

# Algorithms 09

## CS201

Kaustuv Nag

# Representation of Graphs

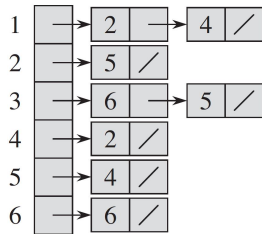
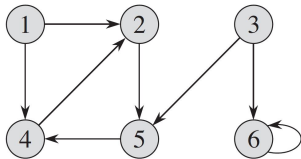
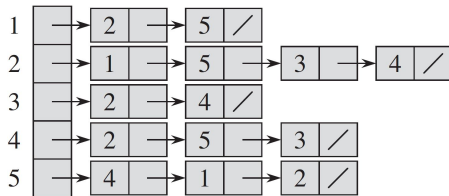
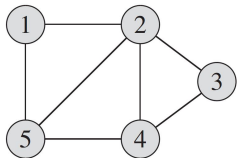
## Two Ways of Representing a Graph $G = (V, E)$

- ▶ **Adjacency-list:** Provides a compact way to represent sparse graphs—those for which  $E$  is much less than  $|V|^2$ .
- ▶ **Adjacency-matrix:** Preferred to represent dense graphs—those for which  $E$  is close to  $|V|^2$  when we need to be able to tell quickly if there is an edge connecting two given vertices.

# Adjacency-List

- ▶ The adjacency-list representation of a graph  $G = (V, E)$  consists of an array  $Adj$  of  $|V|$  lists, one for each vertex in  $V$ .
- ▶ For each  $u \in V$ , the adjacency list  $Adj[u]$  contains all the vertices  $v$  such that there is an edge  $(u, v) \in E$ .
- ▶  $Adj[u]$  consists of all the vertices adjacent to  $u \in G$ . (Alternatively, it may contain pointers to these vertices.)
- ▶ Since the adjacency lists represent the edges of a graph, in pseudocode we treat the array  $Adj$  as an attribute of the graph, just as we treat the edge set  $E$ .
- ▶ If  $G$  is a directed graph, the sum of the lengths of all the adjacency lists is  $|E|$ . If  $G$  is an undirected graph, the sum of the lengths of all the adjacency lists is  $2|E|$ .
- ▶ For both directed and undirected graphs, the adjacency-list representation has the memory requirement of  $\theta(V + E)$ .

# Adjacency-List



# Adjacency-List

- ▶ For both directed and undirected graphs, the adjacency-list representation has the memory requirement of  $\theta(V + E)$ .
- ▶ We can readily adapt adjacency lists to represent weighted graphs, that is, graphs for which each edge has an associated weight, typically given by a weight function  $w : E \rightarrow \mathbb{R}$ .
- ▶ Provides no quicker way to determine whether a given edge  $(u, v)$  present in the graph than to search for  $v$  in the adjacency list  $Adj[u]$ .

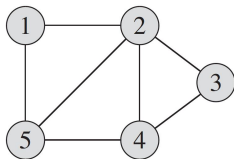
# Adjacency-Matrix

- ▶ We assume that the vertices are numbered  $1, 2, \dots, |V|$  in some arbitrary manner. Then the adjacency-matrix representation of a graph  $G$  consists of a  $|V| \times |V|$  matrix  $A = (a_{ij})$  such that

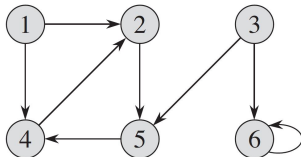
$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

- ▶ The adjacency matrix of a graph requires  $|V|^2$  memory, independent of the number of edges in the graph.
- ▶ Since in an undirected graph,  $(u,v)$  and  $(v,u)$  represent the same edge, the adjacency matrix  $A$  of an undirected graph is its own transpose:  $A = A^T$ .
- ▶ We may store above the diagonal of the adjacency matrix, thereby cutting the memory needed to store the graph almost in half.

# Adjacency-Matrix



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

# Adjacency-Matrix

- ▶ An adjacency matrix can represent a weighted graph.
- ▶ We can store the weight  $w(u, v)$  of the edge  $(u, v) \in E$  as the entry in row  $u$  and column  $v$  of the adjacency matrix. If an edge does not exist, we can store a NIL, 0, or  $\infty$ .



# Breadth-first Search

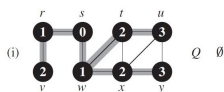
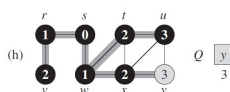
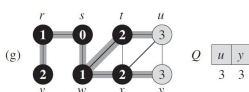
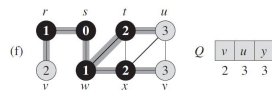
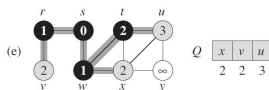
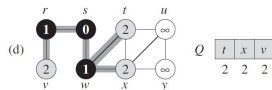
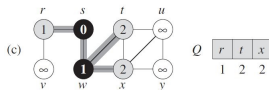
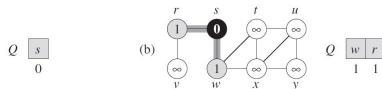
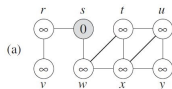
- ▶ Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms.
- ▶ It produces a “breadth-first tree” with root  $s$  that contains all reachable vertices. For any vertex  $v$  reachable from source vertex  $s$ , the simple path in the breadth-first tree from  $s$  to  $v$  corresponds to a “shortest path” from  $s$  to  $v$  in  $G$ , that is, a path containing the smallest number of edges.
- ▶ Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. The algorithm discovers all vertices at distance  $k$  from  $s$  before discovering any vertices at distance  $k + 1$ .

# Breadth-first Search

BFS( $G, s$ )

```

1  for each vertex  $u \in G.V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.Adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE( $Q, v$ )
18      $u.color = \text{BLACK}$ 
    
```



# Breadth-first Search

## Printing Path

```
PRINT-PATH( $G, s, v$ )  
1  if  $v == s$   
2      print  $s$   
3  elseif  $v.\pi == \text{NIL}$   
4      print “no path from”  $s$  “to”  $v$  “exists”  
5  else PRINT-PATH( $G, s, v.\pi$ )  
6      print  $v$ 
```

# Breadth-first Search

## Analysis

- ▶ The overhead for initialization is  $O(|V|)$ .
- ▶ After initialization, breadth-first search never whitens a vertex, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once.
- ▶ The operations of enqueueing and dequeuing take  $O(1)$  time, and so the total time devoted to queue operations is  $O(|V|)$ .
- ▶ Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once.
- ▶ Since the sum of the lengths of all the adjacency lists is  $\Theta(|E|)$ , the total time spent in scanning adjacency lists is  $O(|E|)$ .
- ▶ Thus the total running time of the BFS procedure is  $O(|V| + |E|)$ .

# Depth-first search

- ▶ The strategy followed by depth-first search (DFS) searches “deeper” in the graph whenever possible.
- ▶ DFS explores edges out of the most recently discovered vertex  $v$  that still has unexplored edges leaving it.
- ▶ Once all of  $v$ ’s edges have been explored, the search “backtracks” to explore edges leaving the vertex from which  $v$  was discovered.
- ▶ This process continues until we have discovered all the vertices that are reachable from the original source vertex.
- ▶ If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

# Depth-first search

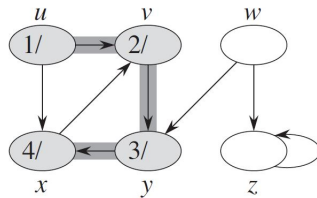
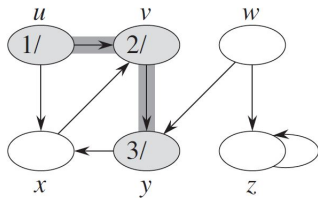
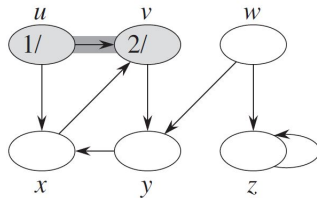
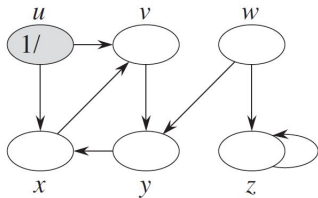
DFS( $G$ )

```
1  for each vertex  $u \in G.V$ 
2       $u.color = \text{WHITE}$ 
3       $u.\pi = \text{NIL}$ 
4   $time = 0$ 
5  for each vertex  $u \in G.V$ 
6      if  $u.color == \text{WHITE}$ 
7          DFS-VISIT( $G, u$ )
```

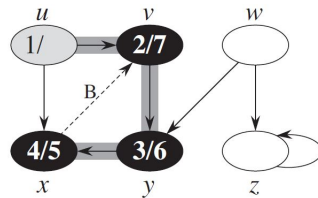
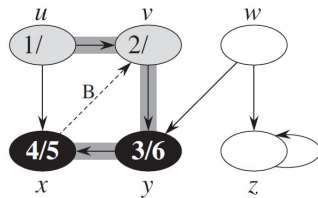
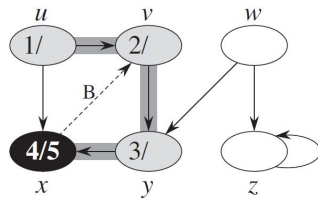
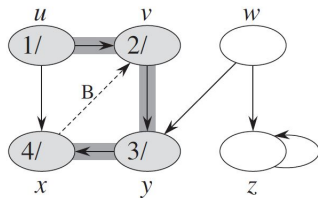
DFS-VISIT( $G, u$ )

```
1   $time = time + 1$ 
2   $u.d = time$ 
3   $u.color = \text{GRAY}$ 
4  for each  $v \in G.Adj[u]$ 
5      if  $v.color == \text{WHITE}$ 
6           $v.\pi = u$ 
7          DFS-VISIT( $G, v$ )
8   $u.color = \text{BLACK}$ 
9   $time = time + 1$ 
10  $u.f = time$ 
```

# Depth-first search

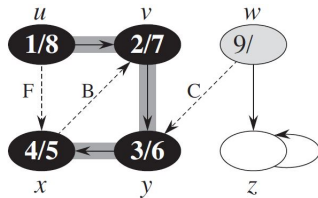
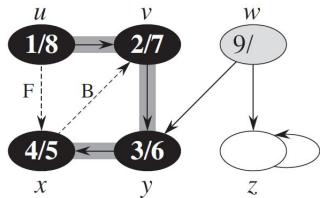
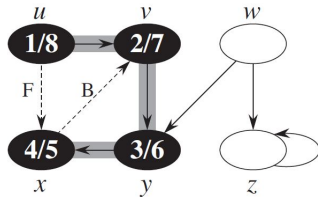
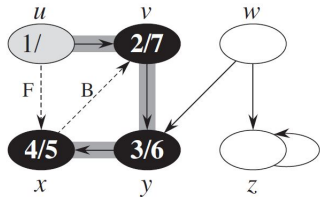


# Depth-first search

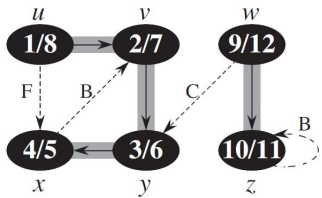
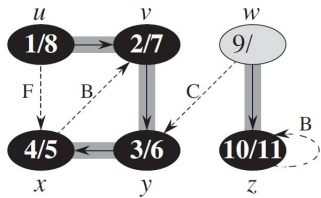
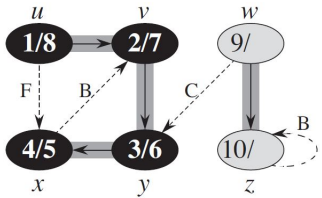
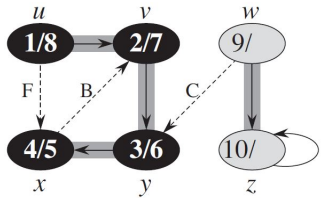




# Depth-first search



# Depth-first search



# Depth-first Search

## Analysis

- ▶ The loops on lines 1–3 and lines 5–7 of DFS take time  $\Theta(|V|)$ , exclusive of the time to execute the calls to DFS-VISIT.
- ▶ The procedure DFS-VISIT is called exactly once for each vertex  $v \in V$ , since the vertex  $u$  on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint vertex  $u$  gray.
- ▶ During an execution of DFS-VISIT  $(G, v)$ , the loop on lines 4–7 executes  $|Adj[v]|$  times.
- ▶ Since

$$\sum_{v \in V} |Adj[v]| = \Theta(|E|)$$

the total cost of executing lines 4–7 of DFS-VISIT is  $\Theta(|E|)$ .

- ▶ The running time of DFS is therefore  $\Theta(|V| + |E|)$ .

# BFS: Lemmas

## Lemma 1

Let  $G = (V, E)$  be a directed or undirected graph, and let  $s \in V$  be an arbitrary vertex. Then, for any edge  $(u, v) \in E$ ,

$$\delta(s, v) \leq \delta(s, u) + 1$$

where  $\delta(s, v)$  is the shortest-path distance from  $s$  to  $v$  defined as the minimum number of edges in any path from vertex  $s$  to vertex  $v$ ; if there is no path from  $s$  to  $v$ , then  $\delta(s, v) = \infty$ .

# BFS: Lemmas

## Lemma 2

Let  $G = (V, E)$  be a directed or undirected graph, and suppose that BFS is run on  $G$  from a given source vertex  $s \in V$ . Then upon termination, for each vertex  $v \in V$ , the value  $v.d$  computed by BFS satisfies  $v.d \geq \delta(s, v)$ .

## Lemma 3

Suppose that during the execution of BFS on a graph  $G = (V, E)$ , the queue  $Q$  contains the vertices  $\langle v_1, v_2, \dots, v_r \rangle$ , where  $v_1$  is the head of  $Q$  and  $v_r$  is the tail. Then,  $v_r.d \leq v_1.d + 1$  and  $v_i.d \leq v_{i+1}.d$  for  $i = 1, 2, \dots, r - 1$ .

## Lemma 4

Suppose that vertices  $v_i$  and  $v_j$  are enqueued during the execution of BFS, and that  $v_i$  is enqueued before  $v_j$ . Then  $v_i.d \leq v_j.d$  at the time that  $v_j$  is enqueued.

# Breadth-first Trees

- For a graph  $G = (V, E)$  with source  $s$ , we define the predecessor subgraph of  $G$  as  $G_\pi = (V_\pi, E_\pi)$ , where

$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$$

and

$$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$$

The predecessor subgraph  $G_\pi$  is a breadth-first tree if  $V_\pi$  consists of the vertices reachable from  $s$  and, for all  $v \in V_\pi$ , the subgraph  $G_\pi$  contains a unique simple path from  $s$  to  $v$  that is also a shortest path from  $s$  to  $v$  in  $G$ . A breadth-first tree is in fact a tree, since it is connected and  $|E_\pi| = |V_\pi| - 1$ . We call the edges in  $E_\pi$  tree edges.

# Predecessor Subgraph of a Depth-first Search

- For a graph  $G = (V, E)$ , predecessor subgraph of a depth-first search can be defined as  $G_\pi = (V, E_\pi)$  where

$$E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq \text{NIL}\}$$

The predecessor subgraph of a depth-first search forms a depth-first forest comprising several depth-first trees. The edges in  $E_\pi$  are tree edges.

# DFS: Theorems

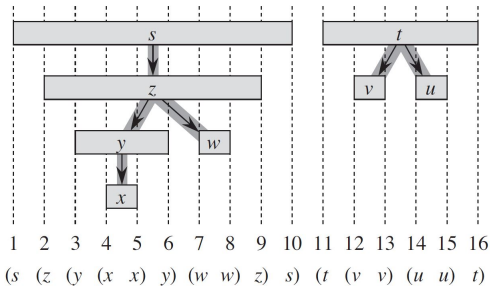
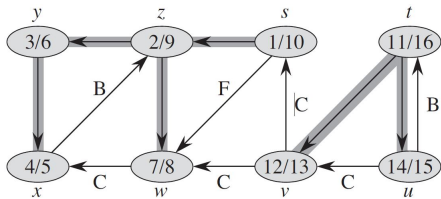
## Parenthesis Theorem

In any depth-first search of a (directed or undirected) graph  $G = (V, E)$ , for any two vertices  $u$  and  $v$ , exactly one of the following three conditions holds:

- ▶ the intervals  $[u.d, u.f]$  and  $[v.d, v.f]$  are entirely disjoint, and neither  $u$  nor  $v$  is a descendant of the other in the depth-first forest,
- ▶ the interval  $[u.d, u.f]$  is contained entirely within the interval  $[v.d, v.f]$ , and  $u$  is a descendant of  $v$  in a depth-first tree, or
- ▶ the interval  $[v.d, v.f]$  is contained entirely within the interval  $[u.d, u.f]$ , and  $v$  is a descendant of  $u$  in a depth-first tree.



# DFS: Theorems



# DFS: Theorems and Corollaries

## Corollary: Nesting of Descendants' Intervals

Vertex  $v$  is a proper descendant of vertex  $u$  in the depth-first forest for a (directed or undirected) graph  $G$  if and only if  $u.d < v.d < v.f < u.f$ .

## Theorem: White-path Theorem

In a depth-first forest of a (directed or undirected) graph  $G = (V, E)$ , vertex  $v$  is a descendant of vertex  $u$  if and only if at the time  $u.d$  that the search discovers  $u$ , there is a path from  $u$  to  $v$  consisting entirely of white vertices.

# Classification of Edges

## Tree edges

Tree edges are edges in the depth-first forest  $G_\pi$ . Edge  $(u, v)$  is a tree edge if  $v$  was first discovered by exploring edge  $(u, v)$ .

## Back edges

Back edges are those edges  $(u, v)$  connecting a vertex  $u$  to an ancestor  $v$  in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.

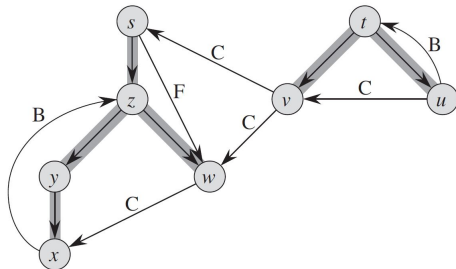
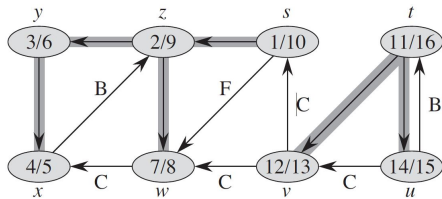
## Forward edges

Forward edges are those nontree edges  $(u, v)$  connecting a vertex  $u$  to a descendant  $v$  in a depth-first tree.

## Cross edges

Cross edges are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

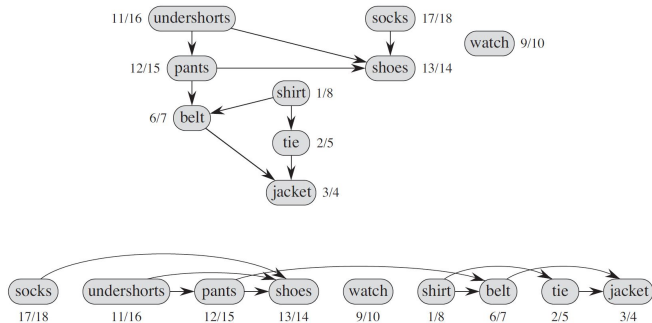
# Classification of Edges



# Topological Sort

- ▶ A topological sort of a dag  $G = (V, E)$  is a linear ordering of all its vertices such that if  $G$  contains an edge  $(u, v)$ , then  $u$  appears before  $v$  in the ordering.
- ▶ We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.
- ▶ If the graph contains a cycle, then topological sorting is not possible.
- ▶ Many applications use directed acyclic graphs to indicate precedences among events.

# Topological Sort



# Topological Sort

## Algorithm

TOPOLOGICAL-SORT( $G$ )

- 1 call DFS( $G$ ) to compute finishing times  $v.f$  for each vertex  $v$
- 2 as each vertex is finished, insert it onto the front of a linked list
- 3 **return** the linked list of vertices

## Analysis

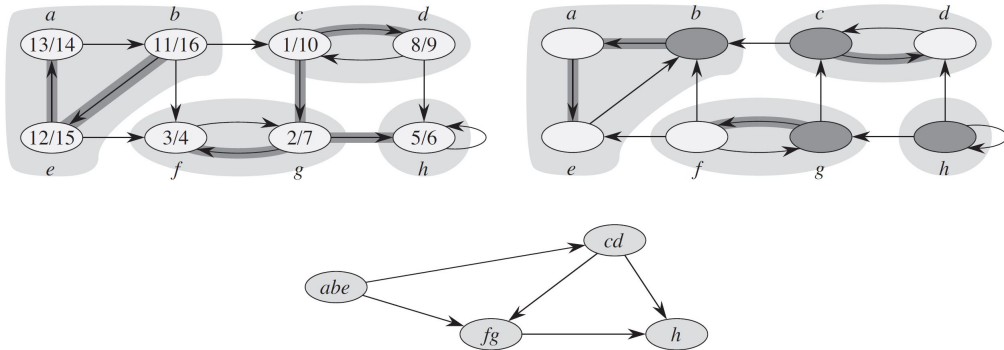
- ▶ We can perform a topological sort in time  $\Theta(|V|+|E|)$ , since depth-first search takes  $\Theta(|V|+|E|)$  time and it takes  $O(1)$  time to insert each of the  $|V|$  vertices onto the front of the linked list.
- ▶ A directed graph  $G$  is acyclic if and only if a depth-first search of  $G$  yields no back edges.

# Strongly Connected Component

- ▶ A strongly connected component of a directed graph  $G = (V, E)$  is a maximal set of vertices  $C \subseteq V$  such that for every pair of vertices  $u$  and  $v$  in  $C$ , we have both  $u \rightsquigarrow v$  and  $v \rightsquigarrow u$ , that is, vertices  $u$  and  $v$  are reachable from each other.
- ▶ The transpose of a graph  $G = (V, E)$  is  $G^T = (V, E^T)$ , where  $E^T = \{(u, v) : (v, u) \in E\}$ .
- ▶  $G$  and  $G^T$  have exactly the same strongly connected components:  $u$  and  $v$  are reachable from each other in  $G$  if and only if they are reachable from each other in  $G^T$ .



# Strongly Connected Component



# Strongly Connected Component

## Algorithm

STRONGLY-CONNECTED-COMPONENTS( $G$ )

- 1 call DFS( $G$ ) to compute finishing times  $u.f$  for each vertex  $u$
- 2 compute  $G^T$
- 3 call DFS( $G^T$ ), but in the main loop of DFS, consider the vertices in order of decreasing  $u.f$  (as computed in line 1)
- 4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

# White Board

# White Board