

COSC 2P95 – Lab Exercise 8 – Graphs

Lab Exercises are uploaded via the Sakai page.

Since you need to submit both sample executions and your source files, package it all up into a .zip to submit.

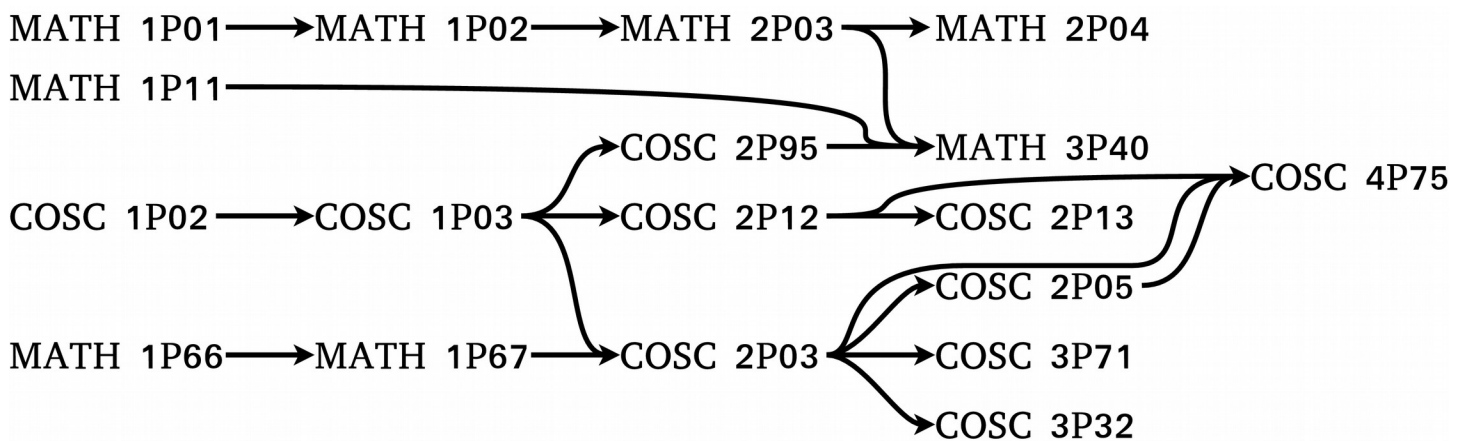
Directed graphs have numerous uses (travel plans, compiling, communications, planning, etc.). Several of those uses rest on the idea that Z must, at some point, follow A.

For compiling, this can mean things like dependencies (implementation An depends on interface/specification As; library L depends on components As, Bs, and Cs; etc.).

But let's look at something simpler. Suppose a student is part-time, getting a second degree, and may only take a single course at a time. This student wishes to get a rough feel for which courses would be available to take in which order.

For example, one would not try taking MATH 1P02 prior to MATH 1P01, since 1P01 is a *prerequisite* of 1P02. Taking COSC 2P03 must follow taking COSC 1P03, but may be before or after COSC 2P12.

Let's look at one possible subset of courses for consideration:



Of course, one wouldn't (couldn't) really take this particular selection of courses. Prerequisites have been simplified and mixed to the point where they don't actually align with any specific program on the calendar.

We can look at this as a *directed acyclic graph*.

- Why a graph? Because we have vertices/nodes with edges to other vertices
- Why directed? Because there's a clear progression of time
- Why acyclic? Because “you must take 1P01 both before *and* after 1P02” doesn't make much sense

*disclaimer: when depicting dependencies, it isn't uncommon to have the arrows drawn in the reverse direction as above (e.g. an arrow from MATH 1P02 → MATH 1P01, to show that 1P02 is *dependent on* 1P01). However, this version is far better suited to our needs.

So, we have a DAG. Now what?

- All we need is a listing of the courses, such that prerequisites will certainly be fulfilled, right?
 - This is *immensely* easy to do by hand (for a problem size this small)
 - We need an automated version that will *always* work

To accomplish this, we can perform a simple **topological sort**.

Topological Sorts:

A topological sort of a graph is simply a listing of the vertices in such a sequence that, for every pair of vertices, u and v , if there exists a path from u to v , then u will be listed before v in the sequence.

- That's why it has to be directed *acyclic* graph; you couldn't list u both before and after v

There are a few ways to solve this, but there's one super-easy method. We just need one more definition first.

For an *undirected graph* (i.e. not what we're using here), the *degree* of a vertex is the *number of edges* connecting it.

For a *directed graph*, the *indegree* of a vertex is the number of *inbound edges* connecting into the vertex.

So, for the graph above, MATH 1P01 has an indegree of 0, while COSC 4P75 has an indegree of 3.

For many graphs (including the one above), there are *multiple* potential solutions for a topological sort. Since the only risk of making the wrong choice is to accidentally select a vertex before one of its ancestors, we know that it's *always* safe to select a vertex with an indegree of 0.

- e.g. for my first choice, I could pick MATH 1P01, MATH 1P11, COSC 1P02, or MATH 1P67. It wouldn't matter which, since they're all safe
 - Similarly, any of the remaining ones would be entirely safe for the second selection
 - However, if we were to pick MATH 1P01, then MATH 1P02 would *also* be safe, right? Why?

Once we make a selection, we've satisfied the requirements of its outbound edges, and thus reduced the prerequisite concerns for anything that depended on it.

Knowing all of this, our algorithm is pretty simple:

- Determine the indegrees of all vertices in the graph
- Pick *any* unvisited vertex, v , with an indegree of zero
 - Process (in this case, print) v
 - For each outbound edge of v , leading to vertex w , reduce the indegree of w by 1
- Repeat until either all vertices have been exhausted, or there are no remaining vertices with 0 indegree

You might notice this also provides a makeshift test for whether the graph is cyclic or acyclic. If you stop before processing all vertices, then there must have been a cycle.

For the sake of output, since you'll *probably* be finding out whether or not it's cyclic when you're partway through the algorithm, you might want to queue up your print statements into a `stringstream` (to then retrieve the `.str()` at the end to print).

If you go this route, don't forget to include `sstream`.

For testing, you've been provided with two sample files: `prereqs.txt` (which will work), and `dangit.txt` (which is cyclic, so won't).

You can see sample executions from both on the last page. Don't forget that there are multiple possible answers, so your sort might not look like mine!

Requirements for Submission:

For your submission, in addition to including your source files, also include a sample execution on both data files (or on two other data files, assuming they're still representative of the two possible outcomes).

Pack it all up into a `.zip` to submit.

Sample Execution:

For acyclic graph:

Graph filename: `prereqs.txt`

Using `prereqs.txt`

File loaded.

Loaded graph.

Vertices:

```
[0:MATH1P01], [1:MATH1P02], [2:MATH2P03], [3:MATH2P04], [4:MATH1P11], [5:COSC2P95],  
[6:MATH3P40], [7:COSC4P75], [8:COSC1P02], [9:COSC1P03], [10:COSC2P12], [11:COSC2P13],  
[12:COSC2P05], [13:MATH1P66], [14:MATH1P67], [15:COSC2P03], [16:COSC3P71], [17:COSC3P32]
```

Edges:

```
MATH1P01 -> MATH1P02  
MATH1P02 -> MATH2P03  
MATH2P03 -> MATH2P04,MATH3P40  
MATH2P04 ->  
MATH1P11 -> MATH3P40  
COSC2P95 -> MATH3P40  
MATH3P40 ->  
COSC4P75 ->  
COSC1P02 -> COSC1P03  
COSC1P03 -> COSC2P95,COSC2P12,COSC2P03  
COSC2P12 -> COSC4P75,COSC2P13  
COSC2P13 ->  
COSC2P05 -> COSC4P75  
MATH1P66 -> MATH1P67  
MATH1P67 -> COSC2P03  
COSC2P03 -> COSC4P75,COSC2P05,COSC3P71,COSC3P32  
COSC3P71 ->  
COSC3P32 ->
```

Topological Sort found!

```
MATH1P01 MATH1P02 MATH2P03 MATH2P04 MATH1P11 COSC1P02 COSC1P03 COSC2P95 MATH3P40 COSC2P12  
COSC2P13 MATH1P66 MATH1P67 COSC2P03 COSC2P05 COSC4P75 COSC3P71 COSC3P32
```

For cyclic graph:

Graph filename: `dangit.txt`

Using `dangit.txt`

File loaded.

Loaded graph.

Vertices:

```
[0:MATH1P01], [1:MATH1P02], [2:MATH2P03], [3:MATH2P04], [4:MATH1P11], [5:COSC2P95],  
[6:MATH3P40], [7:COSC4P75], [8:COSC1P02], [9:COSC1P03], [10:COSC2P12], [11:COSC2P13],  
[12:COSC2P05], [13:MATH1P66], [14:MATH1P67], [15:COSC2P03], [16:COSC3P71], [17:COSC3P32]
```

Edges:

```
MATH1P01 -> MATH1P02  
MATH1P02 -> MATH2P03  
MATH2P03 -> MATH2P04,MATH3P40  
MATH2P04 ->  
MATH1P11 -> MATH3P40  
COSC2P95 -> MATH3P40  
MATH3P40 ->  
COSC4P75 -> MATH1P66  
COSC1P02 -> COSC1P03  
COSC1P03 -> COSC2P95,COSC2P12,COSC2P03
```

```
COSC2P12 -> COSC4P75,COSC2P13
COSC2P13 ->
COSC2P05 -> COSC4P75
MATH1P66 -> MATH1P67
MATH1P67 -> COSC2P03
COSC2P03 -> COSC4P75,COSC2P05,COSC3P71,COSC3P32
COSC3P71 ->
COSC3P32 ->
Cyclic dependencies; no topological sort possible.
```