Graphs

- Graphs are very useful mathematical structures for modeling problems.

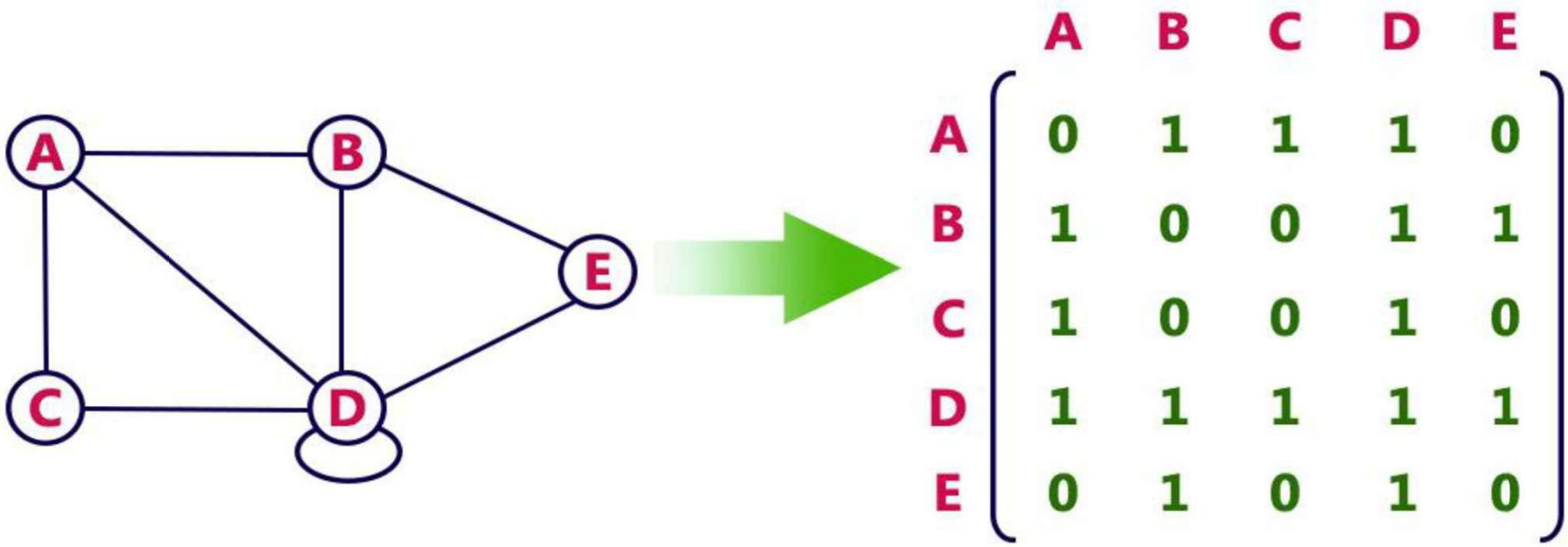
$G = (V,E)$

- Set of vertices V
- Set of edges E
  - E is a subset of pairs (v,v'): $E \subseteq V \times V$
  - Undirected graph: (v,v') and (v',v) are the same edge
  - Directed graph:
    - (v,v') is an edge from v to v'
    - Does not guarantee that (v',v) is also an edge

Graph Representation

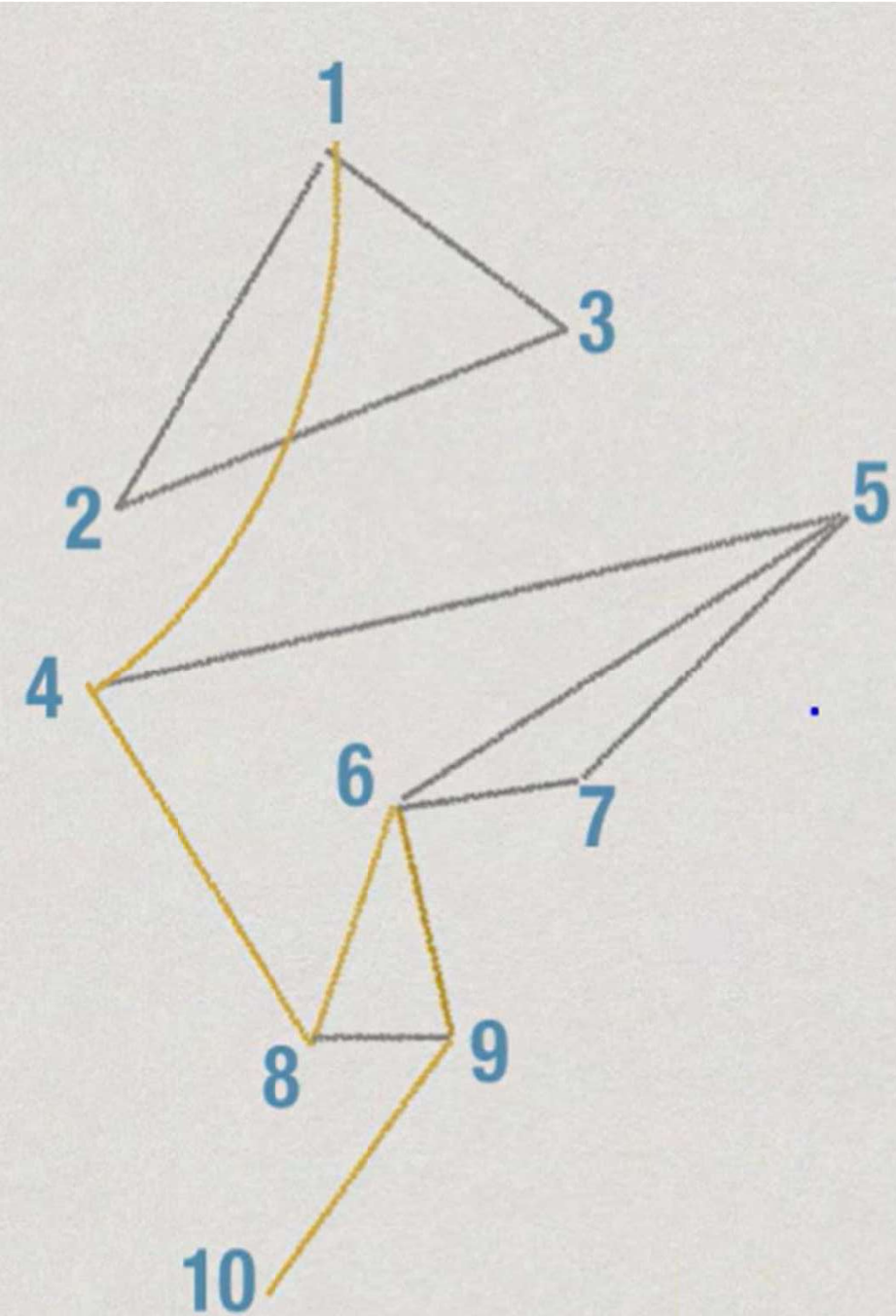
- Let V have n vertices
  - We can assume vertices are named 1,2,…,n
- Each edge is now a pair (i,j), where $1 \leq i,j \leq n$
- Let A(i,j) = 1 if (i,j) is an edge and 0 otherwise
- A is an n x n matrix describing the graph
  - **Adjacency matrix**

Adjacency Matrix



- It is a 2D array(say adj[][]) of size |V|x|V| where |V| is the number of vertices in a graph.
  - If adj[i][j] = 1, then there is an edge from vertex i to vertex j.
  - For a weighted graph if adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w
- Adjacency matrix for undirected graph is always symmetric.

Adjacency Matrix



	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	0	0	0	0	0	0
2	1	0	1	0	0	0	0	0	0	0
3	1	1	0	0	0	0	0	0	0	0
4	1	0	0	0	1	0	0	1	0	0
5	0	0	0	1	0	1	1	0	0	0
6	0	0	0	0	1	0	1	1	1	0
7	0	0	0	0	1	1	0	0	0	0
8	0	0	0	1	0	1	0	0	1	0
9	0	0	0	0	0	1	0	1	0	1
10	0	0	0	0	0	0	0	0	1	0

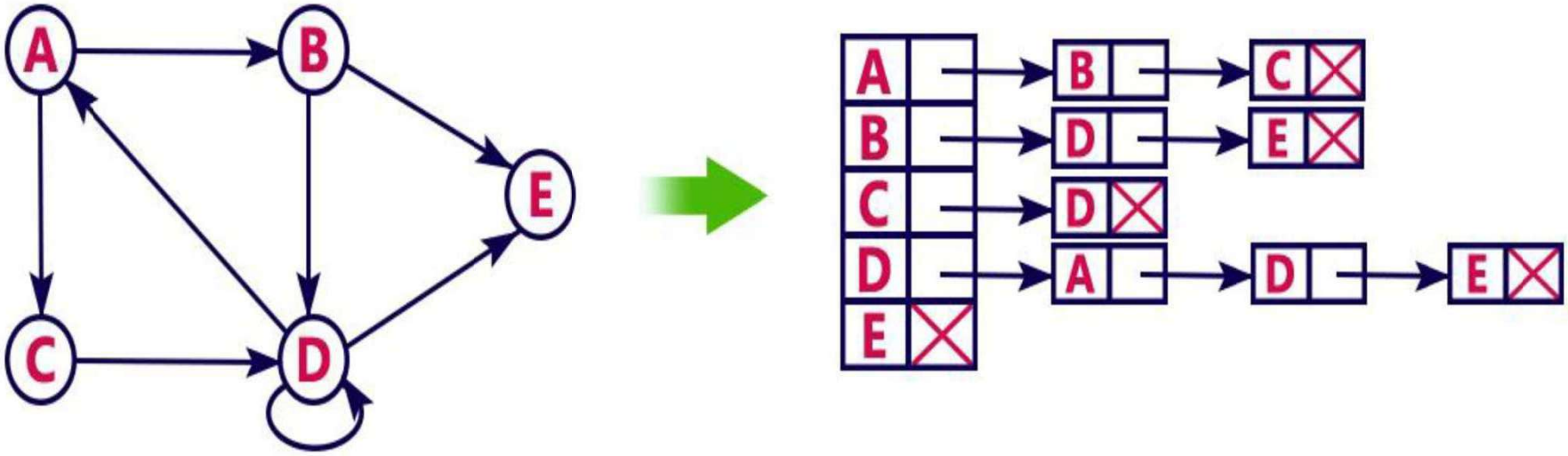
- Neighbours of i
  - Any column j in row i with entry 1
  - Scan row i from left to right to identify all neighbours
- Neighbours of 4 are {1,5,8}

Finding a path

- Start with v_s
  - New Delhi is 1
- Mark each neighbour as reachable
- Explore neighbours of marked vertices
- Check if target is marked
  - v_t =10 = Trivandrum

	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	0	0	0	0	0	0
2	1	0	1	0	0	0	0	0	0	0
3	1	1	0	0	0	0	0	0	0	0
4	1	0	0	0	1	0	0	1	0	0
5	0	0	0	1	0	1	1	0	0	0
6	0	0	0	0	1	0	1	1	1	0
7	0	0	0	0	1	1	0	0	0	0
8	0	0	0	1	0	1	0	0	1	0
9	0	0	0	0	0	1	0	1	0	1
10	0	0	0	0	0	0	0	0	1	0

Adjacency List



- An array of linked lists is used.
- Size of the array is equal to number of vertices.
- An entry array[i] represents the linked list of vertices adjacent to the **i**th vertex.
- This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists.

Type of Graphs

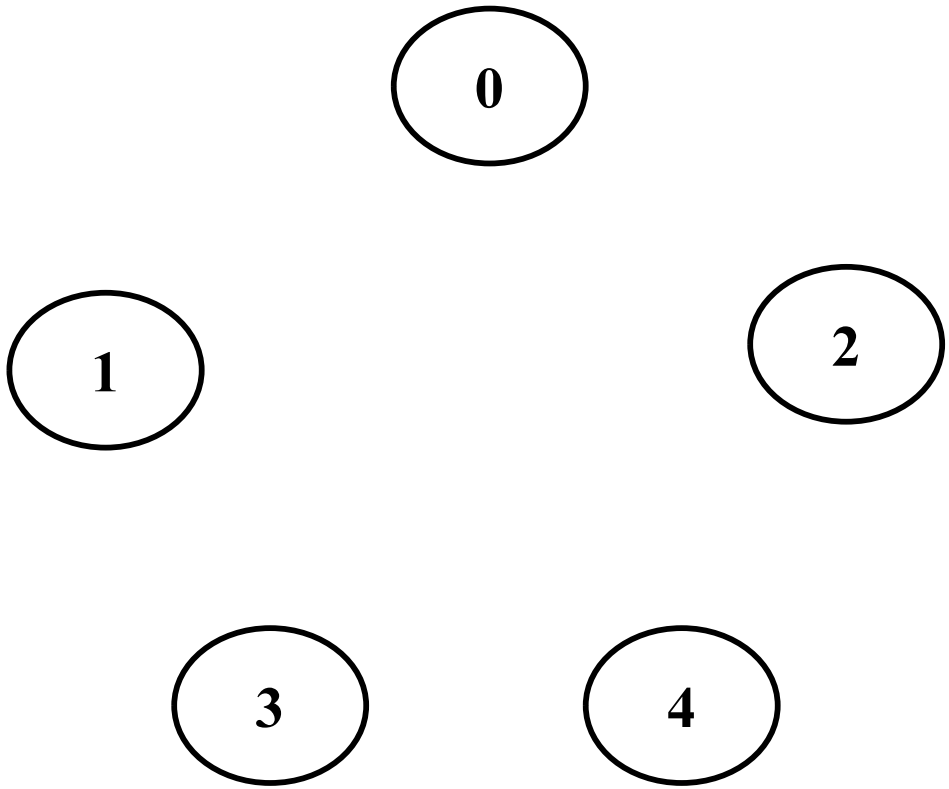
- **Undirected Graph:** A graph with only undirected edges.
- **Directed Graph:** A graph with only directed edges.
- **Directed Acyclic Graphs(DAG):** A directed graph with no cycles.
- **Cyclic Graph:** A directed graph with at least one cycle.
- **Weighted Graph:** It is a graph in which each edge is given a numerical weight.
- **Disconnected Graphs:** An undirected graph that is not connected.

Qtn) Consider a complete undirected graph with vertex set {0, 1, 2, 3, 4}. Entry Wij in the matrix W below is the weight of the edge {i, j}. Construct the graph.

$$W = \begin{pmatrix} 0 & 1 & 8 & 1 & 4 \\ 1 & 0 & 12 & 4 & 9 \\ 8 & 12 & 0 & 7 & 3 \\ 1 & 4 & 7 & 0 & 2 \\ 4 & 9 & 3 & 2 & 0 \end{pmatrix}$$

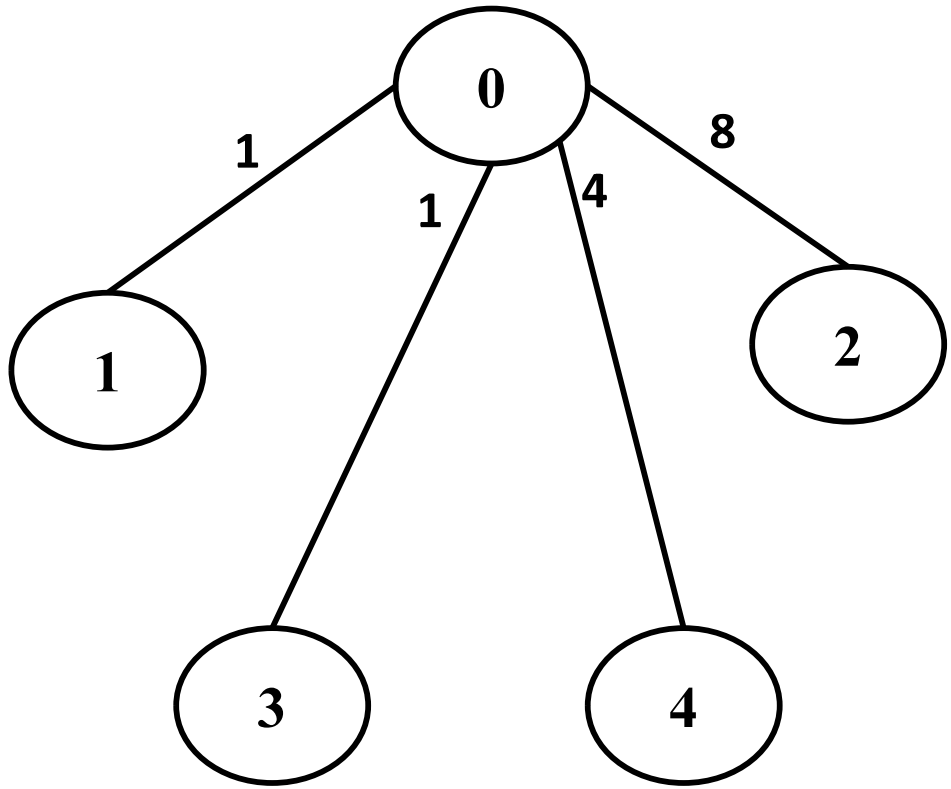
Qtn) Consider a complete undirected graph with vertex set {0, 1, 2, 3, 4}. Entry Wij in the matrix W below is the weight of the edge {i, j}. Construct the graph.

$$W = \begin{pmatrix} 0 & 1 & 8 & 1 & 4 \\ 1 & 0 & 12 & 4 & 9 \\ 8 & 12 & 0 & 7 & 3 \\ 1 & 4 & 7 & 0 & 2 \\ 4 & 9 & 3 & 2 & 0 \end{pmatrix}$$



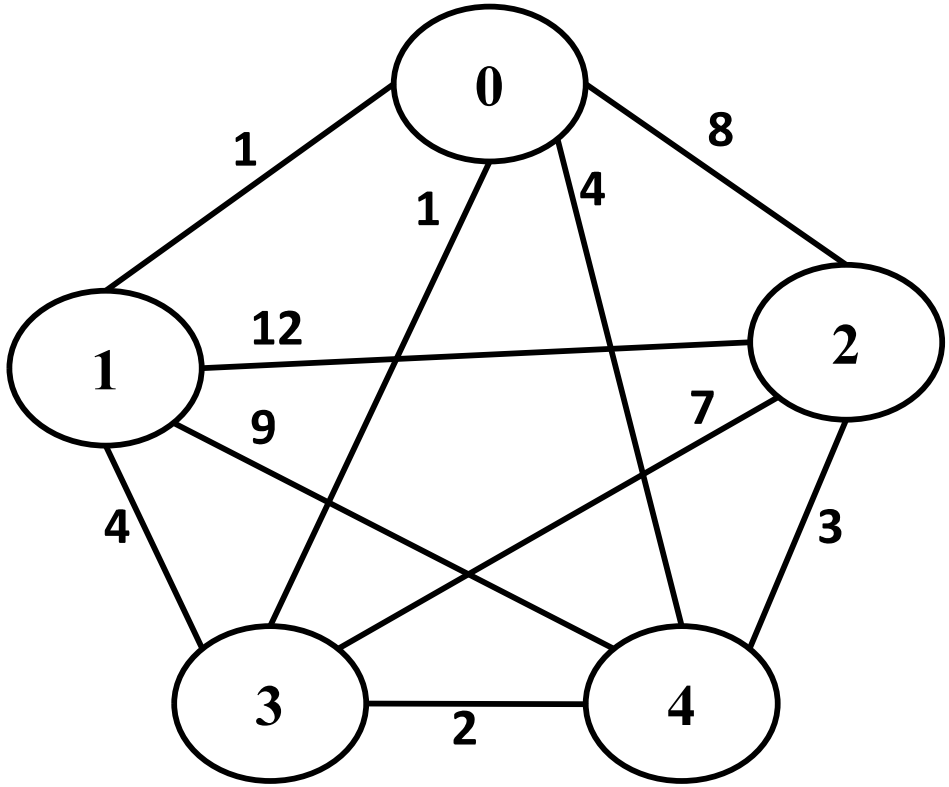
Qtn) Consider a complete undirected graph with vertex set {0, 1, 2, 3, 4}. Entry Wij in the matrix W below is the weight of the edge {i, j}. Construct the graph.

$$W = \begin{pmatrix} 0 & 1 & 8 & 1 & 4 \\ 1 & 0 & 12 & 4 & 9 \\ 8 & 12 & 0 & 7 & 3 \\ 1 & 4 & 7 & 0 & 2 \\ 4 & 9 & 3 & 2 & 0 \end{pmatrix}$$



Qtn) Consider a complete undirected graph with vertex set {0, 1, 2, 3, 4}. Entry Wij in the matrix W below is the weight of the edge {i, j}. Construct the graph.

$$W = \begin{pmatrix} 0 & 1 & 8 & 1 & 4 \\ 1 & 0 & 12 & 4 & 9 \\ 8 & 12 & 0 & 7 & 3 \\ 1 & 4 & 7 & 0 & 2 \\ 4 & 9 & 3 & 2 & 0 \end{pmatrix}$$



Graph Traversal Algorithms

- Breadth First Search(BFS)
- Depth First Search(DFS)

Graph Search Algorithms

Graph Search Algorithms

- Systematic search of every edge and every vertex of a graph.
- Graph G=(V,E) is either directed or undirected.
- Applications
  - Compilers
  - Networks
    - Routing, Searching, Clustering, etc.

Graph Traversal

- Breadth First Search (BFS)
  - Start several paths at a time, and advance in each one step at a time
- Depth First Search (DFS)
  - Once a possible path is found, continue the search until the end of the path

Breadth-First Search

- Breadth-first search starts at a given vertex *s*, which is at level 0.
- In the first stage, we visit all the vertices that are at the distance of one edge away. When we visit there, we paint as "**visited**," the vertices adjacent to the start vertex *s* - these vertices are placed into level 1.
- In the second stage, we visit all the new vertices we can reach at the distance of two edges away from the source vertex s. These new vertices, which are adjacent to level 1 vertices and not previously assigned to a level, are placed into level 2, and so on.
- The BFS traversal terminates when every vertex has been visited.

Breadth-First Search

The algorithm maintains a queue Q to manage the set of vertices and starts with s, the source vertex

Initially, all vertices except s are colored white, and s is gray.

BFS algorithm maintains the following information for each vertex u:

- color[u] (white, gray, or black) : indicates status
  - **white** = not discovered yet
  - **gray** = discovered, but not finished
  - **black** = finished
- **d[u]** : distance from s to u
- **π[u]** : predecessor of u in BF tree

Each vertex is assigned a color.

In general, a vertex is **white** before we start processing it, it is **gray** during the period the vertex is being processed, and it is **black** after the processing of the vertex is completed.

BFS Algorithm

Inputs: Inputs are a graph(directed or undirected) G =(V, E) and a source vertex s, where s is an element of V. The adjacency list representation of a graph is used in this analysis.

Outputs: The outputs are a predecessor graph, which represents the paths travelled in the BFS traversal, and a collection of distances, which represent the distance of each of the vertices from the source vertex.

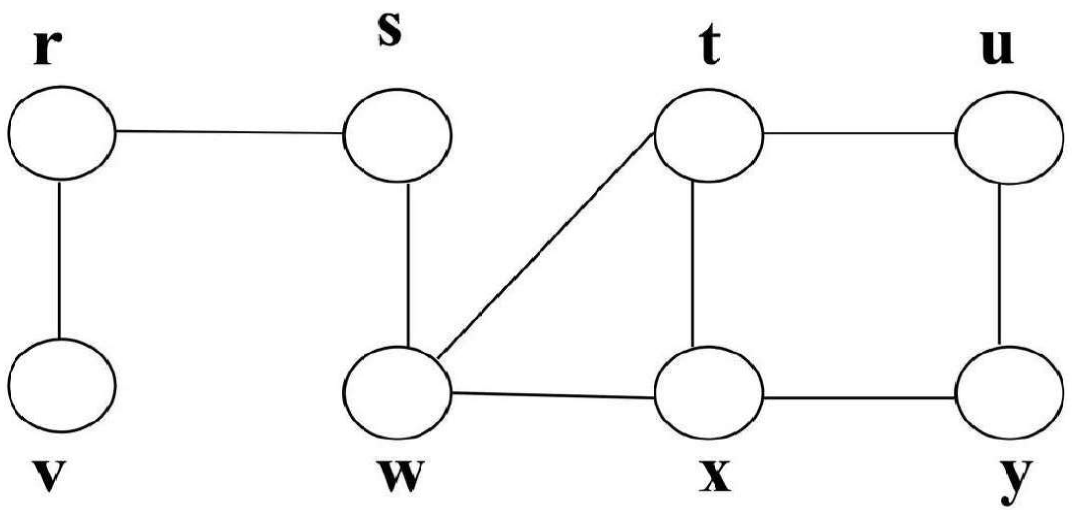
BFS ALGORITHM

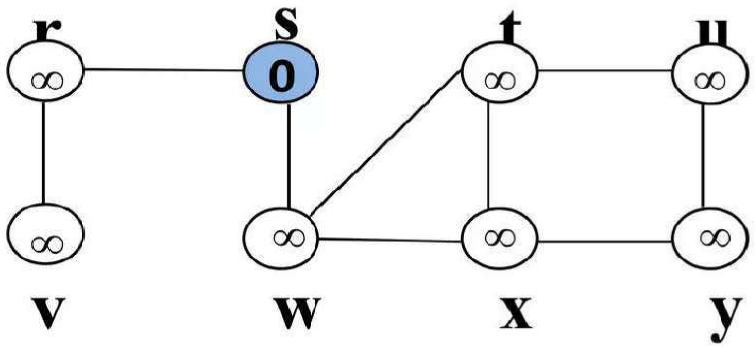
```
1.      for each u in V − {s}
2.          do color[u] ← WHITE
3.             d[u] ← infinity
4.             π[u] ← NIL
5.      color[s] ← GRAY
6.      d[s] ← 0
7.      π[s] ← NIL
8.      Q ← {}
9.      ENQUEUE(Q, s)
10      while Q is non-empty
11.         do u ← DEQUEUE(Q)
12.            for each v adjacent to u
13.                do if color[v] ← WHITE
14.                      then  color[v] ← GRAY
15.                            d[v] ← d[u] + 1
16.                            π[v] ← u
17.                            ENQUEUE(Q, v)

18.         color[u] ← BLACK
```

```
1. Enqueue (Q,s)
2. while Q ≠ ∅
3.     do u ← Dequeue(Q)
4.     for each v adjacent to u
5.         do if color[v] = white
6.             then color[v] ← gray
7.                  d[v] ← d[u] + 1
8.                  π[v] ← u
9.                  Enqueue(Q,v)
10.    color[u] ← black
```

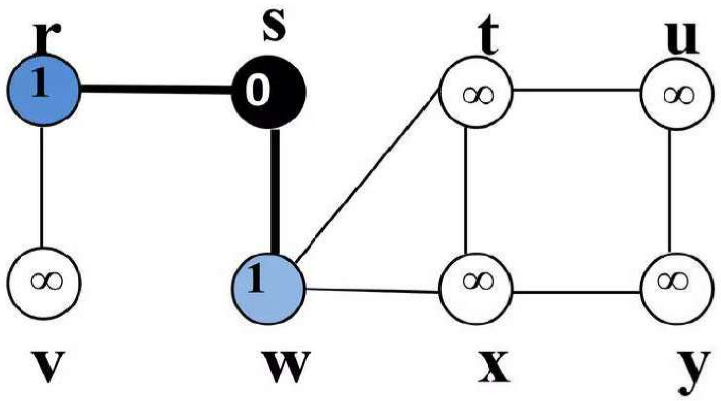
Note: If G is not connected, then BFS will not visit the entire graph (without some extra provisions in the algorithm)

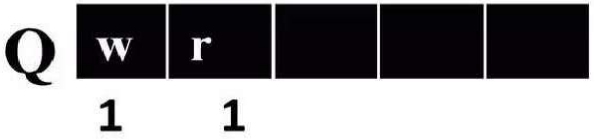
Graph G=(V, E)

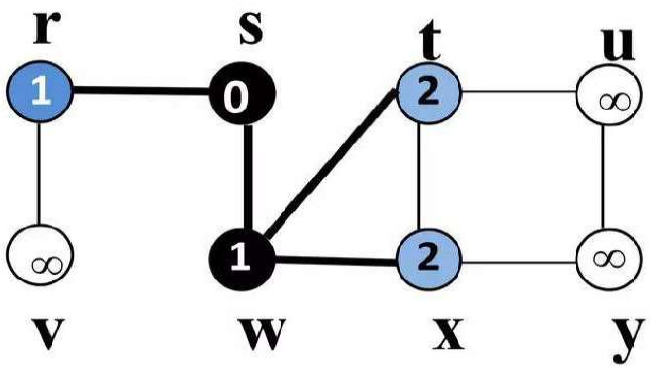



r
s
t
u
∞
0
∞
∞
∞
∞
∞
v
w
x
y

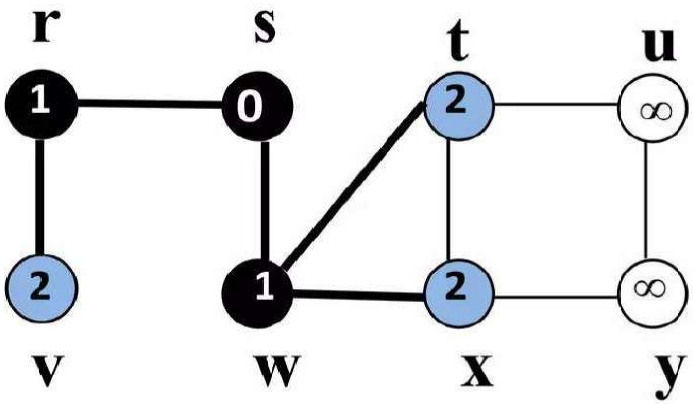

Q
s
0


r
s
t
u
1
0
∞
∞
∞
1
∞
∞
v
w
x
y

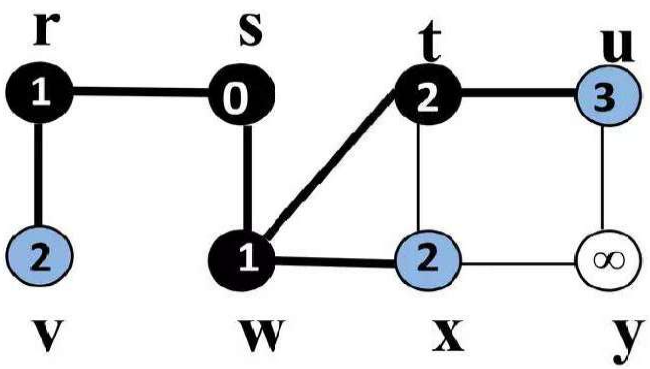

Q
w
r
1
1

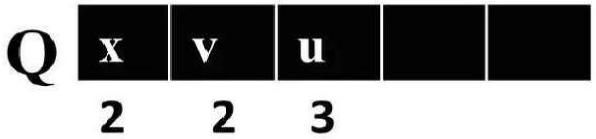


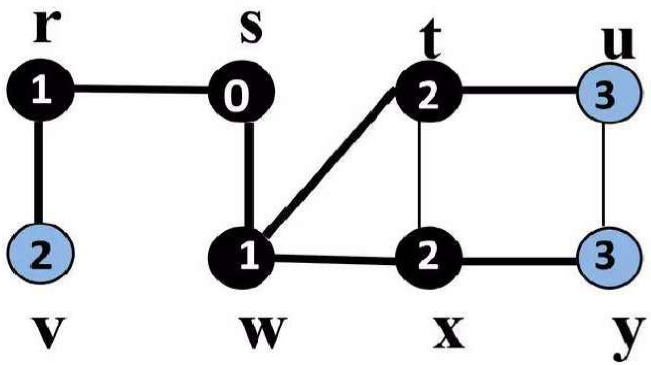
Q | r | t | x | | |
1 2 2

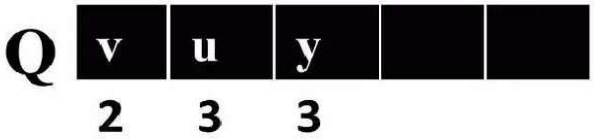


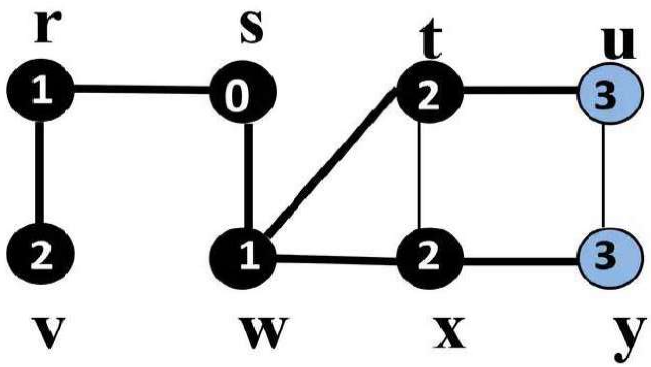
Q | t | x | v | | |
1 2 2

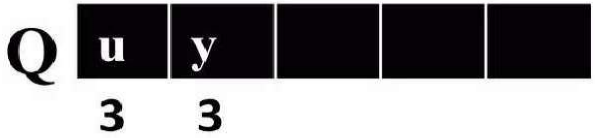

r
s
t
u
1
0
2
3
2
1
2
∞
v
w
x
y

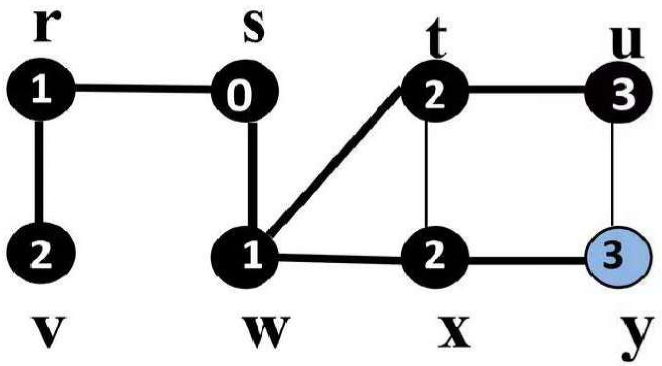

Q
x
v
u
2
2
3


r
s
t
u
1
0
2
3
2
1
2
3
v
w
x
y

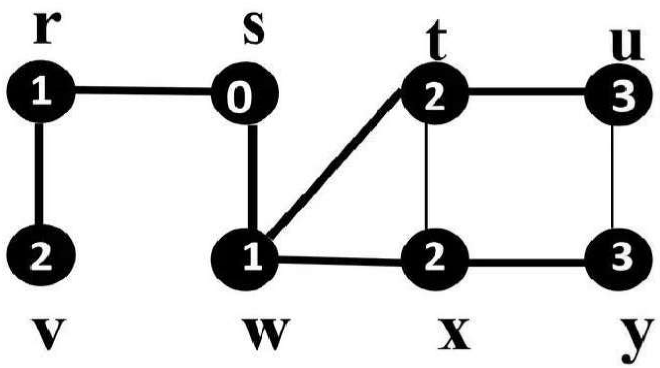

Q
v
u
y
2
3
3

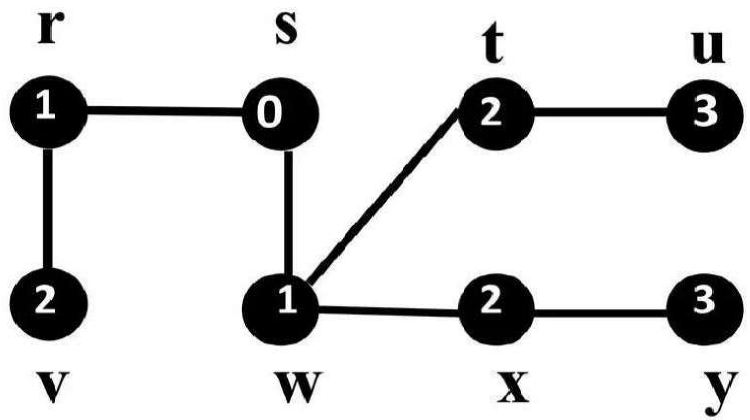

r
s
t
u
1
0
2
3
2
1
2
3
v
w
x
y


Q
u
y
3
3


r
s
t
u
1
0
2
3
2
1
2
3
v
w
x
y


Q
y
3





Spanning Tree

Running Time of BFS

```
BFS (G, s)
 0. For all i ≠ s, d[i]=∞; d[s]=0
 1. Enqueue (Q,s)
 2. while Q ≠ ∅
 3.     do u ← Dequeue(Q)
 4.     for each v adjacent to u
 5.         do if color[v] = white
 6.             then color[v] ← gray
 7.                  d[v] ← d[u] + 1
 8.                  π[v] ← u
 9.                  Enqueue(Q,v)
10.     color[u] ← black
```

Complexity

- each node enqueued and dequeued once = O(V) time
- each edge considered once (in each direction) = O(E) time

Running Time of BFS

We will denote the running time by T(m, n)

where m is the number of edges and n is the number of vertices.

Let V ′ subset of V be the set of vertices enqueued on Q.

Then,

$$T(m, n) = O(m+ n)$$

Analysis of Breadth-First Search

Shortest-path distance $\delta(s,v)$: minimum number of edges in any path from vertex s to v. If no path exists from s to v, then $\delta(s,v) = \infty$

The ultimate goal of the proof of correctness is to show that $d[v] = \delta(s,v)$ when the algorithm is done and that a path is found from s to all reachable vertices.

Analysis of Breadth-First Search

Lemma 1:
Let G = (V, E) be a directed or undirected graph, and let s be an arbitrary vertex in G. Then for any edge (u, v) $\in$ E,

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

Case 1: If u is reachable from s, then so is v. In this case, the shortest path from s to v can't be longer than the shortest path from s to u followed by the edge (u,v).

Case 2: If u is not reachable from s, then $\delta(s,u) = \infty$

In either case, the lemma holds.

Show that $d[v] = \delta(s,v)$ for every vertex v.

Lemma 2:

Let G = (V, E) be a graph, and suppose that BFS is run from vertex s. Then for each vertex v, the value $d[v] \geq \delta(s,v)$.

By induction on the number of enqueues (line 9).

Basis: When s is enqueued, $d[s] = 0 = \delta(s,s)$ and $d[v] = \infty \geq \delta(s,v)$ for all other nodes.

Ind.Step: Consider a white vertex v discovered from vertex u. implies that $d[u] \geq \delta(s,u)$. Then

$$\begin{aligned} d[v] &= d[u] + 1 \\ &\geq \delta(s,u) + 1 \\ &\geq \delta(s,v). \end{aligned}$$

Therefore, d[v] bounds $\delta(s,v)$ from above.

Lemma 3:
For every vertex $(v_1,v_2,...,v_r)$ on Q during BFS (such that v_1 is Q head and v_r is Q rear),

$d[v_r] \leq d[v_1] + 1$ and $d[v_i] \leq d[v_{i+1}]$ for i = 1,2,..., r-1.

By induction on the number of queue operations.

Basis: Q contains only s, so lemma trivially holds.

Ind.Step:

Case 1: Show lemma holds after every dequeue. Suppose v_1 is dequeued and v_2 becomes new head. Then

$d[v_1] \leq d[v_2]$

$d[v_r] \leq d[v_1] + 1$

$\leq d[v_2] + 1$, so lemma holds after every dequeue.

Lemma 3:
For every vertex $(v_1,v_2,...,v_r)$ on Q during BFS (such that v_1 is Q head and v_r is Q rear), $d[v_r] \leq d[v_1] + 1$ and $d[v_i] \leq d[v_{i+1}]$ for i = 1,2,..., r-1

Case 2: Show lemma holds after every enqueue. Suppose v is enqueued, becoming v_{r+1} after vertex u is dequeued (i.e., v is adjacent to u). Then for the new head v_1

$$\begin{aligned} d[u] &\leq d[v_1] \\ d[v_r] &\leq d[u] + 1 \\ &= d[v] \\ &= d[v_{r+1}] \end{aligned}$$

So after the enqueue, we have that $d[v_r] = d[v] = d[u] + 1 \leq d[v_1] + 1$.
Also, $d[v_r] \leq d[v_{r+1}]$ and for all other nodes on Q, $d[v_i] \leq d[v_{i+1}]$.
As a consequence of Lemma 3, if v_i is enqueued before v_j, then $d[v_i] \leq d[v_j]$ (also, if v_i is enqueued before v_j, then it is also dequeued before v_j by definition of a queue).

Theorem 4: (Correctness of BFS)

Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run from a given source vertex $s \in V$. Then, during execution, BFS discovers every vertex $v \neq s$ that is reachable from the source s, and upon termination, $d[v] = \delta(s,v)$ for every reachable or unreachable vertex v.

Proof by contradiction.

Assume that for some vertex v that $d[v] \neq \delta(s,v)$. Also, assume that v is the vertex with *minimum* $\delta(s,v)$ that receives an incorrect d value. By Lemma 2, it must be that $d[v] > \delta(s,v)$.

Case 1: v is not reachable from s. Then $\delta(s,v) = \infty = d[v]$. This is a contradiction, and the Thm holds.

Case 2: v is reachable from s. Let u be the vertex immediately preceding v on a shortest path from s to v, so that $\delta(s,v) = \delta(s,u) + 1$. Because $\delta(s,u) < \delta(s,v)$ and because v is the vertex with the *minimum* $\delta(s,v)$ that receives an incorrect d value, $d[u] = \delta(s,u)$.

So we have $d[v] > \delta(s,v) = \delta(s,u) + 1 = d[u] + 1$ (by L.1 and how we chose v as *minimum* $\delta(s,v)$ that receives an incorrect d value).

Consider the time t when u is dequeued. At time t, v is either white, gray, or black. We can derive a contradiction in each of these cases.

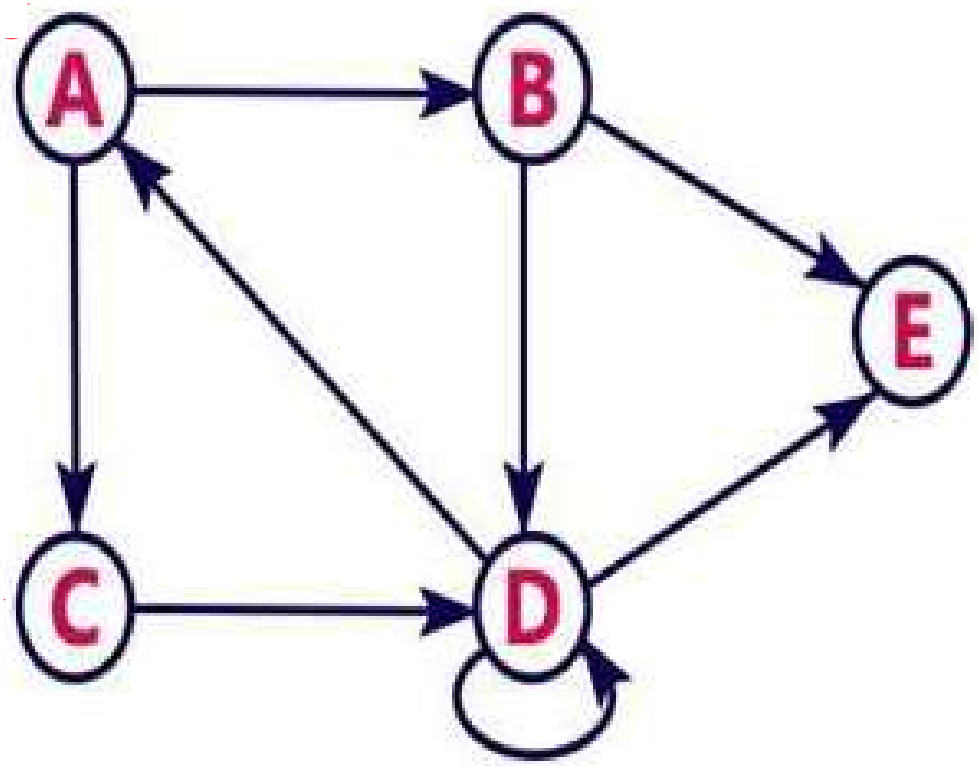
Case 1: v is white. Then in line 7, $d[v] = d[u]+1$.

Case 2: v is black. Then v was already dequeued, and therefore $d[v] \leq d[u]$ (by L. 3).

Case 3: v is gray. Then v turned gray when it was visited from some vertex w, which was dequeued before u.
Then $d[v] = d[w] + 1$. Since $d[w] \leq d[u]$ (by L 3), $d[v] \leq d[u] + 1$.

Each of these cases is a contradiction to $d[v] > \delta(s,v)$, so we conclude that $d[v] = \delta(s,v)$.

Breadth First Search(BFS) Example



BFS Traversal:

Q:[]

BFS Algorithm Complexity

- If the graph is represented as an **adjacency list**
  - Each vertex is enqueued and dequeued atmost once.
  - Each queue operation take O(1) time.
  - So the time devoted to the queue operation is **O(V).**

  - The adjacency list of each vertex is scanned only when the vertex is dequeued.
  - Each adjacency list is scanned at most once.
  - Sum of the lengths of all adjacency list is |E|.
  - Total time spend in scanning adjacency list is **O(E)**

  - Time complexity of BFS = O(V) + O(E) = **O(V+ E)**

BFS Algorithm Complexity

- If the graph is represented as an **adjacency list**
  - **In a dense graph:**
    - $E=O(V^2)$
    - Time complexity= $O(V) + O(V^2) = \mathbf{O(V^2)}$

- If the graph is represented as an **adjacency matrix**
  - There are $|V|^2$ entries in the adjacency matrix. Each entry is checked once
  - Time complexity of BFS = $\mathbf{O(V^2)}$

BFS Applications

- Finding shortest path between 2 nodes u and v, with path length measured by number of edges
- Testing graph for bipartiteness
- Minimum spanning tree for unweighted graph
- Finding nodes in any connected component of a graph
- Serialization/deserialization of a binary tree
- Finding nodes in any connected component of a graph

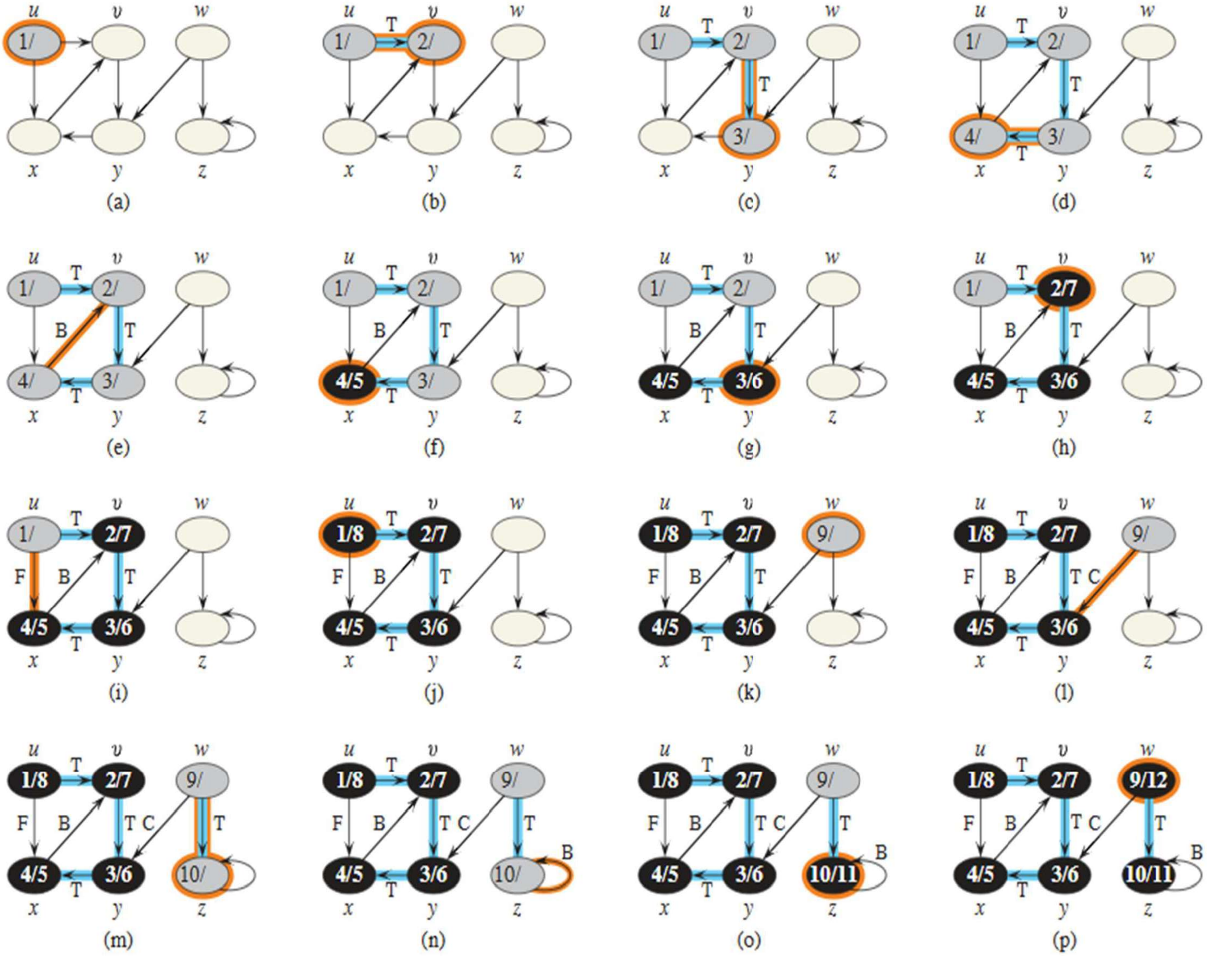
Depth First Search(DFS) Algorithm

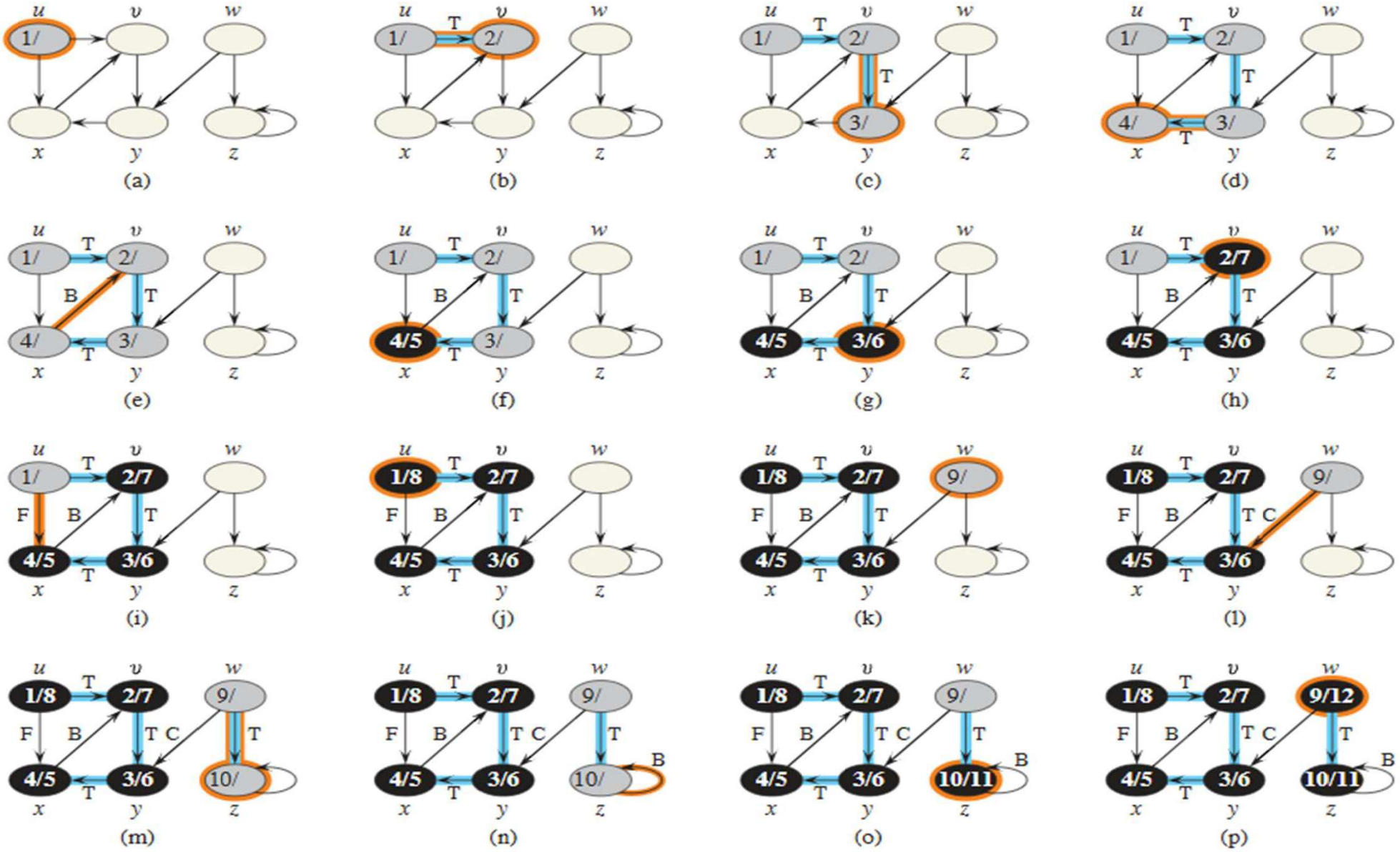
- **DFS forest** means the **collection of DFS trees** produced when Depth-First Search is applied to a graph.
- Depth-First Search does **not always produce a single tree**.
  - If the graph is **connected** → DFS produces **one DFS tree**
  - If the graph is **disconnected** → DFS produces **multiple DFS trees**
- The **set of all these trees together** is called a **DFS forest**.
- Each node has fields for predecessor (π), discovery time (d), finish time (f), and color (c)

DFS

- Start from i, visit a neighbour j
- Suspend the exploration of i and explore j instead
- Continue till you reach a vertex with no unexplored neighbours
- Backtrack to nearest suspended vertex that still has an unexplored neighbour

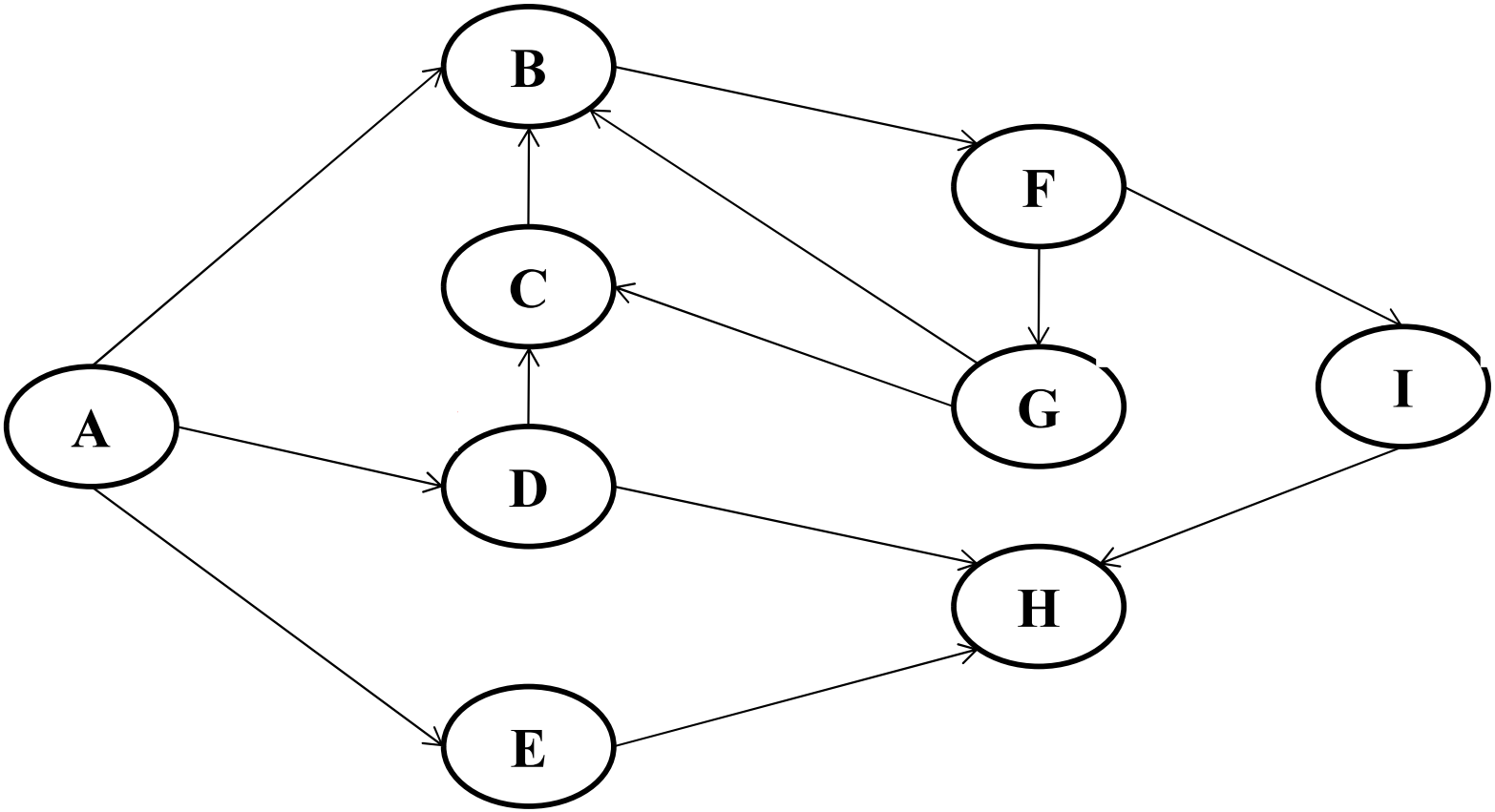
```
DFS(G)
 1  for each vertex u ∈ G.V
 2      u.color = WHITE
 3      u.π = NIL
 4  time = 0
 5  for each vertex u ∈ G.V
 6      if u.color == WHITE
 7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
 1  time = time + 1                  // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each vertex v in G.Adj[u]    // explore each edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  time = time + 1
 9  u.f = time
10  u.color = BLACK                  // blacken u; it is finished
```


u v w
x y z
1/
2/
3/
4/
4/5
3/6
2/7
1/8
9/
10/
9/12
10/11
T
B
F
C
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
(m)
(n)
(o)
(p)



The progress of the DFS on a directed graph. Edges are classified as they are explored: tree edges are labelled T, back edges B, forward edges F, and cross edges C. Timestamps within vertices indicate discovery time / finish times. Tree edges are highlighted in blue. Orange highlights indicate vertices whose discovery or Finish times change and edges that are explored in each step.

Depth First Search(DFS) Example



DFS Algorithm Complexity

- If the graph is represented as an **adjacency list**
  - Each vertex is visited atmost once. So the time devoted is **O(V)**
  - Each adjacency list is scanned atmost once. So the time devoted is **O(E)**
  - Time complexity = **O(V+ E).**

- If the graph is represented as an **adjacency matrix**
  - There are $|V|^2$ entries in the adjacency matrix. Each entry is checked once.
  - Time complexity = $\mathbf{O(V^2)}$

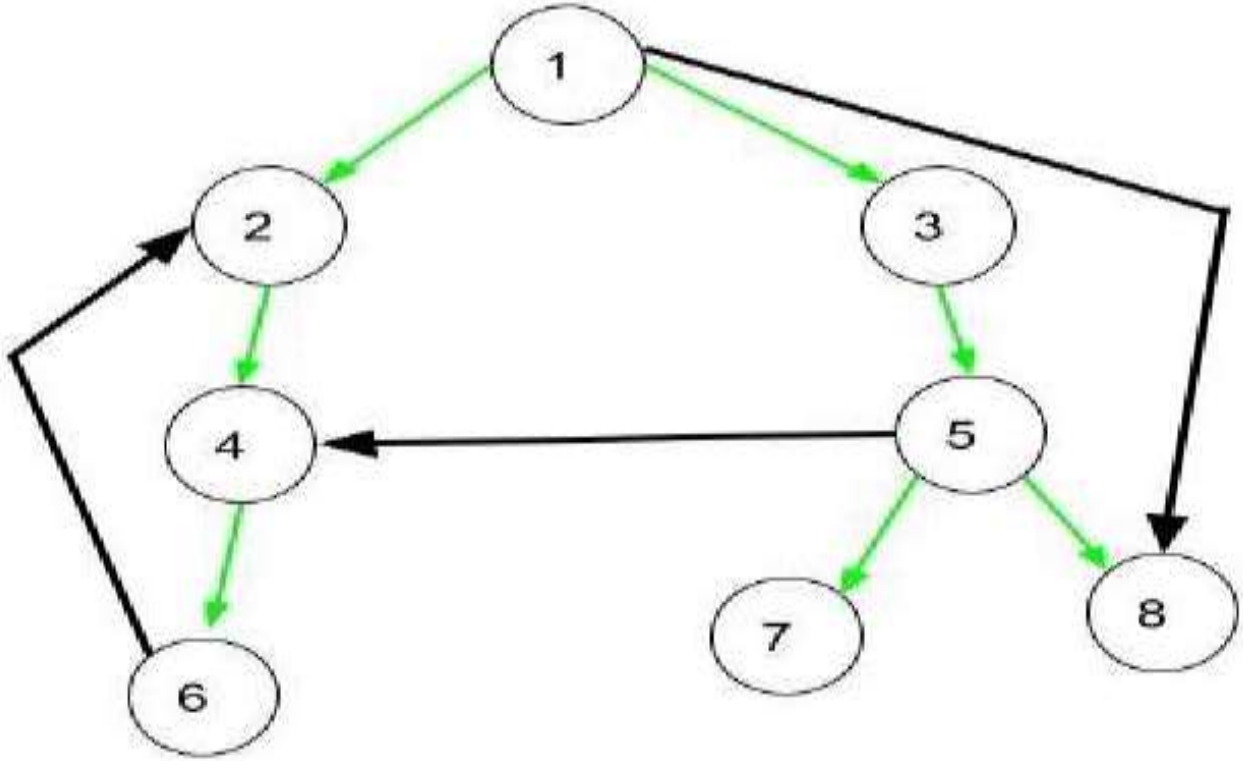
Applications of DFS

- Finding connected components in a graph
- Topological sorting in a DAG
- Scheduling problems
- Cycle detection in graphs
- Finding 2-(edge or vertex)-connected components
- Finding 3-(edge or vertex)-connected components
- Finding the bridges of a graph
- Finding strongly connected components
- Solving puzzles with only one solution, such as mazes
- Finding biconnectivity in graphs

Classification of edges

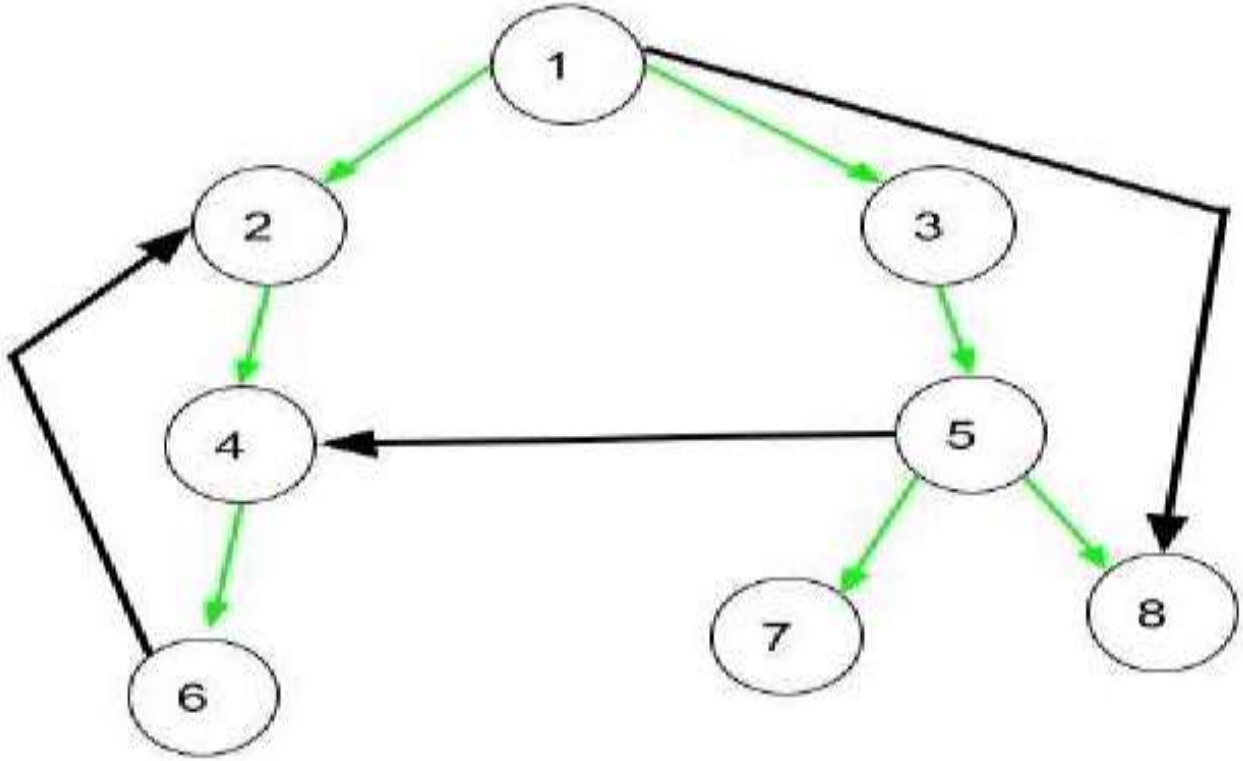
1. ***Tree edges*** are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
2. ***Back edges*** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.
3. ***Forward edges*** are those nontree edges (u, v) connecting a vertex u to a proper descendant v in a depth-first tree.
4. ***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 Based on DFS



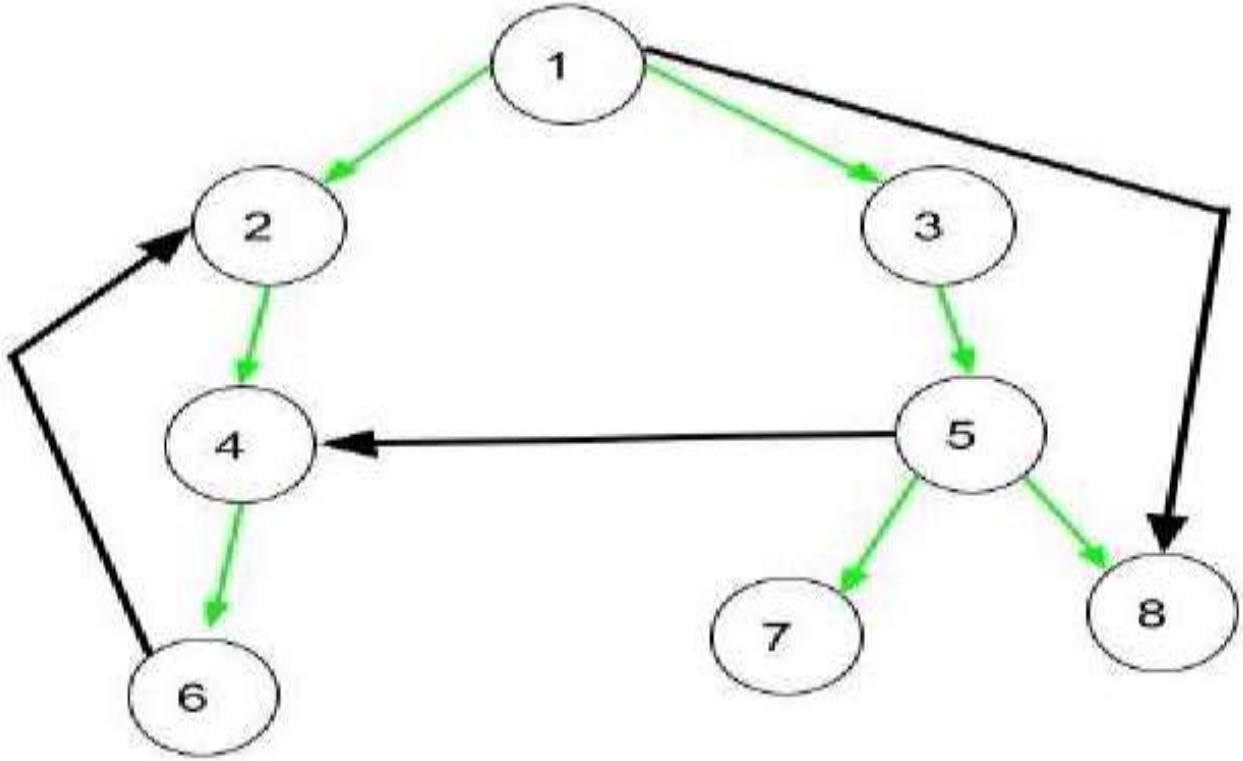
- The DFS traversal of the above graph is 1 2 4 6 3 5 7 8

Classification of Edges Based on DFS



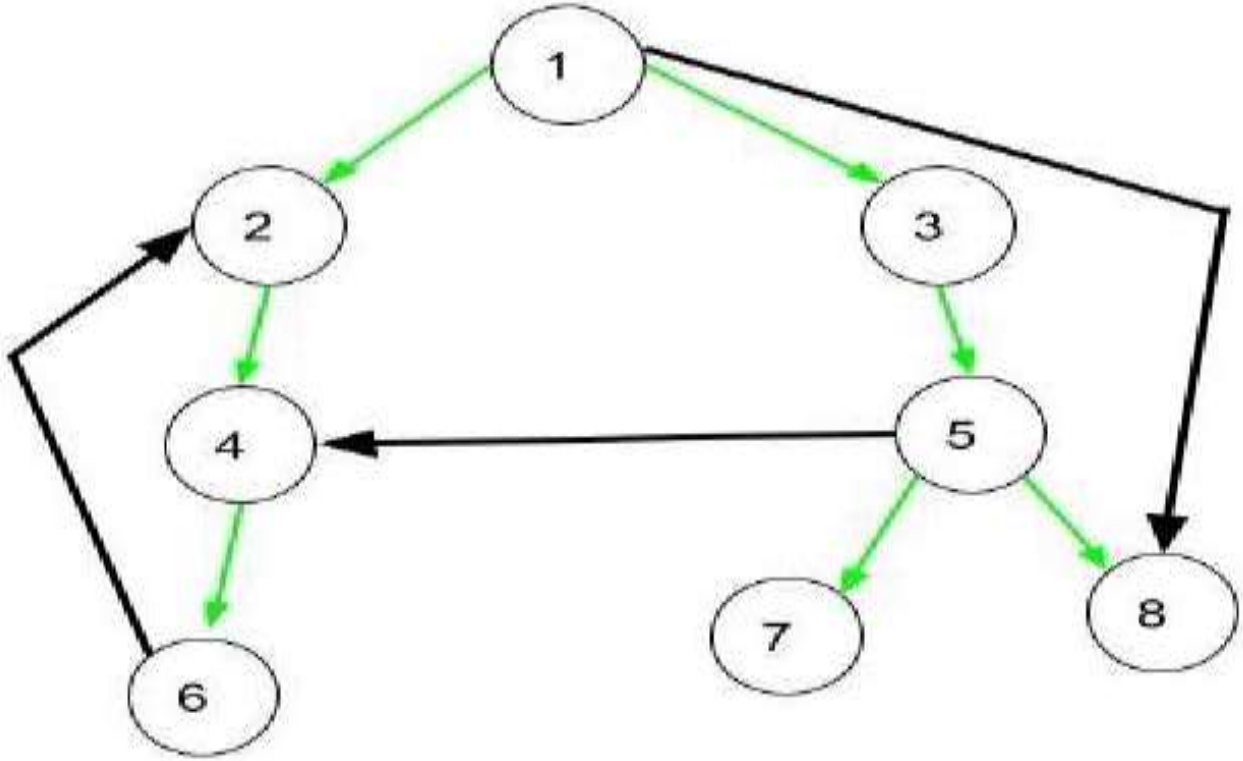
- The DFS traversal of the above graph is 1 2 4 6 3 5 7 8
- **Tree Edge:** It is a edge in tree obtained after applying DFS on the graph
  - Eg: (1,2), (2,4), (4,6), (1,3), (3,5), (5,7) and (5,8)

Classification of Edges Based on DFS



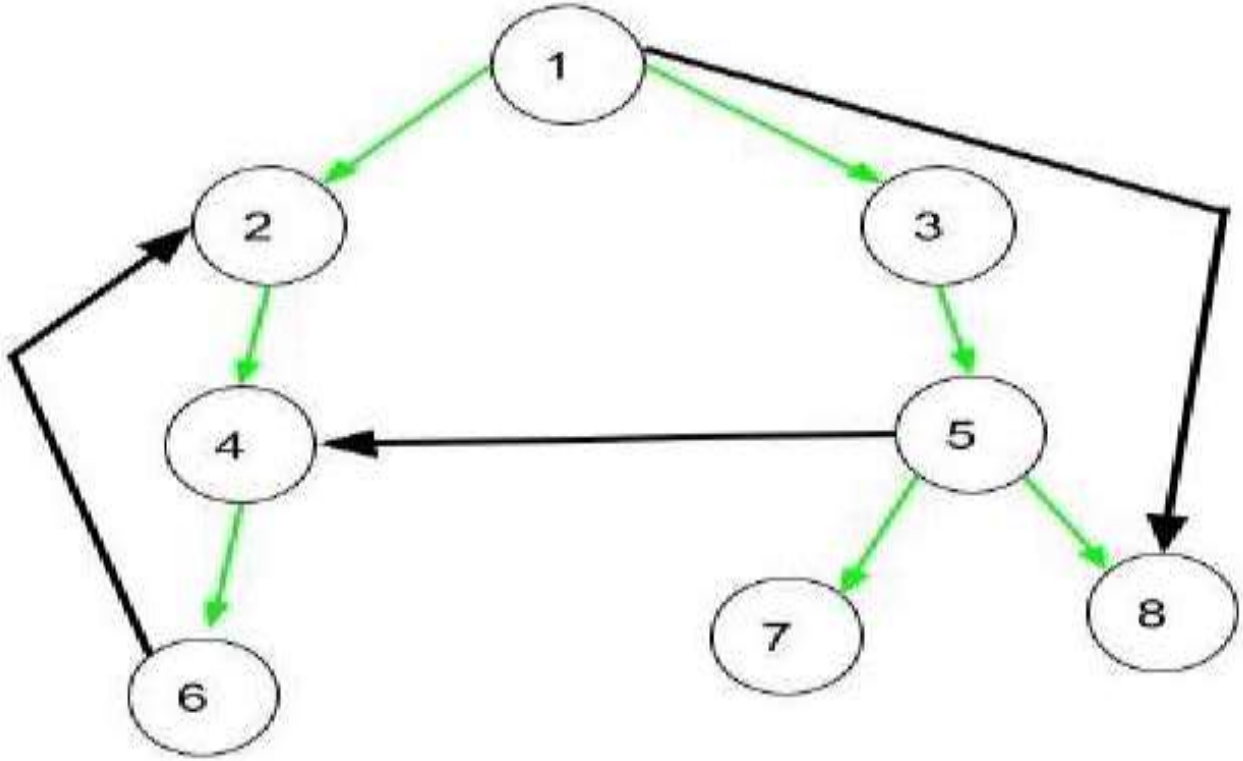
- The DFS traversal of the above graph is 1 2 4 6 3 5 7 8
- **Forward Edge:** It is an edge (u, v) such that v is descendant but not part of the DFS tree
  - Eg: (1,8)

Classification of Edges Based on DFS



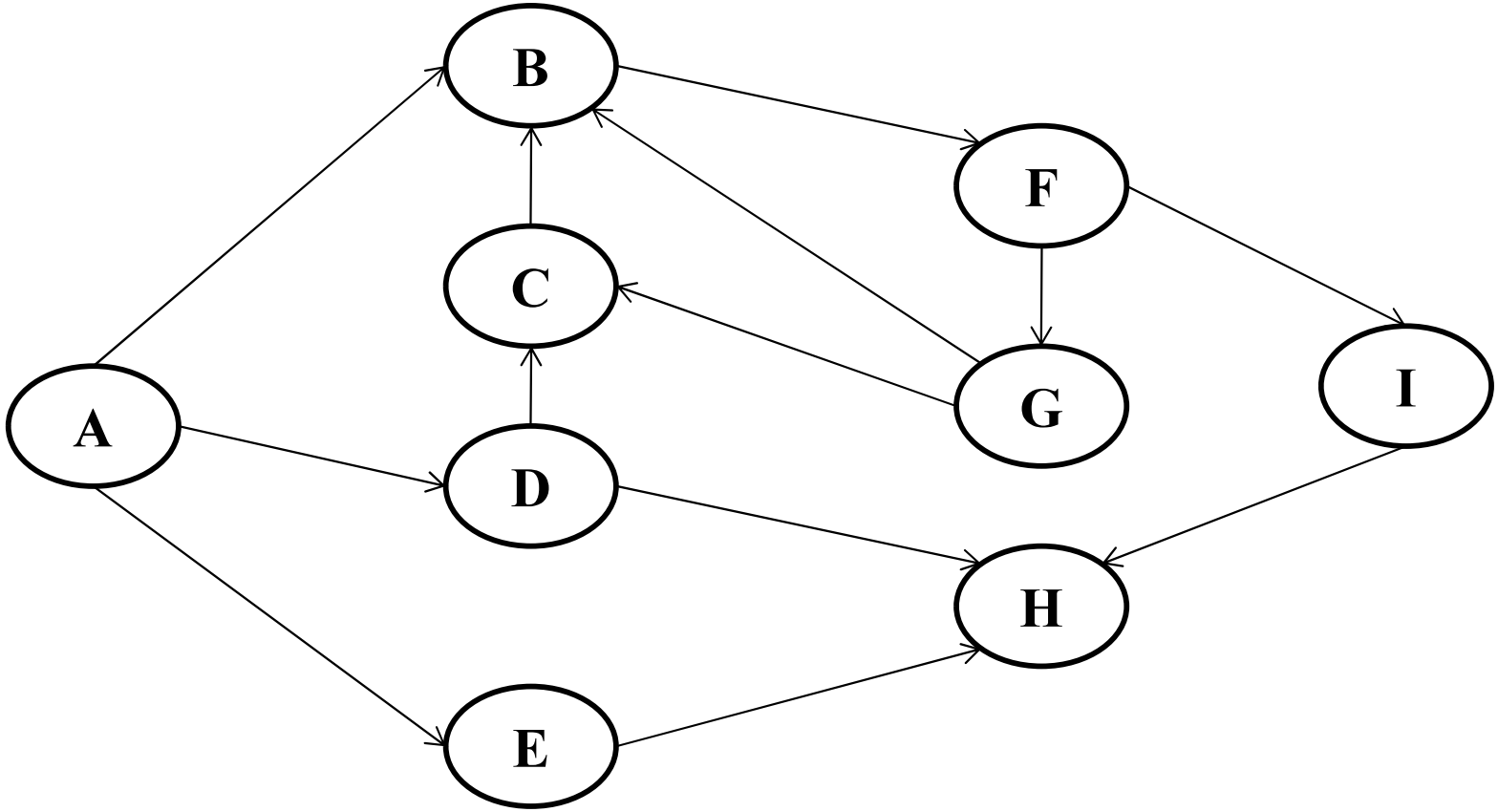
- The DFS traversal of the above graph is 1 2 4 6 3 5 7 8
- **Backward Edge:** It is an edge (u, v) such that v is ancestor of edge u but not part of DFS tree
  - Eg: (6,2)

Classification of Edges Based on DFS

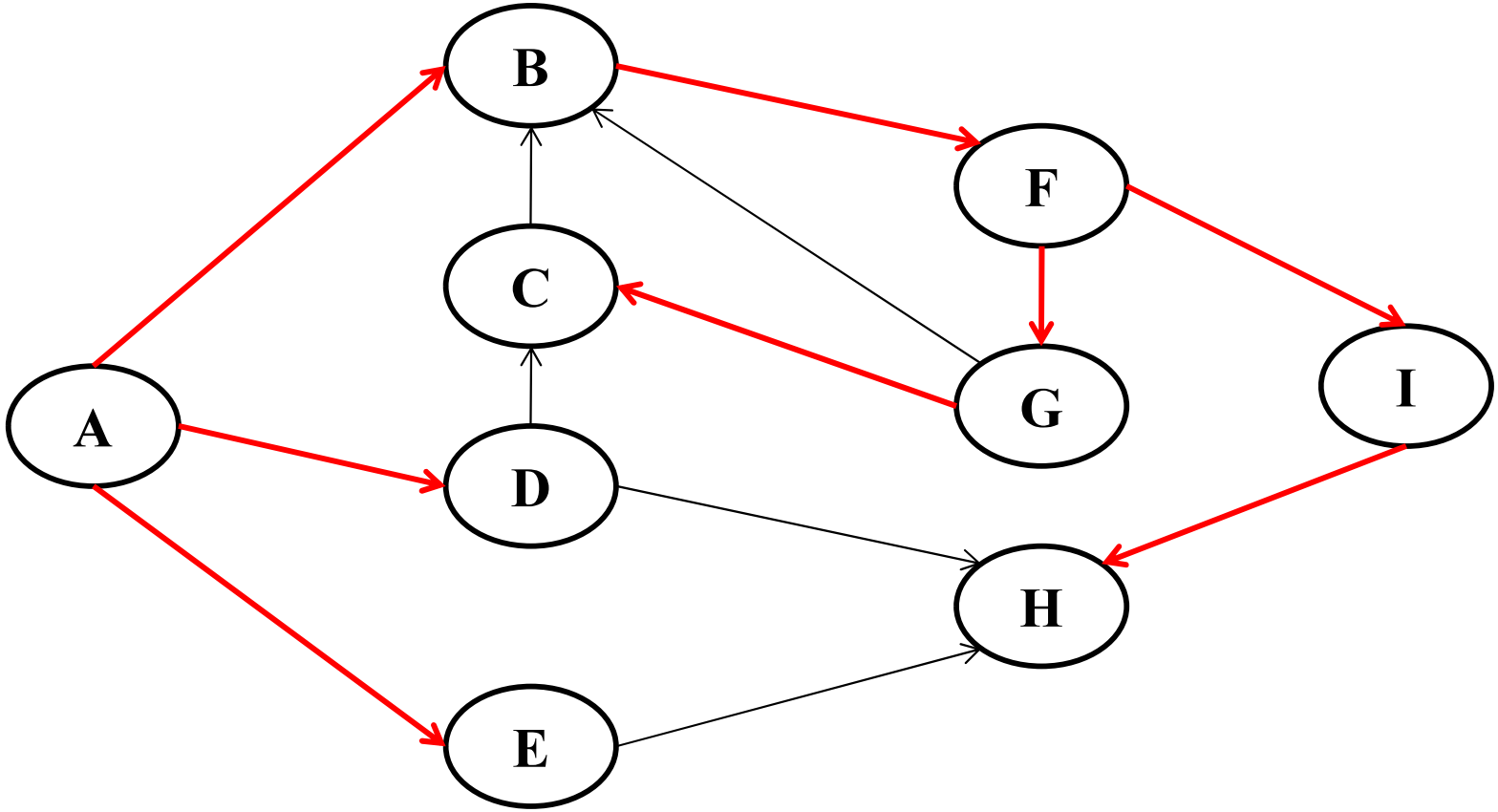


- The DFS traversal of the above graph is 1 2 4 6 3 5 7 8
- **Cross Edge:** It is a edge which connects two node such that they do not have any ancestor and a descendant relationship between them.
  - Eg: (5,4)

Classify the edges of the given graph

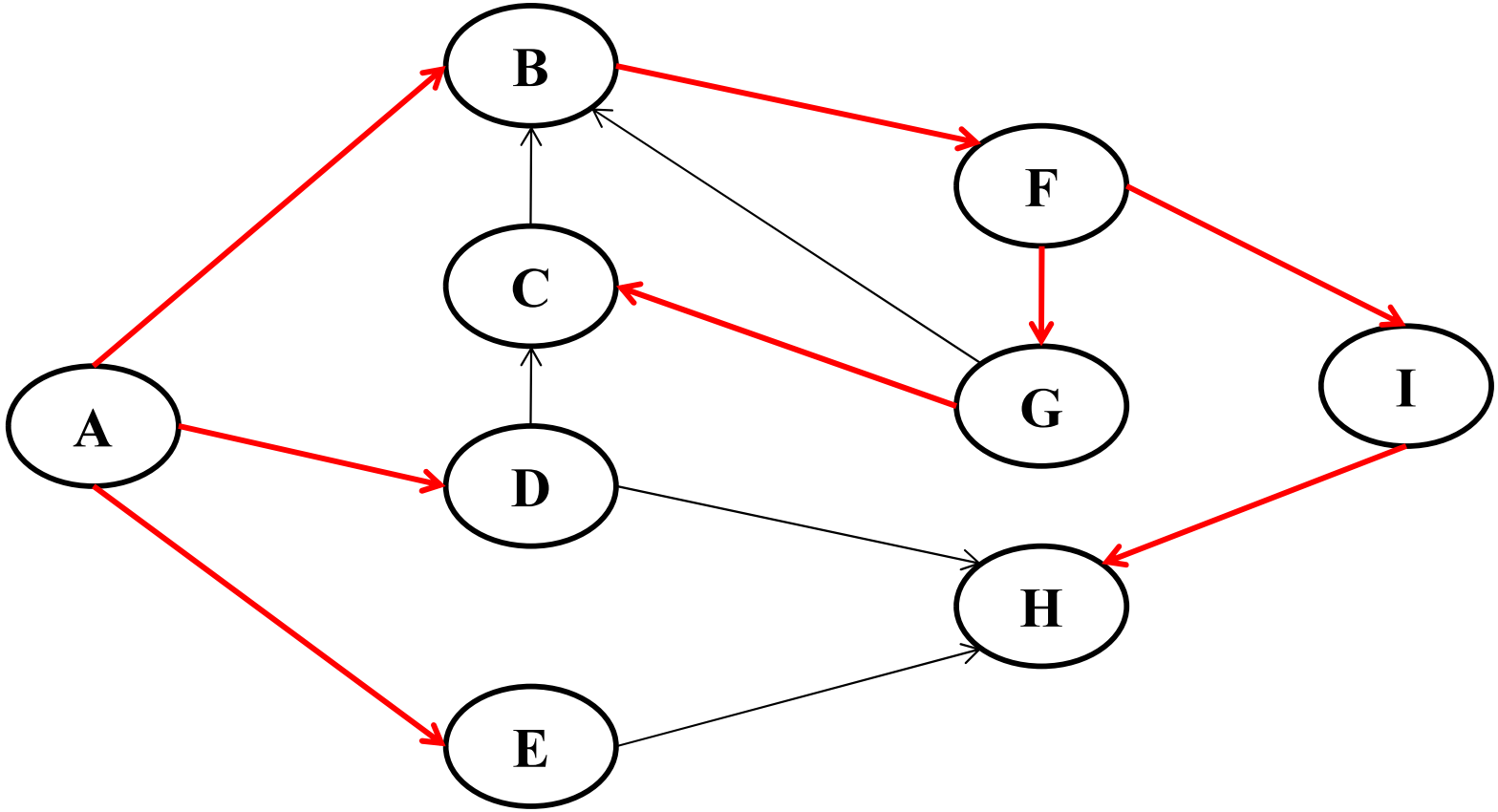


Classify the edges of the given graph



DFS Traversal : A B F G C I H D E

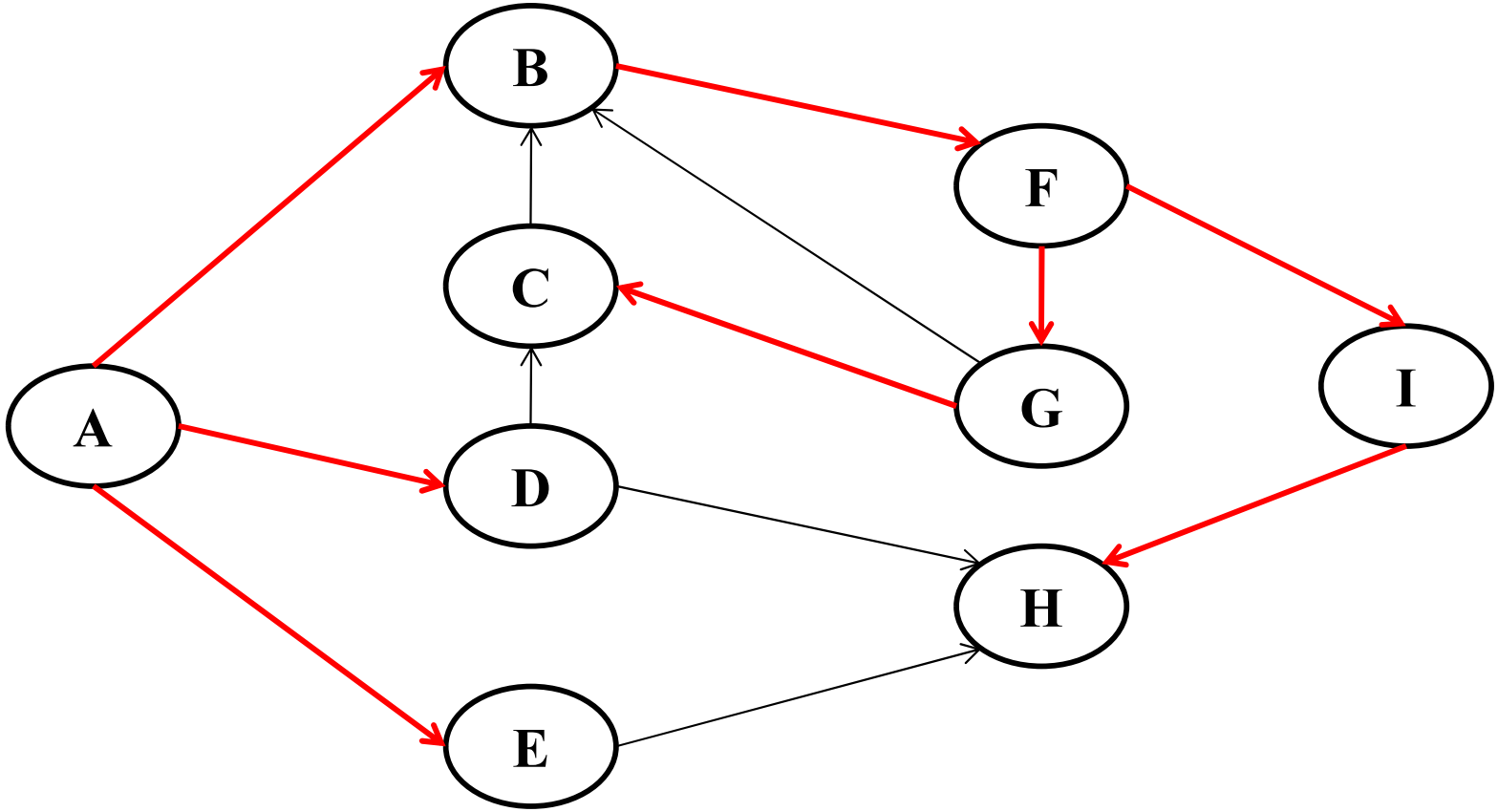
Classify the edges of the given graph



DFS Traversal : A B F G C I H D E

Tree Edge :(A,B),(B,F),(F,G),(G,C),(F,I),(I,H),(A,D),(A,E)

Classify the edges of the given graph

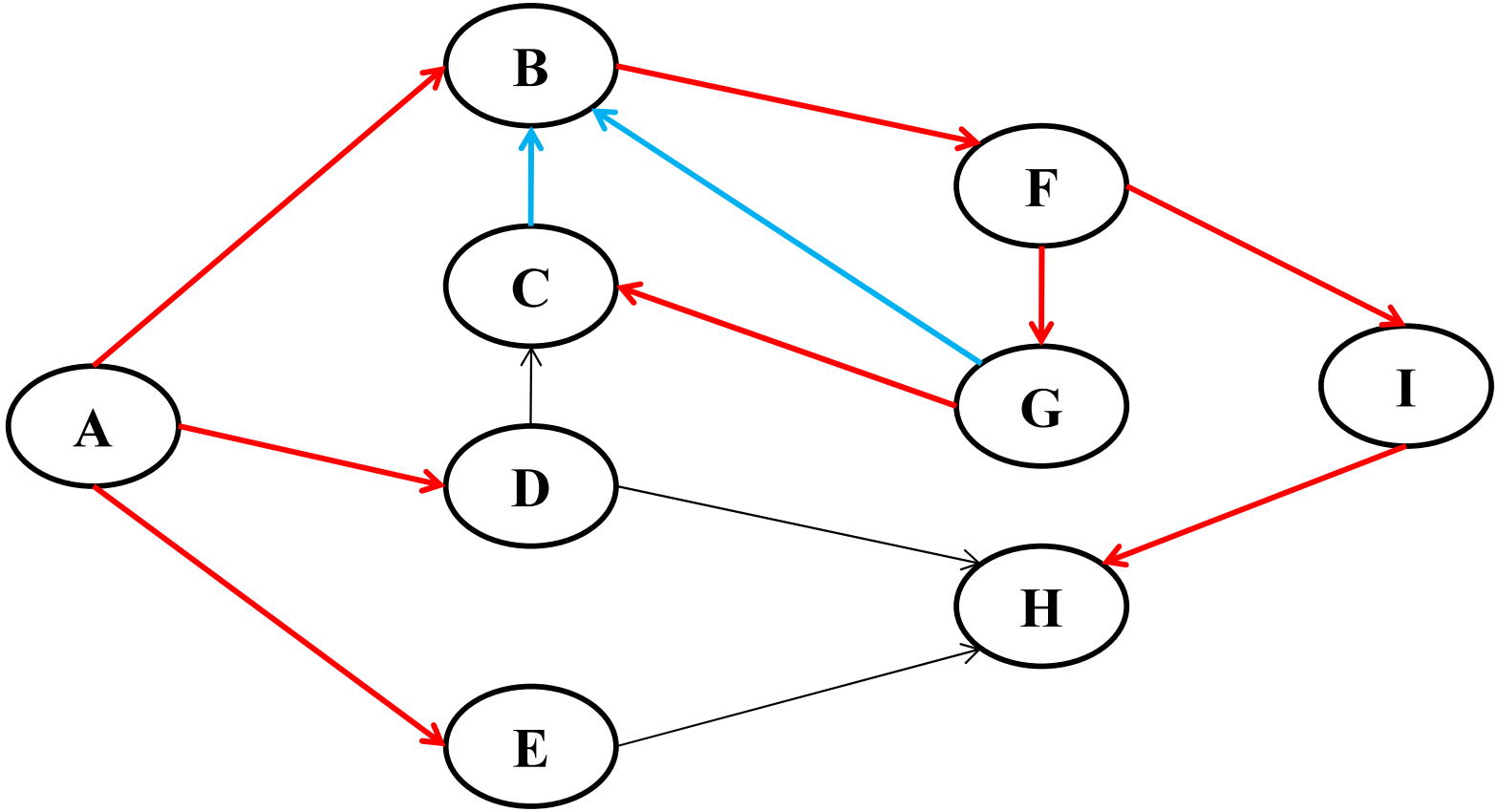


DFS Traversal : A B F G C I H D E

Tree Edge :(A,B),(B,F),(F,G),(G,C),(F,I),(I,H),(A,D),(A,E)

Forward Edge : --

Classify the edges of the given graph



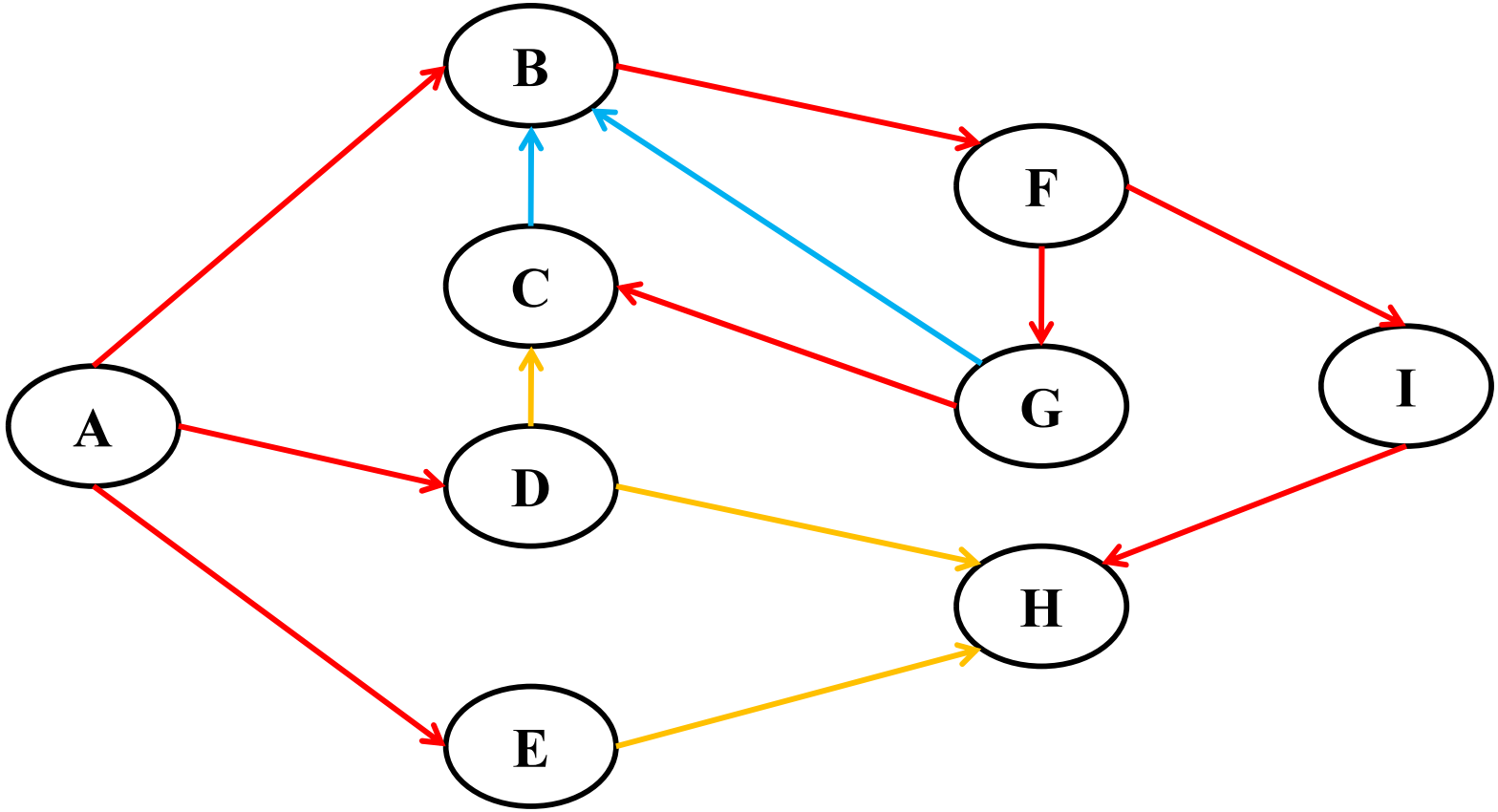
DFS Traversal : A B F G C I H D E

Tree Edge :(A,B),(B,F),(F,G),(G,C),(F,I),(I,H),(A,D),(A,E)

Forward Edge : --

Backward Edge :(G,B),(C,B)

Classify the edges of the given graph



DFS Traversal : A B F G C I H D E

Tree Edge :(A,B),(B,F),(F,G),(G,C),(F,I),(I,H),(A,D),(A,E)

Forward Edge : --

Backward Edge :(G,B),(C,B)

Cross Edge :(D,C),(D,H),(E,H)