CSCI 2134 Assignment 3

Due date: 11:59pm, Friday, March 19, 2021, submitted via Git

Objectives

Practice debugging code, using a symbolic debugger, and fixing errors.

Preparation:

Clone the Assignment 3 repository

https://git.cs.dal.ca/courses/2021-winter/csci-2134/assignment3/????.git

where ???? is your CSID.

Problem Statement

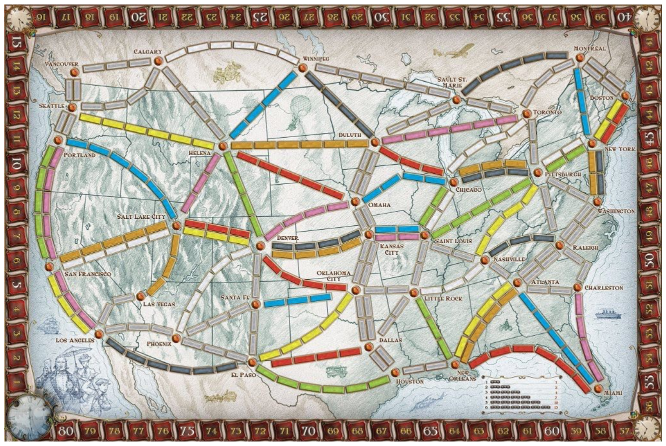
Take a piece of buggy code, debug it, and fix it.

Background

You have inherited some buggy code for computing shortest path solutions to the board game Ticket to Ride. Your boss has fired the previous developer because they did not do any testing and did not fix the bugs! She has hired you to debug and fix the code. She will provide you with some unit tests (some of which fail), sample input and sample output of what should be produced. Your job is to fix the bugs: Both the bugs exhibited by the unit tests and the ones by the input. Good luck!

You will be provided with a full buggy codebase for JSON comparison, a specification, a set of unit tests using JUnit5, sample input and expected output. Your job is to identify and fix all the bugs.

Given a game board of rail segments and a list of routes (pairs of cities), the code is supposed to compute the total cost of building a network between the given routes, assuming that the shortest distance for each route is chosen. This can be computed by computing shortest paths for each route using Dijkstra's shortest path algorithm.



You will be provided with a full buggy codebase for distance computation, a specification, a set of unit tests using JUnit5, sample input and expected output. Your job is to identify and fix all the bugs.

Task

1. Review the specification (**`specification.pdf`**) in the **`docs`** directory. You will absolutely need to understand it and the code you are debugging. The main method for the program is in **RouteCost`.java`**. Note that your boss finally got the buggy **makeTree** method in **City.java** from the previous developer. Spend some time tracing through the code and creating a diagram of how the classes and code are put together. This will help you a lot later on!
2. **Fix all bugs** that are identified by the tests generated by the unit tests in the following classes:
    - `City.java`
    - `CityComparator.java`
    - `Link.java`
3. See **`buglist.txt`** file in the **`docs`** directory. One sample entry is included. For each bug that you fix add an entry to this file that includes:
    a. The file/class name where the bug was.
    b. The method where the bug was
    c. The line number(s) where the buggy code was
    d. A description of what the bug was
    e. A description of what the fix was.
4. The previous developer made a set of example input and expected output in the **`input_tests`** directory. These tests will likely not pass yet even after fixing the bugs identified by the unit tests.
    - See the README.txt in this directory for help running the tests. The easiest method is to copy your .java files from src to this directory and run the test.sh script in a terminal or git bash command line shell.
    - Compare the output in the .out files with the expected .gold files.
    - For each output that differs from the expected output, debug the code and determine the reason for the mismatch. Fix any identified bugs missed by the unit tests.
5. Record any new bugs found and fixed from Step 4 in the previously created **`buglist.txt`**
6. **Commit and push back** the bug fixes and the **`buglist.txt`** file to the remote repository.

Submission

All fixes and files must be committed and pushed back to the remote Git repository.

Grading

The following grading scheme will be used:

Task	4/4	3/4	2/4	1/4	0/4
Bugs found [unit tests] (20%)	4 to 5 bugs are correctly identified and documented.	Three (3) bugs are correctly identified and documented.	Two (2) bugs are correctly identified and documented.	One (1) bug is correctly identified and documented.	Zero (0) bugs are correctly identified and documented.
Bugs fixed [unit tests] (20%)	4 to 5 bugs are correctly fixed. All unit tests pass.	Three (3) bugs are correctly fixed.	Two (2) bugs are correctly fixed.	One (1) bug is correctly fixed.	Zero (0) bugs are correctly fixed.
Bugs found [input tests] (20%)	2 to 3 bugs are correctly identified and documented.	N/A	One (1) bug is correctly identified and documented.	N/A	Zero (0) bugs are correctly identified and documented.
Bugs fixed [input tests] (30%)	2 to 3 bugs are correctly fixed. All input tests pass.	2 to 3 bugs are correctly fixed.	One (1) bug is correctly fixed. Some input tests pass	N/A	Zero (0) bugs are correctly fixed.
Document `[buglist.txt]` Clarity (10%)	Document looks professional, includes all information, and easy to read	Document looks ok. May be hard to read or missing some information.	Document is sloppy, inconsistent, and has missing information	Document is very sloppy with significant missing information	Document is illegible or not provided.

Hints

1. You will need to use a symbolic debugger to make headway. Using print-statements will be possible but extremely painful.
2. You will need to step through the code to find the bugs.
3. There are about 2-3 bugs in the code (in addition to the ones identified by the unit tests). The single bug report should cover all of them.