CSCI 2600 — Principles of Software

Homework 6: Generics and Least-Cost Paths

Due: Thursday, Apr. 10, 2025, 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 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 `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 LEGO Paths

Problem 1: Making Your Graph Generic

In a subsequent homework assignment, 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` package — there is no need to copy or duplicate code. Also remember that you need to keep your Graph ADT implementation as general and universal as possible. In particular, it means that you are not allowed to have any bounds on type parameters because it would restrict the client from using your Graph ADT implementation in a certain way. For example, if some client wanted to create a graph with node data as `Object` and edge labels as `Object`, it should be possible.

When you are done, your previously-written Homework 4 and Homework 5 tests, as well as ProfessorPaths 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 the 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. Autograding on Submitty will also be configured to treat any generics warnings as compilation errors.

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.
```

Getting the LEGO Data

You will write a modified version of your ProfessorPaths application in which your application finds paths between LEGO parts instead of professors and uses Dijkstra's algorithm instead of BFS.

Before you get started, obtain LEGO datasets. We have not added these files to your Git repositories because some of them are very large. Instead, download the files from the Course Web site:

- LEGO dataset up to year 1960 (inclusive)
- LEGO dataset up to year 1970 (inclusive)
- LEGO dataset up to year 1980 (inclusive)
- LEGO dataset up to year 1990 (inclusive)
- LEGO dataset up to year 2000 (inclusive)
- LEGO dataset up to year 2010 (inclusive)
- LEGO dataset up to year 2020 (inclusive)
- LEGO dataset up to 04/07/2024 (inclusive)

Store the files in `data/` directory. You might have to add the `data/` directory under the project root. As you can see, the size of the file grows rapidly with the year number because more and more LEGO parts and sets are introduced every month. We provide you with a variety of datasets to help you test the performance of your application. With the largest `lego2024.csv` dataset your code may take up to several minutes to construct the graph and up to a minute to find paths. You might also need to increase the maximum amount of heap memory allocated by JVM to your application. Please read Oracle Documentation for the java command for more details. If your code is taking substantially longer to run or if it fails with the out of memory error, consider revising your code to make it faster and/or reduce its memory footprint.

IMPORTANT: Do not commit LEGO CSV files that we provided 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 `lego*.csv`. For those not using Eclipse, make sure to add `lego*.csv` to your `.gitignore` file. Note that CSV files that you create yourselves must be added to the repository. When naming your files, make sure they do not fall under the exclusion rule that you set up as described above.

Take a moment to inspect the files. The format is very similar to the format of the RPI Courses dataset from Homework 5. Each line in `lego*.csv` is of the form:

```
"part","set"
```

where `part` is the part number, color, and part name of a LEGO part all combined into a single String. `set` is the set number, version, release year, and set name of a LEGO set all combined into a single String. If a part and a set appear in the same line of a CSV file, it means that this part is included in the set. As before, the CSV format 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 part or set String. However, note that part and set Strings may contain double quotes as a part of the String itself.

Dijkstra's Algorithm in LegoPaths

Dijkstra's algorithm requires weighted edges. To simulate edge weights over the LEGO dataset, the weight of the edge between two parts will be based on how well-connected those two parts are. Specifically, the weight is the inverse of how many LEGO sets two parts are in together (equivalently, the weight is the multiplicative inverse of the number of edges in the multigraph between them). For example, if "6541 Black Technic Brick" and "65486 White Plate" appeared in five sets together, the weight of the edge between them would be $\frac{1}{5}$. Thus, the more well-connected two parts are, the lower the weight and the more likely that a path is taken through them. In summary, the idea with the LEGO 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 part 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 previous 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 parts will have the label containing the multiplicative inverse of how many sets they share.
- You should implement your Dijkstra's algorithm in `LegoPaths` rather than directly in your graph.

Place your new LEGO application in `src/main/java/hw6/LegoPaths.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 **300** seconds using the largest 2024 LEGO dataset. We will set the tests to have a 300,000 ms (300-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 `LegoPaths` is similar to that of `ProfessorPaths` 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 `lego*.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 PART1, String PART2)** finds the path with Dijkstra's algorithm instead of BFS and returns its output in the form:
  `path from` $PART_1$ `to` $PART_N$`:`
  $PART_1$ `to` $PART_2$ `with weight` w_1
  $PART_2$ `to` $PART_3$ `with weight` w_2
  `...`
  $PART_{N-1}$ `to` $PART_N$ `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 parts 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 $PART_1$ and $PART_N$, 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 values of the individual weights with full precision, not the rounded values.

As in Homework 5, a path from a part 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 PART to PART:
total cost:  0.000
```

This only applies to parts in the dataset. A request for a path from a part that is not in the dataset to itself should print the usual "unknown part PART" output.

Also, as in Homework 5, if the user gives two valid node arguments PART1 and PART2 that have no path in the specified graph, output:

`path from` $\texttt{PART}_1$ `to` $\texttt{PART}_N$`:`
`no path found`

The following example illustrates the required format of the output:

```
path from 31367 Green Duplo Egg Base to 98138pr0080 Pearl Gold Tile Round 1 x 1 with Blue, Yellow and Black Minecraft Print:

31367 Green Duplo Egg Base to 3437 Green Duplo Brick 2 x 2 with weight 1.000

3437 Green Duplo Brick 2 x 2 to 3437 Red Duplo Brick 2 x 2 with weight 0.003

3437 Red Duplo Brick 2 x 2 to 41250 Blue Ball, Hard Plastic, 51mm (approx.  6 studs diameter) with weight 0.053

41250 Blue Ball, Hard Plastic, 51mm (approx.  6 studs diameter) to 2780 Black Technic Pin with Friction Ridges Lengthwise and Center Slots with weight 0.100

2780 Black Technic Pin with Friction Ridges Lengthwise and Center Slots to 98138pr0080 Pearl Gold Tile Round 1 x 1 with Blue, Yellow and Black Minecraft Print with weight 1.000

total cost:  2.156
```

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(lp.findPath("31367 Green Duplo Egg Base", "98138pr0080 Pearl Gold Tile Round 1 x 1 with Blue, Yellow and Black Minecraft Print"));`	sample_hw6_output_00.txt
No path exists.	`System.out.print(lp.findPath("880006 Black Stopwatch", "3007d White Brick 2 x 8 without Bottom Tubes, 1 End Slot"));`	sample_hw6_output_01.txt
Character not found.	`System.out.print(lp.findPath("35480 Green Plate Special 1 x 2 Rounded with 2 Open Studs", "27ac01 Light Yellow Window 1 x 2 x 1 (old type) with Extended Lip and Solid Studs, with Fixed Glass"));`	sample_hw6_output_02.txt
Both characters not found.	`System.out.print(lp.findPath("76371pr0201 White Duplo Brick 1 x 2 x 2 with Bottom Tube, Target and Water Splash Print", "75266 White Duplo Car Body, Camper / Caravan Roof"));`	sample_hw6_output_03.txt
A path to the character themselves.	`System.out.print(lp.findPath("3035 Dark Gray Plate 4 x 8 to 3035 Dark Gray Plate 4 x 8", "3035 Dark Gray Plate 4 x 8 to 3035 Dark Gray Plate 4 x 8"));`	sample_hw6_output_04.txt
A path to the unknown character themselves.	`System.out.print(lp.findPath("2412a White Tile Special 1 x 2 Grille with Bottom Groove", "2412a White Tile Special 1 x 2 Grille with Bottom Groove"));`	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. We will be measuring coverage too. You must achieve 85% 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.

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?

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 `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 LegoPaths tests: 16 pts. (auto-graded)

- Code quality (Code organization, Genericity, Rep invariants, Abstraction functions, Specifications, comments): 11 pts.
- Quality of small CSV datasets (number of datasets, their variety, covering edge cases, following testing heuristics reviewed in class, etc.): 2 pts.
- Collaboration and reflection: 1 pt., up to 1 extra credit point (at the discretion of the grader) for particularly insightful, constructive, and helpful reflection statements.

Extra Credit Code Performance Competition

We will run all submissions through a benchmark and award extra credit points to top 50 students whose code took the least amount of time to run the benchmark. Points will be awarded according to the following table:

Position	Extra credit points
1	20
2	17
3	14
4	12
5	10
6	8
7	7
8	6
9	5
10	4
11-20	3
21-50	1

The benchmark will consist of:

- Loading the `lego1970.csv` dataset
- Loading the `lego2024.csv` dataset
- Finding a path of length one
- Finding a path of length two
- Finding a path of length four
- Finding a path of length more than four
- Finding a path when no path exists

All submissions will be tested in the exact same environment on the same machine. Time will be measured in milliseconds using `Java.lang.System.currentTimeMillis()` method. Code that returns incorrect results or causes an out of memory error will be disqualified from the competition.

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 LEGO 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/LegoPaths.java`
- `src/main/java/hw6/*.java` *[Other classes you create, if any]*
- `data/*.csv` *[Your .csv test files.* ***Do not*** *commit lego*.csv that were provided to you]*
- `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 ProfessorPaths]*
- `src/test/java/hw5/*Test.java` *[Your ProfessorPaths 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.