Programming Assignment #6: TopolALLgical

COP 3503, Spring 2019

Due: Sunday, April 14, *before* 11:59 PM

Abstract

In this assignment, you will determine all valid topological sorts for an arbitrary graph. You are required to take a backtracking approach to this problem, which means that by completing this assignment, you will attain a deeper understanding of the general structure of recursive backtracking algorithms.

While it is possible for a graph with |V| nodes to have |V|! valid topological sorts, your backtracking algorithm must be coded in such a way that it avoids going down paths that would very obviously lead to vertex orderings that are *not* topological sorts. Thus, while the worst-case runtime of your algorithm might be $O(|V|!)$, it must have a best-case runtime that is no worse than $O(|V|^2)$.[1]

As you begin to work on this assignment, you might find it helpful to refer to my backtracking notes in Webcourses. In particular, see the section titled, "Basic Anatomy of a Backtracking Algorithm."

Deliverables

TopolALLgical.java

Note! The capitalization and spelling of your filename matter!

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

---

1   You might be able to achieve an even better best-case runtime, depending on the graph representation you choose.

1. Problem Statement

Given a directed graph, use backtracking to generate all valid topological sorts for that graph. For example:
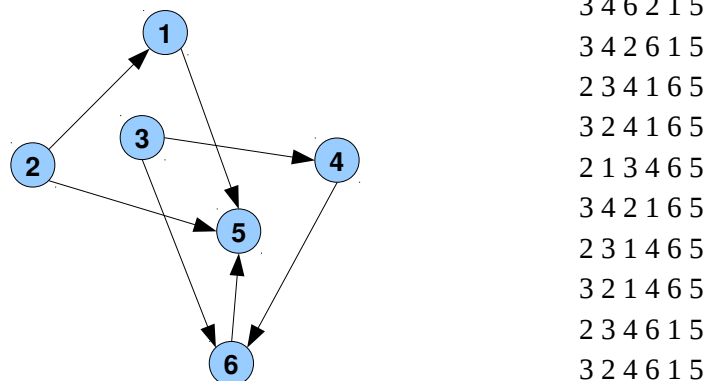


```
3 4 6 2 1 5
3 4 2 6 1 5
2 3 4 1 6 5
3 2 4 1 6 5
2 1 3 4 6 5
3 4 2 1 6 5
2 3 1 4 6 5
3 2 1 4 6 5
2 3 4 6 1 5
3 2 4 6 1 5
```

Figure 1: *A digraph shown alongside a list of all valid topological sorts for that graph.*
Note that the order of the 10 strings in the list does not matter.

Each valid topological sort will be represented as a string and placed into a HashSet. Each string should be a space-separated permutation of vertices. There should not be any spaces at the beginning or end of each string. For further details on the string format, refer to the test cases included with this assignment.

2. Input File Format

This program uses the same input file specification given in Program #5: TopoPaths. Keep in mind that vertices are numbered 1 through |V| (as opposed to 0 through |V| - 1).

3. Special Requirements: Recursive Backtracking and Efficiency

You must solve this problem with a recursive backtracking algorithm. Your algorithm should backtrack when an obviously infeasible state is reached, rather than continuing down a path that is guaranteed to lead to an invalid vertex ordering (i.e., a vertex ordering that does not correspond to a valid topological sort).

So, your algorithm should not simply generate all |V|! possible permutations of the vertices in a graph and then check whether they're valid topological sorts. Certainly, it might be possible for a graph with |V| vertices to have |V|! valid topological sorts, and so the worst-case runtime of your algorithm will be O(|V|!). However, the best-case runtime must be no worse than $O(|V|^2)$.

Also, your algorithm should be written in such a way that it generates each permutation of vertices *at most* once. If it generates a particular permutation multiple times, the HashSet can certainly help get rid of those duplicates, but your runtime will necessarily suffer for having produced the same permutation multiple times. Find a way to avoid this.

4.   Method and Class Requirements

Implement the following methods in a class named *TopolALLgical*. Please note that they are all ***public*** and ***static***.

public static HashSet<String> allTopologicalSorts(String filename)

> Open *filename* and process the graph it contains. Return a HashSet containing strings corresponding to all the topological sorts for the graph. For the required string format, refer to the test cases included with this assignment. If the graph has no topological sorts, or if the specified file does not exist, return an empty HashSet (*not* a null pointer!). If *filename* refers to an existing file, that file will follow the input format indicated above. Your method must throw exceptions as necessary, although in the case where an input file does not exist, you should catch the exception and return an empty HashSet.

public static double difficultyRating()

> Return a double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 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.   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 TopolALLgical.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 *TopolALLgical.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.

6.   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 <u>*above*</u> your import statements.

★   Use end-of-line comments sparingly. Comments longer than three words should always be placed <u>*above*</u> 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 <u>*not*</u> 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.

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:

100% Passes test cases. Your program must use backtracking in order to be eligible for credit. A non-backtracking solution will not receive credit, even if it passes all test cases. This portion of the grade includes tests of the *difficultyRating()* and *hoursSpent()* methods.

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 *TopolALLgical.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 *TopolALLgical.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! 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 *TopolALLgical.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 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.

(*New!*) **With this assignment, I've included fewer test cases than usual.** I want to transfer more responsibility for test case creation to you as you progress through this class and develop a stronger understanding of test driven development.

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