CSCI 2600 — Principles of Software

Homework 6: Generics and Least-Cost Paths

Due: Tuesday, April 18, 2023, 11:59:59 pm

Submission Instructions

- This assignment uses the same repository as Homework assignments 4 and 5, so when you are ready to start working on Homework 6, pull Homework 6 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 add any additional files to 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 `hw6` with the usual directory structure. Write your code under `src/main/java/hw6` and your tests under `src/test/java/hw6` (shows as `hw6` under `src/test/java` in Package Explorer). If your directory structure is incorrect autograding will fail resulting in a grade of 0!

Introduction

This assignment lays the groundwork for an application you'll build in a later homework assignment. This assignment has two main parts. In the first part, you will make your graph class(es) generic. In the second part, you will implement a different path-finding algorithm for graphs known as Dijkstra's algorithm.

Augmenting Your Graph and Marvel Paths

Problem 1: Making Your Graph Generic

In a subsequent homework, your mission will be to find the shortest route to visit a certain set of buildings. A graph is an excellent representation of a map, and luckily you have already specified and implemented a graph. Unfortunately, your graph only stores Strings, whereas the route-finding application needs to store non-String data types in nodes and edges. More generally, your graph would be much more widely useful if the client could choose the data types to be stored in nodes and edges.

Your task is to convert your graph ADT to a generic class. Rather than always storing the data in nodes and edge labels as Strings, it should have two type parameters representing the data types to be stored in nodes and edges. Directly modify your existing classes under hw4 — there is no need to copy or duplicate code.

When you are done, your previously-written HW4 and HW5 tests and MarvelPaths will no longer compile. Modify these classes to construct and use graph objects parameterized with Strings. All code must compile, and all tests must pass when you submit your homework. Depending on your changes, some of your tests may no longer be valid. Try to adapt your tests to your new implementation or discard them and write the new ones as they should help you build confidence in your implementation. On the other hand, don't overdo it: as with any testing, stop when you feel that the additional effort is not being repaid in terms of increased confidence in your implementation.

Build tools and generic code

We want you to configure Eclipse to show generics problems as errors. By default, Eclipse shows generics problems as warnings (indicated by yellow lines and markers). You can configure Eclipse to instead issue errors (indicated by red lines and markers) for these problems. Doing so will help you remember to write acceptable generics code.

In order to properly configure Eclipse, go to **Window → Preferences** and select **Java → Compiler → Errors/Warnings**. Under **Generic types**, change the value of **Unchecked generic type operation** to **Error**.

Note that there is another setting named **Usage of a raw type** that is set to Ignore by default. We recommend that this option be set to **Warning** because it is specific to the Eclipse compiler and checks for more stringent requirements than required by the Java language specification.

Hint: Sometimes you may find that classes which previously compiled are now reporting "[some class] cannot be resolved" errors in Eclipse. You can fix these errors by doing a clean build: go to **Project → Clean...**, select your `csci2600` project, and hit **OK**.

Problem 2: Weighted Graphs and Least-Cost Paths

In a *weighted graph*, the label on each edge is a *length, cost*, or *weight* representing the cost of traversing that edge. Depending on the application, the cost may be measured in time, money, physical distance, etc. The total cost of a path is the sum of the costs of all edges in that path, and the *minimum-cost path* between two nodes is the path with the lowest total cost between those nodes.

Dijkstra's algorithm

You will implement Dijkstra's algorithm, which finds a minimum-cost path between two given nodes in a graph with nonnegative edge weights. Below is a pseudocode algorithm that you may use in your implementation. You are free to modify it as long as you are essentially still implementing Dijkstra's algorithm. Your implementation of the algorithm may assume a graph with Double edge weights.

The algorithm uses a priority queue. The standard Java library includes an implementation of a PriorityQueue.

Dijkstra's algorithm assumes a graph with all nonnegative edge weights.

```
    start = starting node
    dest = destination node
    active = priority queue.  Each element is a path from start to a given node.
             A path's "priority" in the queue is the total cost of that path.
             Nodes for which no path is known yet are not in the queue.
    finished = set of nodes for which we know the minimum-cost path from start.

    // Initially we only know of the path from start to itself, which has a cost
    // of zero because it contains no edges.
    Add a path from start to itself to active

    while active is non-empty:
        // minPath is the lowest-cost path in active and is the minimum-cost
        // path for some node
        minPath = active.removeMin()
        minDest = destination node in minPath

        if minDest is dest:
            return minPath

        if minDest is in finished:
            continue

        for each edge e = ⟨minDest, child⟩:
            // If we don't know the minimum-cost path from start to child,
            // examine the path we've just found
            if child is not in finished:
                newPath = minPath + e
                add newPath to active

        add minDest to finished

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

Dijkstra's Algorithm in MarvelPaths

You will write a modified version of your Marvel Paths application in which your application finds its paths using Dijkstra's algorithm instead of BFS. Dijkstra's algorithm requires weighted edges. To simulate edge weights over the Marvel dataset, the weight of the edge between two characters will be based on how well-connected those two characters are. Specifically, the weight is the inverse of how many comic books two characters are in together (equivalently, the weight is the multiplicative inverse of the number of edges in the multigraph between them). For example, if Amazing Amoeba and Zany Zebra appeared in 5 comic books together, the weight of the edge between them would be 1/5. Thus, the more well-connected two characters are, the lower the weight and the more likely that a path is taken through them. In summary, the idea with the Marvel data is to treat the number of paths from one node to another as a "distance" — if there are several edges from one node to another then that is a "shorter" distance than another pair of nodes that are only connected by a single edge.

Things to note:

- A path from a character to itself is defined to have cost 0.0.
- Calculations for the weight of the edges in your graph should be done when the graph is loaded. This assignment is different from the last one in that when the graph is initialized there is only one edge between nodes and that edge is the weighted edge. The one edge between any two characters will have the label containing the multiplicative inverse of how many books they share.
- You should implement your Dijkstra's algorithm in `MarvelPaths2` rather than directly in your graph.

Place your new Marvel application in `src/main/java/hw6/MarvelPaths2.java` in package `hw6`. In choosing how to organize your code, remember to avoid duplicating code as much as possible. In particular, reuse existing code where possible, and keep in mind that you may need to use the same implementation of Dijkstra's algorithm in a different application.

For this assignment, your program must be able to construct the graph and find a path in less than **10** seconds using the full Marvel dataset. We will set the tests to have a 10,000 ms (10 second) timeout for each test and run with assertions disabled.

As before, you are welcome to write a `main()` method for your application, but you are not required to do so.

The interface of `MarvelPaths2` is the same as that of `MarvelPaths` in Homework 5, but with a few differences in arguments and output:

- **public void createNewGraph(String filename)** is the same. It creates a brand new graph and populates the graph from *filename*, where *filename* is a data file of the format for `marvel.csv` and is located in `data/`. As in Homework 5, relative paths to data files should begin with `data/`. Consult section "Paths to files" in Homework 5 if you are having trouble making Eclipse work with these relative paths.

- **public String findPath(String CHAR1, String CHAR2)** searches with Dijkstra's algorithm instead of BFS and returns its output in the form:
  `path from` $CHAR_1$ `to` $CHAR_N$`:`
  $CHAR_1$ `to` $CHAR_2$ `with weight` w_1
  $CHAR_2$ `to` $CHAR_3$ `with weight` w_2
  `...`
  $CHAR_{N-1}$ `to` $CHAR_N$ `with weight` w_{N-1}
  `total cost:` W
  where W is the sum of w_1, w_2, ..., w_{N-1}.

In other words, the only changes in output from Homework 5 are the way the edge labels are printed and the addition of a "total cost" line at the bottom. The output should remain the same as before when no path is found or the characters are not in the graph. In particular, do not print the "total cost" line in those cases.

If there are two minimum-cost paths between $CHAR_1$ and $CHAR_N$, it is undefined which one is printed.

For readability, the output of `findPath` should print numeric values with exactly 3 digits after the decimal point, rounding to the nearest value if they have more digits. The easiest way to specify the desired format of a value is using format strings. For example, you could create the String "Weight of 1.760" by writing:

```
String.format("Weight of %.3f", 1.759555555555);
```

In `findPath`, the total cost should be computed by summing the full values of the individual weights, not the rounded values.

As in Homework 5, a path from a character to itself should be treated as a trivially empty path. Because this path contains no edges, it has a cost of zero. (Think of the path as a list of edges. The sum of an empty list is conventionally defined to be zero.) So, your `findPath` should produce the usual output for a path but without any edges, i.e.:

```
path from C to C:
total cost:  0.000
```

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 usual "unknown character C" output.

Also, as in Homework 5, if the user gives two valid node arguments CHAR1 and CHAR2 that have no path in the specified graph, output:

```
path from CHAR1 to CHARN:
no path found
```

The following example illustrates the required format of the output:

```
path from PETERS, SHANA TOC to SEERESS:
PETERS, SHANA TOC to KNIGHT, MISTY with weight 1.000
KNIGHT, MISTY to CAGE, LUKE/CARL LUCA with weight 0.017
CAGE, LUKE/CARL LUCA to HULK/DR. ROBERT BRUC with weight 0.032
HULK/DR. ROBERT BRUC to RAVAGE/PROF. GEOFFRE with weight 0.500
RAVAGE/PROF. GEOFFRE to SEERESS with weight 1.000
total cost:  2.549
```

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 minimum-cost path is found.	`System.out.print (mp2.findPath("PETERS, SHANA TOC", "SEERESS"));`	sample_hw6_output_00.txt
No path exists.	`System.out.print (mp2.findPath("GOOM", "HOFFMAN"));`	sample_hw6_output_01.txt
Character not found.	`System.out.print (mp2.findPath("BATMAN", "CAPTAIN AMERICA"));`	sample_hw6_output_02.txt
Both characters not found.	`System.out.print (mp2.findPath("BATMAN", "GREEN LANTERN"));`	sample_hw6_output_03.txt
A path to the character themselves.	`System.out.print (mp2.findPath("SEERESS", "SEERESS"));`	sample_hw6_output_04.txt
A path to the unknown character themselves.	System.out.print (mp2.findPath(”BATMAN”, ”BATMAN”));	sample_hw6_output_05.txt

Problem 3: Testing Your Solution

Just as with Homework 5, create smaller `*.csv` files to test your graph building and path finding. Place them in the `data/` directory. Write tests in JUnit classes and place those tests in `hw6` package (`src/test/java/hw6/` directory). Just as in Homework 4 and 5, run EclEmma to measure your code coverage. The code coverage threshold will be set at 85% for this assignment.

Tests do not directly test the property that your graph is generic. However, Homework 4 and Homework 5 tests use String edge labels, while Homework 6 uses numeric values. Supporting all three test drivers implicitly tests the generic behavior of your graph.

Reflection [0.5 points]

Please answer the following questions in a file named `hw6_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?

Collaboration[0.5 points]

Please answer the following questions in a file named `hw6_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

- Instructor hw4 tests: 5 pts (auto-graded)
- Instructor hw5 tests: 5 pts (auto-graded)
- Quality of hw6 test suite, percent of your tests passed: 5 pts (auto-graded)
- Quality of hw6 test suite, percent coverage: 5 pts (auto-graded)
- Instructor MarvelPaths2 tests: 16 pts (auto-graded)
- Code quality (Code organization, Genericity, Rep invariants, Abstraction functions, Specifications): 13 pts
- Collaboration and reflection: 1 pt

Hints

Documentation

When you add generic type parameters to a class, make sure to describe these type parameters in the class' Javadoc comments so that the client understands what they are for.

As usual, 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 Abstraction function and Rep invariant 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. You may comment out checkRep() when running on Submitty, but leave the commented code in your files.

Code Organization

In deciding how to organize your code, remember that you may need to reuse Dijkstra's algorithm in future homework assignments. These assignments have nothing to do with Marvel and, depending on your implementation, might use a different generic type for nodes. Structure your code so that your implementation of Dijkstra's algorithm is convenient to use for other applications.

What to Turn In

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

- `src/main/java/hw6/MarvelPaths2.java`
- `src/main/java/hw6/*.java` *[Other classes you create, if any]*
- `data/*.csv` *[Your .csv test files. Do not commit marvel.csv]*
- `src/test/java/hw6/*Test.java` *[JUnit test classes you create]*
- `answers/hw6_reflection.pdf`
- `answers/hw6_collaboration.pdf`

Additionally, be sure to commit any updates you make to the following files:

- `src/main/java/hw4/*.java` *[Your graph ADT]*
- `src/test/java/hw4/*Test.java` *[Your graph ADT test classes]*
- `src/main/java/hw5/*.java` *[Your Marvel Paths]*
- `src/test/java/hw5/*Test.java` *[Your Marvel Paths test classes]*

Errata

None yet. Check the Submitty Discussion forum regularly. As usual, we will announce errors there.

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