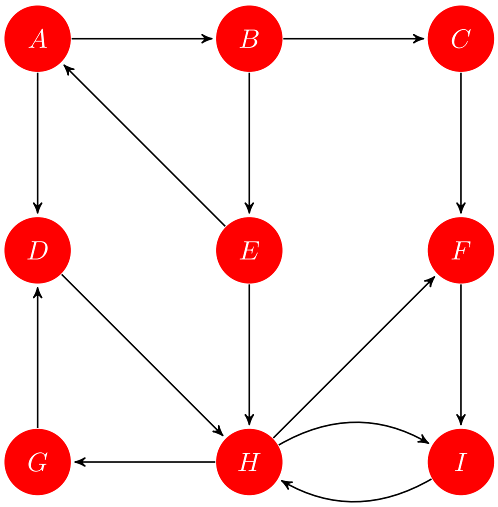
Problem Set 3
Erdun E
September 30, 2024

CS 5800: Algorithm
Problem Set 3

Exercises

Dasgupta 3.4 a-c(apply only to graph ii)

Run the strongly connected components algorithm on the following directed graphs G. When doing DFS on G^R: whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.



(ii)

In each case answer the following questions.

a. In what order are the strongly connected components (SCCs) found?

b. Which are source SCCs and which are sink SCCs?

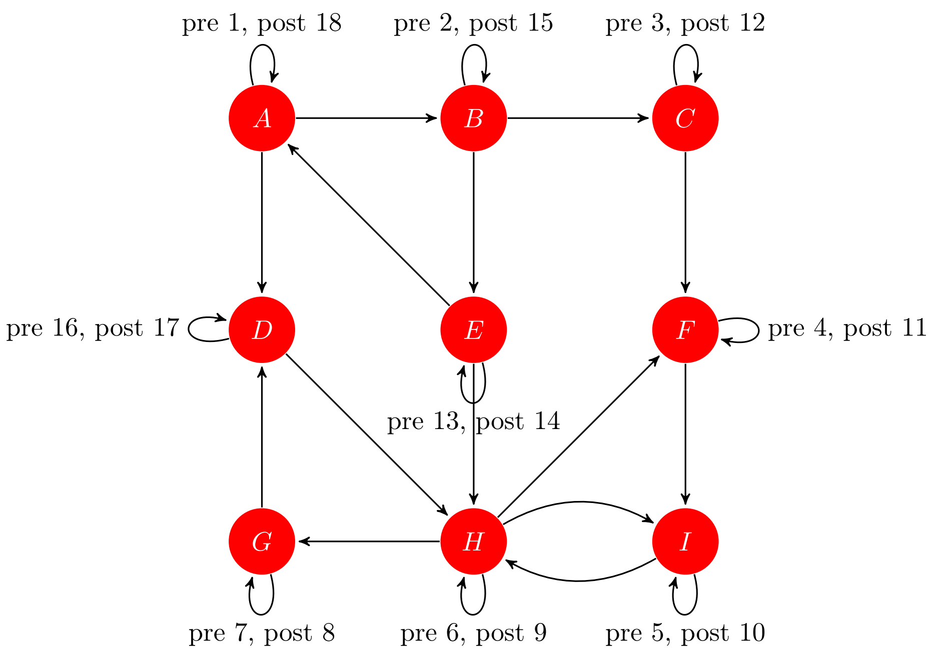
c. Draw the "metagraph" (each meta-node is an SCC of G).

Solution

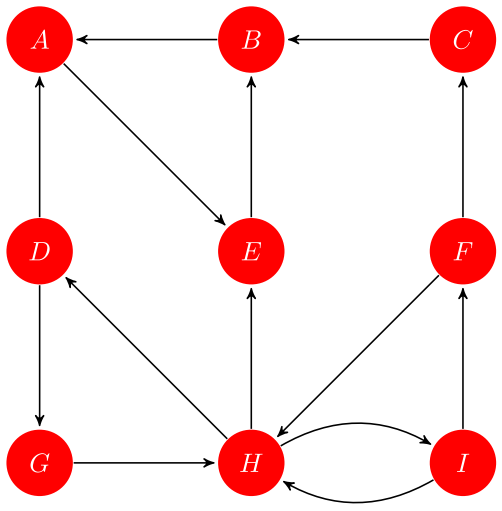
a. In what order are the strongly connected components (SCCs) found?

First DFS on original graph

- Start the first node A, visit B,then C, then F, then I, then H, then G, backtrack
- Backtrack to B and then visit E, then backtrack
- Backtrack to D and finished.



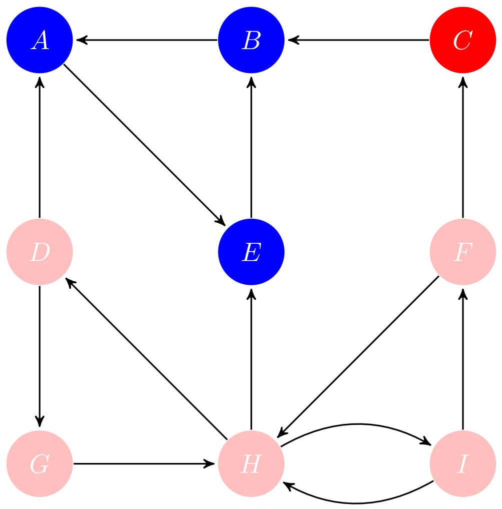
After reversing the edges, the transposed graph



Second DFS Pass on the Transposed Graph

- Start from A. $A \Rightarrow B \Rightarrow E$ forms one SCC that the $\{A, B, E\}$.
- $D \Rightarrow H \Rightarrow G \Rightarrow F \Rightarrow$ I forms another SCC that $\{D, F, G, H, I\}$.
- C left and its own SCC that the $\{C\}$

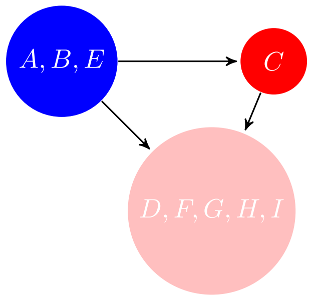
Thus, the strongly connected components are:$\{A, B, E\}$,$\{D, F, G, H, I\}$,$\{C\}$



b. Which are source SCCs and which are sink SCCs?

- The source SCC is $\{A, B, E\}$. The sink SCC is $\{D, F, G, H, I\}$.

c. Draw the "metagraph" (each meta-node is an SCC of G



Dasgupta 3.13 a-c (note that a asks for a formal proof)

Undirected vs. directed connectivity.

a. Prove that in any connected undirected graph G = (V, E) there is a vertex v $\in$ V whose removal leaves G connected. (Hint: Consider the DFS search tree for G.)

b. Give an example of a strongly connected directed graph G = (V, E) such that, for every v $\in$ V, removing v from G leaves a directed graph that is not strongly connected.

c. In an undirected graph with 2 connected components it is always possible to make the graph connected by adding only one edge. Give an example of a directed graph with two strongly connected components such that no addition of one edge can make the graph strongly connected.

Solution

a. Prove that in any connected undirected graph G = (V, E) there is a vertex v $\in$ V whose removal leaves G connected. (Hint: Consider the DFS search tree for G.)

Consider the Depth-First Search (DFS) tree of the graph G. The DFS tree is constructed by starting from any arbitrary vertex in G and exploring as deeply as possible along each branch before backtracking. The properties of the DFS tree are as follows:

- The vertices of the DFS tree are exactly the vertices of G
- The edges of the DFS tree are a subset of the edges in G.
- The DFS tree is connected, meaning there is a path from the root vertex to every other vertex in the tree.

Now, consider any leaf vertex v in the DFS tree. A leaf vertex is a vertex that has no children in the DFS tree. In the original graph G, this leaf vertex may have edges connecting it to other vertices, but within the DFS tree, it only has a single edge that connects it to its parent.

If remove this leaf vertex v from the graph G, the remaining vertices still form a connected subgraph. This is because the removal of a leaf in the DFS tree does not affect the connectivity of the tree, the rest of the tree remains intact, and since the DFS tree reflects the connectivity of the original graph, the entire graph remains connected after the removal of the leaf.

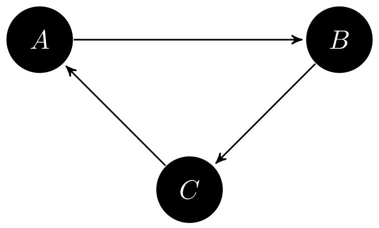
Since removing a leaf vertex from the DFS tree leaves the graph connected, there always exists a vertex in any connected undirected graph whose removal does not disconnect the graph

b. Give an example of a strongly connected directed graph G = (V, E) such that, for every v $\in$ V, removing v from G leaves a directed graph that is not strongly connected.

A strongly connected directed graph is one where there is a directed path between every pair of vertices. This means that from any vertex, you can reach every other vertex, and every vertex can also reach the starting vertex.

There is a simple example with three vertices that A, B, and C. These vertices are connected in a cycle:

- There is a directed edge from $A \Rightarrow B$
- $B \Rightarrow C$
- $C \Rightarrow A$



This graph is strongly connected because:

- From A, you can travel to B, then to C, and back to A.
- Similarly, you can start from B and reach both A and C, and the same goes for C.

Now, if we remove any one vertex, such as C, the remaining graph will no longer be strongly connected. For example, if C is removed:

- There is a path from $A \Rightarrow B$, but can't return from $B \Rightarrow A$, because $C \Rightarrow A$ is gone. So the graph is no longer strongly connected because it's impossible to reach all vertices from any other vertex.

This same logic applies if you remove any of the other vertices. If remove A or B, the remaining two vertices will not be able to form a cycle, and thus, the graph will lose its strong connectivity.

Therefore, in this strongly connected directed graph, removing any vertex results in a graph that is no longer strongly connected. This proves that for this graph, removing any vertex will leave a graph that is not strongly connected.

c. In an undirected graph with 2 connected components it is always possible to make the graph connected by adding only one edge. Give an example of a directed graph with two strongly connected components such that no addition of one edge can make the graph strongly connected.

Consider a graph with two disjoint cycles, where each cycle is a SCC. Adding just one edge between these two components allows travel from one cycle to the other, but not back. This means that adding one edge is not sufficient to make the graph fully strongly connected.

In a strongly connected graph, it must be possible to reach every vertex from any other vertex. Adding a single edge between two SCCs does not make the whole graph strongly connected because it does not create paths in both directions between the components.

For example, connecting one vertex in one SCC to another in the second SCC only enables travel in one direction. To make the entire graph strongly connected, at least two edges are needed: one going from the first SCC to the second and another going from the second SCC back to the first.

This situation is similar to undirected graphs, where two components can be connected with just one edge. However, in directed graphs, more than one edge is required to ensure that travel is possible in both directions.

In conclusion, for a directed graph with two strongly connected components, adding just one edge cannot make the graph strongly connected.

Dasgupta 3.16

Suppose a CS curriculum consists of n courses, all of them mandatory. The prerequisite graph G has a node for each course, and an edge from course v to course w if and only if v is a prerequisite for w. Find an algorithm that works directly with this graph representation, and computes the minimum number of semesters necessary to complete the curriculum (assume that a student can take any number of courses in one semester). The running time of your algorithm should be linear.

Initial the given information

- Given a graph G, each node represents a course.
- And directed edges represent prerequisites.
- The goal is to compute the minimum number of semesters required to complete all courses.
- In each semester, a student can take any number of courses as long as the prerequisites for those courses are already completed.

Implement Approach

- The problem can be solved using topological sorting which processes nodes in order of their in-degrees. The key idea is to assign each course to the earliest semester in which all its prerequisites are completed.

Steps

- Start by calculating the in-degree that the number of prerequisites for each course. Courses with no prerequisites can be taken in the first semester.
- Use a queue to track courses that can be taken in the current semester. Once a course is taken, reduce the in-degree of its dependent courses.
- Each time a set of courses is taken, move to the next semester and continue processing until all courses are completed.
- The number of semesters matches to the number of times a new set of courses is taken

Algorithm: 1 ComputeMinSemesters(G) Psesudocode

```
INPUT Graph G // nodes = courses, edges = prerequisites
OUTPUT Minimum number of semesters required to complete the curriculum.

1. Initial the variables
n = number of course
inDegree[] = array to store the in-degree of each course, initialized to 0
semester[] = array to store the semester in which each course can be taken
queue = empty queue for courses with no prerequisites // in-degree = 0

2. Calculate in-degrees
    For each course v in G :
        For each neighbor w of v: // dependent course
            Increase inDegree[w] // Increment the in-degree for each dependent course

3. Add courses with no prerequisites (in-degree = 0) to the queue:
    For each course v in G:
        If inDegree[v] == 0: // No prerequisites
            Add v to the queue
            Set semester[v] = 1 // First semester for courses with no prerequisites

4. Process the queue:
    While queue is not empty:
        Remove course v from the queue
        For each dependent course w of v:
            Decrease inDegree[w] by 1 // Reduce prerequisite count
            If inDegree[w] == 0: // If no remaining prerequisites, add to queue
                Add w to the queue
                Set semester[w] = semester[v] + 1 // Next semester

5. Return the maximum value in the semester[] // // Number of semesters required
```

This algorithm works in linear time, since each course and each edge is processed exactly once