SCHOOL OF ENGINEERING

VANDERBILT UNIVERSITY

CS 3250 Algorithms Kosaraju's Algorithm

VANDERBILT UNIVERSITY

Announcements

- **HW2** is due Wednesday, February 7th by 9 AM.
- There are two parts:
  1. **Brightspace Hashing Quiz**. Formative assessment. Graded but not timed.
  2. **Gradescope Written Questions.** Remember to keep your Gradescope answers:
     - Clear
     - Correct
     - Concise
- HW3 will be released this week.

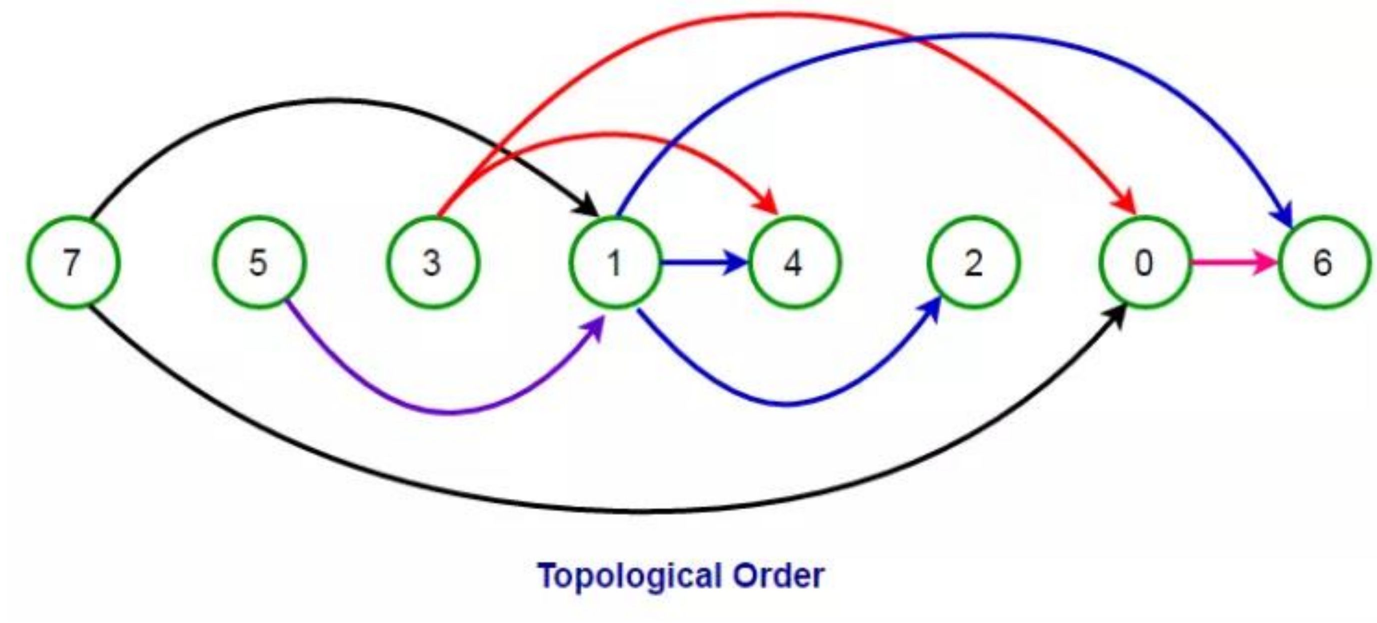
Announcements

- **HW1 Regrade Requests Feb 6th thru Feb 20th:**
  - Locate the question on Gradescope and click the Regrade Request Button at the bottom.
  - Be professional with your request.
  - State your reasoning clearly.
  - It is very difficult to argue for points back on clarity/conciseness. The TAs have seen over 110 student answers and have a good sense of what is clear and concise for the problem they are grading.

VANDERBILT UNIVERSITY

Depth-First Search: Topological Sort

- Another popular use of DFS is Topological Sort which is often used as the basis in task scheduling.
- A **topological sort** is a linear ordering of the vertices in a graph G such that for every **directed** edge uv, vertex u comes before v in the ordering.



Topological Sort: Observations

Proof (Inductive): Every DAG has a topological ordering

- **Base case:** n = 2; n is number of vertices in Graph G. Since the edge is directed, the graph must have a topological ordering from target vertex to source vertex.
- **Inductive hypothesis:** Assume our base case holds for k vertices. In other words, a DAG on k vertices has a topological ordering. We will prove that the hypothesis holds for k+1 vertices. That is, a DAG G on k+1 nodes has a topological ordering.
- Given an arbitrary graph G with k+1 vertices that is a DAG, find an arbitrary vertex v in G where v is a node with no outgoing edges. Remove v to create G', a graph on k vertices.

Topological Sort: Observations

Proof (con’t): Every DAG has a topological ordering

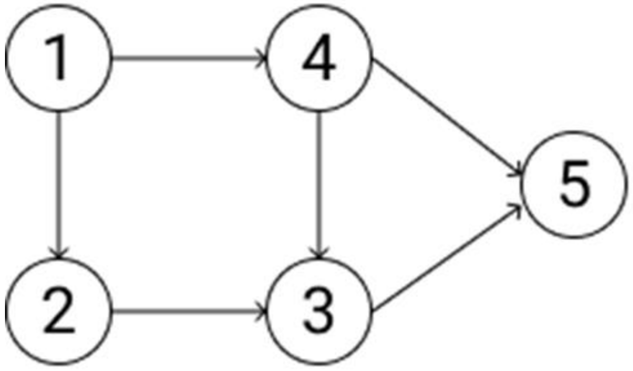
- It must be the case that G’ is a DAG as well. After all, if you have a DAG and you **remove** a vertex, you cannot create a cycle. So, G' is a DAG on k vertices and it has a topological ordering by the inductive hypothesis
- We can concatenate Topological ordering (G’) + v to get a topological ordering of G.
- QED

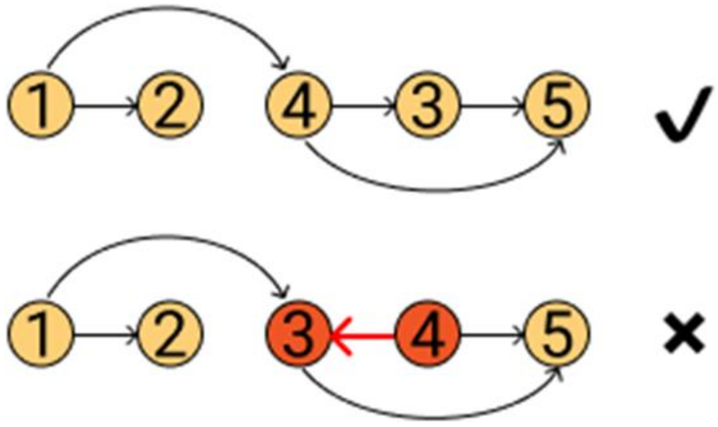
Topological Sort: Observations

- What if the graph G was disconnected? Then you can’t visit n vertices by following edges backwards.
- **Answer:** True, we’ll say we consider any connected component on $h \leq n$ vertices.
- What if there aren’t any connected components in the graph G?
- **Answer:** In other words, the graph G has no edges. Well, in that case G still must be a DAG since no vertex has any outgoing edges.

Topological Sort w/o Depth-First Search

- Nadha Skolar tries writing a TopSort. He calls this version TopoSkolar (he mistakenly thinks inventing his own sort will impress all the big companies).
- **Nadha's Idea:** Locate the sink index. Remove the sink vertex and any incoming edges to it. Repeat the process. The last vertex removed is the source vertex and the start of the topological ordering.





Topological Sort w/o Depth-First Search

- **Nadha's Idea:** Locate the sink index. Remove the sink vertex and any incoming edges to it. Repeat the process. The last vertex removed is the source vertex and the start of the topological ordering.
- **Analysis:** Assume graph G with V vertices…
  - **Locate the sink vertex** = at most V comparisons.
  - **Remove sink and incoming edges =** at most E
  - **Repeat proves for each vertex**… = Total O(V(V+E))…could be V^3 in a dense graph.
  - Ugh.

Topological Sort via Depth-First Search

How can DFS help us?

1. DFS will automatically find the sink node. Great!
2. DFS will automatically backtrack to right before the sink node. Even better!
3. **Anything else?** If we can just keep track of an ordering value to assign to each vertex as we back up, we're golden.

Topological Sort via Depth-First Search

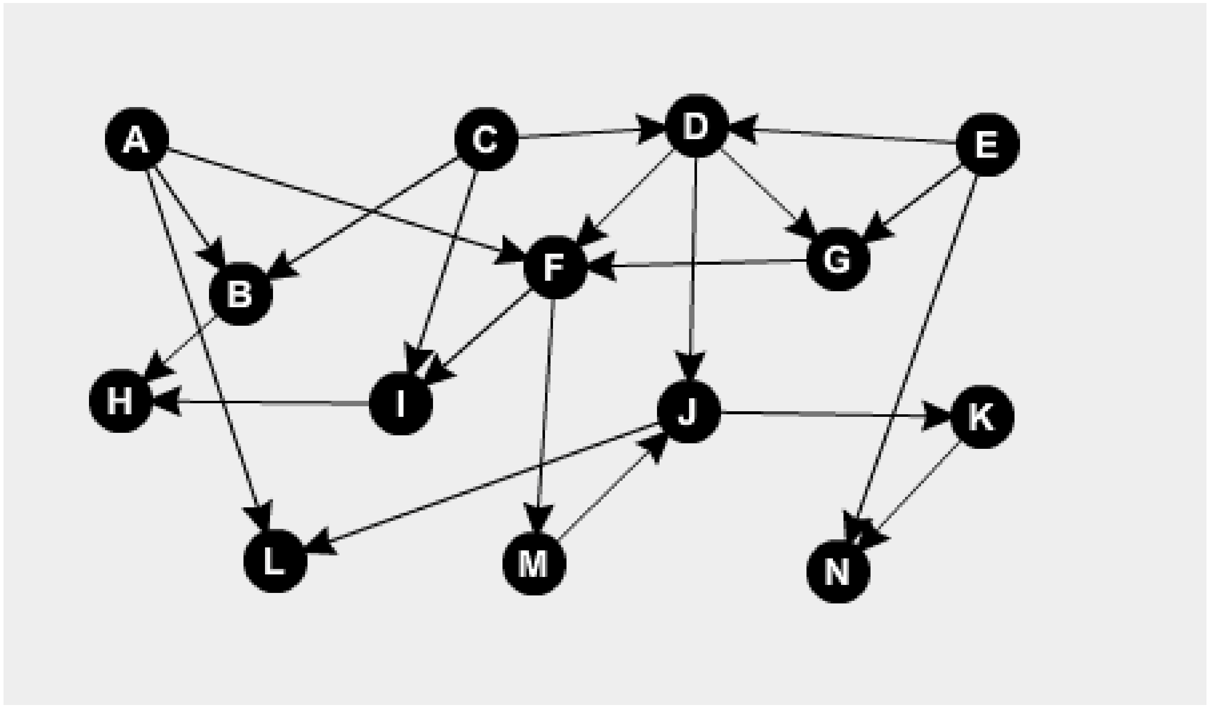
```
FOR each vertex u in V(G)
  state[u] = undiscovered
  parent[u] = nil
 time = 0
END FOR
//number backwards topologically
//use a global variable curNumber
curNumber = n
FOR each vertex u in V(G)
  IF (state[u] != discovered) THEN
       DFS(G, u)
END FOR
```

```
DFS(G, s)
  state[s] = discovered
  FOR each v adjacent to s
    IF (state[v] != discovered)
       parent[v] = s
       DFS(G, v)
  END FOR
  state[s] = processed
  topOrder[s] = curNumber
  curNumber--
END DFS
```

Total = Θ(V+E) or O(V+E)

On Your Own: Topological Sort

- **Practice On Your Own:** Using A as the starting vertex, perform a topological sort and determine a valid ordering. Use lexicographic order when given a choice.



Graphs and Connectivity

- **Connectivity:** There is a path from each vertex to every other vertex
- **Undirected graph:**
  - Is the graph connected?
  - What are the graph's connected components?
  - Which vertex is an articulation point?
- **Directed graph:**
  - Is the graph strongly connected?
  - What are the graph's strongly connected components?

Directed Graphs and Connectivity

- Our next application is an algorithm for determining a **directed** graph’s **strongly connected components**.
- The **strongly connected components** (SCCs) of a **directed** graph G can be defined as the equivalence classes on the relation uRV where uRv iff you can get from u to v via some directed path and you can get from v to u via some directed path.

déjà vu
noun [U]
the strange feeling that in some way you have
experienced what is happening no
• When I met her, I ha
CFG Law

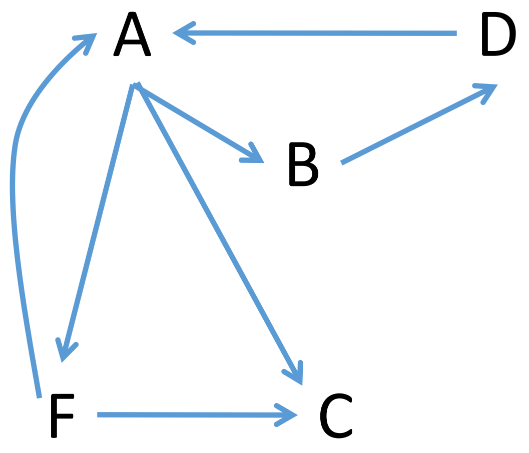
Directed Graphs and Connectivity

- When we talk about SCCs we always speak about **maximal** SCCs.
- **Huh?** In other words, if we find an SCC, it must be the case that we cannot add any other vertex from G to the SCC and still maintain the SCC relation.
- How many SCCs are in the graph below?

$$\begin{tikzcd}
 & A \arrow[dr] \arrow[ddl] \arrow[ddr] & & D \arrow[ll] \\
 & & B \arrow[ur] & \\
F \arrow[uur, bend left] \arrow[rr] & & C &
\end{tikzcd}$$

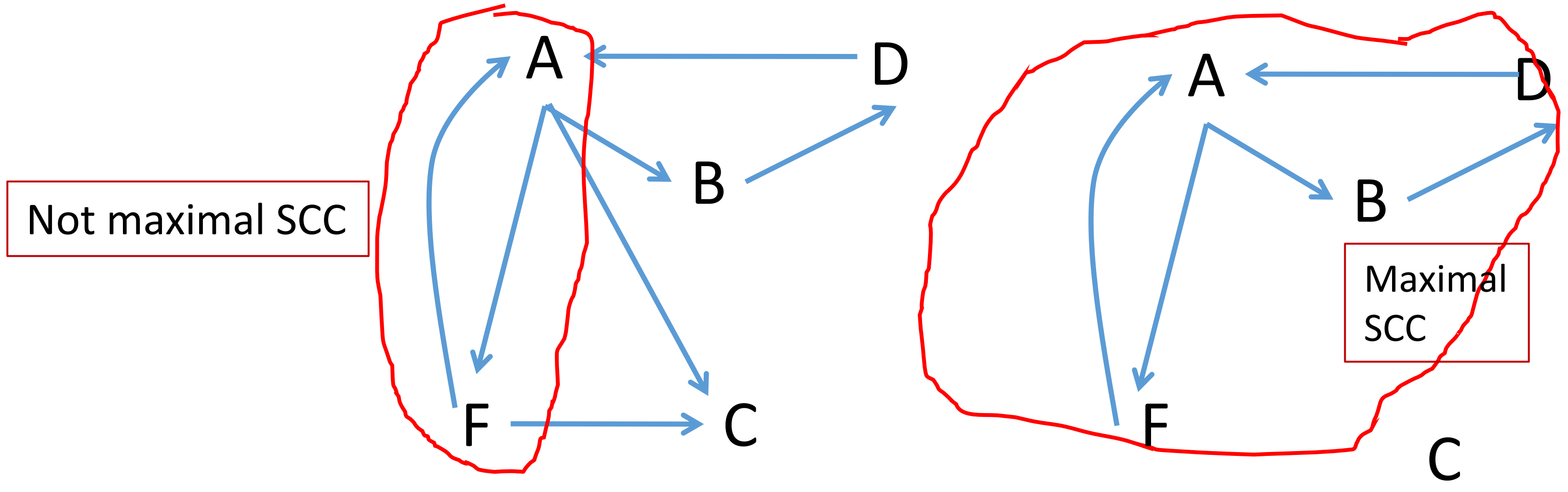
Directed Graphs and Connectivity

- How many SCCs are in the graph below?
- **Answer:** 2
  1. SCC #1: A-B-D-F
  2. SCC #2: C


$$\begin{tikzcd}
A \arrow[dr] \arrow[ddr] \arrow[dd] & & D \arrow[ll] \\
& B \arrow[ur] & \\
F \arrow[uu, bend left] \arrow[r] & C &
\end{tikzcd}$$

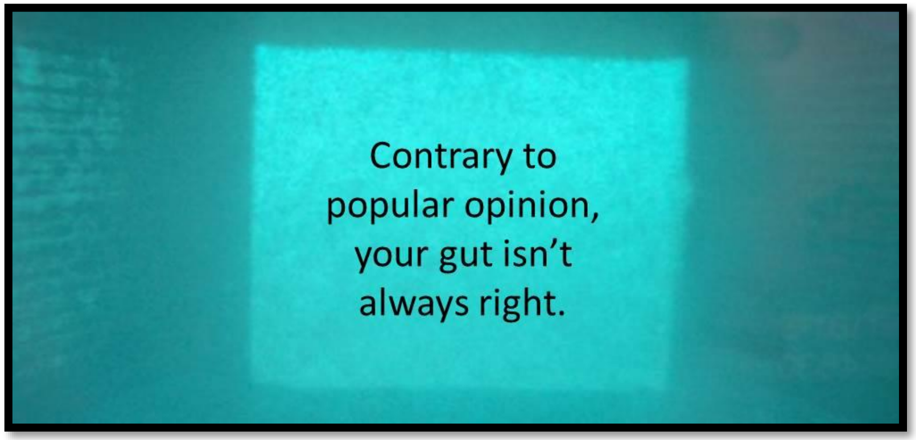
Directed Graphs and Connectivity

- Even though the vertices A, F below satisfy the definition of being strongly connected, it is not a strongly connected component.
- **Why?** The equivalence class is not **maximal** because we can add vertices B, D to the relation and still satisfy the definition (CS 2212 to the rescue!).



Directed Graphs and SCCs

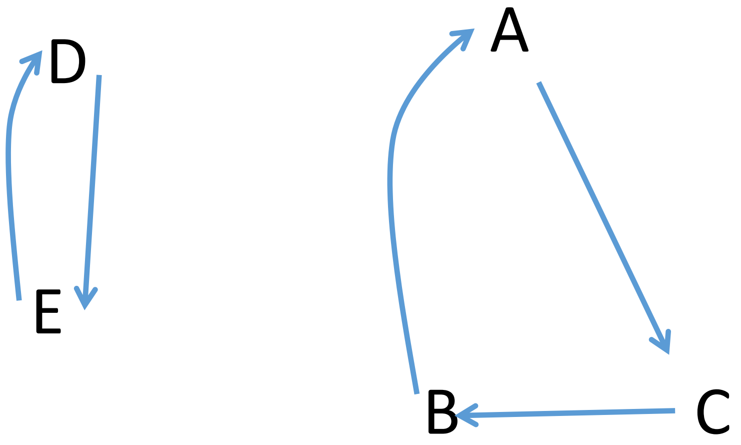
- We previously looked at graph connectivity for an **undirected** graph and saw that we could modify BFS to easily determine connected components using our wrapper or controller/manager loop.
- Our first instinct might be to try a similar approach for a directed graph (i.e., use DFS with a wrapper loop to determine the SCCs).



Contrary to popular opinion, your gut isn't always right.

Directed Graphs and SCCs

- **Intuition:** Look below and try it out.
- If we run a DFS on vertex A, we correctly identify the SCCs {A, B, C} and {D, E}. Maybe our instinct was right? DFS got the job done.

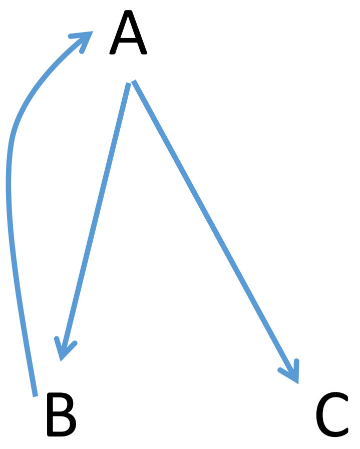


Directed Graphs and SCCs

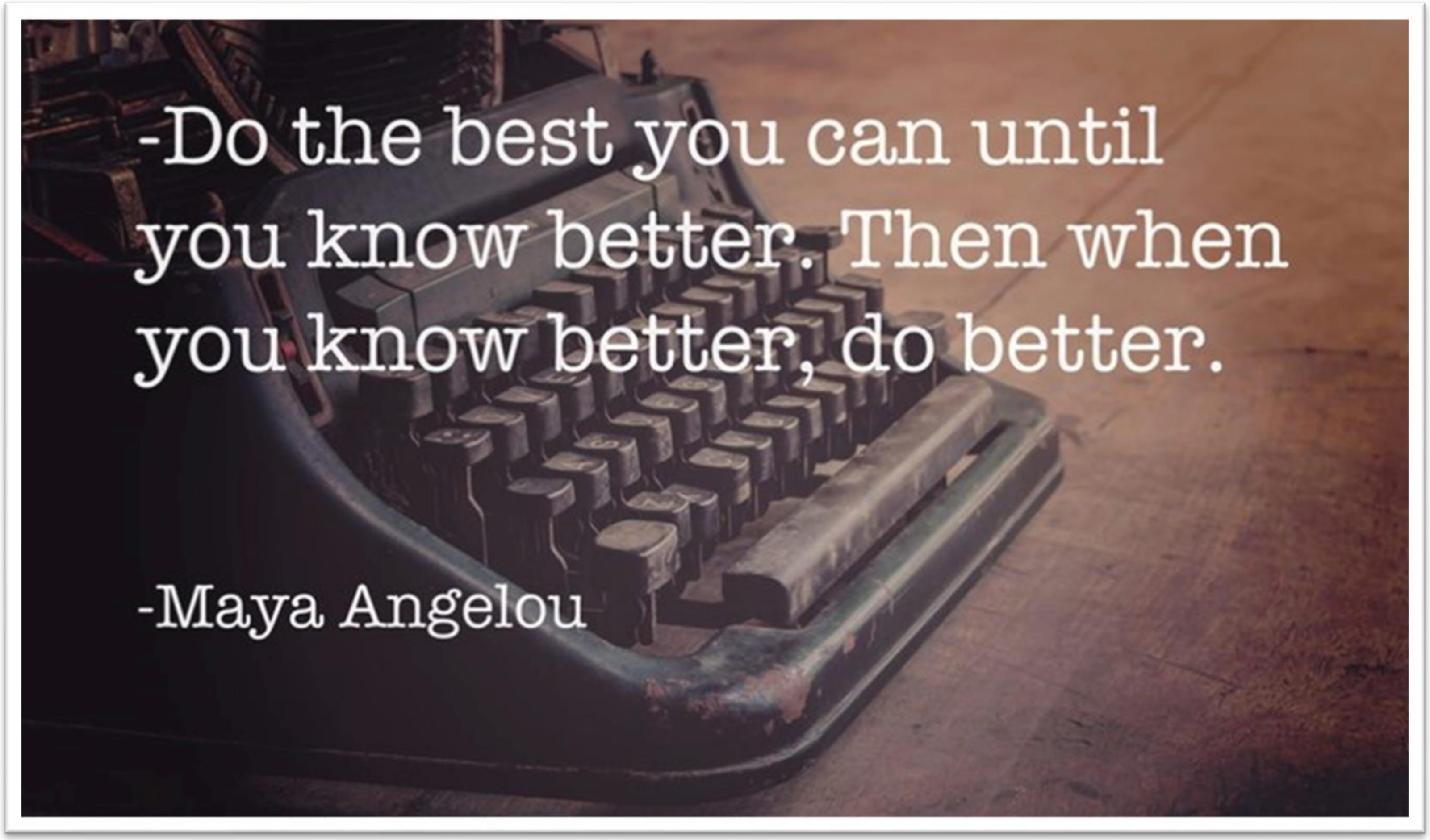
- **Question:** Can you devise a **directed** graph G that's a counterexample and demonstrates why a simple DFS with a wrapper loop fails trying to locate all SCCs of a graph?

Directed Graphs and SCCs

- The strongly connect components in the graph below are {A, B} and {C}.
- However, if doing a DFS starting at B, we will conclude that the SCC is {B, A, C}. This is not the case since C has no way of getting back to A.
- Conclusion: DFS alone may not correctly locate all SCCs.

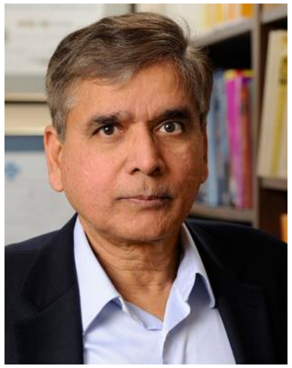

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & \\
B \arrow[ur, bend left] & & C
\end{tikzcd}$$

Directed Graphs and SCCs


-Do the best you can until you know better. Then when you know better, do better.
-Maya Angelou

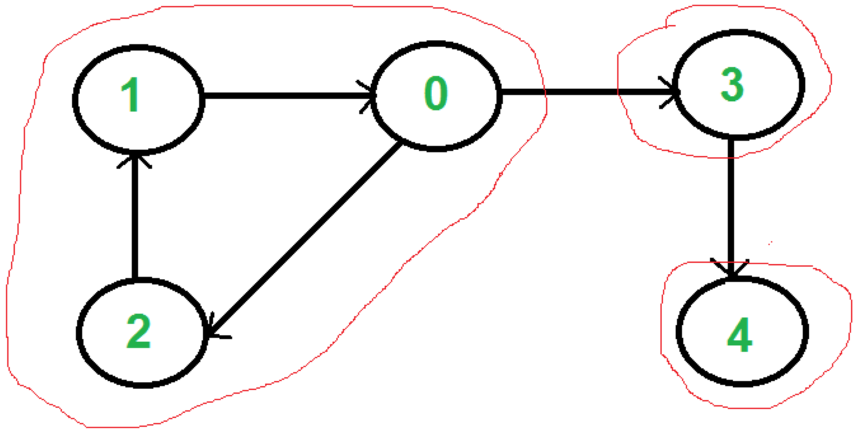
Directed Graphs and SCCs

- **Enter Rao "Magic" Kosaraju.** Kosaraju's Algorithm is an amazingly subtle algorithm based on DFS that computes the SCCs in a directed graph G.
- Here's how it works…
  1. Run DFS on G.
  2. Compute the transpose or reverse of G, namely G^{rev}. This is the graph G with all arcs reversed.
  3. Run DFS on G^{rev}. That's it!
  4. Total = $\Theta(V+E)$



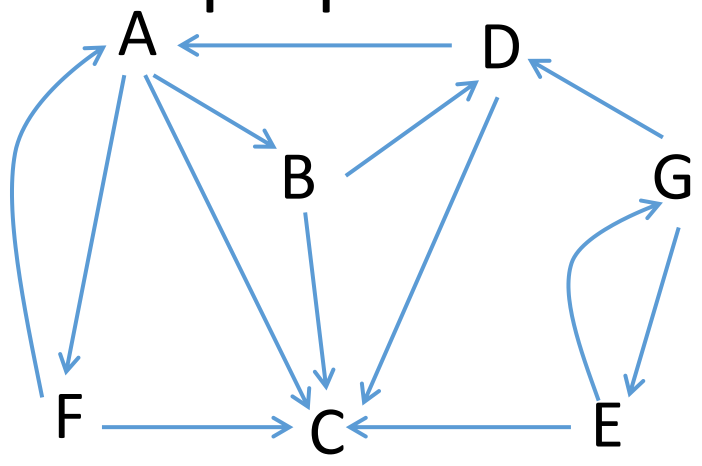
Kosaraju's Algorithm and SCCs

- Does this little bit of magic using DFS really work?
- Yes! How does it work?
  - The first pass of the DFS on G determines the order in which to visit the vertices for the SCCs.
  - The second pass follows that ordering using DFS to determine the vertices in each SCC.



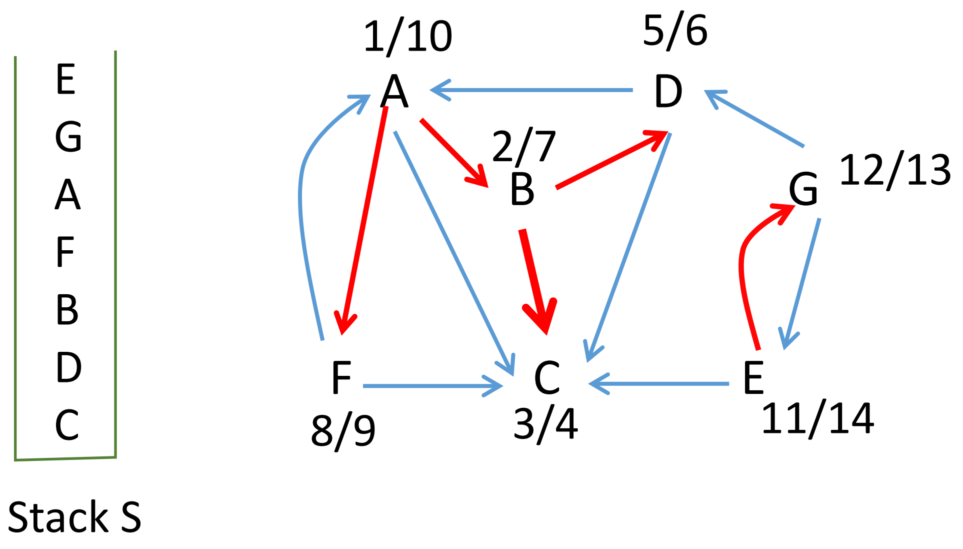
Example: Kosaraju's Algorithm for SCCs

- Let's do a walkthrough of Kosaraju's algorithm on the DFS below.
- **Step 1:** Perform DFS starting at A. For practice, let's note the entry/exit time for each vertex using 1 as our start time. Use lexicographic ordering.
  1. When a vertex is **finished** processing push it onto a stack S in preparation for the second DFS pass.



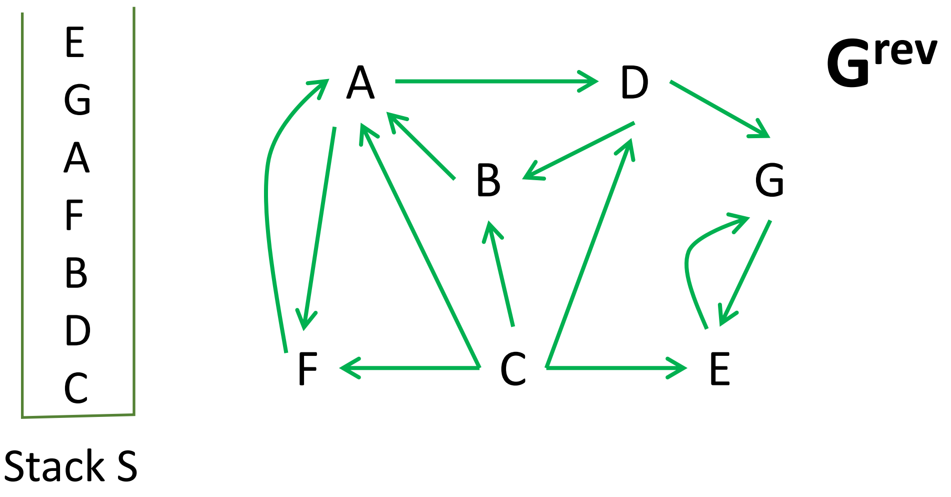
Example: Kosaraju's Algorithm for SCCs

- After Step 1, we have the following (tree edges are denoted in red):



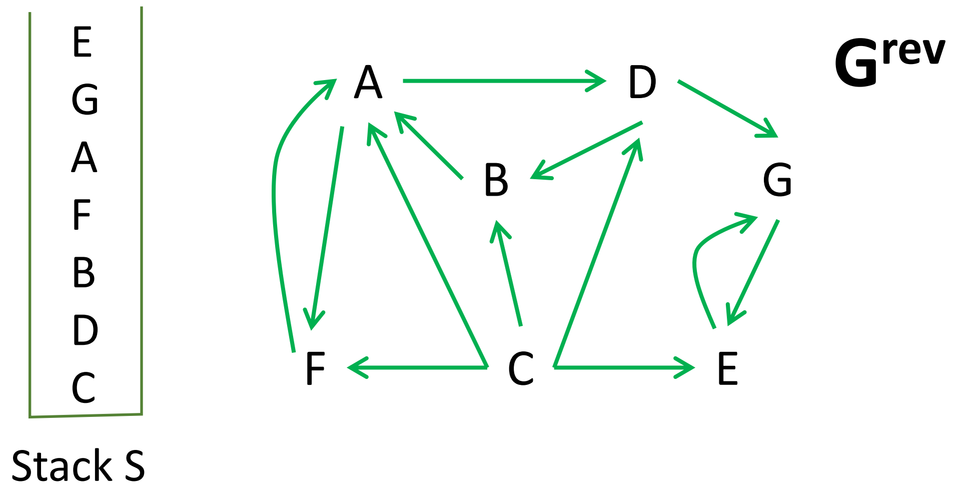
Kosaraju’s Algorithm and SCCs

- **Step 2:**
  - Compute the transpose/reverse of the graph G, call it G^T or G^{rev}
  - I’ll leave the details of “how” to do this to you. Reversing the pointers takes Θ(V+E) time. This part was a recent tech interview question for Yahoo.



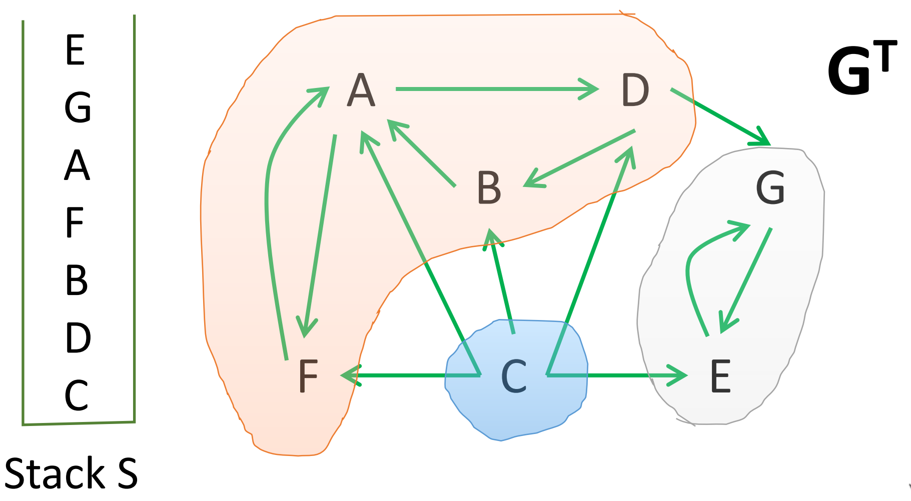
Kosaraju’s Algorithm and SCCs

- **Step 3:**
  - Run a new DFS on G^{rev} using any **unvisited** items on the stack to find the SCCs.
  - The vertices discovered during each DFS of a stack item belong to the same SCC.



Kosaraju's Algorithm and SCCs

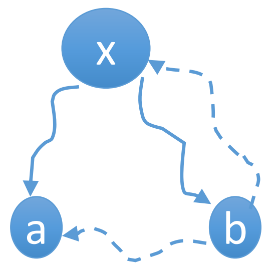
- After Step 3, we have the following:
  - The three SCCs have been correctly determined.



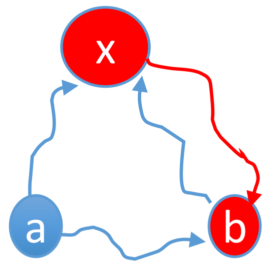
Kosaraju’s Algorithm: Why Does it Work?

- **Why does Kosaraju’s work?** The problem with determining the SCCs via straight DFS is that there is no control as to how we explore the graph.
- In a DFS forest of a graph G, several SCC’s may be in one DFS tree.
- With the transpose G^{rev}, each tree in the DFS forest obtained from G^{rev} contains exactly one SCC.

Original graph G

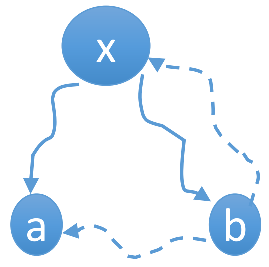


Transpose graph G_T

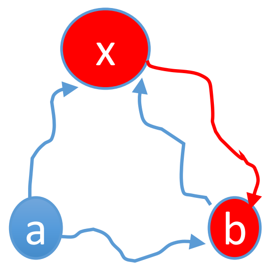


Kosaraju's Algorithm: Why Does it Work?

- In Kosaraju's, the order is enforced from our first pass using DFS. Then, each unvisited vertex that is popped off the stack becomes the root of an SCC.
  - Everything root x can reach in G^{rev} is in one SCC.
  - x was placed on the stack when finished in the original graph G.
  - In G, any vertex x can reach is below it on the stack because it finished before x.



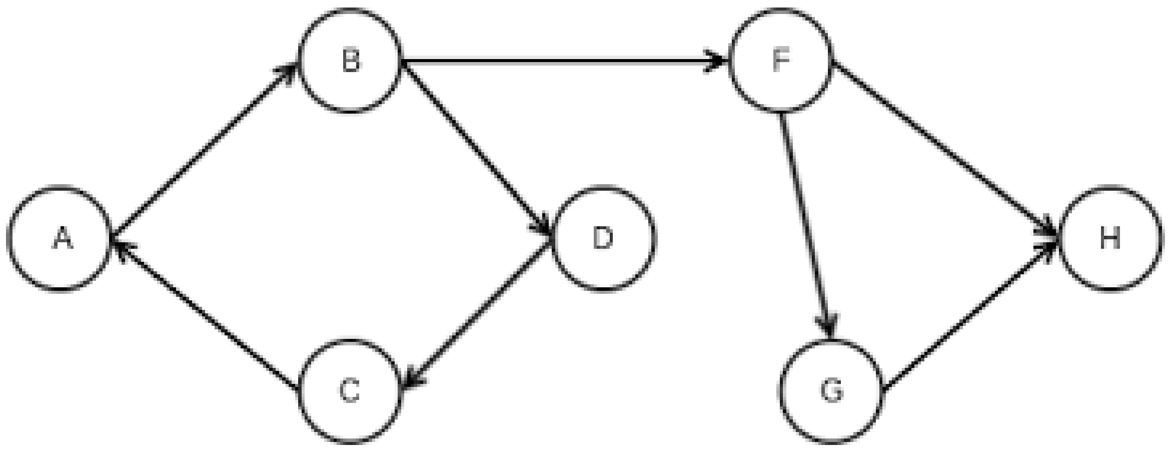
Original graph G



Transpose graph G_T

Practice: Kosaraju’s Algorithm and SCCs

- **On Your Own:** Perform Kosaraju’s Algorithm on the graph below and identify the SCCs of the graph. Be sure to indicate G, the stack items, G^{rev} and the SCCs.



That’s All For Now…

- Coming to a Slideshow Near You Soon…
  1. Prim’s algorithm analysis.
  2. Proof of Prim’s algorithm.

That’s All For Now

VANDERBILT UNIVERSITY