Programming Assignment #5: ConstrainedTopoSort

COP 3503, Summer 2019

Due: Sunday, July 21, *before* 11:59 PM

Abstract

In this assignment, you will determine whether an arbitrary directed graph has a valid topological sort in which some vertex, x, comes before some other vertex, y. 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

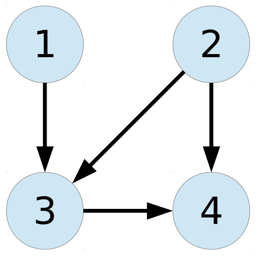
ConstrainedTopoSort.java

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

1. Problem Statement

Given a directed graph, G, and two integers, *x* and *y*, determine whether G has a valid topological sort in which vertex *x* comes before vertex *y*. (Notice that the problem does **not** ask whether *x* comes *directly* before *y*.) For example:



G_1

In G_1, there is a valid topological sort in which vertex 2 comes before vertex 1 (**2, 1, 3, 4**). There is also a valid topological sort in which vertex 1 comes before vertex 2 (**1, 2, 3, 4**). Both of those are also valid topological sorts in which vertex 2 comes before vertex 4. (Notice that vertex 2 does not have to come *directly* before vertex 4.) However, there is no valid topological sort in which vertex 4 comes before vertex 1.

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 file corresponds to the graph G_1 that is pictured above:

g1.txt

```
4
1 3
2 3 4
1 4
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 `ConstrainedTopoSort`.

> **`public`** `ConstrainedTopoSort(String filename)`
>
> This constructor opens the file named *filename* and reads the graph it contains into either an adjacency matrix or adjacency list. We will process multiple *xy* queries for this graph, but we only want to load it into memory once. This method should throw exceptions as necessary.

> **`public boolean`** `hasConstrainedTopoSort(int x, int y)`
>
> Given integers x and y such that $1 \le x \le n$ and $1 \le y \le n$, if this graph has a valid topological sort in which vertex x precedes vertex y, return *true*. Otherwise, return *false*. Do this in $O(n^2)$ time.

> **`public static double`** `difficultyRating()`
>
> Return a double on the range 1.0 (ridiculously easy) to 5.0 (insanely difficult).

> **`public static double`** `hoursSpent()`
>
> Return a realistic estimate (greater than zero) of the number of hours you spent on this assignment.

5. Style Restrictions (Same as in Program #1) (*Super Important!*)

Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

★ Capitalize the first letter of all class names. Use lowercase for the first letter of all method names.

★ Any time you open a curly brace, that curly brace should start on a new line.

★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.

★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.

★ Please avoid block-style comments: /* *comment* */

★ Instead, please use inline-style comments: // *comment*

★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"

★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed *above* your import statements.

★ Use end-of-line comments sparingly. Comments longer than three words should always be placed *above* the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be intended with two tabs.

★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.

★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.

★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do not have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)

★ When defining or calling a method, do not leave a space before its opening parenthesis. For example: use *System.out.println("Hi!")* instead of *System.out.println ("Hi!")*.

★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.

★ Use meaningful variable names that convey the purpose of your variables. (The exceptions here are when using variables like *i*, *j*, and *k* for looping variables or *m* and *n* for the sizes of some inputs.)

★ Do not use *var* to declare variables.

6. Compiling and Testing on Eustis (and the *test-all.sh* Script!)

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 my test cases:

```
javac ConstrainedTopoSort.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.

4. I've also included a script, *test-all.sh*, that will compile and run all test cases for you. You can run it on Eustis by placing it in a directory with *ConstrainedTopoSort.java* and all the test case files and typing:

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

   Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.

7. Grading Criteria and Miscellaneous Requirements

Important Note: When grading your programs, we will use different test cases from the ones we've released with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to this program that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

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

Your program must be submitted via Webcourses.

Important Note! Additional point deductions may be imposed for poor commenting and whitespace. Significant point deductions may be imposed for violating the style restrictions listed above. You should also still include your name and NID in your source code.

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.

Important! Programs that do not compile on Eustis will receive zero credit. When testing your code, you should ensure that you place *ConstrainedTopoSort.java* alone in a directory with the test case files (source files, the *sample_output* directory, and the *test-all.sh* script), and no other files. That will help ensure that your *ConstrainedTopoSort.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. 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! You should not print anything to the screen. Extraneous output will result in severe point deductions. The required methods you write in *ConstrainedTopoSort.java* should not print anything to the screen.

Important! No file writing. Please do not write to any files.

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 implement a required method, or failing to make certain methods *public*, *private*, *static*, and/or non-static (as required), 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.

Test your code thoroughly. Please be sure to create your own test cases and thoroughly test your code. You're welcome to share test cases with each other, as long as your test cases don't include any solution code for the assignment itself.

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