Programming Assignment #6: TopoPath

COP 3503, Summer 2018

Due: Sunday, July 22, *before* 11:59 PM

Abstract

In this assignment, you will determine whether an arbitrary directed graph contains a *topopath* – an ordering of its vertices that simultaneously corresponds to a valid path in the graph *and* a valid topological sort.[1,2] More than anything, this program should serve as a relatively short critical thinking exercise.

You will gain experience reading graphs from an input file, representing them computationally, and writing graph theory algorithms. You will also solidify your understanding of topological sorts, sharpen your problem solving skills, and get some practice at being clever, because your solution to this problem must be O(n^2). In coming up with a solution, I recommend focusing first on developing a working algorithm, and *then* analyzing its runtime. (Focusing *too* much on the runtime restriction might cloud your thinking as you cook up a solution to this problem.)

If you use any code that I have given you so far in class, you should probably include a comment to give me credit. The intellectually curious student will, of course, try to write the whole program from scratch.

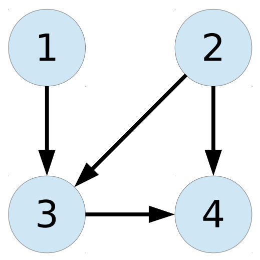
Deliverables

TopoPath.java

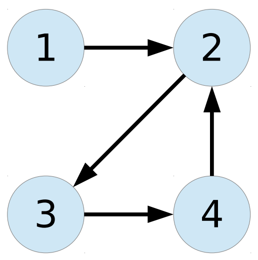
1 *Topopath* is a word I made up. Please don't go out into the real world expecting other people to know what it means.
2 An *ordering* is just a permutation. Recall that in a permutation, the elements are simply shuffled; they are not repeated. So, a topopath would have to include each vertex from the graph *exactly* once – no more, no less.

1. Problem Statement

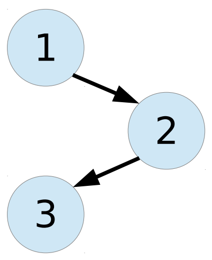
Given a directed graph, determine whether it has a *topopath* – an ordering of its vertices that simultaneously corresponds to a valid path in the graph *and* a valid topological sort. For example:



G_1
(No topopath.)



G_2
(No topopath.)



G_3
(Contains a topopath.)

In G_1, the vertex ordering **1, 2, 3, 4** is a valid topological sort, but does not correspond to a valid path in the graph (i.e., **1** → **2** → **3** → **4** is not a path in G_1). In fact, none of the topological sorts for G_1 correspond to actual paths through that graph.

In contrast, the vertex ordering **1, 2, 3, 4** corresponds to a valid path in G_2 (i.e., **1** → **2** → **3** → **4** is an actual path in G_2), but is not a valid topological sort for the graph. None of the paths in G_2 correspond to a valid topological sort of that graph's vertices.

In G_3, the ordering **1, 2, 3** corresponds to both a valid path (**1** → **2** → **3**) *and* a valid topological sort.

2. Input File Format

Each input file contains a single digraph. The first line contains a single integer, $n \geq 2$, indicating the number of vertices in the graph. (Vertices in these graphs are numbered 1 through *n*.) The following *n* lines are the adjacency lists for each successive vertex in the graph, with a small twist: each adjacency list begins with a single non-negative integer, *k*, indicating the number of vertices that follow. The list of vertices that follows will contain *k* distinct integers (i.e., no repeats) on the range 1 through *n*. For example, the following text files correspond to the graphs G_1, G_2, and G_3 that are pictured above:

g1.txt
```
4
1 3
2 3 4
1 4
0
```

g2.txt
```
4
1 2
1 3
1 4
1 2
```

g3.txt
```
3
1 2
1 3
0
```

3. Special Restriction: Runtime Requirement

Please note that you must implement a solution that is $O(n^2)$, where $n = |V|$. Recall from our formal definition of big-oh that a faster solution is still considered $O(n^2)$.

4. Method and Class Requirements

Implement the following methods in a class named `TopoPath`. Notice that all these methods are both **`public`** and **`static`**.

> **`public static boolean`** `hasTopoPath(String filename)`
>
> Open *filename* and process the graph it contains. If the graph has a topopath (explained above), return *true*. Otherwise, return *false*. The string *filename* will refer to an existing file, and it will follow the format indicated above. You may have your method throw exceptions as necessary. Note that this function must execute in O(n^2) time.
>
> **`public static double`** `difficultyRating()`
>
> Return a double on the range 1.0 (ridiculously easy) through 5.0 (insanely difficult).
>
> **`public static double`** `hoursSpent()`
>
> Return an estimate (greater than zero) of the number of hours you spent on this assignment.

5. Compiling and Testing Your Code on Eustis

Recall that your code must compile, run, and produce precisely the correct output on Eustis in order to receive full credit. Here's how to make that happen:

1. To compile your program with one of the given test cases:

```
javac TopoPath.java TestCase01.java
```

2. To run this test case and redirect your output to a text file:

```
java TestCase01 > myoutput01.txt
```

3. To compare your program's output against the sample output file I've provided for this test case:

```
diff myoutput01.txt sample_output/TestCase01-output.txt
```

If the contents of `myoutput01.txt` and `TestCase01-output.txt` are exactly the same, `diff` won't print anything to the screen. It will just look like this:

```
seansz@eustis:~$ diff myoutput01.txt sample_output/TestCase01-output.txt
seansz@eustis:~$ _
```

Otherwise, if the files differ, `diff` will spit out some information about the lines that aren't the same.

6. Using the test-all.sh Script

As always, the `test-all.sh` script is the safest way to ensure your code is compatible with my test cases. You can run it on Eustis by placing it in a folder with `TopoPath.java`, all the test case files, and the `sample_output` subdirectory, and typing:

```
bash test-all.sh
```

7. Grading Criteria and Miscellaneous Requirements

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

90%	program passes test cases using an O(n^2) algorithm
10%	`difficultyRating()` and `hoursSpent()` are implemented correctly

Note! Point deductions might still be imposed for those who do not employ good coding style (comments and whitespace).

Please be sure to submit your `.java` file, not a `.class` file (and certainly not a `.doc` or `.pdf` file). Your best bet is to submit your program in advance of the deadline, then download the source code from Webcourses, re-compile, and re-test your code in order to ensure that you uploaded the correct version of your source code. Programs that do not compile will receive zero credit.

Important! You might want to remove `main()` and then double check that your program compiles without it before submitting. Including a `main()` method can cause compilation issues if it includes references to home-brewed classes that you are not submitting with the assignment. Please remove.

Important! Your program should not print anything to the screen. Extraneous output is disruptive to the grading process and will result in severe point deductions. Please do not print to the screen.

Important! No file writing. Please do not write to any files from `TopoPath.java`.

Important! Please do not create a java `package`. Articulating a `package` in your source code could prevent it from compiling with our test cases, resulting in severe point deductions.

Important! Name your source file, class(es), and method(s) correctly. Minor errors in spelling and/or capitalization could be hugely disruptive to the grading process and may result in severe point deductions. Similarly, failing to write any one of the required methods, or failing to make them `public` and/or `static` (as appropriate) may cause test case failure. Please double check your work!

Input specifications are a contract. We promise that we will work within the confines of the problem statement when creating the test cases that we'll use for grading. Please reflect carefully on the kinds of edge cases that might cause unusual behaviors for any of the methods you're implementing, and be sure to write your own test cases to thoroughly test your code.

Start early! Work hard! Ask questions! Good luck!