Project 3

COP 4530, Spring 2020

Due April 22, 2020

1 Overview

In this project, you will implement topological sorting in a directed acyclic graph. In addition, your implementation will:

- represent the graph as an adjacency matrix
- always choose the vertex with the lowest ID when given a choice (e.g., when iterating through the neighbors of a vertex and when selecting a vertex of in-degree 0 to start)
- throw an exception if it encounters a cycle

I have provided you with a driver and a preliminary header file for the project. The provided header is just to get you started–you are allowed to change it however you like; however, your solution must be compatible with the provided driver file. You are strongly encouraged to add more data members and member functions to the class defined in the header.

2 Submitted code

You should submit a zip archive containing two files: `digraph.h` and `digraph.cpp`, which define and implement a DigraphMatrix class. This class must have a constructor that accepts a string (or `const string&`), and it must have a function `topologicalSort` that accepts no parameters and returns a vector of integers. The constructor must be able to construct the graph based on the name of an input file to read. (See section 3 for a description of the graph file format.) Moreover, it must represent the corresponding graph using an adjacency matrix.

The `topologicalSort` function should return the vertices of the vertices of the graph, in topologically sorted order if the directed graph does not contain a cycle. If the directed graph *does* contain a cycle, though, `topologicalSort` should throw an exception. The type of exception it throws (and its `what` function) are not important, but it must be an `std::exception` object or one that inherits from `std::exception`.

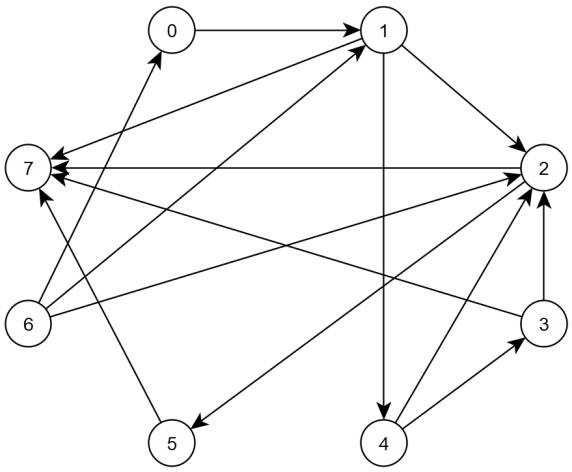
3 Input file

The input graph is described in edge list format. The first line of the file has the number of vertices and edges of the graph, separated by a space (n and m, respectively).

The following m lines have two nonnegative integers each. These integers, u and v, will be in the range 0 to $n-1$, and they represent the endpoints of an edge in the graph. Specifically, they represent an edge from u to v.

4 Example 1

Consider running the topological sorting algorithm described in class on the graph below:

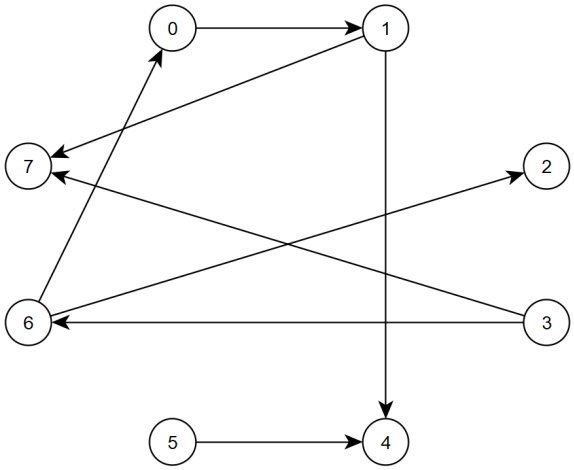


We start by finding a node with no incoming edges. In this graph, only vertex 6 has no incoming edges, so we start our DFS at vertex 6 and mark it as discovered. The neighbors of 6 are 0, 1, and 2. Since we should consider neighbors in sorted order, we recurse on the 0 and mark it as discovered. The only neighbor of 0 is 1, so we recurse on 1 and mark it as discovered. The first neighbor of 4 is 2, so we recurse on 2 and mark discovered. The neighbors of 2 are 5 and 7, so we recurse on 5 and mark it discovered. The only neighbor of 5 is 7, so we recurse on 7 and mark it discovered.

Vertex 7 has no outgoing neighbors, so we mark it explored and add it to our output array (7). When we return to 5, it has no other neighbors, so we mark it explored and add it to the array (7, 5). Returning to 2, we see that 7 has already been explored, so we mark 2 explored and add it to the array (7, 5, 2). Returning to 4, the other neighbor of 4 is 3, which we have not visited yet, so we recurse on 3 and mark it discovered. The neighbors of 3 are 2 and 7, both of which are explored, so we mark 3 as explored and add it to the array (7, 5, 2, 3). Returning back to 4, we've explored both of its neighbors, so we mark 4 explored and add it to the array (7, 5, 2, 3, 4). When we return to 1, we see that the other neighbors are 2 and 7, both of which are explored, so we mark 1 explored and add it to the array (7, 5, 2, 3, 4, 1). We return to 0, but its only neighbor was 1, so we mark it explored and add to the array (7, 5, 2, 3, 4, 1, 0). Finally, we return to 6, and all of its neighbors (indeed, every other vertex of the graph) has been marked explored, so it too is marked explored and added to the array (7, 5, 2, 3, 4, 1, 0, 6). Reversing this array yields the final answer (6, 0, 1, 4, 3, 2, 5, 7). (Due to the structure of the graph, this is the only correct answer.)

5 Example 2

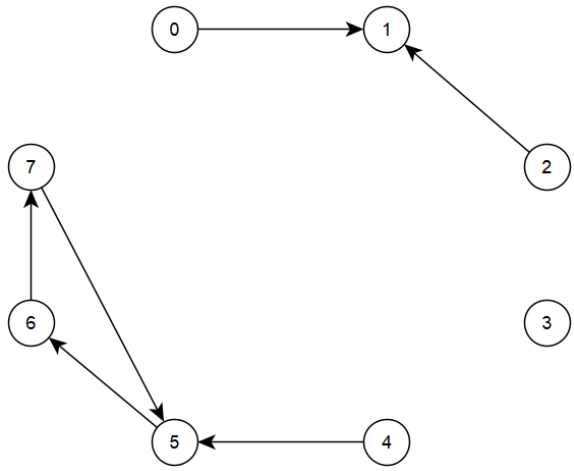
Consider running the topological sorting algorithm described in class on the graph below:



This graph has two vertices with in-degree zero, 3 and 5. Since we need to select the vertex with lowest ID, we start with 3. Vertex 3 has two neighbors, 6 and 7, so after marking it discovered, we recurse on 6 and mark 6 discovered. The neighbors of 6 are 0 and 2, so we recurse on 0 and mark it discovered. The only neighbor of 0 is 1, so we recurse on 1 and mark it discovered. The neighbors of 1 are 4 and 7, so we recurse on 4 and mark it discovered. Vertex 4 has no outgoing neighbors, so we mark it explored, add it to the list (4), and return to 1. Vertex 1 continues to its neighbor 7, which we mark as discovered. Vertex 7 has no outgoing neighbors, so we mark it explored and add it to the list (4, 1). When we return to 1, we have explored all outgoing neighbors, so we mark 1 explored and add it to the list (4, 7, 1). We then return to 0, which has no other neighbors, so we mark 0 explored and add it to the list (4, 7, 1, 0). We return from 6 and continue to the next neighbor, which is 2, and mark it discovered. Vertex 2 has no neighbors, so we mark it explored, add it to the list (4, 7, 1, 0, 2), and return to 6. Vertex 6 has no other neighbors after 2, so we mark it explored and add it to the list (4, 7, 1, 0, 2, 6). Finally, we return to 3, mark it explored, and add it to the list (4, 7, 1, 0, 2, 6, 3). Since we have not yet discovered vertex 5, we explore it, and as 4 has already been explored, we mark 5 explored and add it to the list (4, 7, 1, 0, 2, 6, 3, 5). Reversing this list yield the final answer (5, 3, 6, 2, 0, 1, 7, 4).

6 Example 3

We will trace the topological sorting algorithm on the graph below:



This graph has 4 vertices with in-degree zero: 0, 2, 3, and 4. We start at 0 and mark it discovered. The only neighbor of 0 is 1, so we mark it discovered. Vertex 1 has no outgoing neighbors, so we mark it explored, add it to the list (1), and return to 0. Since 0 has no other neighbors, we mark it explored and add it to the list (1, 0). We then start again at 2. After marking 2 as discovered, we see that its only neighbor, 1, has been marked explored, so we mark 2 explored and add it to the list (1, 0, 2). We continue on to vertex 3. We mark 3 as discovered, and since it has no neighbors, we mark it explored and add it to the list (1, 0, 2, 3). When we continue to 4, we mark it as discovered and continue on to 5. We mark vertex 5 discovered and continue to vertex 6, which we mark as discovered and continue to vertex 7. At vertex 7, we see that vertex 5 is a neighbor, which creates a cycle (5, 6, 7, 5). At this point, your code should throw an exception.

Note that the graphs formed by removing vertex 4 or reversing the edge (4, 5) would also have cycles. You may wish to trace the topological sorting algorithm on one of these graphs, as well.

7 Grading

Your code will be evaluated based on whether it produces the correct output for some number of test cases. You may use any development environment that you like when developing the code; however, it will be compiled and run using `g++` in a Linux environment. Code that does not compile will not receive substantial credit, so be sure that your code can be compiled using `g++`.

8 Important note

You are allowed to use any code posted for this course on Canvas, and you are allowed to discuss this project with the course TAs or professor. However, you are *not* allowed to use code from the internet or your peers. Your code will be run through plagiarism-detection software, and violators will be dealt with accordingly.