Programming Assignment #5: TopoPath

COP 3503, Fall 2016

Due: Sunday, November 6, *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]

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 creative problem solving skills, and get some practice at being clever, because your solution to this problem must be $O(n^2)$.

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

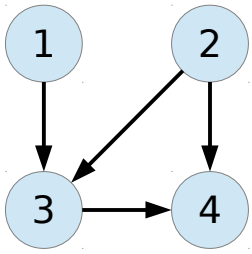
TopoPath.java

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.

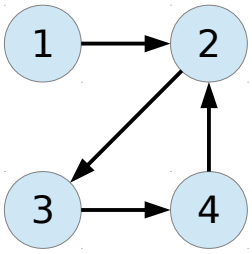
2 An *ordering* is just a permutation. Recall that in a permutation, the elements are simply shuffled; they are not repeated. So, a topopath would have to include each vertex from the graph *exactly* once – no more, no less.

1. Problem Statement

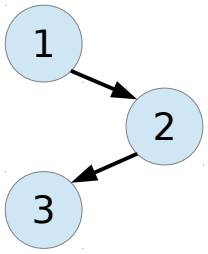
Given a directed graph, determine whether it has a *topopath* – an ordering of its vertices that simultaneously corresponds to a valid path in the graph *and* a valid topological sort. For example:



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. 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. Runtime Requirements

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.

`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. Grading Criteria and Miscellaneous Requirements

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

80%	program passes test cases
10%	`difficultyRating()` and `hoursSpent()` return doubles in the specified ranges
10%	adequate comments and whitespace

Important! Programs that do not compile will receive zero credit. When testing your code, you should ensure that you place TopoPath.java alone in a directory with the test case files (source files and sample_output directory), and no other files. That will help ensure that your TopoPath.java is not relying on external support classes that you've written in separate .java files but won't be including with your program submission.

Important! You might want to remove `main()` and then double check that your program compiles without it before submitting.

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! 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, capitalization, or other method signature details could be hugely disruptive to the grading process and may result in severe point deductions. Please double check your work!

Test your code thoroughly. Please be sure to create your own test cases and thoroughly test your code.

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