Project 3

COP 4530, Spring 2021

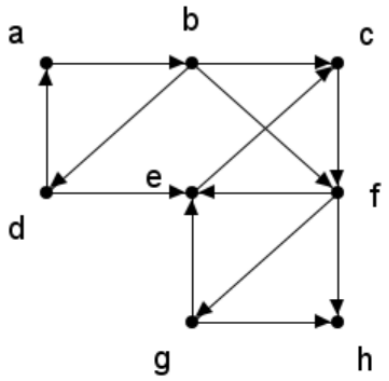
Due April 29, 2021

1 Overview

In this project, you will implement *strongly connected component* detection in a directed graph (digraph). 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 to start the traversal)

A *strongly connected component* (SCC) in a digraph is a maximal set of vertices so that every vertex in the set is connected to every other vertex (i.e., there exists a directed path between them). The difference between an SCC in a digraph and a component in an undirected graph is that in order for vertices u and v to be in the same SCC, there must be directed paths from u to v *and* from v to u (in an undirected graph, you can use the same path both ways). For example, the digraph below contains 3 SCCs: $\{a, b, d\}$, $\{c, e, f, g\}$, and $\{h\}$. In particular, b and c are in different SCCs because there is no path from c back to b. If the edges were undirected, though, there would only be 1 component. (In other words, this digraph has 3 SCCs and 1 *weakly connected component*.)



The algorithm for finding the strongly connected components (SCCs) in a digraph is based on a two-pass DFS algorithm and is described in section 2.

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, and you must implement the graph using an adjacency matrix. (You may use the adjacency matrix implementation described in class or the representation described in the textbook.) You will need to add more member functions to the class defined in the header; the provided header is not sufficient to solve the problem.

2 How to compute strongly connected components in a digraph

The algorithm described in this document calculates the SCCs of a digraph in two DFS passes. In the first DFS pass, you should add the vertices of the graph to a stack as you *explore* them. Note that when you iterate through the neighbors of a vertex v in a digraph, you should iterate through the outgoing neighbors (*out neighbors*) of v: the vertices that have an edge pointing to them from v.

Important: the second DFS pass uses the in neighbors (vertices with incoming edges) of a vertex instead of the out neighbors. You may either write separate functions to find the in neighbors and the out neighbors of a vertex, or you may write a single function for the out neighbors and another function to reverse the edges in the graph.

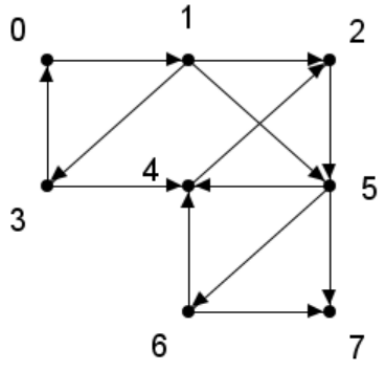
For the second pass, you should initialize each vertex to be undiscovered as normal, but instead of iteratively selecting the undiscovered vertex with lowest ID, you should perform the recursive DFS on the first undiscovered vertex in the stack (i.e., the outer loop of the second pass iterates in stack order rather than sorted order). Each iteration of this loop represents a different SCC, and you should assign the same component ID to all of the vertices encountered in the recursive function. Component IDs start numbering at 0 (i.e., component 0 should be the SCC containing the last vertex explored in the first DFS) and count upwards. As discussed previously, the recursive function processes the incoming neighbors of a vertex rather than the outgoing neighbors.

3 Algorithm output

The `stronglyConnectedComponents` function you will implement must return a vector containing an array-based (vector-based) map that associates each vertex in the graph with its component ID. Component IDs should start numbering at 0 (i.e., the component that contains the last vertex explored during the first pass should have ID 0), and they should count upwards from there (0, 1, 2, 3, etc.). In the first DFS pass, you must always choose the vertex with the min ID when given a choice; i.e., you must iterate through the neighbors of a vertex in sorted order, and you must start your DFS each iteration at the undiscovered vertex with the min ID. It should not matter which order you iterate through the neighbors of each vertex in the second pass, as long as the outer loop iterates through the vertices in stack order. Each input graph should have only a single correct solution for its SCC IDs.

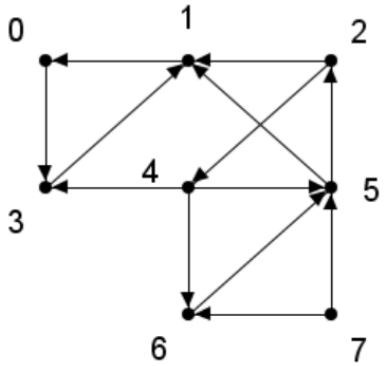
4 Example 1

Consider running the algorithm described above on the digraph below:



In the first pass of DFS, we start by discovering vertex 0. Vertex 0 only has one outgoing neighbor, so we recurse and discover vertex 1. The first undiscovered out neighbor of 1 is 2, so we recurse on 2 and mark it discovered. The only out neighbor of 2 is 5, so we recurse on 5 and mark it discovered. The first undiscovered out neighbor of 5 is 4, but 4 has no out undiscovered out neighbors (2 is discovered), so we mark 4 as explored. The next undiscovered out neighbor of 5 is 6, so we recurse on 6. From 6, we have already explored 4, so we continue to 7. Vertex 7 has no undiscovered out neighbors, so we mark it explored and return to 6. Vertex 7 was the last outneighbor of 6, so 6 is explored and returns to 5. Vertex 7 is already explored, so 5 is explored and returns to 2, which is marked explored and returns to 1. Vertex 3 is the next undiscovered out neighbor of 1, so we discover, explore, and return from vertex 3. Finally, we explore and return from vertices 1 and 0. The vertices of this digraph were explored in the order 4, 7, 6, 5, 2, 3, 1, 0. (Discover 0, Discover 1, Discover 2, Discover 5, Discover 4, Explore 4, Discover 6, Discover 7, Explore 7, Explore 6, Explore 5, Explore 2, Discover 3, Explore 3, Explore 1, Explore 0.) Thus, the stack contains (top-to-bottom) 0, 1, 3, 2, 5, 6, 7, 4 after the first pass.

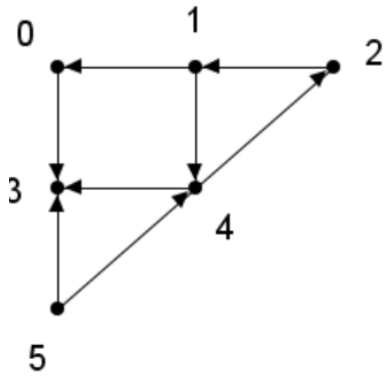
For these examples, we reverse all of the edges in the graph after the first pass so that the second pass will continue to use the outgoing neighbors of each vertex. The "reversed" graph appears below:



Starting with vertex 0 (top of the stack) in the reverse graph, we explore 3 then 1 before backtracking twice, so vertices 0, 1, and 3 belong to component 0. We skip vertices 1 and 3 in the stack (as they are already explored), and when we start at vertex 2, we discover 4, ignore 3 (already explored), discover 5, ignore 1, explore 5, discover 6, explore 6, explore 4, and explore 2, so vertices 2, 4, 5, and 6 belong to component 1. We then skip 5 and 6 in the stack, label 7 as component 2, skip 4, then return. The final component vector for this graph should contain [0, 0, 1, 0, 1, 1, 1, 2].

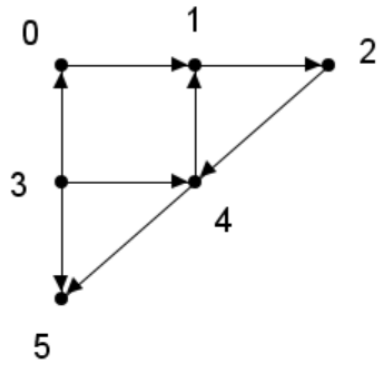
5 Example 2

Consider running the algorithm on the graph below:



Starting at vertex 0, we discover 0, discover 3, explore 3, and explore 0. Then starting at 1, we discover 1, discover 4, discover 2, explore 2, explore 4, and explore 1. Then, we discover and explore 5. Our exploration order was 3, 0, 2, 4, 1, 5, so our stack contains (top-to-bottom): 5, 1, 4, 2, 0, 3.

When we reverse the edges of the original graph, we get the graph below:



Starting at vertex 5, we discover and explore 5, so 5 is component 0. Then, starting at 1, we discover 1, discover 2, discover 4, ignore 5, explore 4, explore 2, and explore 1, so vertices 1, 2, and 4 are component 1. We skip 4 and 2 in the stack since they are explored, then we discover and explore 0, so vertex 0 is component 2. Finally, we discover and explore 3 (0, 4, and 5 are already explored), so vertex 3 is component 3. The final component vector should contain [2, 1, 1, 3, 1, 0].

6 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 a directed edge in the graph. Specifically, they represent an edge from u to v. Note that in a digraph, the the edge from u to v and the edge from v to u are separate, independent edges. A digraph may contain one, the other, both, or neither.

7 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 `stronglyConnectedComponents` that accepts no parameters and returns a vector containing the component ID for each vertex in the graph (see section 3 for description of the required output). The constructor must be able to construct the graph based on the name of an input file to read (see section 6 for a description of the graph file format.) Moreover, it must represent the corresponding graph using an adjacency matrix, and it must be compatible with the provided driver file (`scc-driver.cpp`).

8 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++`.

9 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.