


# Data Structures and Algorithms <sup>1</sup>

BITS-Pilani K. K. Birla Goa Campus

---

<sup>1</sup>Material for the presentation taken from Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms, Fourth Edition*; 

# Course plan

- ▶ Ch. 20, 21, 22.3 (Graph Algorithms)

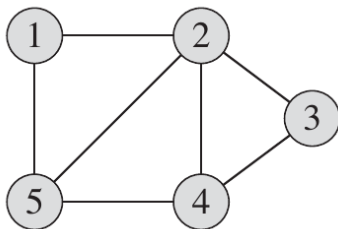
# Course plan

- ▶ Ch. 20, 21, 22.3 (Graph Algorithms)
- ▶ Ch. 12, 13 (Binary Search Tree and Red-Black Tree)

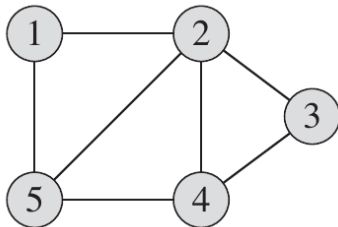
# Course plan

- ▶ Ch. 20, 21, 22.3 (Graph Algorithms)
- ▶ Ch. 12, 13 (Binary Search Tree and Red-Black Tree)
- ▶ Ch. 11 (Hash Tables)

# Module IV: Graph Algorithms

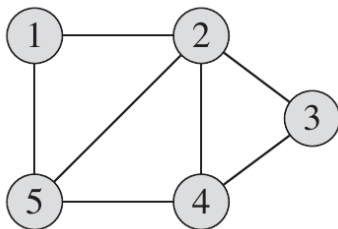


# Module IV: Graph Algorithms



- ▶ Graphs are mathematical structures consisting of vertices and edges.

# Module IV: Graph Algorithms



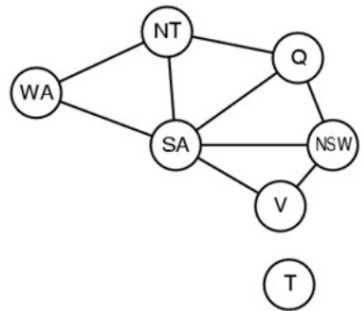
- ▶ Graphs are mathematical structures consisting of vertices and edges.
- ▶ The input to the graph algorithms will be a graph represented as  $G = (V, E)$ .

# Map coloring problem

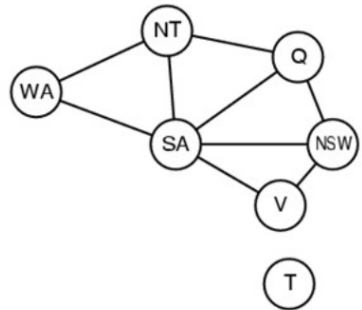




# Vertex coloring problem



# Vertex coloring problem



# Module IV: Graph Algorithms

## ► Ch. 20: Elementary Graph Algorithms

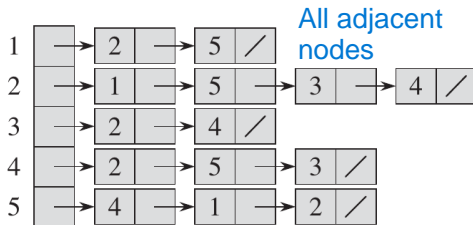
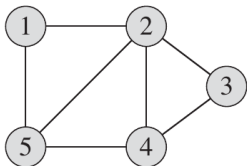
# Module IV: Graph Algorithms

## ► Ch. 20: Elementary Graph Algorithms

Breadth-first search, Depth-first search, Topological sort

# Representing undirected graphs

## ► Adjacency-list representation of undirected graphs



# Representing undirected graphs

- ▶ Suppose we sum the lengths of all the adjacency lists of an undirected graph. What will be the sum?

# Representing undirected graphs

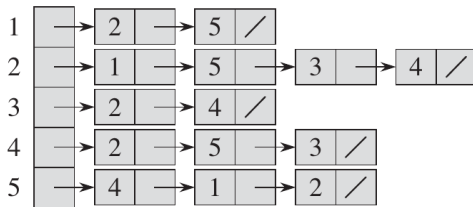
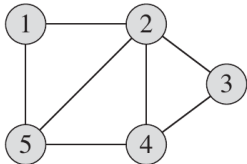
- ▶ Suppose we sum the lengths of all the adjacency lists of an undirected graph. What will be the sum?

- (i)  $V$       (ii)  $E$       (iii)  $V^2$       (iv)  $VE$   
(v) None of the above

# Representing undirected graphs

- Suppose we sum the lengths of all the adjacency lists of an undirected graph. What will be the sum? **2E**

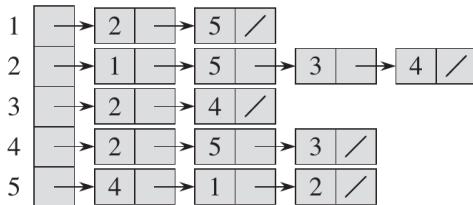
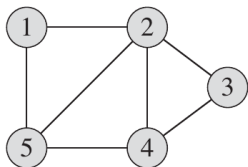
- (i)  $V$     (ii)  $E$     (iii)  $V^2$     (iv)  $VE$   
(v) None of the above





# Representing undirected graphs

## ► Adjacency-matrix representation of undirected graphs

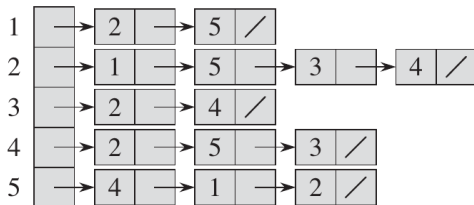
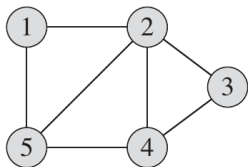


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

1 if adjacent

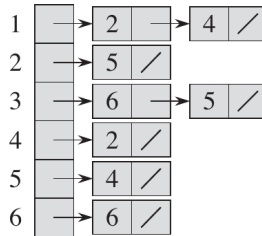
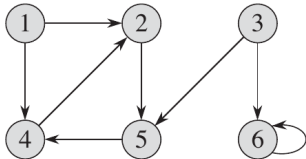
# Representing undirected graphs

- Will the Adjacency-matrix for an undirected graph be symmetric? **Yes**



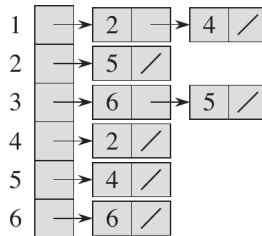
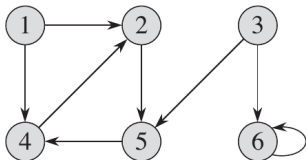
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

# Representing directed graphs



adjacency only 1-way not 2-way

# Representing directed graphs



	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

# Representing directed graphs

- ▶ Suppose we sum the lengths of all the adjacency lists of a directed graph. What will be the sum?

# Representing directed graphs

- ▶ Suppose we sum the lengths of all the adjacency lists of a directed graph. What will be the sum?  $E$
- ▶ Will the Adjacency-matrix for a directed graph be symmetric?

# Elementary Graph Algorithms

- ▶ We will refer to adjacency list of vertex  $u$  as  $G.Adj[u]$ .

# Elementary Graph Algorithms

- ▶ We will refer to adjacency list of vertex  $u$  as  $G.Adj[u]$ .
- ▶ Graph algorithms assume that the input graph is represented either using adjacency-list or using adjacency-matrix.



# Elementary Graph Algorithms

- ▶ We will refer to adjacency list of vertex  $u$  as  $G.Adj[u]$ .
- ▶ Graph algorithms assume that the input graph is represented either using adjacency-list or using adjacency-matrix.
- ▶ Sparse graphs:  $|E|$  is much less than  $|V|^2$ .

# Elementary Graph Algorithms

- ▶ We will refer to adjacency list of vertex  $u$  as  $G.Adj[u]$ .
- ▶ Graph algorithms assume that the input graph is represented either using adjacency-list or using adjacency-matrix.
- ▶ Sparse graphs:  $|E|$  is much less than  $|V|^2$ .
- ▶ We may use adjacency-list to represent sparse graphs.

# Elementary Graph Algorithms

- ▶ We will refer to adjacency list of vertex  $u$  as  $G.Adj[u]$ .
- ▶ Graph algorithms assume that the input graph is represented either using adjacency-list or using adjacency-matrix.
- ▶ Sparse graphs:  $|E|$  is much less than  $|V|^2$ .
- ▶ We may use adjacency-list to represent sparse graphs.
- ▶ Dense graphs:  $|E|$  is close to  $|V|^2$ .

# Graph Representation : Weighted graphs

- ▶ Weighted graphs: Graphs where each edge has a weight associated with it.

# Graph Representation : Weighted graphs

- ▶ Weighted graphs: Graphs where each edge has a weight associated with it.

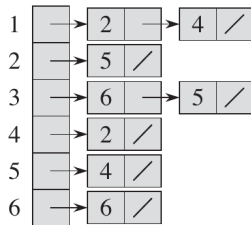
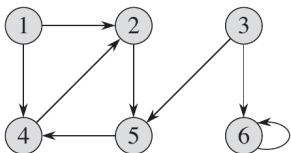
Weight function  $w : E \rightarrow \mathbb{R}$ .

# Graph Representation : Weighted graphs

- ▶ Adjacency-list and Adjacency-matrix can be adapted to represent weighted graphs.

# Graph Representation : Weighted graphs

- Adjacency-list and Adjacency-matrix can be adapted to represented weighted graphs.

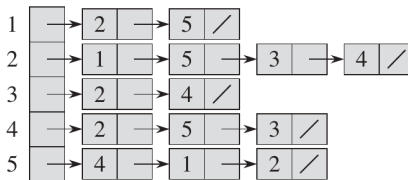
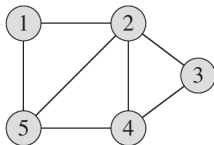


	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

[https://algorithmtutor.com/images/graph\\_representation\\_weighted.png](https://algorithmtutor.com/images/graph_representation_weighted.png)

# Graph Representation

- Can we have an edge which is a self-loop in an undirected graph?



	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



# Graph Representation

- ▶ Adjacency-list representation requires  $\Theta(V + E)$  memory compared to  $\Theta(V^2)$  memory for Adjacency matrix.

# Graph Representation

- ▶ Adjacency-list representation requires  $\Theta(V + E)$  memory compared to  $\Theta(V^2)$  memory for Adjacency matrix.
- ▶ Suppose we wish to check whether a given edge  $(u, v)$  is present in a graph  $G(V, E)$ . Which representation will take more time to check this? Adjacency-list or adjacency-matrix?

Matrix will be  $O(2 \cdot V)$ , List will be  $O(V + E^*)$ , where  $E^*$  would be number of edges for that vertex, ie, degree of that vertex.

Directed graph can double time but pretty much same complexity.

# Breadth-first search algorithm

- ▶ Breadth-first search finds the shortest distance (smallest number of edges) from source vertex  $s$  to all the vertices.

# Breadth-first search algorithm

- ▶ Breadth-first search finds the shortest distance (smallest number of edges) from source vertex  $s$  to all the vertices.
- ▶ We use Queue data structure in Breadth-first search.

# Breadth-first search algorithm

- ▶ Breadth-first search finds the shortest distance (smallest number of edges) from source vertex  $s$  to all the vertices.
- ▶ We use Queue data structure in Breadth-first search.
- ▶ Also, we have an attributes  $d$  (distance) and *color* for each node.

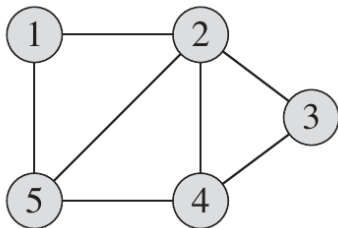
# Breadth-first search algorithm

- ▶ Breadth-first search finds the shortest distance (smallest number of edges) from source vertex  $s$  to all the vertices.
- ▶ We use Queue data structure in Breadth-first search.
- ▶ Also, we have an attributes  $d$  (distance) and *color* for each node.
- ▶ Operation of Breadth-first search (P. 557)

# Breadth-first search algorithm

- ▶ Breadth-first search finds the shortest distance (smallest number of edges) from source vertex  $s$  to all the vertices.
- ▶ We use Queue data structure in Breadth-first search.
- ▶ Also, we have an attributes  $d$  (distance) and *color* for each node.
- ▶ Operation of Breadth-first search (P. 557)
- ▶ Breadth-first search pseudocode (P.556)

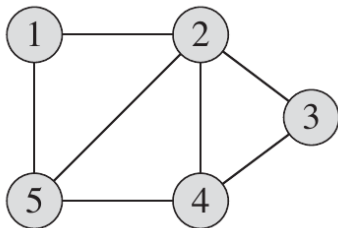
## Breadth-first search : Undirected graph



- Suppose we do a Breadth-first search starting from node 5. Which of the following orders of node visits is not possible under Breadth-first search?
- a. 5,4,1,2,3
  - b. 5,1,2,4,3
  - c. 5,1,2,3,4    c, since 4 is at less dist than 3

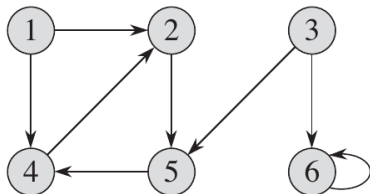


# Breadth-first search : Undirected graph



- ▶ If we start from node 5, what will be the distance of node 3 found by the Breadth-first search?

# Breadth-first search : Directed graph

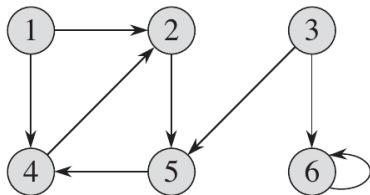


► Suppose we do a Breadth-first search starting from node 1. Which of the following orders of node visits is not possible under Breadth-first search?

- a. 1,2,4,5,3,6
- b. 1,2,5,4
- c. 1,2,4,5

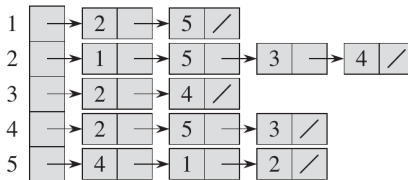
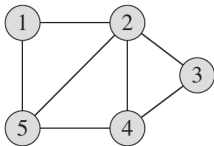
Only c is possible

# Breadth-first search : Directed graph

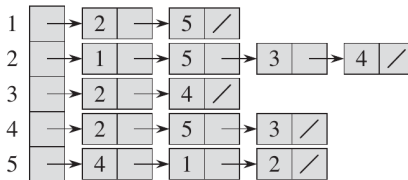
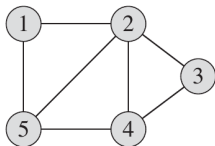


- ▶ What will be the distance of node 3 found by the Breadth-first search?

# Breadth-first search

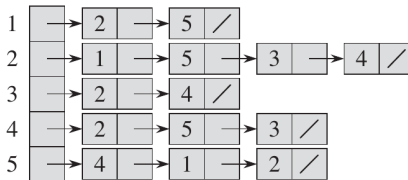
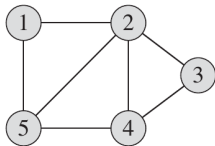


# Breadth-first search



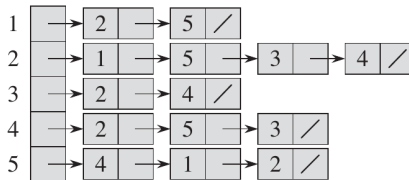
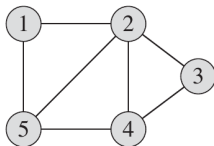
► Running time of BFS is

# Breadth-first search



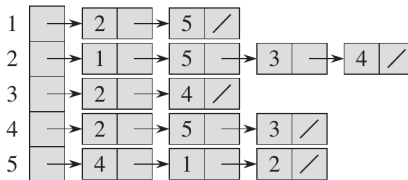
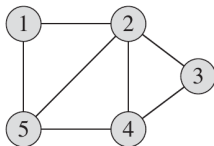
- ▶ Running time of BFS is  $O(V + E)$ .

# Breadth-first search



- ▶ Running time of BFS is  $O(V + E)$ . (Aggregate analysis)

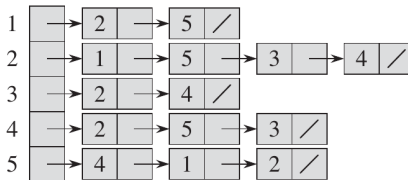
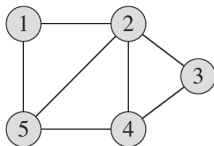
# Breadth-first search



- ▶ Running time of BFS is  $O(V + E)$ . (Aggregate analysis)
- ▶ Why is it called Breadth-first search?

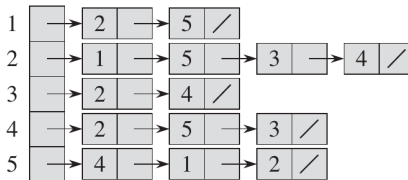
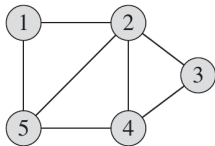


# Breadth-first search



- ▶ Running time of BFS is  $O(V + E)$ . (Aggregate analysis)
- ▶ Why is it called Breadth-first search?
- ▶ When the BFS algorithm terminates, the attribute  $v.d$  contains the shortest path (minimum number of edges) from node  $s$  to  $v$ .

# Predecessor attribute $\pi$



- ▶ The attribute  $\pi$  stores the parent (predecessor) of each node in the breadth-first tree. We can use  $\pi$  attributes to find the shortest path from  $s$  to any vertex  $v$ .

# Breadth-first search

- Q. What will be the running time of BFS if we represent its input graph by an adjacency matrix and modify the algorithm to handle this form of graph input?

# Breadth-first tree

- ▶ Predecessor subgraph  $G_\pi = (V_\pi, E_\pi)$ :

# Breadth-first tree

- ▶ Predecessor subgraph  $G_\pi = (V_\pi, E_\pi)$ :  
$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$$

# Breadth-first tree

- ▶ Predecessor subgraph  $G_\pi = (V_\pi, E_\pi)$ :

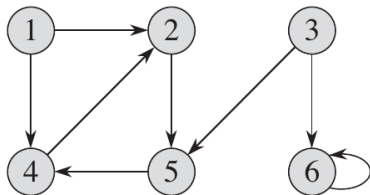
$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$$

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

# Breadth-first tree

- ▶ Predecessor subgraph  $G_\pi = (V_\pi, E_\pi)$ :  
$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$$
$$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$$
- ▶ Predecessor subgraph obtained after breadth-first search is called a breadth-first tree.

# Breadth-first tree



- ▶ Suppose we do a Breadth-first search starting from node 1. Find the Breadth-First tree  $G_\pi = (V_\pi, E_\pi)$  obtained?

Tree consists of parts of graph reachable from given node



# Shortest path in a maze

$$\text{maze} = \begin{bmatrix} S & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & E \end{bmatrix}$$

# Depth-first search

- ▶ Depth-first search explores edges out of the most recently discovered vertex  $v$ .

# Depth-first search

- ▶ Depth-first search explores edges out of the most recently discovered vertex  $v$ .
- ▶ DFS operation (p. 566)

# Depth-first search

- ▶ Depth-first search explores edges out of the most recently discovered vertex  $v$ .
- ▶ DFS operation (p. 566)
- ▶ We will once again use color attributes for the nodes.

# Depth-first search

- ▶ Depth-first search explores edges out of the most recently discovered vertex  $v$ .
- ▶ DFS operation (p. 566)
- ▶ We will once again use color attributes for the nodes.
- ▶ Gray color: when a node is discovered.

# Depth-first search

- ▶ Depth-first search explores edges out of the most recently discovered vertex  $v$ .
- ▶ DFS operation (p. 566)
- ▶ We will once again use color attributes for the nodes.
- ▶ Gray color: when a node is discovered.  
Black color: when the adjacency list of a node is examined completely.

# Depth-first search

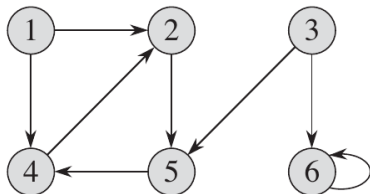
- ▶ Depth-first search explores edges out of the most recently discovered vertex  $v$ .
- ▶ DFS operation (p. 566)
- ▶ We will once again use color attributes for the nodes.
- ▶ Gray color: when a node is discovered.  
Black color: when the adjacency list of a node is examined completely.
- ▶ Discovered and finished timestamps.

# Depth-first search

- ▶ Depth-first search explores edges out of the most recently discovered vertex  $v$ .
- ▶ DFS operation (p. 566)
- ▶ We will once again use color attributes for the nodes.
- ▶ Gray color: when a node is discovered.  
Black color: when the adjacency list of a node is examined completely.
- ▶ Discovered and finished timestamps.
- ▶ DFS pseudocode (p. 565)

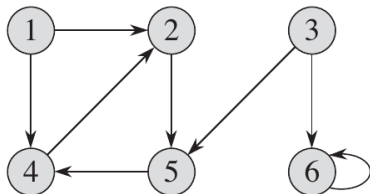


# Depth-first search : Directed graph



- ▶ Suppose that the first node to be discovered is 3. What are the possible finish timestamps for node 4?

# Depth-first search : Directed graph



- ▶ Suppose that the first node to be discovered is 3. What are the possible finish timestamps for node 4?
- ▶ Which node remains white when node 3 is finished?

# Predecessor subgraph

- ▶ Tree - connected acyclic graph

# Predecessor subgraph

- ▶ Tree - connected acyclic graph
- ▶ Forest - set of trees

# Predecessor subgraph

- ▶ Tree - connected acyclic graph
- ▶ Forest - set of trees
- ▶ DFS helps us generate a depth-first forest  $G_\pi$ .

# Predecessor subgraph

- ▶ Tree - connected acyclic graph
- ▶ Forest - set of trees
- ▶ DFS helps us generate a depth-first forest  $G_\pi$ .
- ▶ Depth-first forest depends on the order in which vertices are visited (Line 5 of DFS procedure).

# Predecessor subgraph

- ▶ Tree - connected acyclic graph
- ▶ Forest - set of trees
- ▶ DFS helps us generate a depth-first forest  $G_\pi$ .
- ▶ Depth-first forest depends on the order in which vertices are visited (Line 5 of DFS procedure).
- ▶ Running time of DFS is

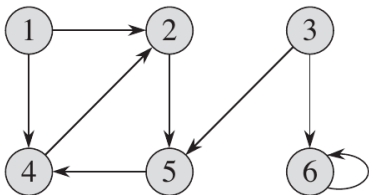
# Predecessor subgraph

- ▶ Tree - connected acyclic graph
- ▶ Forest - set of trees
- ▶ DFS helps us generate a depth-first forest  $G_\pi$ .
- ▶ Depth-first forest depends on the order in which vertices are visited (Line 5 of DFS procedure).
- ▶ Running time of DFS is  $\Theta(V + E)$ .

Forest is set of trees obtained on performing DFS from various initial nodes

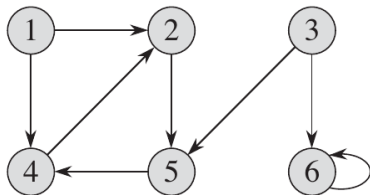


# Depth-first Tree



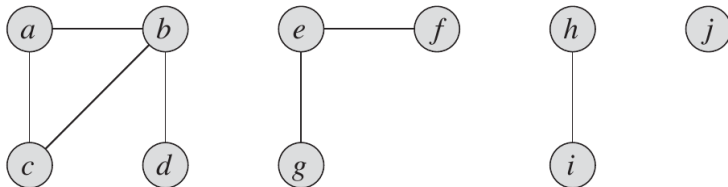
- ▶ DFS helps us generate a depth-first forest  $G_\pi$ .

# Depth-first Tree



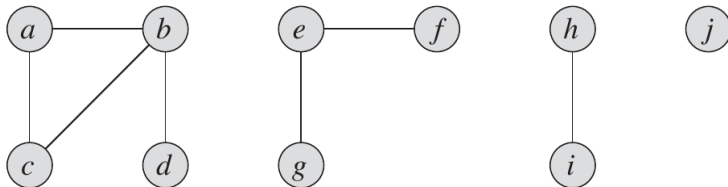
- ▶ DFS helps us generate a depth-first forest  $G_\pi$ .
- ▶ Suppose that the first node to be discovered is 3. What will the depth-first forest  $G_\pi$  look like?

# Depth-first Forest



- Consider the undirected graph shown above.

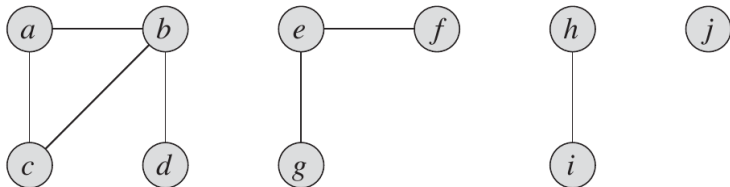
# Depth-first Forest



- Consider the undirected graph shown above.

$$|V| = 10, |E| = 7.$$

# Depth-first Forest



- ▶ Consider the undirected graph shown above.  
 $|V| = 10$ ,  $|E| = 7$ .
- ▶ How many trees will the Depth-first forest  $G_\pi$  contain?

# Classification of Edges

- ▶ The color of a vertex can help us detect a *back edge*.

# Classification of Edges

- ▶ The color of a vertex can help us detect a *back edge*.
- ▶ Back edge is an edge from a descendent node to a parent node during a DFS.

# Classification of Edges

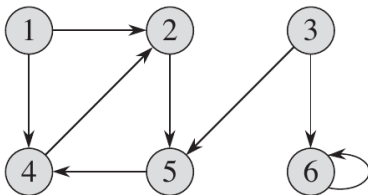
- ▶ The color of a vertex can help us detect a *back edge*.
- ▶ Back edge is an edge from a descendent node to a parent node during a DFS.
- ▶ Suppose we discover the nodes in the following order:  
1 , 2 , 5 , 4 , 3 , 6 .



# Classification of Edges

- ▶ The color of a vertex can help us detect a *back edge*.
- ▶ Back edge is an edge from a descendent node to a parent node during a DFS.
- ▶ Suppose we discover the nodes in the following order:

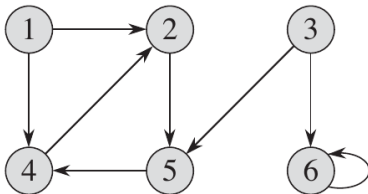
1 , 2 , 5 , 4 , 3 , 6 .



# Classification of Edges

- ▶ The color of a vertex can help us detect a *back edge*.
- ▶ Back edge is an edge from a descendent node to a parent node during a DFS.
- ▶ Suppose we discover the nodes in the following order:

1 , 2 , 5 , 4 , 3 , 6 .

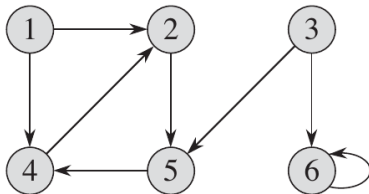


- ▶ What will be a back edge in the above graph?

# Classification of Edges

- ▶ The color of a vertex can help us detect a *back edge*.
- ▶ Back edge is an edge from a descendent node to a parent node during a DFS.
- ▶ Suppose we discover the nodes in the following order:

1 , 2 , 5 , 4 , 3 , 6 .



- ▶ What will be a back edge in the above graph? 4,2 is back edge
- ▶ Back edge helps us detect a cycle in a graph.

# Classification of Edges

- ▶ During DFS of a directed graph, suppose we reach  $v$  by following  $adj[u]$

# Classification of Edges

- ▶ During DFS of a directed graph, suppose we reach  $v$  by following  $adj[u]$
- ▶ That is, we reach  $v$  by following the directed edge  $(u, v)$ .

# Classification of Edges

- ▶ During DFS of a directed graph, suppose we reach  $v$  by following  $adj[u]$
- ▶ That is, we reach  $v$  by following the directed edge  $(u, v)$ .
- ▶ **Tree edge** if  $v$  is WHITE.

# Classification of Edges

- ▶ During DFS of a directed graph, suppose we reach  $v$  by following  $adj[u]$
- ▶ That is, we reach  $v$  by following the directed edge  $(u, v)$ .
- ▶ **Tree edge** if  $v$  is WHITE.
- ▶ How can we detect a Back edge using the *COLOR* attribute?

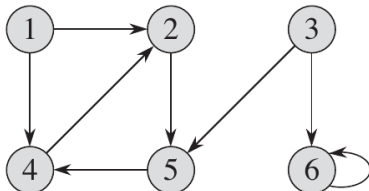
# Classification of Edges

- ▶ During DFS of a directed graph, suppose we reach  $v$  by following  $adj[u]$
- ▶ That is, we reach  $v$  by following the directed edge  $(u, v)$ .
- ▶ **Tree edge** if  $v$  is WHITE.
- ▶ How can we detect a Back edge using the *COLOR* attribute?
- ▶ **Back edge** if  $v$  is GRAY.



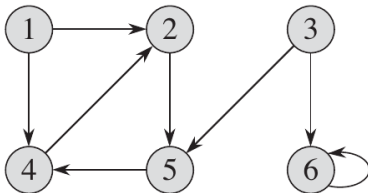
# Classification of Edges

- Suppose we discover the nodes in the following order:  
1, 4, 2, 5, 3, 6.



# Classification of Edges

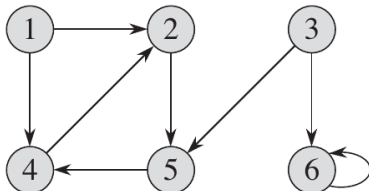
- Suppose we discover the nodes in the following order:  
1, 4, 2, 5, 3, 6.



- How can we detect a Forward edge using the *COLOR* attribute?

# Classification of Edges

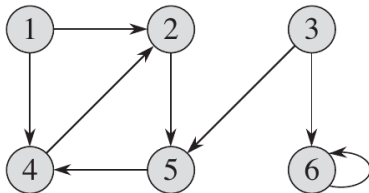
- Suppose we discover the nodes in the following order:  
1, 4, 2, 5, 3, 6.



- How can we detect a Forward edge using the *COLOR* attribute?
- Forward edge** if  $v$  is BLACK (not a sufficient condition).

# Classification of Edges

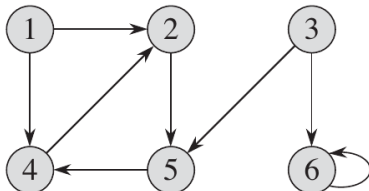
- Suppose we discover the nodes in the following order:  
1, 4, 2, 5, 3, 6.



- How can we detect a Forward edge using the *COLOR* attribute?
- Forward edge** if  $v$  is BLACK (not a sufficient condition).
- All the remaining edges are Cross edges.

# Classification of Edges

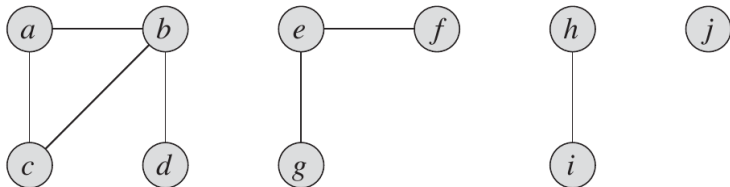
- Suppose we discover the nodes in the following order:  
1, 4, 2, 5, 3, 6.



color implemented by map  
or hash table

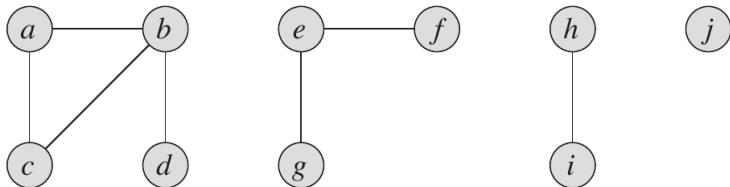
- How can we detect a Forward edge using the *COLOR* attribute?
- Forward edge** if  $v$  is BLACK (not a sufficient condition).
- All the remaining edges are Cross edges.
- Cross edge** : if  $v$  is BLACK.

# Connected components of an undirected graph



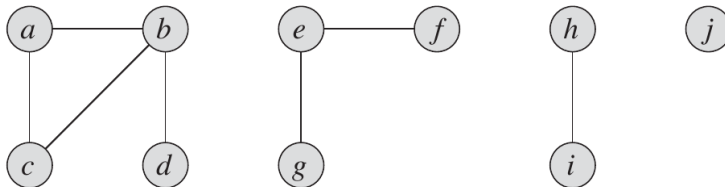
- ▶ Subgraph of a graph  $G$  is another graph formed from a subset of the vertices and edges of  $G$ .

# Connected components of an undirected graph



- ▶ Subgraph of a graph  $G$  is another graph formed from a subset of the vertices and edges of  $G$ .
- ▶ The vertex subset must include all endpoints of the edge subset, but may also include additional vertices.

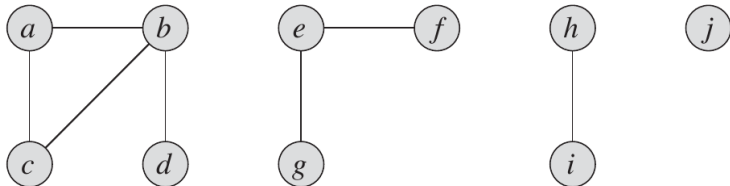
# Connected components of an undirected graph



- ▶ Subgraph of a graph  $G$  is another graph formed from a subset of the vertices and edges of  $G$ .
- ▶ The vertex subset must include all endpoints of the edge subset, but may also include additional vertices.
- ▶ Connected component is a connected subgraph that is not part of any larger connected subgraph.

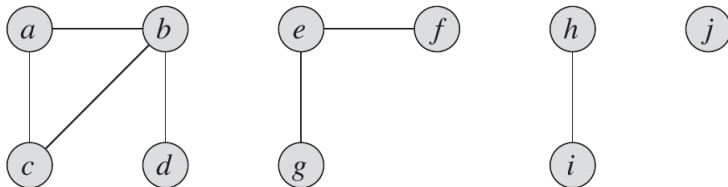


# Counting the connected components



- ▶ How many connected components are there in the above graph?

# Counting the connected components



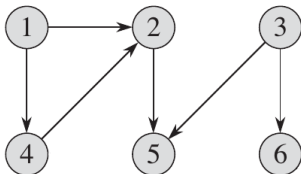
- ▶ How many connected components are there in the above graph?
- ▶ Can we modify the DFS algorithm to count the number of connected components?

# Topological sort

- ▶ A directed graph without any back edge is called a Directed acyclic graph (DAG).

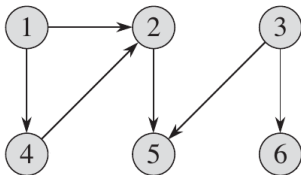
# Topological sort

- ▶ A directed graph without any back edge is called a Directed acyclic graph (DAG).



# Topological sort

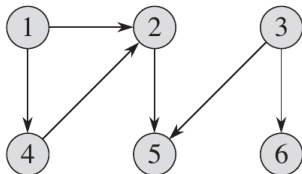
- ▶ A directed graph without any back edge is called a Directed acyclic graph (DAG).



- ▶ Topological sort gives us a linear ordering of vertices such that if there is a directed edge  $(u, v)$ , then  $u$  comes before  $v$  in the linear ordering.

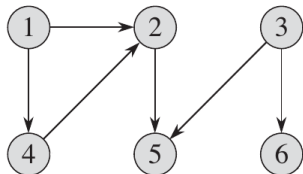
# Topological sort

- ▶ A directed graph without any back edge is called a Directed acyclic graph (DAG).



- ▶ Topological sort gives us a linear ordering of vertices such that if there is a directed edge  $(u, v)$ , then  $u$  comes before  $v$  in the linear ordering.
- ▶ Which of the following is a topological sort?
  - 1, 4, 2, 5, 3, 6
  - 1, 4, 2, 3, 5, 6

# Topological sort

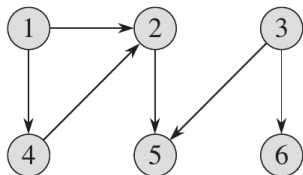


► Which of the following is a topological sort?

a. 1, 4, 2, 5, 3, 6

b. 1, 4, 2, 3, 5, 6

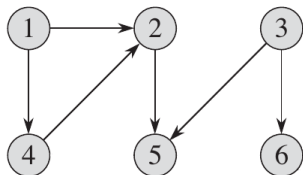
# Topological sort



- ▶ Which of the following is a topological sort?
  - a. 1, 4, 2, 5, 3, 6
  - b. 1, 4, 2, 3, 5, 6
- ▶ Can we have another topological sort for the same graph?

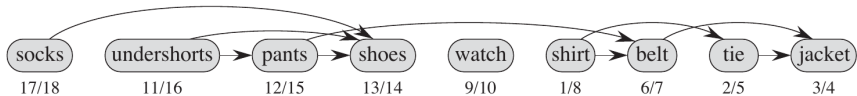
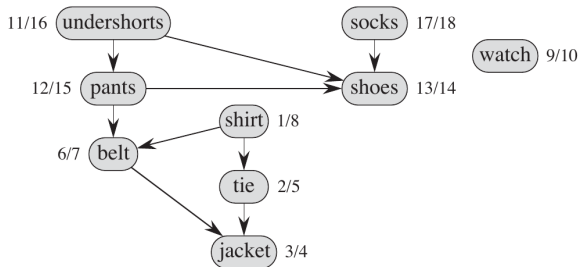


# Topological sort



- ▶ Which of the following is a topological sort?
  - a. 1, 4, 2, 5, 3, 6
  - b. 1, 4, 2, 3, 5, 6
- ▶ Can we have another topological sort for the same graph?
  - c. 3, 6, 1, 4, 2, 5

# Topological sort example

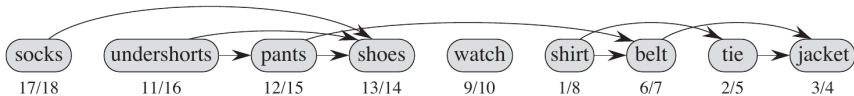
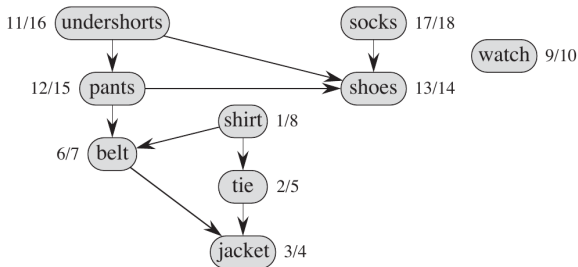


# Topological sort pseudocode

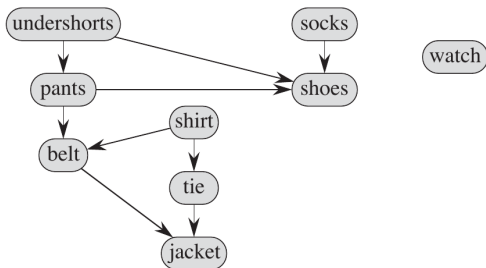
## TOPOLOGICAL-SORT( $G$ )

- 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

# Topological sort example



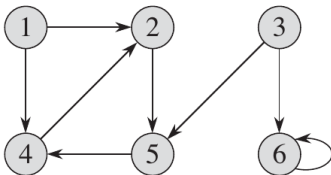
# Topological sort example 2



- ▶ Let us say we consider nodes in the following order:  
Belt, Shoes, Shirt, Undershorts, Watch, Socks

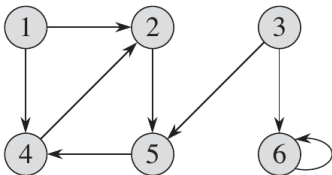
# Topological sort

- ▶ Topological sorting should fail if we detect a back edge.



# Topological sort

- ▶ Topological sorting should fail if we detect a back edge.



- ▶ Why does topological sort algorithm work?

# Applications of DFS and Topological sort

- ▶ Operating system deadlock detection



# Applications of DFS and Topological sort

- ▶ Operating system deadlock detection
- ▶ Course schedule problem

# Applications of DFS and Topological sort

- ▶ Operating system deadlock detection
- ▶ Course schedule problem
- ▶ Job scheduling