

香港中文大學(深圳)

The Chinese University of Hong Kong

CSC3100 Data Structures

Lecture 21: DAG and SCC computation

Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen



Outline

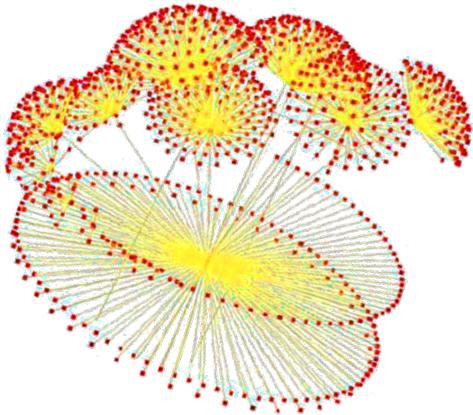
- ▶ We focus on directed graphs in this lecture
- ▶ Directed acyclic graph (DAG) checking
 - What is DAG?
 - How to check the existence of DAGs?
- ▶ Strongly connected component (SCC) computation
 - What is SCC?
 - How to detect SCCs?



Directed graphs are everywhere



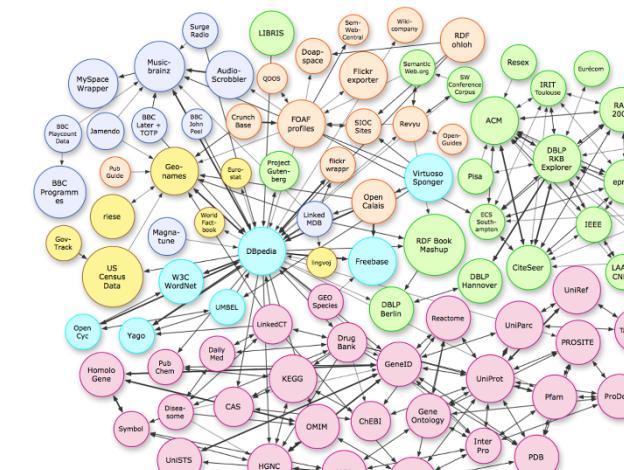
Social networks



Internet networks



DBLP co-author network



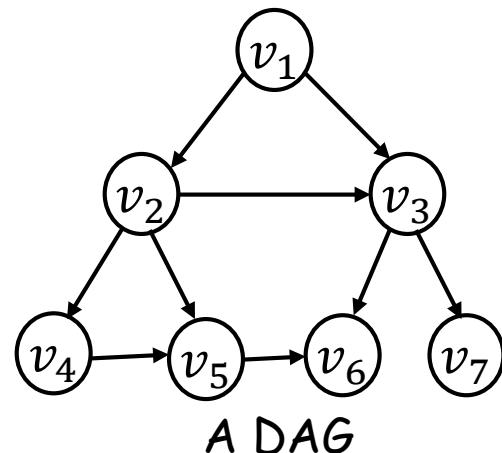
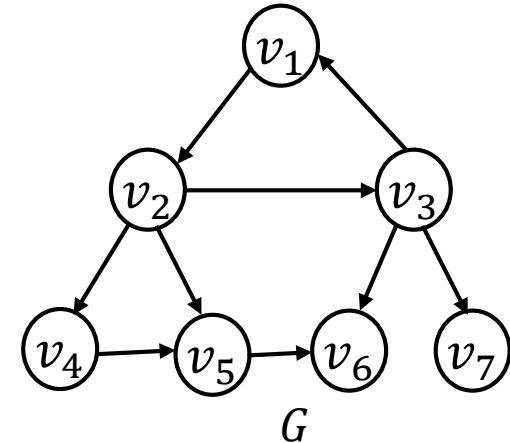
Knowledge graphs

As of March 2009



Directed Acyclic Graph (DAG)

- ▶ Cycle: A simple path that starts and ends at the same node
 - In directed graph G
 - Path $P = (v_1, v_2, v_3, v_1)$ is a cycle
- ▶ Directed acyclic graph (DAG)
 - A directed graph that contains no cycles





DAG checking: using DFS

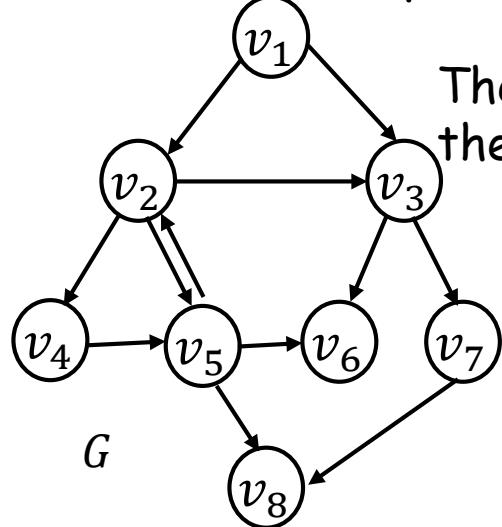
- ▶ Doing DFS on the entire graph G
 - The DFS we learned has an input source s

- ▶ To apply to the entire graph:
 - Randomly generate a permutation of the nodes and repeat the following until there is no white node
 - Pick the first white node s in the permutation and do DFS (during DFS, we will color nodes, and record timestamps)

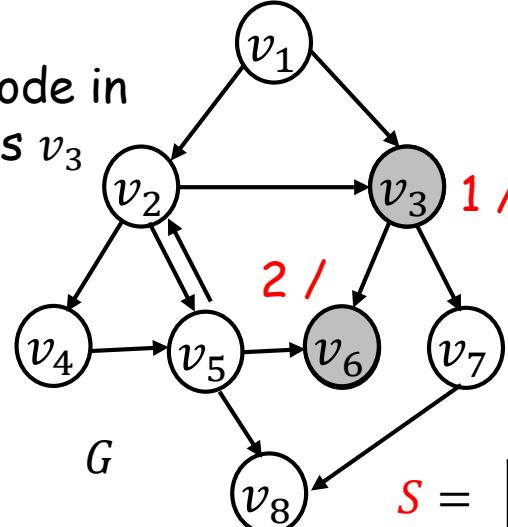


A running example

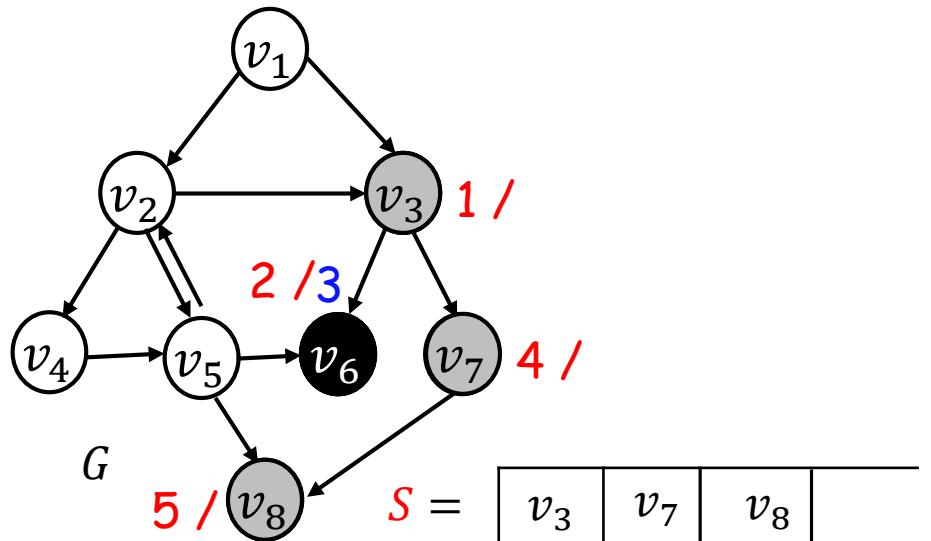
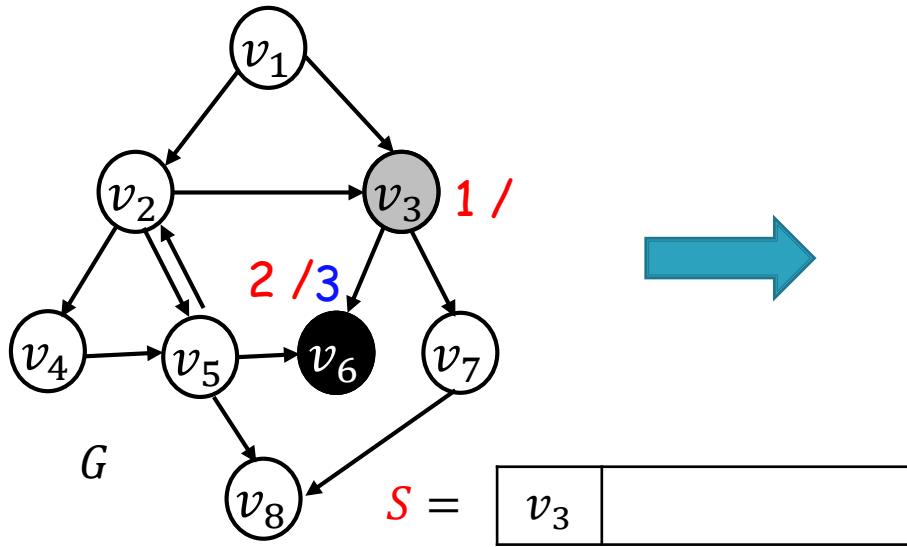
Assume that the permutation is $(v_3, v_7, v_1, v_6, v_4, v_5, v_8, v_2)$



The first white node in
the permutation is v_3



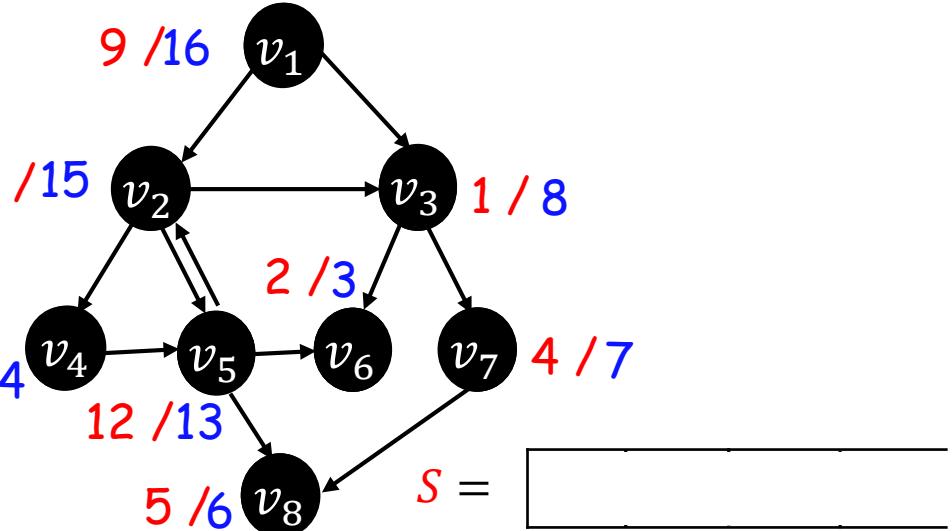
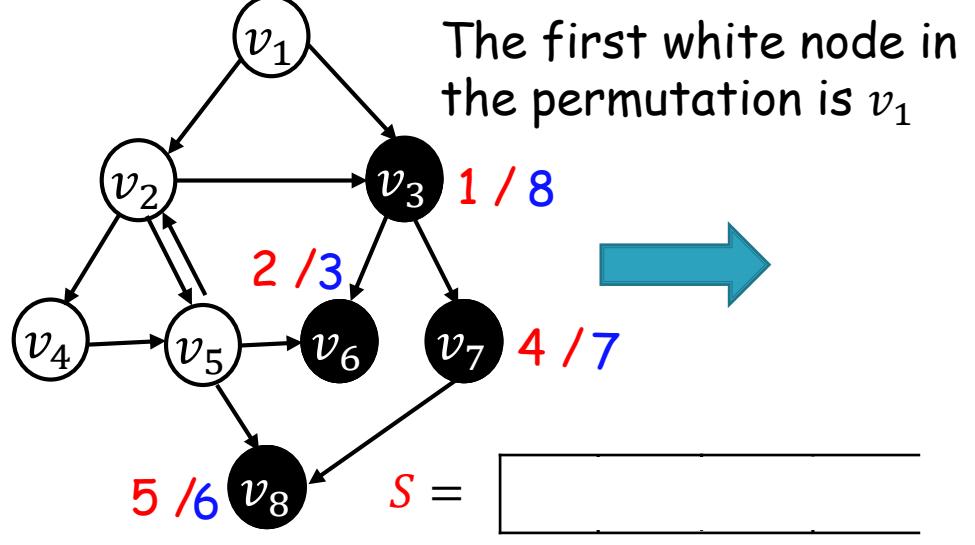
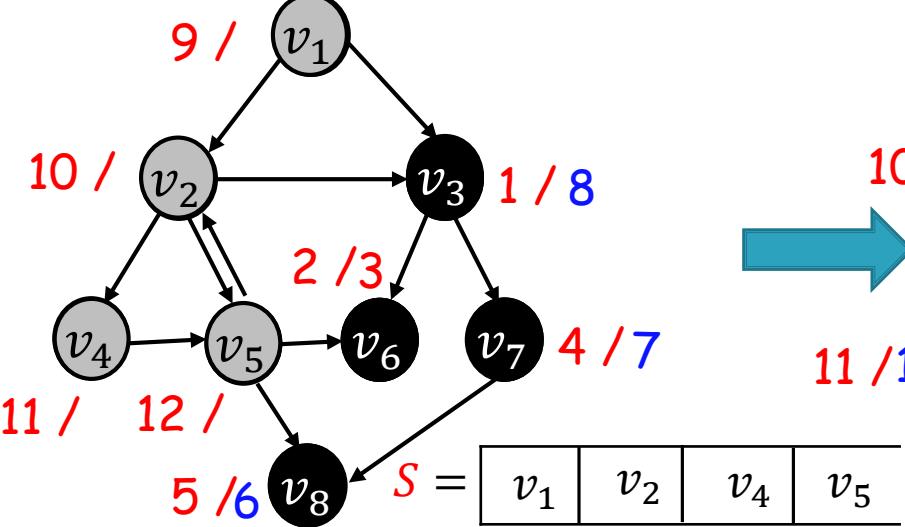
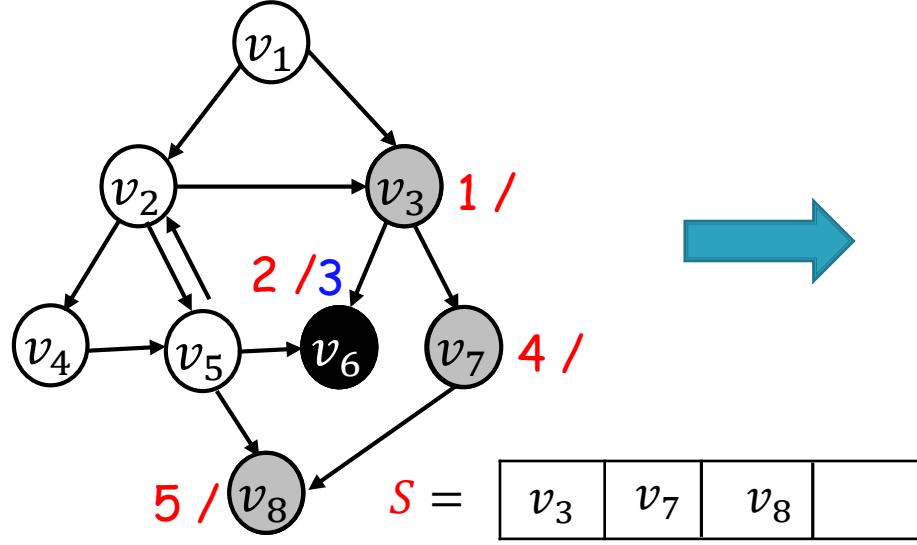
$$S = \boxed{v_3 \quad v_6}$$





A running example

Assume the permutation is $(v_3, v_7, v_1, v_6, v_4, v_5, v_8, v_2)$

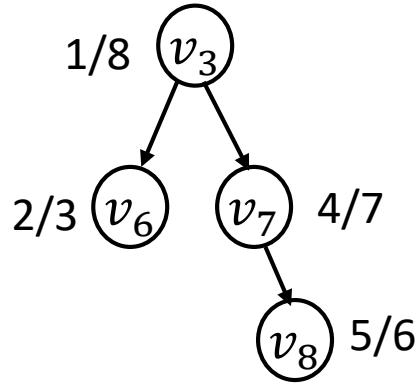


The first white node in the permutation is v_1

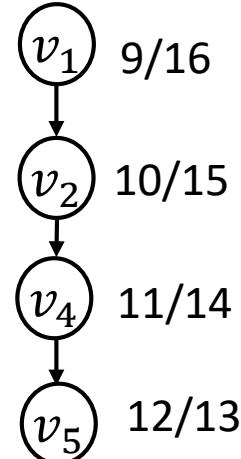


Edge classifications (i)

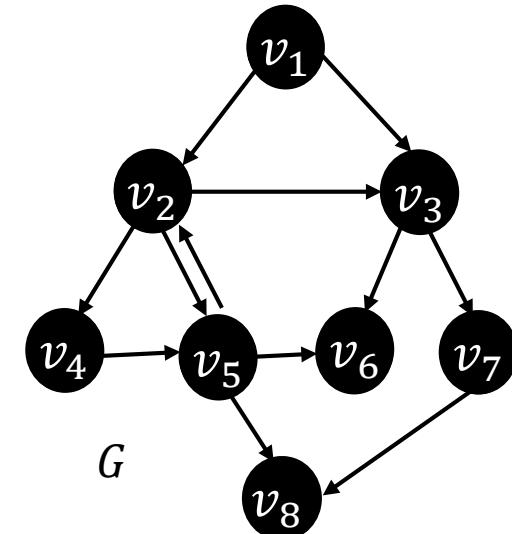
► Results of the DFS-trees on graph G



DFS-Tree
rooted at v_3



DFS-Tree
rooted at v_1

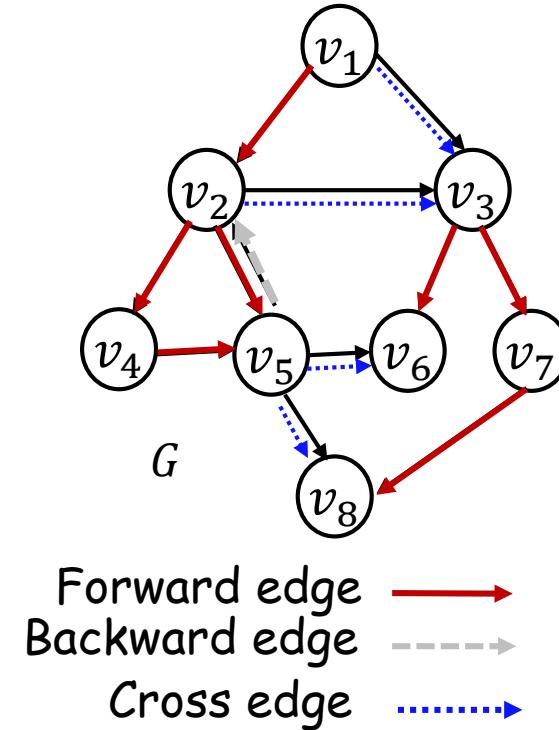
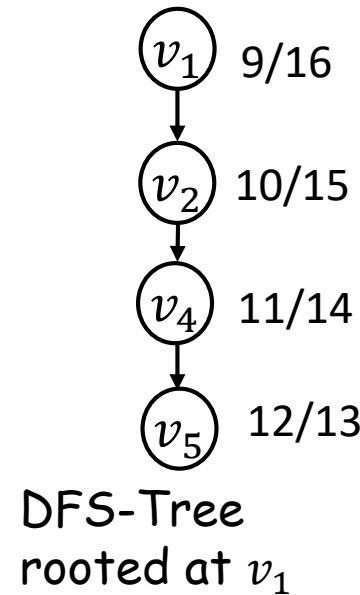
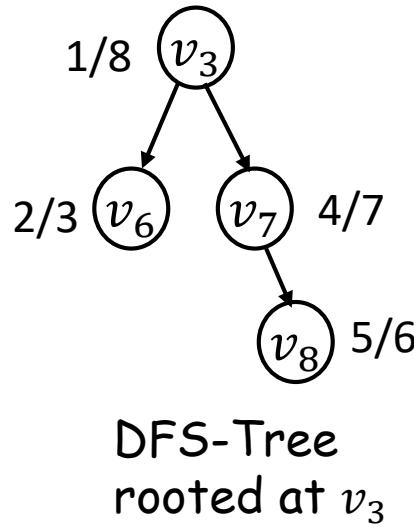


- v_3 is an ancestor of v_8 in the DFS tree rooted at v_3
- v_5 is a descendant of v_1 in the DFS tree rooted at v_1
- Neither v_1 or v_3 is the descendant of the other



Edge classifications (ii)

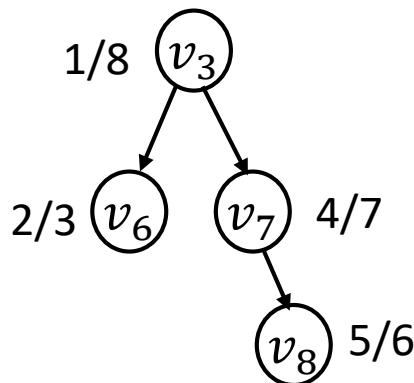
- Assume we have done DFS on graph G . Let $\langle u, v \rangle$ be an edge in G . It can be classified into three types:
 - Forward edge: if u is an ancestor of v in one of the DFS-trees
 - Backward edge: if u is a descendant of v in one of the DFS-trees
 - Cross edge: if none of the above happens



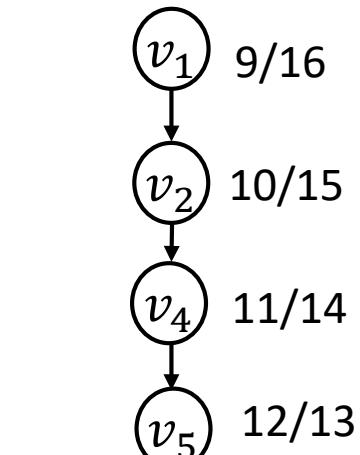


Recap: interval property

- ▶ Interval $I(u)$ of node u is $[u.d, u.f]$, where $u.d$ is the **first discovery time** and $u.f$ is the **finish time**
 - We will only have three cases for two nodes u and v
 - $I(u) \subset I(v)$, u is the descendant of v
 - $I(v) \subset I(u)$, v is the descendant of u
 - $I(u) \cap I(v) = \emptyset$, neither one is the descendant of the other.



DFS-Tree
rooted at v_3



DFS-Tree
rooted at v_1

$I(v_5) \subset I(v_1)$: v_5 is the descendant of v_1

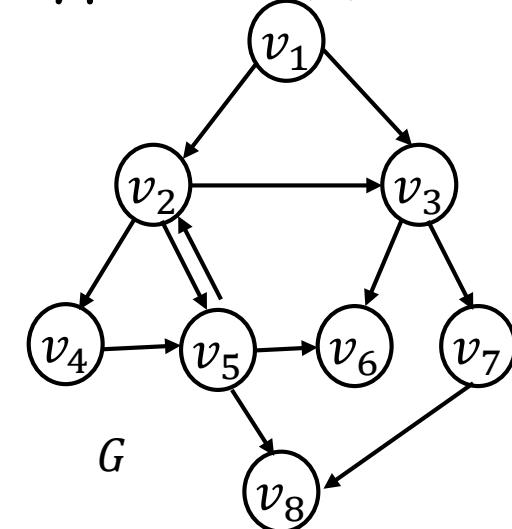
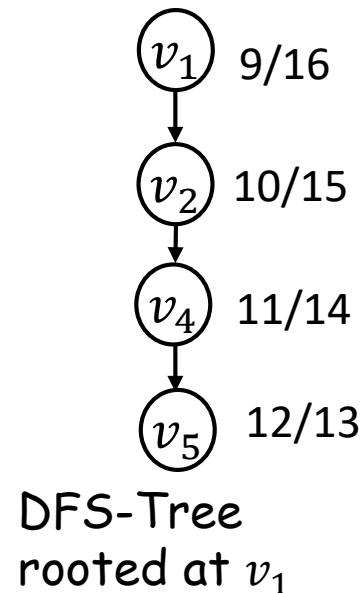
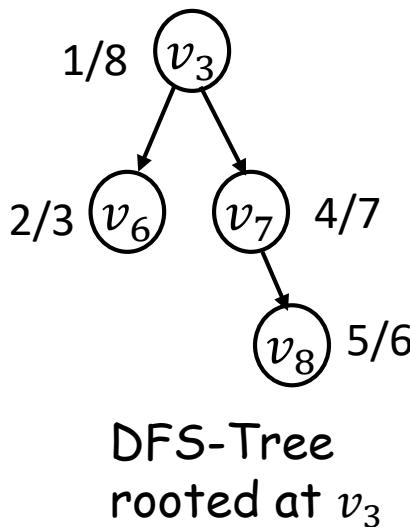
$I(v_6) \cap I(v_7) = \emptyset$: neither one is the descendant of the other

How about v_3 and v_8 ?
How about v_3 and v_1 ?



Cost for edge classifications

- For an edge $\langle u, v \rangle$, we can check edge type in $O(1)$ time given the interval information
 - $I(u) \subset I(v)$, backward edge
 - $I(v) \subset I(u)$, forward edge
 - $I(u) \cap I(v) = \emptyset$, cross edge



$\langle v_2, v_3 \rangle: I(v_2) = [10, 15],$
 $I(v_3) = [1, 8]$
 $I(v_2) \cap I(v_3) = \emptyset$. Cross edge

How about $\langle v_2, v_5 \rangle$ and $\langle v_5, v_2 \rangle$?



Cycle theorem

Theorem 1: Given the DFS result on graph G , then G contains a cycle **if and only if** there is a **backward edge** in the DFS result on G .

Proof: (i) there is a backward edge $\langle u, v \rangle$, then G contains a cycle. This part can be proved according to the definition and will be left as exercise.

(ii) Prove that if there is a cycle, then there will exist a backward edge. Assume that the cycle is $(v_1, v_2, v_3, \dots, v_l, v_1)$. Then actually, we know path $(v_2, v_3, \dots, v_l, v_1, v_2)$ is also a cycle, and so on for the other paths starting from v_3, v_4, \dots, v_l .

Assume that v_i is the first node to be pushed onto the stack when doing DFS from a source s . Then, since there is a path from v_i to any other nodes $v_1, v_2, \dots, v_{i-1}, v_{i+1}, \dots, v_l$, all these nodes will be visited during this DFS traversal with source s , and will be descendant of v_i .

Therefore, we have an edge $\langle v_{i-1}, v_i \rangle$, and v_{i-1} is an descendant of v_i , which is a backward edge according to the definition. Proof done.



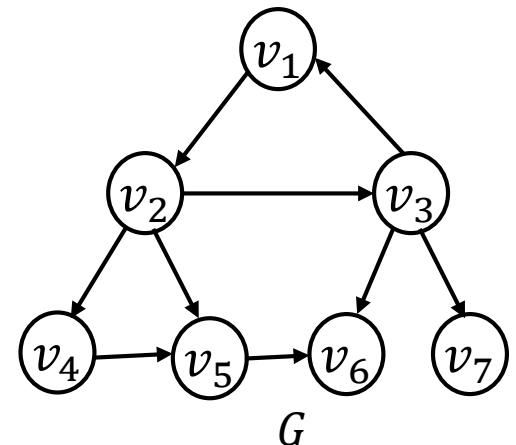
Cycle detection: putting it all together

- ▶ Step 1: Do DFS traversal on graph G
 - Time complexity: $O(n + m)$ (permutation can be done in $O(n)$)
- ▶ Step 2: Classify edges according to the interval of each node derived with DFS
 - Time complexity: $O(m)$
- ▶ Step 3: If there exists a backward edge, G contains a cycle, otherwise, G is a DAG
- ▶ Total time complexity: $O(n + m)$



Practice

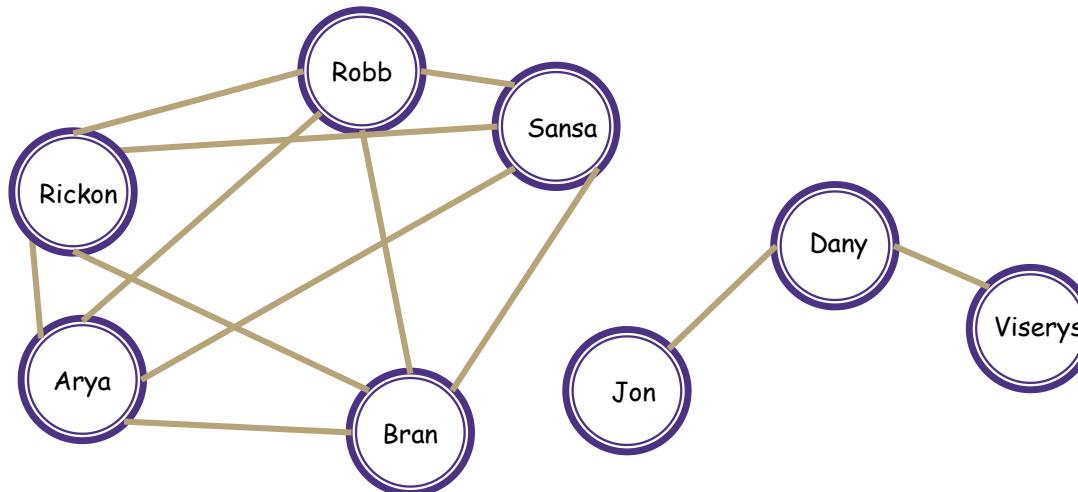
- Given a graph G , and assume that the permutation generated for the nodes is: $(v_3, v_2, v_4, v_5, v_7, v_6, v_1)$
 - Verify if the graph is a DAG by using DFS step by step
 - In your solution, you should explicitly output the type of each edge





Connected [Undirected] Graphs

- ▶ Connected graph - a graph where every vertex is connected to every other vertex via some path
 - It is not required for every vertex to have an edge to every other vertex
 - There exists some way to get from each vertex to every other vertex
- ▶ Connected Component - a subgraph in which any two vertices are connected via some path, but is connected to no additional vertices in the supergraph
 - ▶ A vertex with no edges is itself a connected component

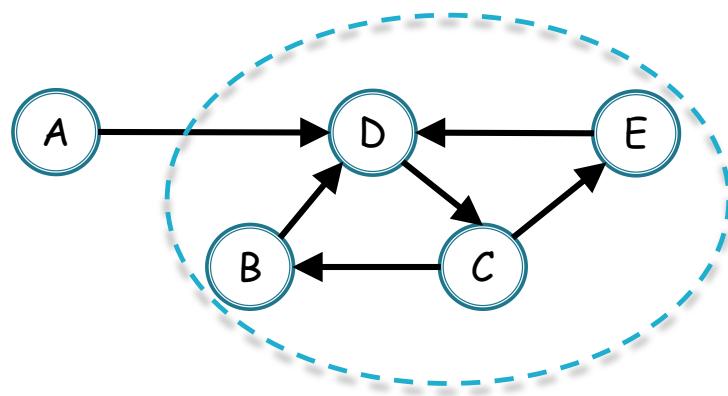




Strongly Connected Component (SCC)

Strongly Connected Component (SCC)

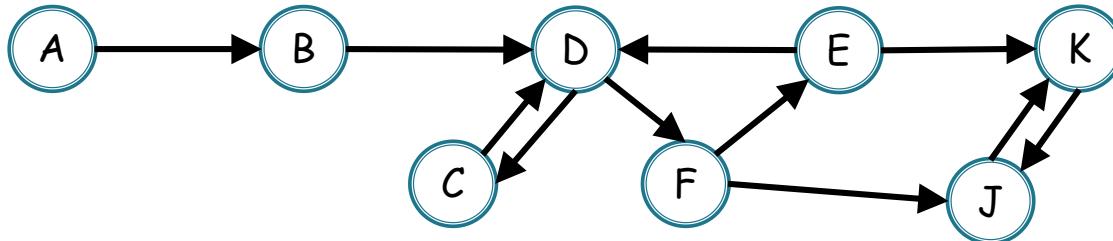
A subgraph C such that every pair of vertices in C is connected via some path in **both directions**, and there is no other vertex which is connected to every vertex of C in both directions.



- ▶ Note: the direction of edges matters!



SCC Problem



$\{A\}, \{B\}, \{C,D,E,F\}, \{J,K\}$

Strongly Connected Components Problem

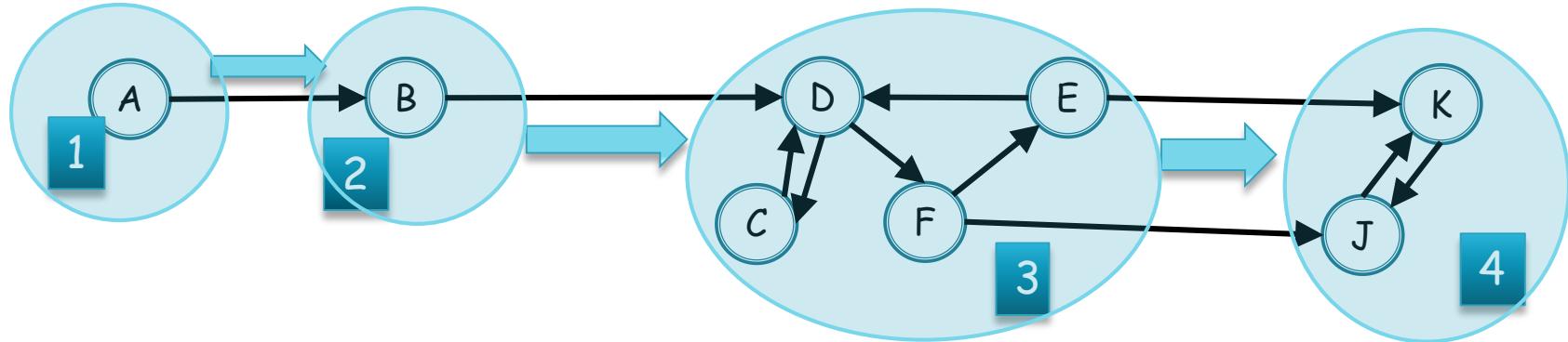
Given: A directed graph G

Find: All the strongly connected components of G



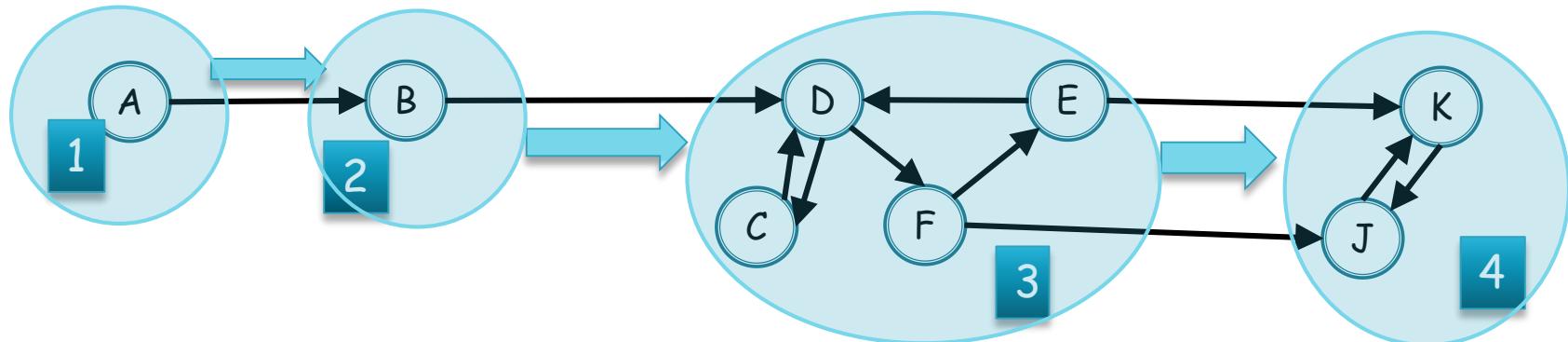
Why Find SCCs?

- Given the SCCs of G , let's build a new graph out of them! Call it H
 - Have a vertex for each of the SCCs
 - Add an edge from component 1 to component 2 if there is an edge from a vertex inside 1 to one inside 2





Why Find SCCs?



- ▶ That's awful meta. Why?
- ▶ This new graph summarizes reachability information of the original graph
 - I can get from A in 1 to F in 3, if and only if I can get from 1 to 3 in H

H is always a DAG (do you see why?)



How to solve the SCC problem?

- ▶ A naïve approach: $O(n^2(n+m))$

For each i, j in nodes:

If i is reachable from j and vice versa

Then i and j are in the same SCC

- ▶ Another approach: $O(n(n+m))$

Array of bool reachable

For each i in nodes:

DFS and put the visited array inside reachable of i

For each i, j in nodes:

If $\text{reachable}[i][j]$ and $\text{reachable}[j][i]$

Then i, j are in the same SCC.



Three algorithms with linear time

- ▶ Kosaraju-Sharir algorithm [1]
 - Run DFS on G , and get a topological order
 - Run DFS on G^T and output SCCs
- ▶ Path-based algorithm [2]
 - A single DFS with sub-path contraction
- ▶ Tarjan's algorithm [3]
 - A single DFS; Each SCC corresponds to a sub-tree

[1] M. Sharir, "A strong-connectivity algorithm and its applications in data flow analysis," Computers & Mathematics with Applications, vol. 7, no. 1, pp. 67-72, 1981.

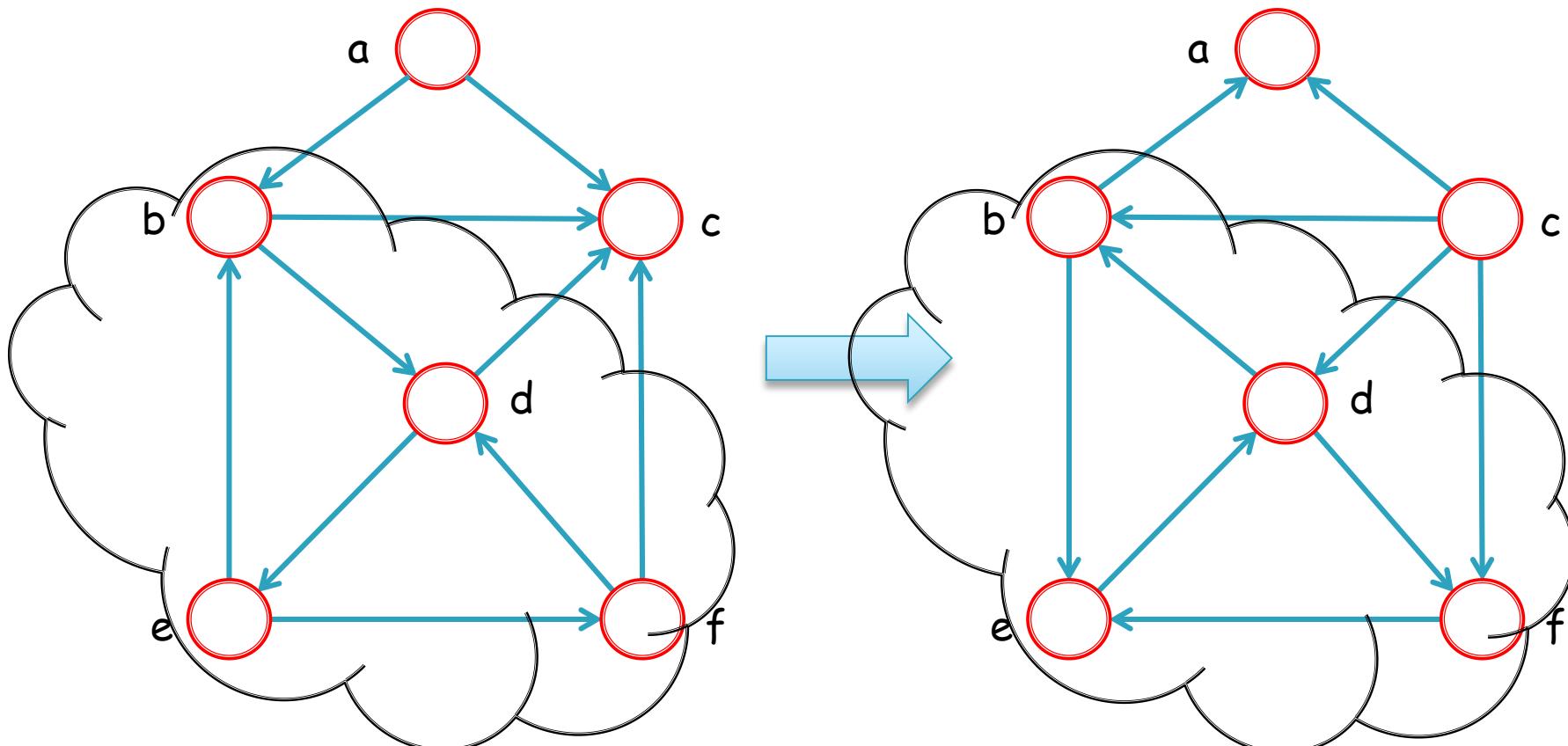
[2] H. N. Gabow, "Path-based depth-first search for strong and biconnected components," Information Processing Letters, vol. 74, no. 3-4, pp. 107-114, 2000.

[3] https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm



Kosaraju-Sharir algorithm

- Fact: the transpose graph (the same graph with the direction of every edge reversed) has exactly the same SCCs as the original graph





Kosaraju-Sharir algorithm

Input: a directed graph $G=(V, E)$

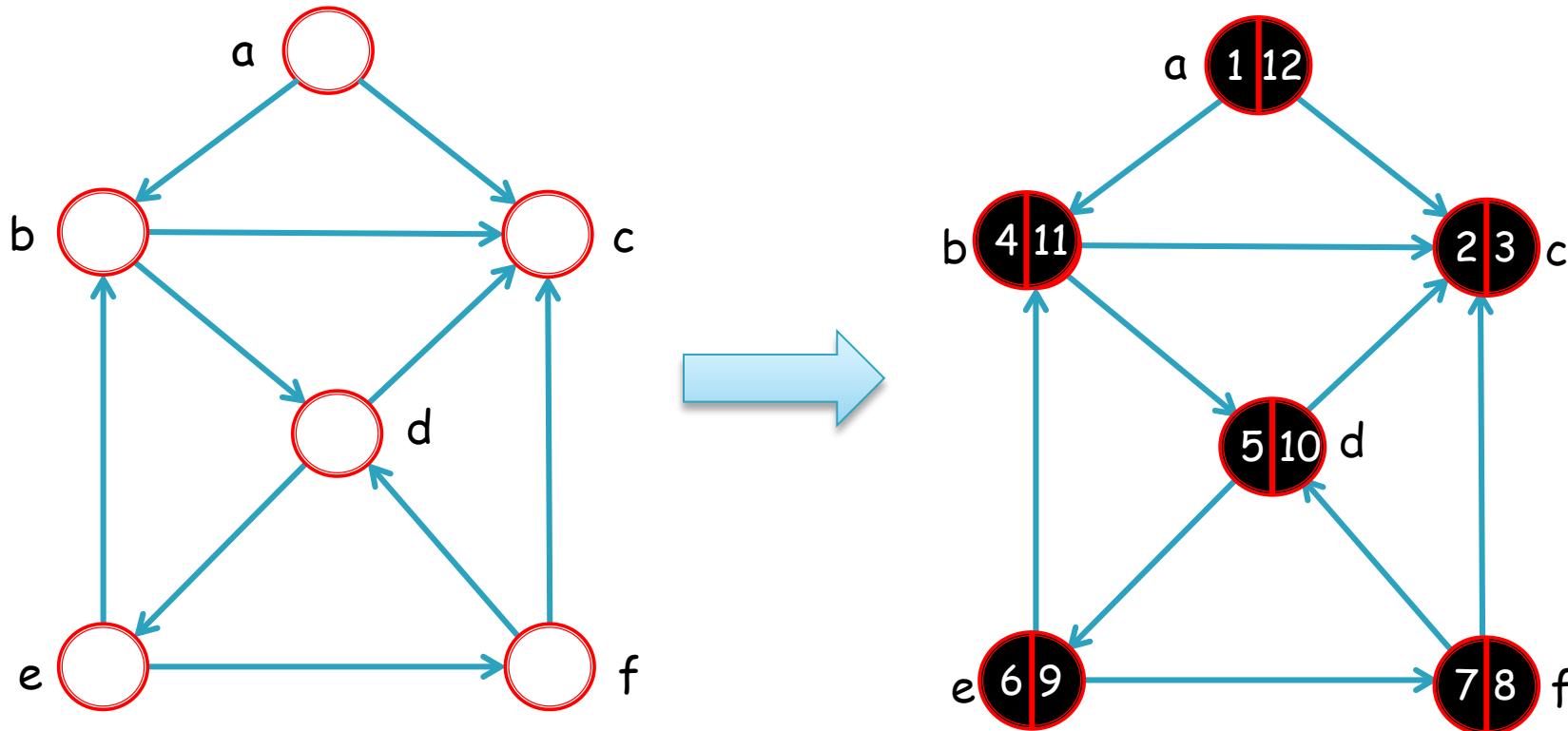
Output: all the *SCCs* of G

1. Run DFS on G , during which we compute the first discovery time and finish time of each vertex
2. Build the transpose graph $G^T=(V^T, E^T)$
3. Run DFS on G^T , by considering the vertices' first discovery time and finish time in descending order
4. Output the vertex set in each DFS traversal as an *SCC*



Kosaraju-Sharir algorithm

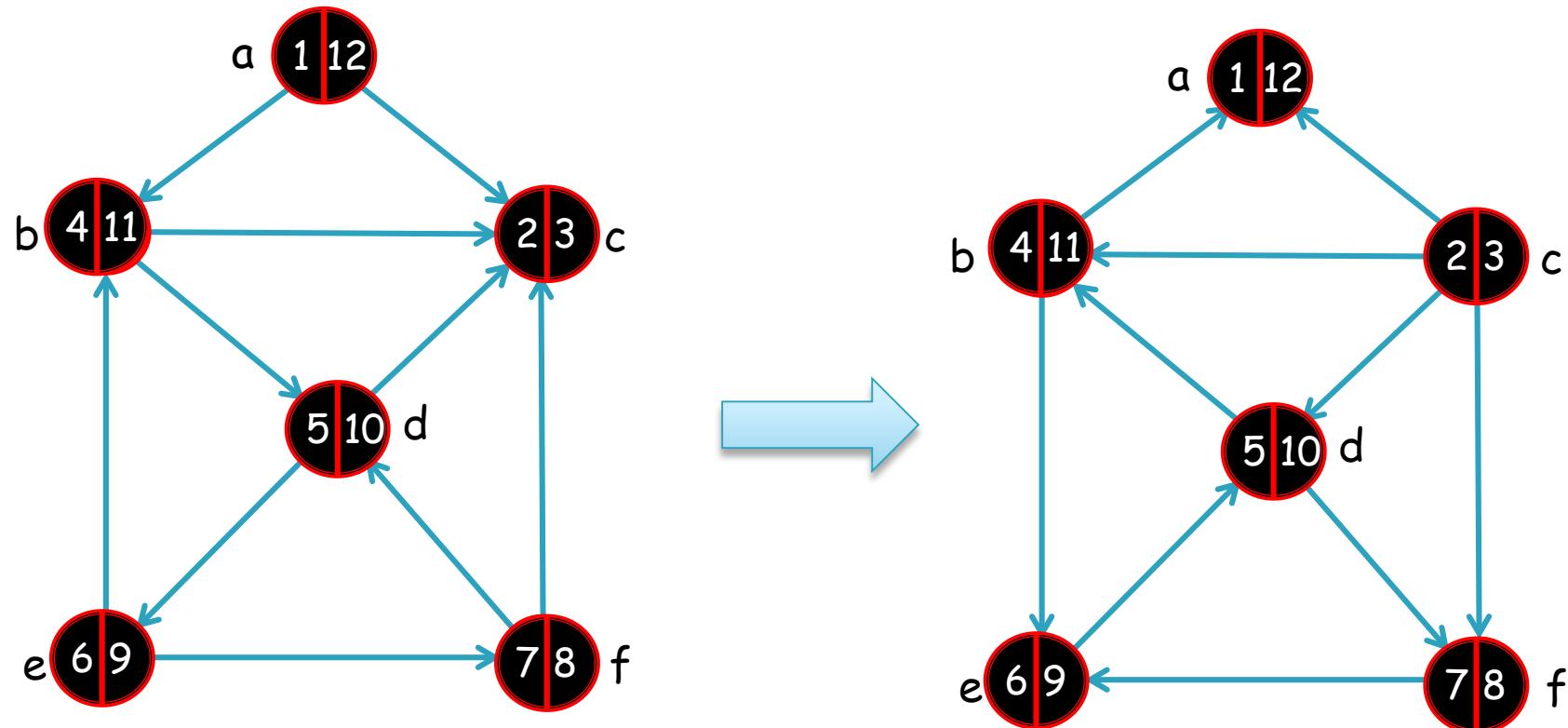
- Run DFS and compute the first discovery time and finishing time of each vertex





Kosaraju-Sharir algorithm

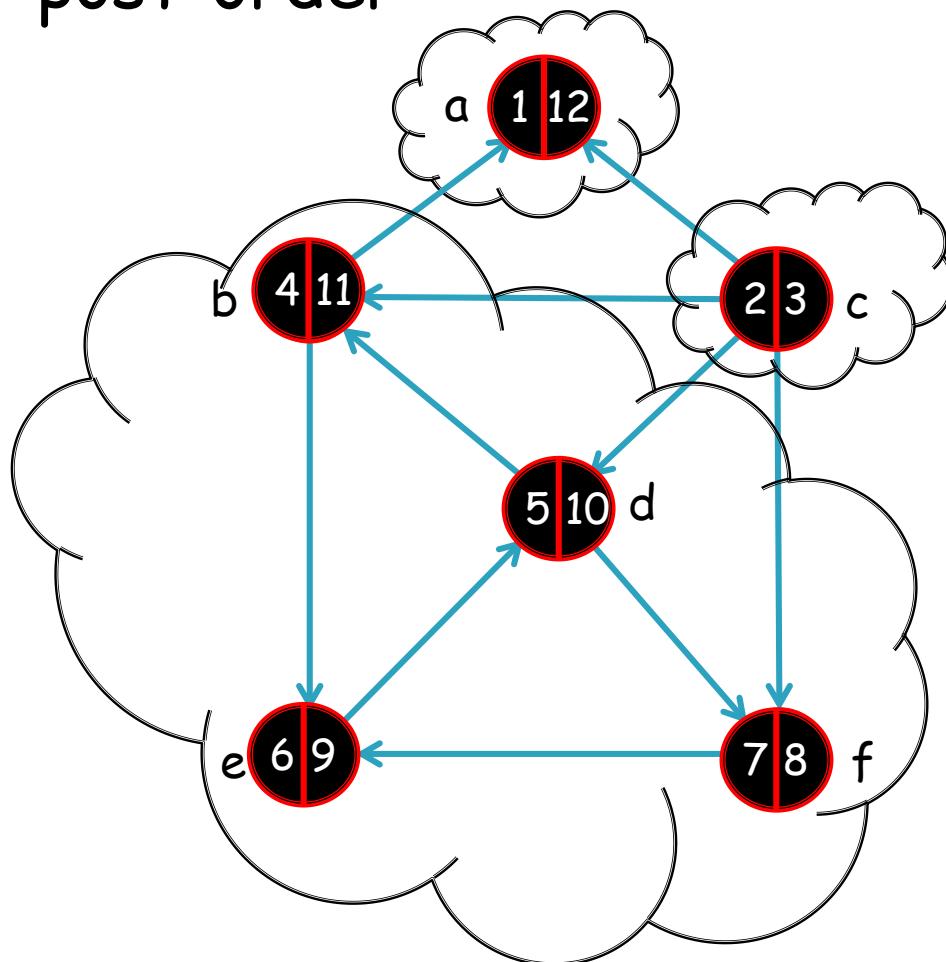
- ▶ Reverse the edge directions





Kosaraju-Sharir algorithm

- Run DFS and consider vertices in the decreasing post-order



SCC list:

{a}

{b, d, e, f}

{c}



Kosaraju-Sharir algorithm

► What's the overall time complexity?

Input: a directed graph $G=(V, E)$

Output: all the SCCs of G

- $O(n+m) \longrightarrow$ 1. Run DFS on G , during which we compute the first discovery time and finish time of each vertex
- $O(m) \longrightarrow$ 2. Build the transpose graph $G^T=(V^T, E^T)$
- $O(n+m) \longrightarrow$ 3. Run DFS on G^T , by considering the vertices' first discovery time and finish time in descending order
- $O(n) \longrightarrow$ 4. Output the vertex set in each DFS traversal as an SCC

The overall time complexity: $O(n+m)$



Recommended reading

- ▶ Reading this week
 - DAG checking and SCC computation, Chapter 22

- ▶ Next lecture
 - Shortest path algorithm, Textbook Chapters 24-25