no path found" output isn't appropriate here. But there are no edges to print, either. So your test driver should print the header line

```
path from C to C:
```

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

This only applies to characters in the dataset: a request for a path from a character that is not in the dataset to itself should print the the usual "unknown character *C*" output.

Sample testing files

Several sample test files demonstrating the new commands are provided in `hw5/test`.

Paths to Files

To match the behavior of Eclipse, the autograder and Ant are configured to run your program from your `cse331` project directory. This means that your program must load files using a relative pathname that begins with `src`, e.g. `src/hwN/data/foo.txt`. Your test driver in particular must look for files specifically within `src/hw5/data`, as we copy some additional data files to that directory when running the autograder.

If you usually run your program at the command line rather than through Eclipse, you have probably been running it from the `bin` directory. From `bin`, your program won't be able to find files using relative pathnames of the form above (i.e. starting with `src`), so you need to run it directly from `cse331`. To do so, `cd` into the `cse331` directory, then tell Java where to look for your class files by adding `bin` to the classpath:

```
java -cp bin hw5.YourClassName
```

Alternatively, you can also run your tests with Ant by running `ant test` from the `src/hw5` directory. Unlike `ant validate`, this command runs on any machine and does not require your changes to be committed first.

Lastly, be aware that machine specs affect not only how fast your program runs but also how much memory it is allowed to use (more precisely, the maximum heap size). If you're working on a souped-up machine at home, it's particularly important that you test on attu.

Hints

Performance

An easy way to time your program is to run `ant test` on attu, just like you run `ant validate`. It will run all your implementation and specification tests and write how long each test took to a file. You can also run `ant test.impl` or `ant test.spec` to run just the implementation or specification tests, respectively.

If your program takes an excessively long time to construct the graph for the full Marvel dataset, 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 to 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?

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 static 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 Turn In

You should add and commit the following files to SVN:

- `hw5/MarvelPaths.java`
- `hw5/*.java` *[Other classes you create, if any (there may be none!)]*
- `hw5/answers/changes.txt`
- `hw5/answers/reflection.txt`
- `hw5/answers/collaboration.txt`
- `hw5/test/*.test`
- `hw5/test/*.expected`
- `hw5/data/*.tsv`
- `hw5/test/*Test.java` *[Other JUnit test classes you create, if any]*

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:

- `hw4/*.java` *[Your graph ADT]*
- `hw4/HW4TestDriver.java`
- `hw5/MarvelParser.java`
- `hw5/test/HW5TestDriver.java`
- `hw5/test/ImplementationTests.java`

Finally, remember to run `ant validate` to verify that you have submitted all files and that your code compiles and runs correctly when compiled with `javac` on attu (which sometimes behaves differently from the Eclipse compiler).

Errata

Added clarification about reflexive edges to The MarvelPaths Application section: the Marvel Graph should not store reflexive edges.

Q & A

Q: I have created a `.tsv` file for testing. I get this error:

```
java.lang.Exception: Line should contain exactly one tab
```

even though I used my tab key.

A: Your text editor might not be saving tab characters in the file. Just because you type a particular key on the keyboard does not mean that you editor saves a particular character in a file. In fact, if you are editing in Eclipse using the CSE-331-recommended settings, this is a feature: the tab key is very useful to help you move around your file, fix indentation, etc., but the tab character should never appear in any source code file. To solve the problem, use a different editor to edit your `.tsv` file.

For problems or questions regarding this page, contact: cse331-staff@cs.washington.edu.