CS 3510: Design & Analysis of Algorithms (Sections A & B)

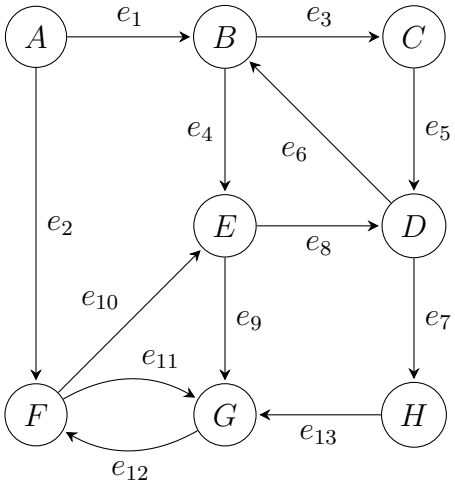
Homework 5: Intro to Graph Algorithms

Professors Dana Randall and Gerandy Brito **Due: 10/27/2022 11:59pm**

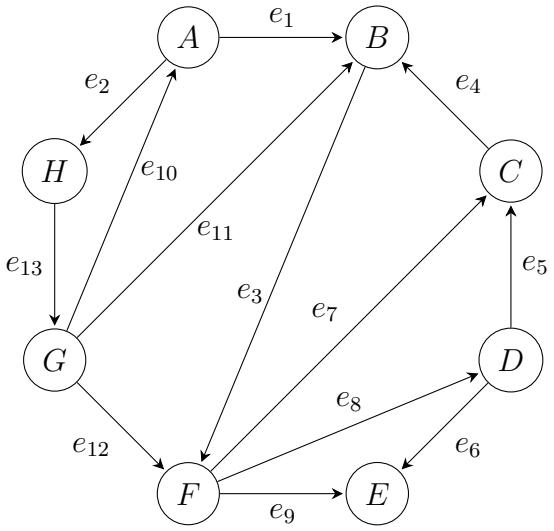
Name: Anthony Wong

1.) (25 points) Perform a depth-first search on each of the following graphs; whenever there's a choice of multiple vertices, pick the one that is alphabetically first.

Graph X



Graph Y



(a.) Classify each edge in Graph X as a tree edge, forward edge, back edge, or cross edge.

Solution:

Tree Edges: $e_1, e_3, e_5, e_7, e_{10}, e_{12}, e_{13}$
Forward Edges: e_2, e_4
Back Edges: e_6, e_8, e_9, e_{11}

(b.) Give the `pre` and `post` number of each vertex in Graph X.

Solution:

In the format $Vertex(pre, post)$:
$A(0, 15), B(1, 14), C(2, 13), D(3, 12), H(4, 11), G(5, 10), F(6, 9), E(7, 8)$

(c.) Classify each edge in Graph Y as a tree edge, forward edge, back edge, or cross edge.

Solution:

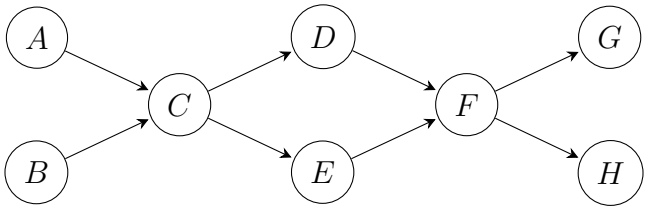
Tree Edges: $e_1, e_2, e_3, e_6, e_7, e_8, e_{13}$
Forward Edges: e_9
Back Edges: e_4, e_{10}
Cross Edges: $e_5, e_{11}.e_{12}$

(d.) Give the `pre` and `post` number of each vertex in Graph Y.

Solution:

In the format $Vertex(pre, post)$:
$A(0, 15), B(1, 10), F(2, 9), C(3, 4), D(5, 8), E(6, 7), H(11, 14), G(12, 13)$

2.) (25 points) Run the DFS-based topological ordering algorithm on the following graph. Whenever you have a choice of vertices to explore, always pick the one that is alphabetically first.



(a.) Indicate the `pre` and `post` numbers of the vertices.

Solution:

In the format $Vertex(pre, post)$:
$A(0, 13), C(1, 12), D(2, 9), F(3, 8), G(4, 5), H(6, 7), E(10, 11), B(14, 15)$

(b.) What are the sources and sinks of the graph?

Solution:

Sources: A & B
Sinks: G & H

(c.) What topological ordering is found by the algorithm?

Solution:

The ordering is:
B, A, C, E, D, F, H, G

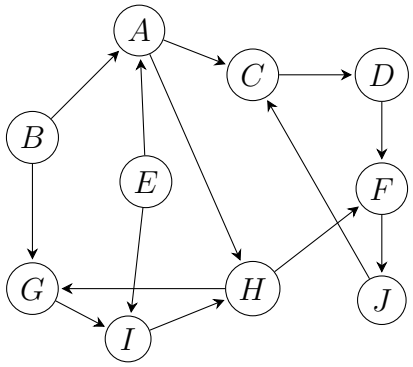
(d.) How many topological orderings does this graph have?

Solution:

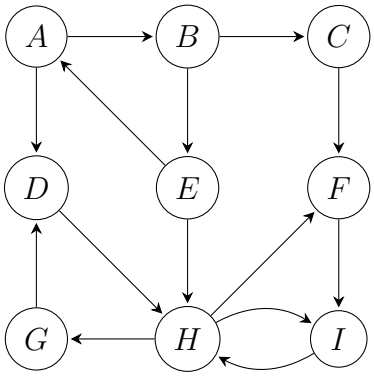
Given we have 3 spots where there are 2 nodes with the same degree, the number is given by $2^3 = 8$

3.) (25 points) Run the strongly connected components algorithm on the following directed graphs. When running DFS for this problem, if there is a choice of vertices to explore, always pick the one that is alphabetically first.

Graph X



Graph Y



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

Solution:

{C, D, F, J}, {H, G, I}, {A}, {E}, {B}

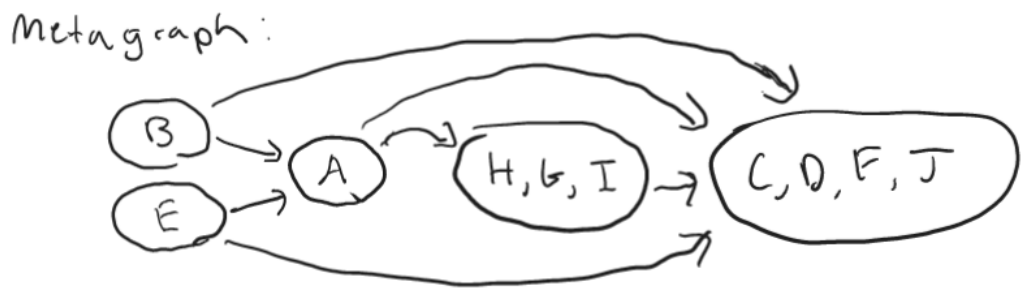
(b.) Which are source SCCs and which are sink SCCs for Graph X?

Solution:

Sources SCCs: {E}, {B}
Sink SCCs: {C, D, F, J} [Write your answer here]

(c.) Draw the "metagraph" of Graph X (each meta-node is a SCC of Graph X).

Solution:



(d.) What is the minimum number of edges you must add to Graph X to make it strongly connected?

Solution:

We need the sinks to link to the sources. Thus the minimum number of edges you must add is 2.

(e.) In what order are the strongly connected components (SCCs) found for Graph Y?

Solution:

{D, H, F, I, G}, {C}, {A, B, E}

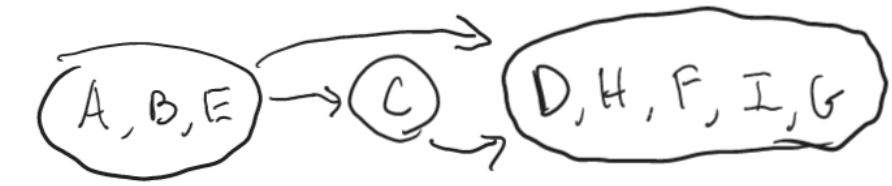
(f.) Which are source SCCs and which are sink SCCs for Graph Y?

Solution:

Source SCCs: {A, B, E}
Sink SCCs: {D, H, F, I, G}

(g.) Draw the "metagraph" of Graph Y (each meta-node is a SCC of Graph Y).

Solution:



(h.) What is the minimum number of edges you must add to Graph Y to make it strongly connected?

Solution:

We need the sinks to link to the sources. Thus the minimum number of edges you must add is 1.

4.) (25 points) Let $G = (V, E)$ be a directed graph for which you have an adjacency list. A vertex v is called a *global sink* if and only if it meets the following two conditions:

1. v has no outgoing edges
2. for every other vertex u, there is a path from u to v

Give an algorithm that determines if G has a global sink and, if the answer is yes, returns the global sink. Your algorithm should have running time $O(|V| + |E|)$.

Solution:

Design:

First, like the SCC algorithm, we reverse the edge direction of all the edges in graph G. Let's call this graph G'. We then should call **DFS** on G', keeping track of previsit and postvisit times for each node as we have done in class. Once we have run **DFS** on G', the node with the largest postvisit time is our candidate global sink, which in the context of a reversed graph, is a source. Just to cover the case that our graph G is a strongly connected component itself, we need to check if this candidate node has any nodes adjacent to it in the adjacency list. If it has adjacent nodes, we can say that the graph has no sinks, else we move on. With this candidate node, we then run **Explore** on G' with the candidate node. If the length of our visited set at the end of this function call is equal to the number of nodes in G, return the candidate node as we can conclusively say it is a global sink.

Correctness:
We reverse graph G and find the node with the largest postvisit time because that means that node is the last node seen during **DFS**. This is typically a source of the reversed graph G'. Now that we have the candidate node, we know that this node is a global sink only if we can reach every other node in G' from it. This is correct because if there is a path to any node in G', then there is a path from any other node to our candidate in G. By checking the length of our visited set versus the number of nodes in G, we can confirm that our candidate can reach all other nodes in G' and that all other nodes can reach it in G, making it a global sink. There is an edge case where our graph G is a strongly connected component itself, which means there is no sink. We can simply check this by seeing if our sink candidate has any adjacent nodes in the adjacency list. If there is we can say there are no sinks.

Runtime:
In this algorithm, we first build G' which has a runtime of $O(|E|)$ since it iterates through all the edges to reverse them. We then run **DFS** on G' which iterates through all edges and vertices, which results in $O(|V|+|E|)$. Our adjacency list check is constant. We then run **Explore** on G' which iterates through all edges and vertices, which again results in $O(|V|+|E|)$. Then, our final checking of lengths is also constant. Our overall runtime is then $O(2|V|+3|E|) = O(|V|+|E|)$.