Programming Assignment #6: TopolALLgical

COP 3503, Fall 2016

Due: Sunday, November 20, *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 n nodes to have $n!$ 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 $O(n!)$, it must have a best-case runtime that is no worse than $O(n^2)$.[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, "Supplementary: Basic Anatomy of a Backtracking Algorithm."

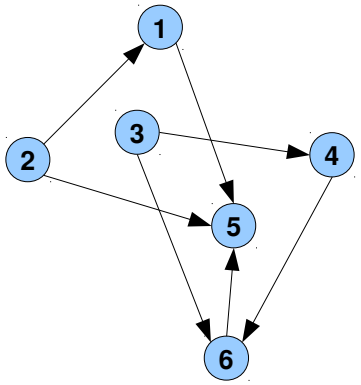
Deliverables

TopolALLgical.java

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: TopoPath. Keep in mind that vertices are numbered 1 through *n* (as opposed to 0 through *n* - 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 $n!$ 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 n vertices to have $n!$ valid topological sorts, and so the worst-case runtime of your algorithm will be $O(n!)$. However, the best-case runtime must be no worse than $O(n^2)$.

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`. Notice that all these methods are both **`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 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:

90%	program passes test cases[2]
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 TopolALLgical.java alone in a directory with the test case files (source files and the input_files and sample_output directories), and no other files. That will help ensure that your TopolALLgical class 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 refers to home-brewed classes that you are not submitting with this 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! 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,

2 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.

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!

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 behavior for any of the methods you're implementing.

You should create additional test cases. Please be aware that the test cases included with this assignment writeup are by no means comprehensive, and serve as a bare minimum to get you started with testing your code. Please be sure to create your own test cases and thoroughly test your code. Sharing test cases with other students is allowed, but you should challenge yourself to think of edge cases before reading other students' test cases.

New! With this assignment, I've actually 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!