JAIN DEEMED-TO-BE UNIVERSITY | SCHOOL OF COMPUTER SCIENCE AND IT

Master of Computer Applications

Data Structures

Module 5

GRAPH DATA STRUCTURES

SCHOOL OF COMPUTER SCIENCE AND IT

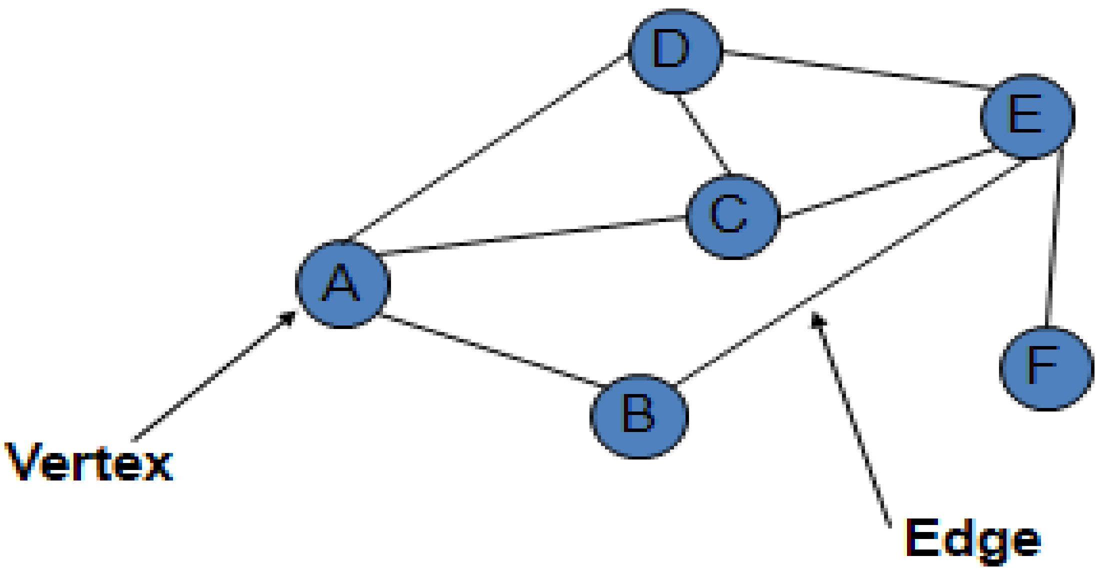
Syllabus Contents – Graphs ADT

- Introduction to graph representation and Terminology
- Types of Graphs
- Graph traversal using Stack and Queue
- Applications of Depth First and Breadth First Traversal
- Applications of graph,
- Detect Cycle in a Directed Graph and in an undirected graph,
- Transitive Closure of a Graph using DFS.
- Topological sorting of Directed Acyclic Graphs.

Graphs

- Consist of:
  - Vertices
  - Edges
  - it is an ordered pair of sets G(V,E) is called Graph.

Extremely useful tool in modeling problems



Vertices can be considered "sites" or locations.

Edges represent connections.

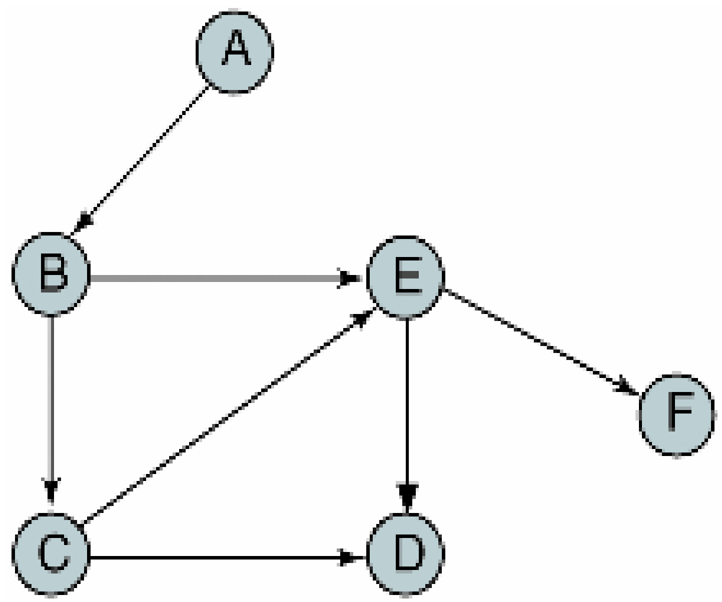
Types of Graphs

SCHOOL OF COMPUTER SCIENCE AND IT

- Directed Graphs
  - Where there **is direction arrow** at the **end of the edge**
- Undirected Graphs
  - **No arrows in edges** (Bi-direction between two nodes)
- Weighted Graphs
  - **Cost will be assigned** in each edge in the graph

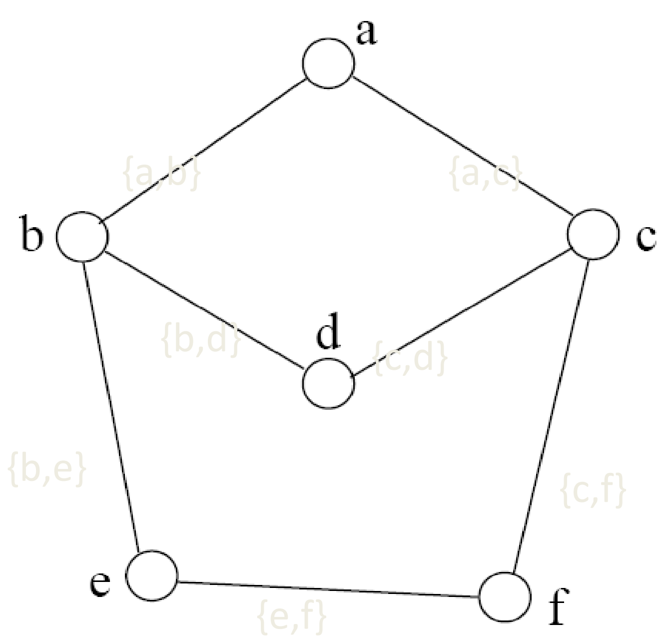
Directed Graph

- A graph is directed if **direction is assigned to each edge**. We call the directed edges ***arcs*.**
  - An edge is denoted as an ordered pair (u, v)



UnDirected Graph

An undirected graph is **specified by an ordered pair** (V,E), where V is the **set of vertices** and E is the **set of edges**

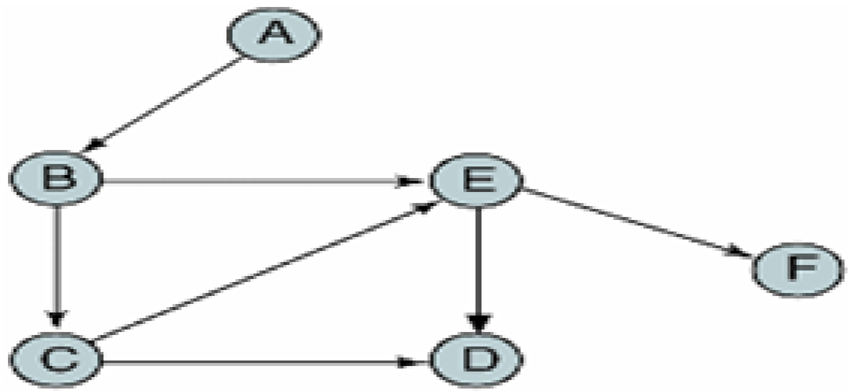


$$V = \{a, b, c, d, e, f\}$$

$$E = \{\{a, b\}, \{a, c\}, \{b, d\}, \{c, d\}, \{b, e\}, \{c, f\}, \{e, f\}\}$$

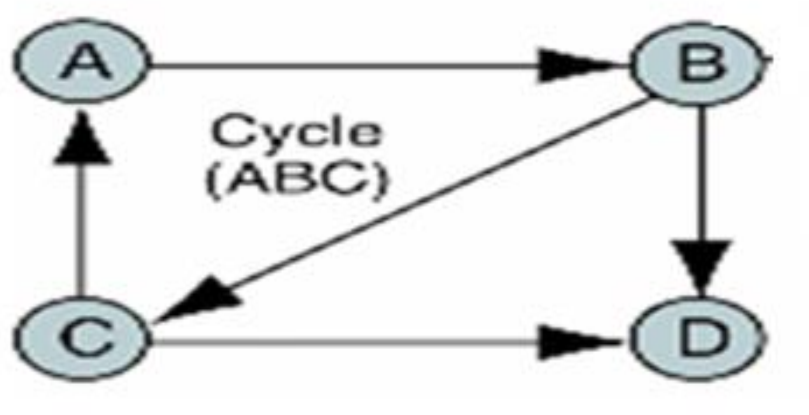
Few more Types of Graph

1. Acyclic: No circuit format of the Graph.



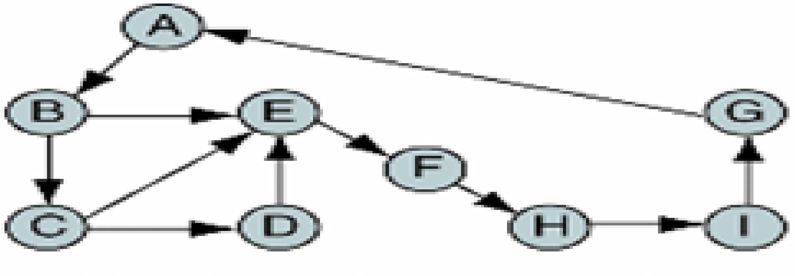
(a) **No cycle is found**

2. Cycle: A cycle is a path along the directed edges from a vertex to itself.



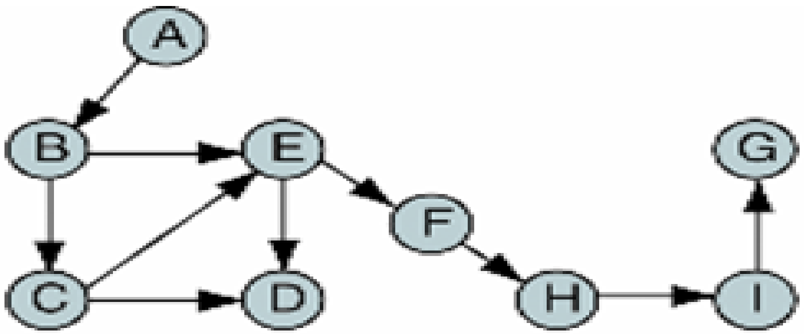
Few more Types of Graph

3. Strongly Connected Graph: It is strongly connected if there is a path from each vertex to every other vertex, considering direction.



(b) Strongly connected

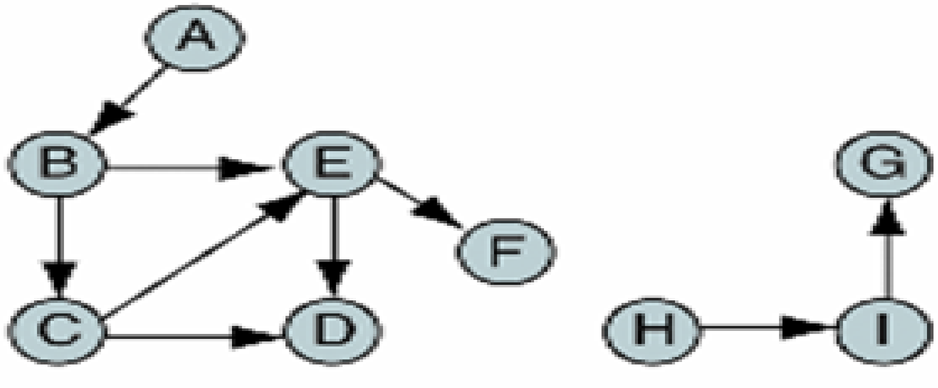
4. Weakly Connected Graph: It is strongly connected if there is a path from each vertex to every other vertex, considering direction, otherwise, it is weakly connected.



(a) Weakly connected

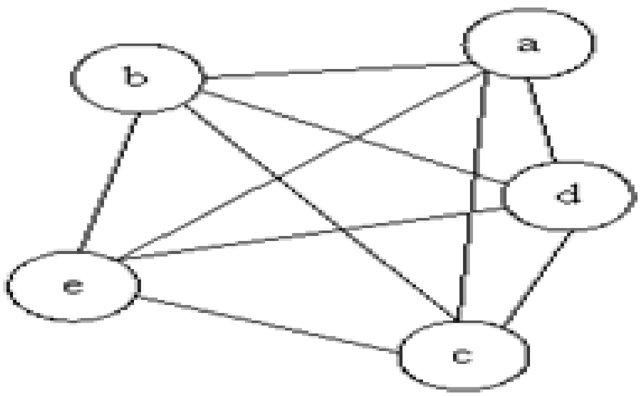
Few more Types of Graph

5. Disjoint Graph: A graph is disjoint if it is not connected



(c) Disjoint graph

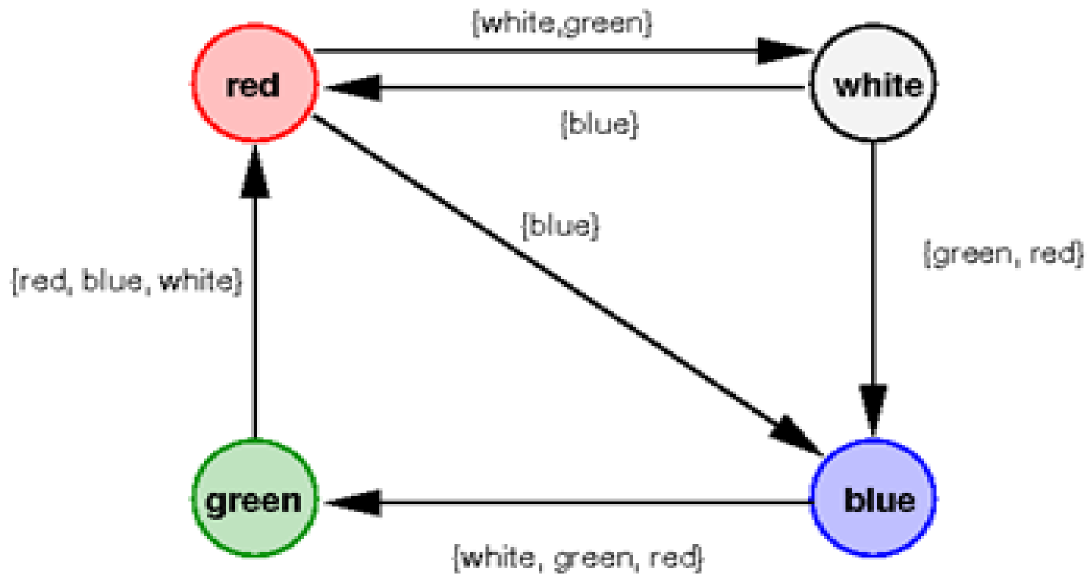
6. Complete Graph: a graph that has the maximum number of edges, A graph in which every vertex is directly connected to every other vertex.



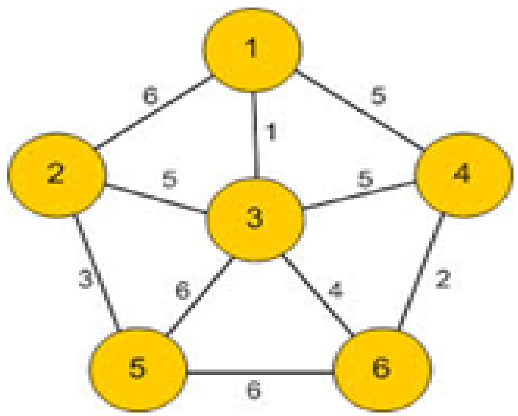
(e)

Few more Types of Graph

7. Labeled Graph: The Graph having names of the edges

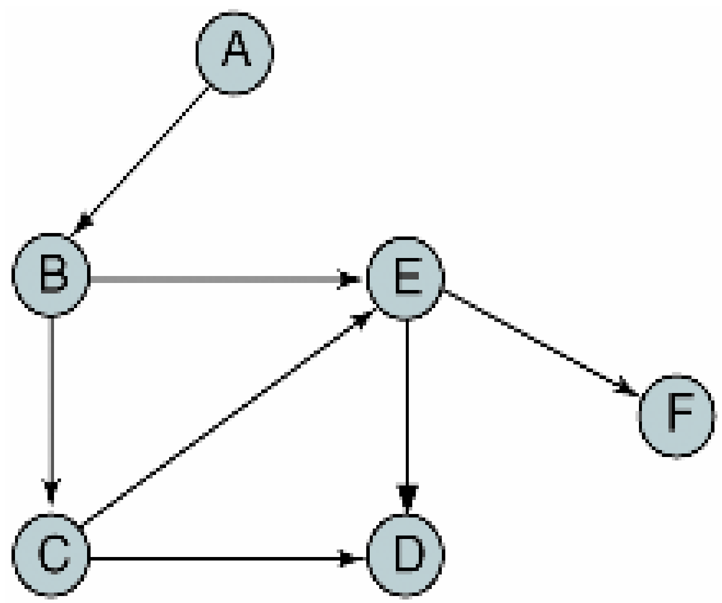


8. Weighted Graph: The Graph having weights of the edges, A graph in which each edge carries a value.



Graph Terminologies

1. Path: A **sequence of vertices** that **connects two nodes** in a graph.



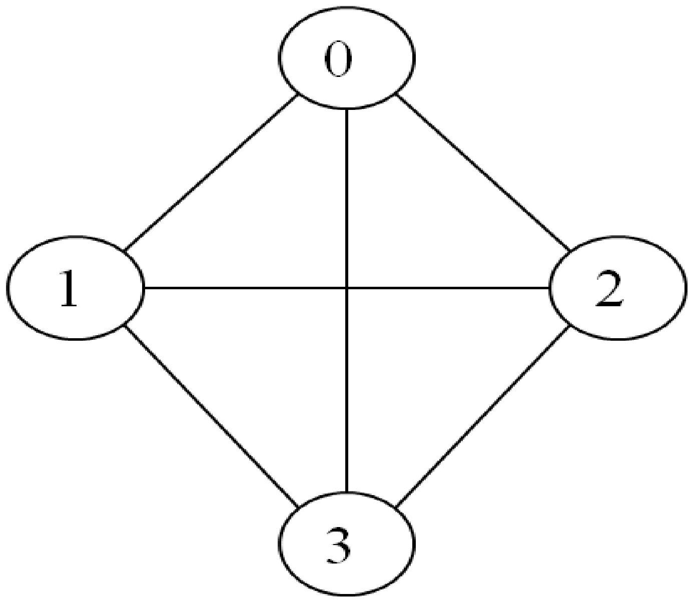
Path **A->B->C->E->F,**
A->B->E->F

Graph Terminologies

2. Adjacent Vertices:

Two vertices in a graph that are **connected by an edge**. Sequence of vertices in which **each vertex is adjacent to next one**. Two vertices are adjacent (or **neighbors**) **if there is a direct path** connecting them

- ✓1 is adjacent to 0
- ✓1 is adjacent to 3
- ✓1 is adjacent to 2



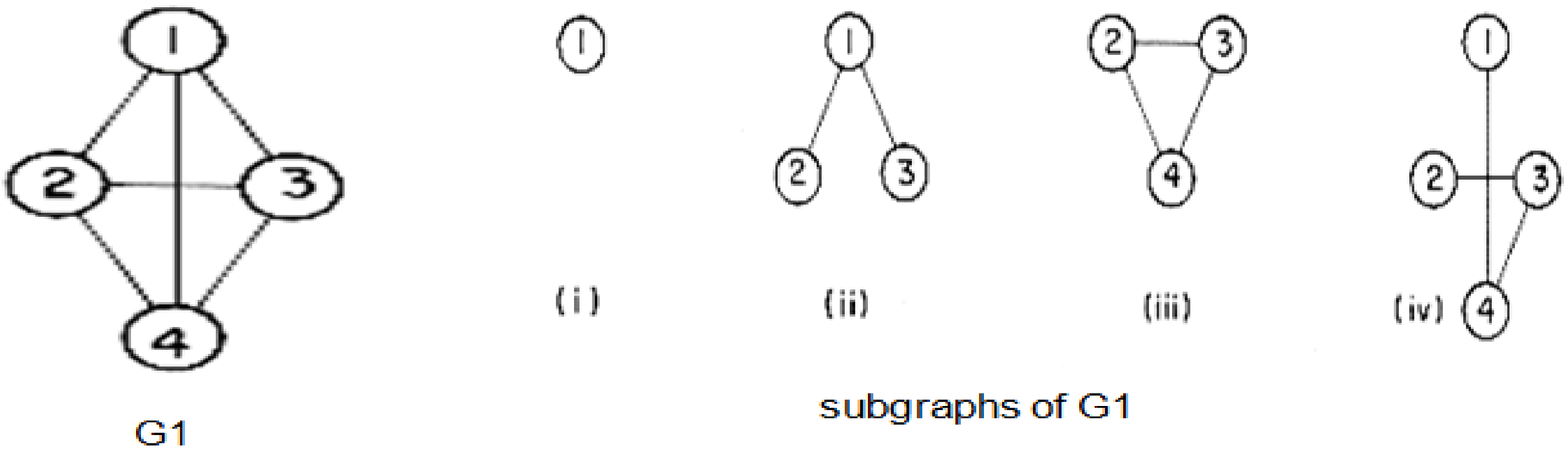
G1

V(G1)={0,1,2,3}
E(G1)={(0,1), (0,2), (0,3),(1,2), (1,3), (2,3)}

Graph Terminologies

3) Subgraph: G'(V',E') is subgraph of G(V, E)

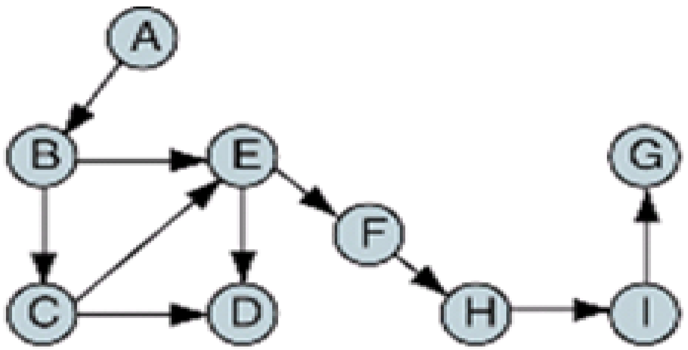
- $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$



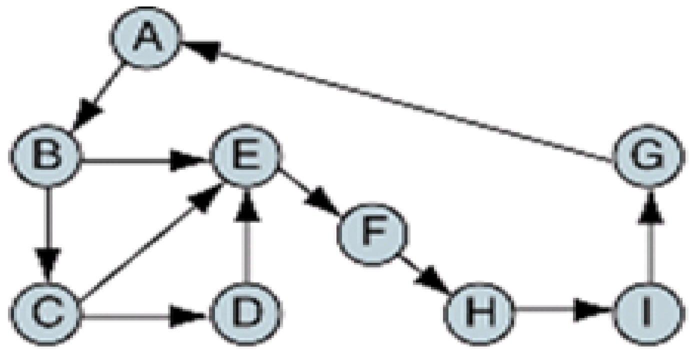
Graph Terminologies

4) Degree of Graph: The degree of a vertex is the number of lines incident to it.

- In Figure (a), the degree of vertex B is 3 and the degree of vertex E is 4



(a)

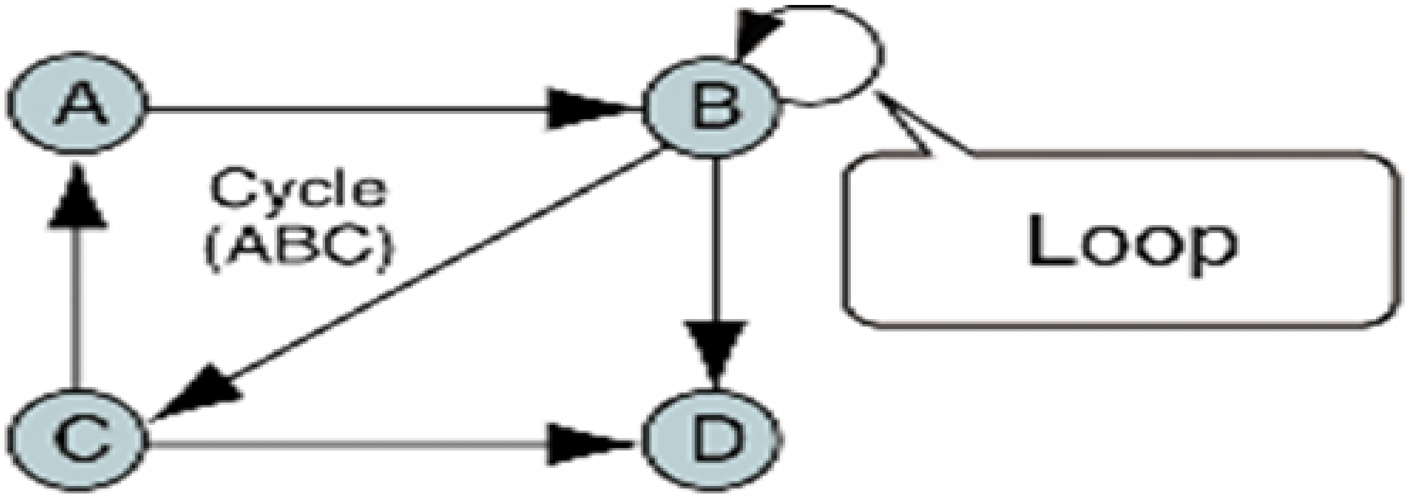


(b)

- The outdegree and indegree of a vertex
  - In Figure (a), the indegree of vertex B is 1 and its outdegree is 2
  - In Figure (b), the indegree of vertex E is 3 and its outdegree is 1

Graph Terminologies

5. Loop: single arc begins and ends at the same vertex.



(c)

ABC is cycle

Graph Representation

- Two **popular computer representations techniques are available,** but both represent the **vertex set and the edge set** in different ways.

1. Adjacency Matrix

   Use a **2D matrix** to represent the graph

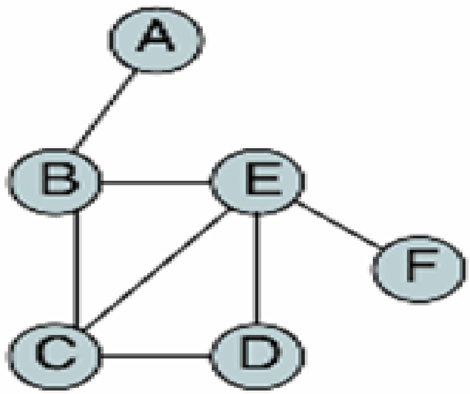
2. Adjacency List

   Use a **1D array of linked lists**

Adjacency Matrix

Adjacency Matrix with UnDirected Graph:

A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices. Data on edges and vertices must be stored externally. Only the cost for one edge can be stored between each pair of vertices.



Vertex vector
A
B
C
D
E
F

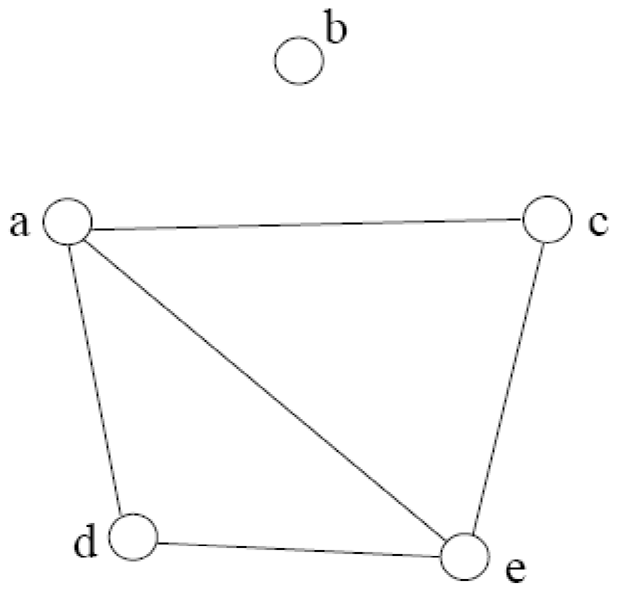
Vertex vector

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

Adjacency matrix

(a) Adjacency matrix for nondirected graph

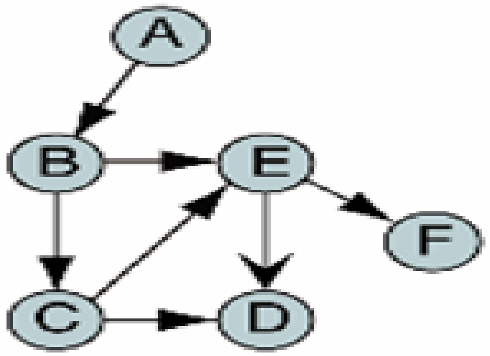
Adjacency Matrix



	a	b	c	d	e
a	0	0	1	1	1
b	0	0	0	0	0
c	1	0	0	0	1
d	1	0	0	0	1
e	1	0	1	1	0

- 2D array A[0..n-1, 0..n-1], where ***n* is the number of vertices** in the graph
- Each **row and column** is indexed by the **vertex id**.
  - e,g **a=0, b=1, c=2, d=3, e=4**
- An **array entry A [i] [j] is equal to 1** if there is an edge connecting vertices i and j. Otherwise, **A [i] [j] is 0.**
- The storage requirement is **$\Theta(n^2)$. Not efficient if the graph has few edges**.
- We can **detect in O(1) time** whether **two vertices are connected**.

Adjacency Matrix



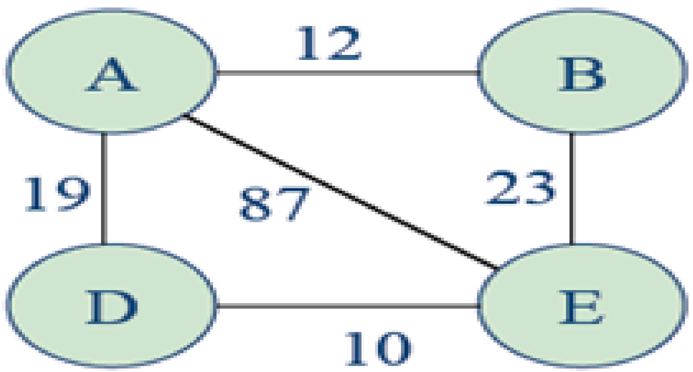
Vertex vector
A
B
C
D
E
F

Adjacency matrix

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

(b) Adjacency matrix for directed graph

✓ c) Adjacency Matrix with Weighted Graph



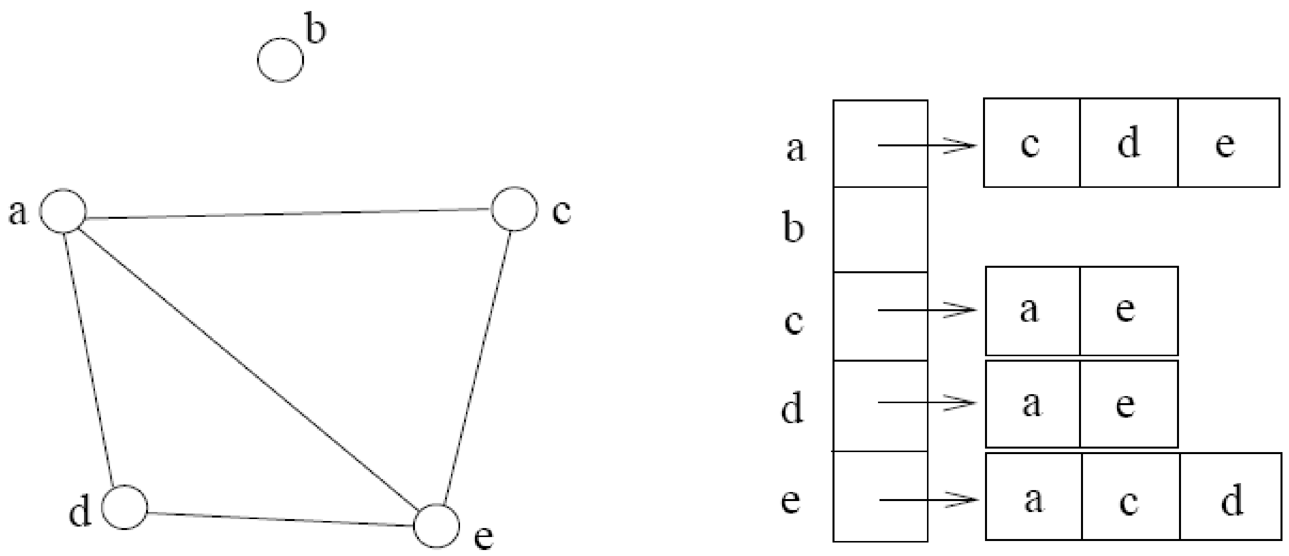
⇔

	A	B	D	E
A		12	19	87
B	12			23
D	19			10
E	87	23	10	

Notice anything about this matrix?
What would be the effect if the arcs were directed?

Adjacency list

Module No.5
Non-Linear DS

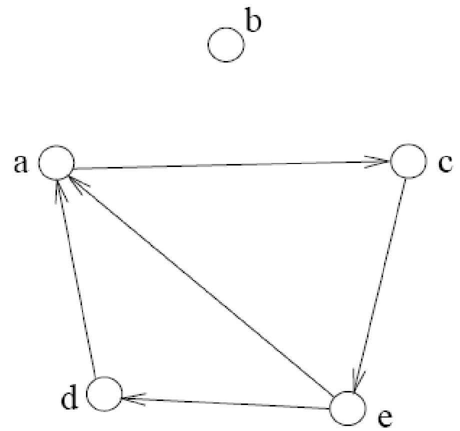


- The adjacency list is an array A[0..n-1] of lists, where n is the number of vertices in the graph.
- Each array entry is indexed by the vertex id (as with adjacency matrix)
- The list A[i] stores the ids of the vertices adjacent to i.

Directed Graph Representations

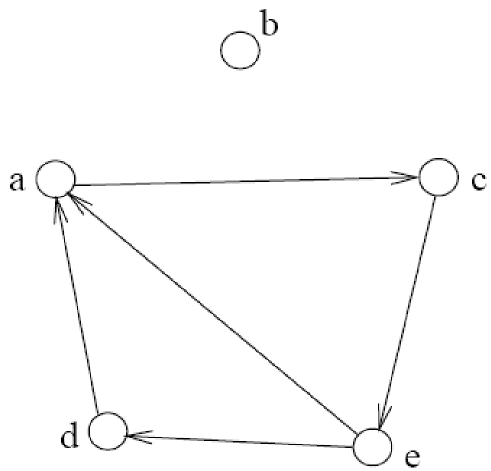
- The adjacency matrix and adjacency list can be used

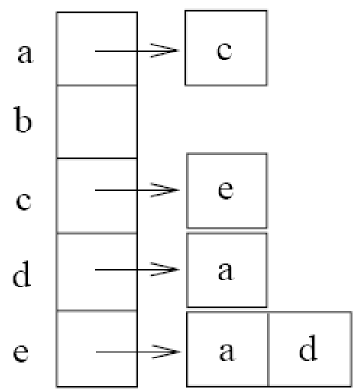
1. Adjacency Matrix



	a	b	c	d	e
a	0	0	1	0	0
b	0	0	0	0	0
c	0	0	0	0	1
d	1	0	0	0	0
e	1	0	0	1	0

2. Adjacency List





Adjacency Lists vs. Matrix

- **Adjacency Lists**
  - More compact than adjacency matrices if graph has few edges
  - Requires more time to find if an edge exists
- **Adjacency Matrix**
  - Always require n^2 space
    - This can waste a lot of space if the number of edges are sparse
  - Can quickly find if an edge exists

Directed Graph

- A graph is directed if direction is assigned to each edge. We call the directed edges *arcs*.
  - An edge is denoted as an ordered pair (u, v)
- Recall: for an undirected graph
  - An edge is denoted {u,v}, which actually corresponds to two arcs (u,v) and (v,u)

SCHOOL OF COMPUTER SCIENCE AND IT

Directed Acyclic Graph

- A directed path is a sequence of vertices $(v_0, v_1, \ldots, v_k)$
  - Such that (v_i, v_{i+1}) is an *arc*

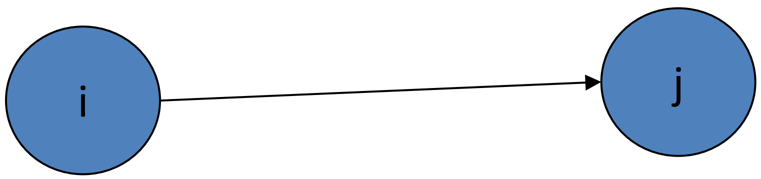
- *A directed cycle* is a directed path such that the first and last vertices are the same.

- A directed graph is *acyclic* if it does not contain any directed cycles

Directed Graphs Usage

SCHOOL OF COMPUTER SCIENCE AND IT

- Directed graphs are often **used to represent order-dependent tasks**
- That is we cannot start a task before another task finishes
- We can model this task dependent constraint using *arcs*
- An *arc (i,j)* means *task j* cannot start until *task i* is finished



Task j cannot start until task i is finished

- Clearly, for the system not to hang, the graph must be acyclic.

Graph Traversal

Breadth-First Search (BFS)

- BFS **strategy looks similar to level-order**.
- From **a given node v**, **it first visits itself**. Then, **it visits every node adjacent to v** before visiting any other nodes.

Depth-First Search (DFS)

- From a **given node v, it first visits itself**. Then, **recursively visit its unvisited neighbors one by one**.
- Strategy **looks similar to pre-order**

BFS Traversal

SCHOOL OF COMPUTER SCIENCE AND IT

✓**Visit all children of a node**, **then all grandchildren**, etc;

✓Principle is **Queue(LEVEL ORDER).**

1. Visit v
2. Visit all v’s neighbors
3. Visit all v’s neighbors' of neighbors.

BFS algorithm

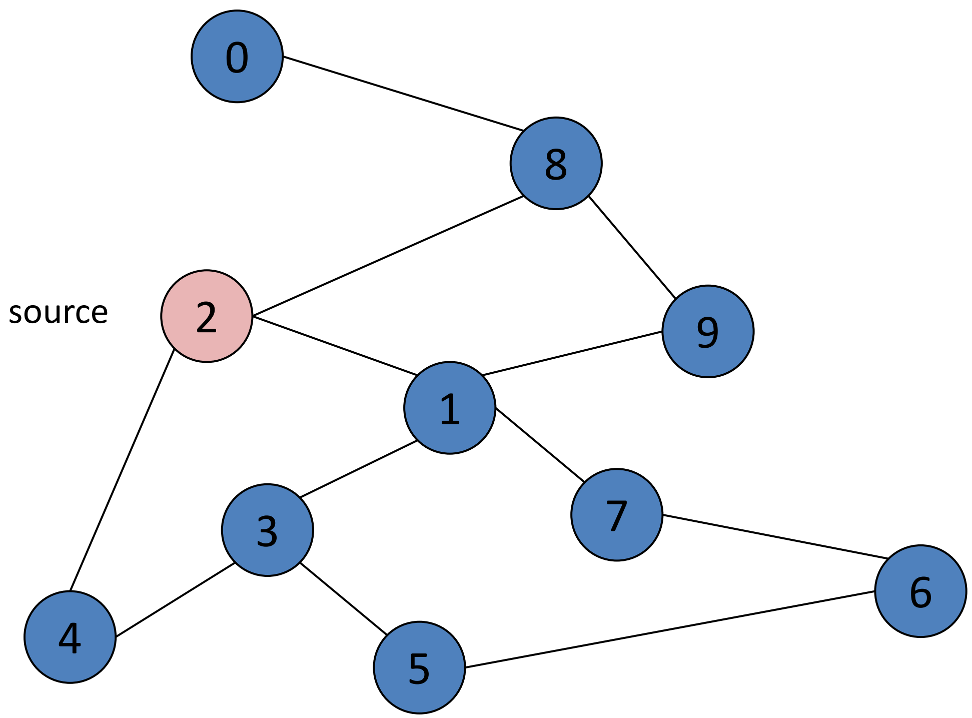
Algorithm *BFS*(s)

Input: s is the source vertex

Output: Mark all vertices that can be visited from s.

1. **for** each vertex v
2.   **do** $\mathit{flag}[v] :=$ false;
3. $Q =$ empty queue;
4. $\mathit{flag}[s] :=$ true;
5. *enqueue*(Q, s);
6. **while** Q is not empty
7.   **do** $v :=$ *dequeue*(Q);
8.   **for** each w adjacent to v
9.   **do if** $\mathit{flag}[w] =$ false
10.   **then** $\mathit{flag}[w] :=$ true;
11.   *enqueue*(Q, w)

Example



Adjacency List

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

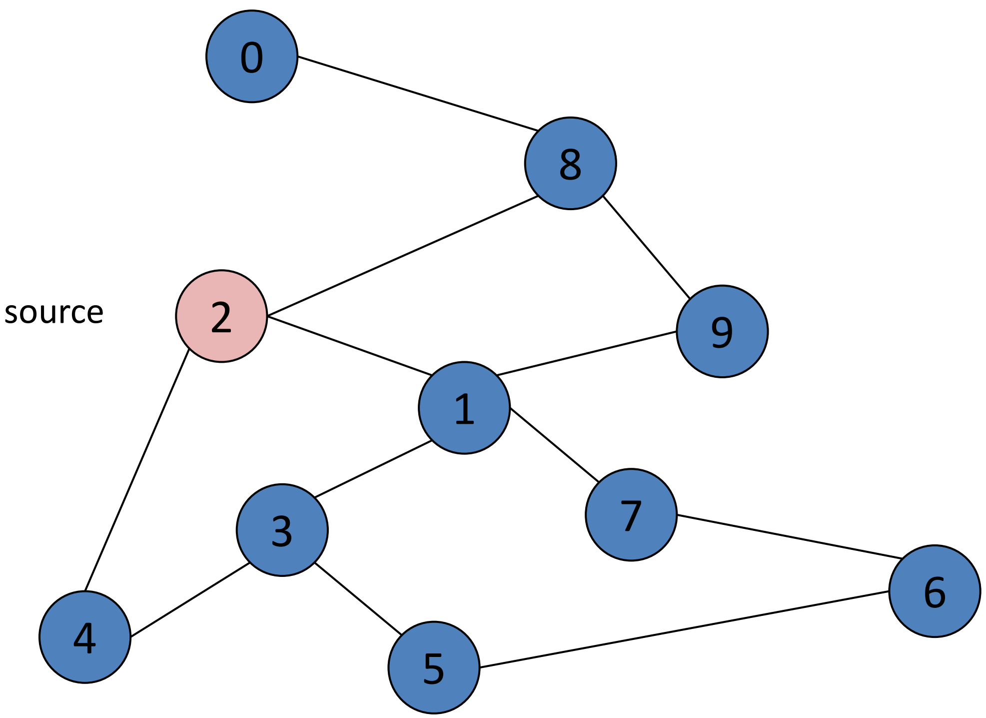
Visited Table (T/F)

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

Initialize visited table (all False)

Q = { }   Initialize Q to be empty

Example



Adjacency List

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

Visited Table (T/F)

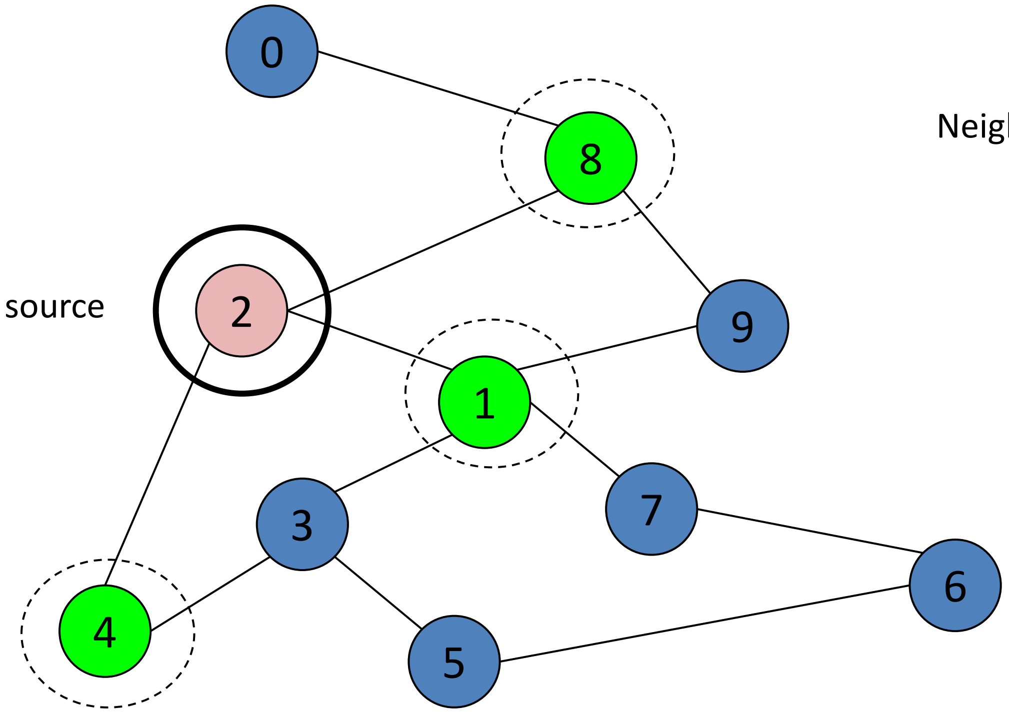
0	F
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	F
9	F

Flag that 2 has been visited.

Q = { 2 }

Place source 2 on the queue.

Example



Adjacency List

Neighbors →

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

Visited Table (T/F)

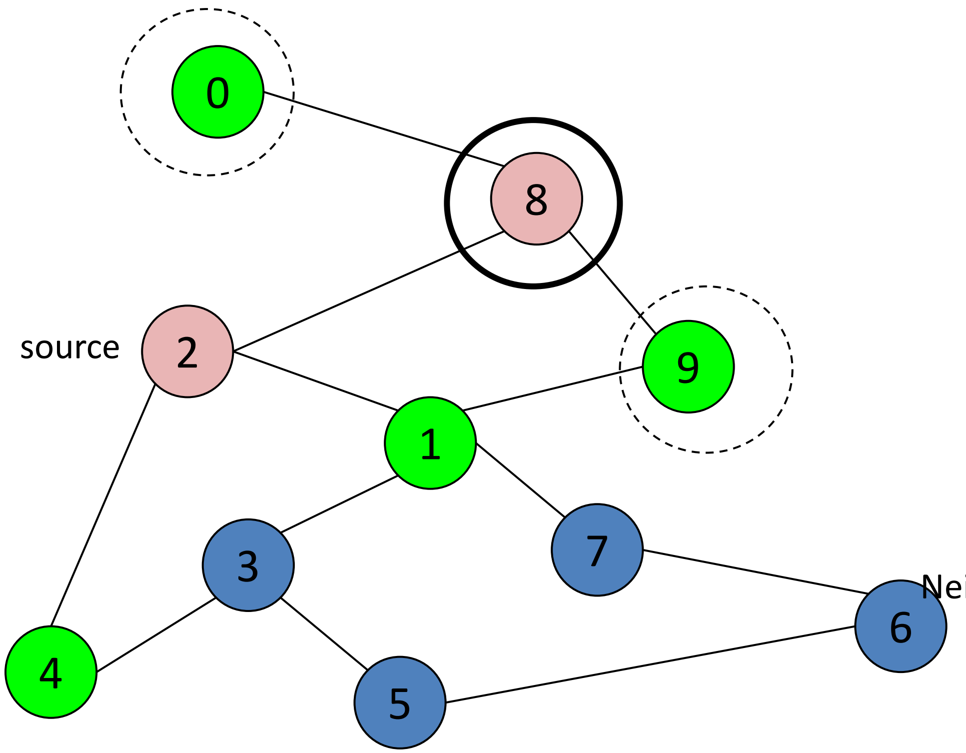
0	F
1	T
2	T
3	F
4	T
5	F
6	F
7	F
8	T
9	F

Mark neighbors as visited.

Q = {2} → { 8, 1, 4 }

Dequeue 2. Place all unvisited neighbors of 2 on the queue

Example



Adjacency List

Neighbors → 8

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

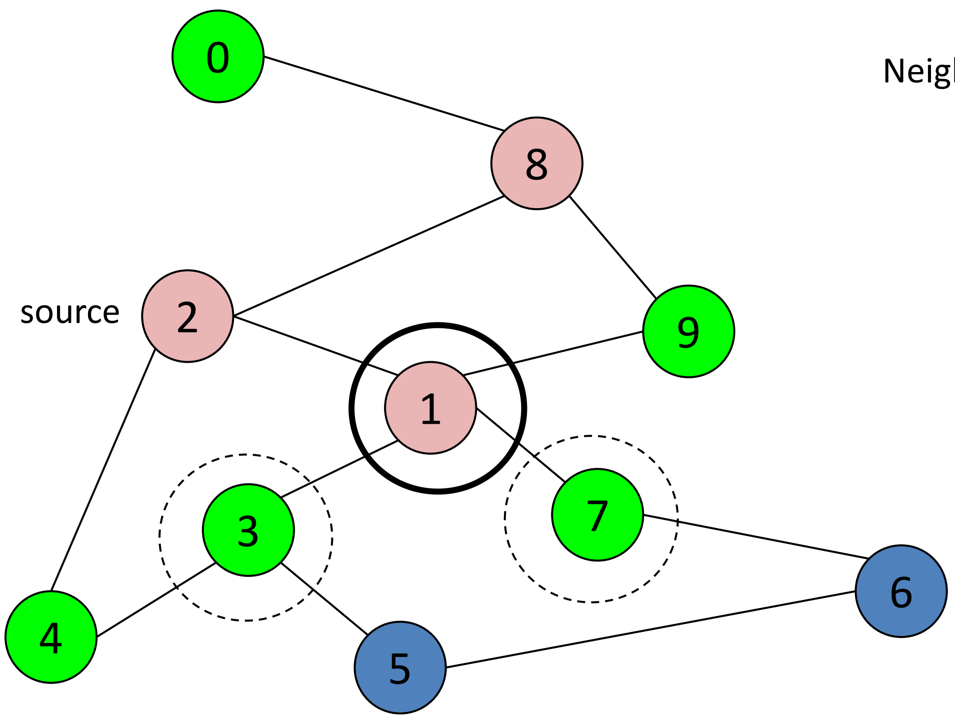
Visited Table (T/F)

0	T
1	T
2	T
3	F
4	T
5	F
6	F
7	F
8	T
9	T

Mark new visited Neighbors.

Q = { 8, 1, 4 } → { 1, 4, 0, 9 }
Dequeue 8.
-- Place all unvisited neighbors of 8 on the queue.
-- Notice that 2 is not placed on the queue again, it has been visited!

Example



Adjacency List

Neighbors →

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

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Mark new visited Neighbors.

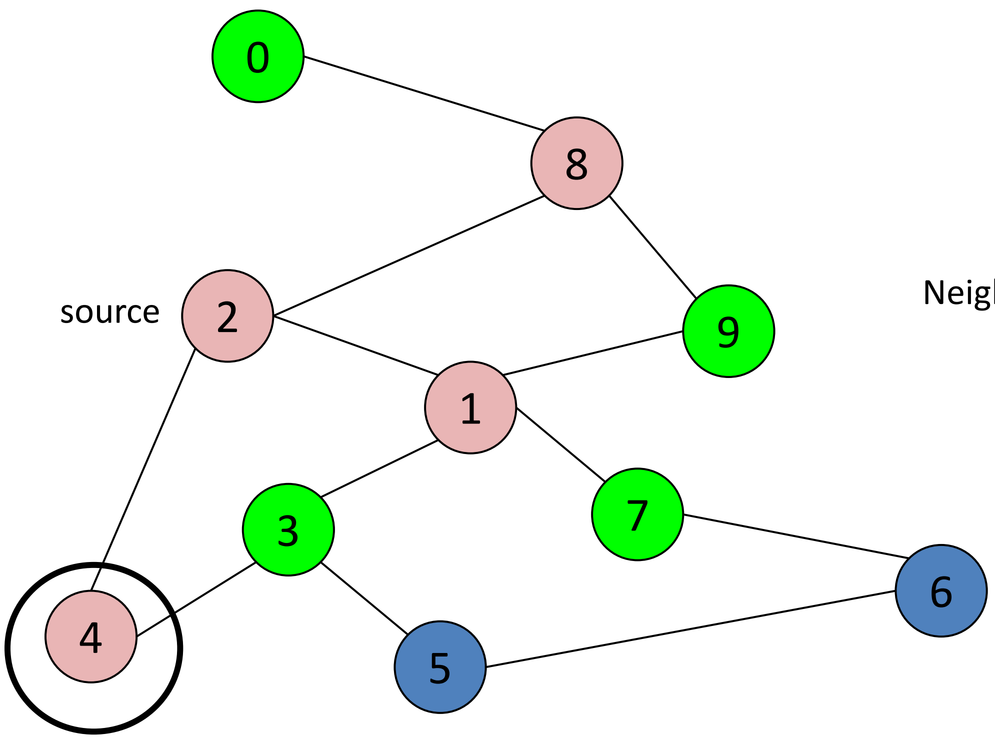
Q = { 1, 4, 0, 9 } → { 4, 0, 9, 3, 7 }

Dequeue 1.

-- Place all unvisited neighbors of 1 on the queue.

-- Only nodes 3 and 7 haven't been visited yet.

Example



Adjacency List

Neighbors →

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

Visited Table (T/F)

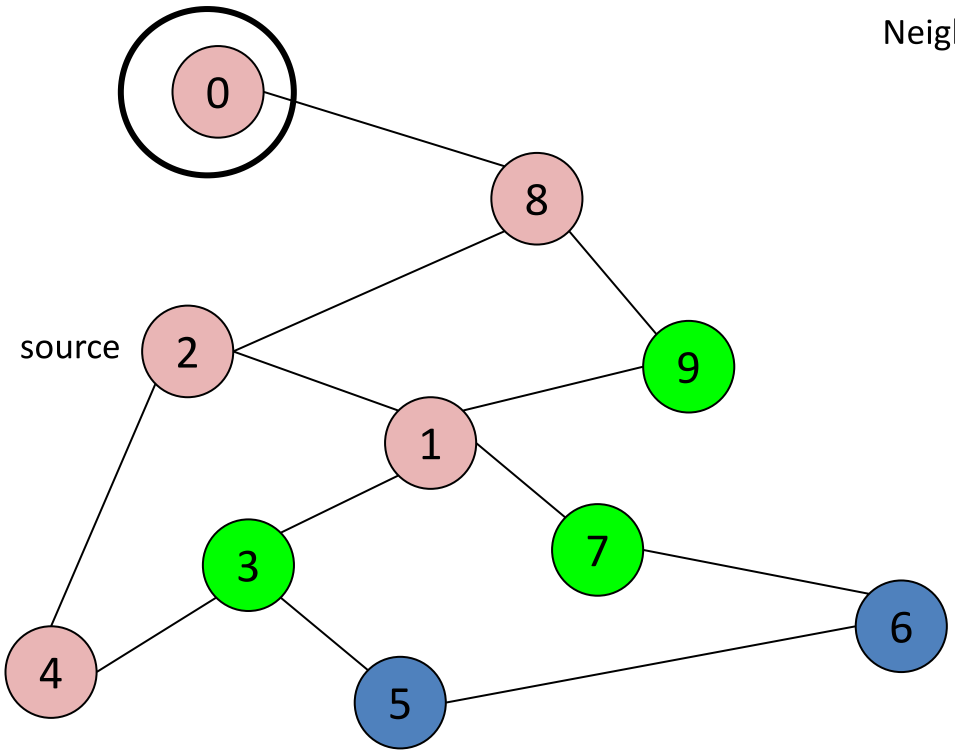
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Q = { 4, 0, 9, 3, 7 } → { 0, 9, 3, 7 }

Dequeue 4.

-- 4 has no unvisited neighbors!

Example



Adjacency List

Neighbors →

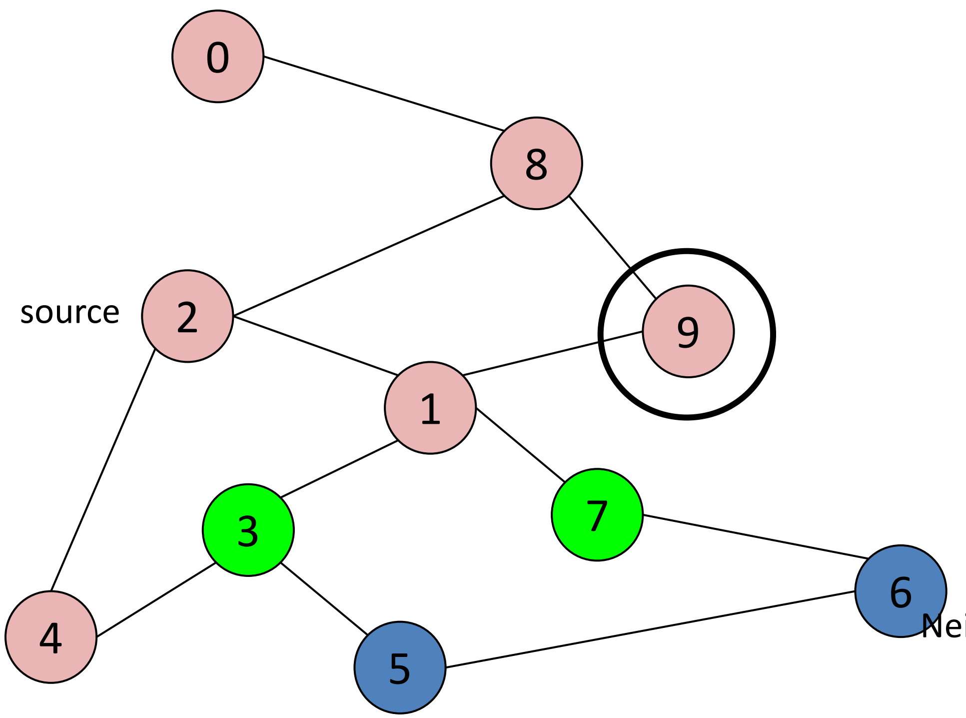
0	8			
1	3	7	9	2
2	8	1	4	
3	4	5	1	
4	2	3		
5	3	6		
6	7	5		
7	1	6		
8	2	0	9	
9	1	8		

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Q = { 0, 9, 3, 7 } → { 9, 3, 7 }
Dequeue 0.
-- 0 has no unvisited neighbors!

Example



Neighbors

Adjacency List

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

Visited Table (T/F)

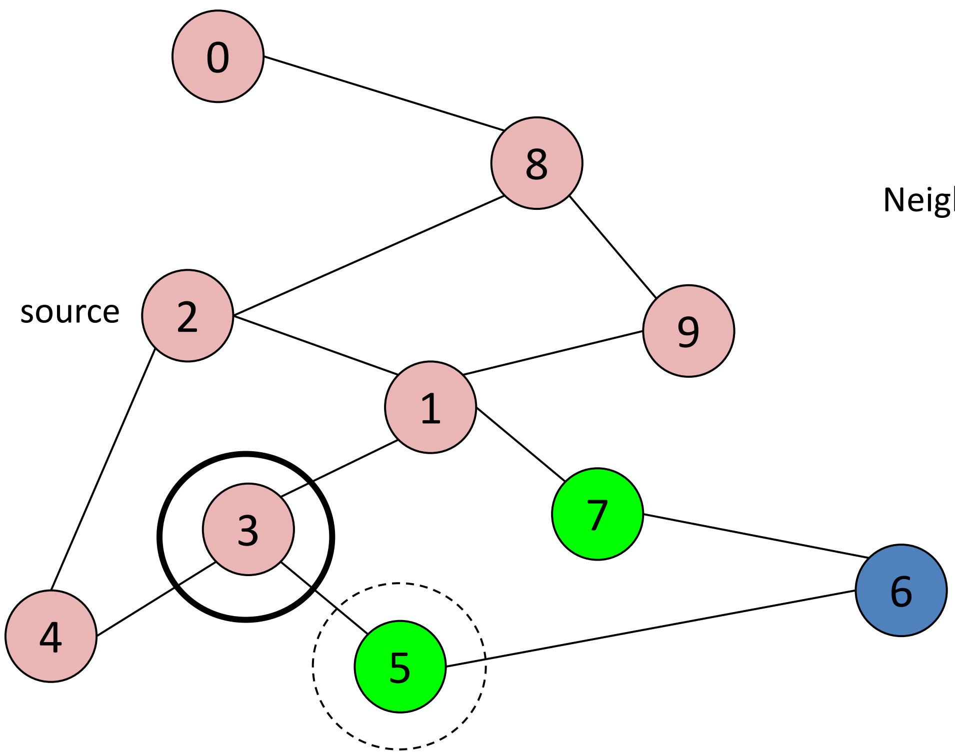
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Q = { 9, 3, 7 } → { 3, 7 }

Dequeue 9.
-- 9 has no unvisited neighbors!

Example

SCHOOL OF COMPUTER SCIENCE AND IT



Adjacency List

Neighbors →

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

Visited Table (T/F)

0	T
1	T
2	T
3	T
4	T
5	T
6	F
7	T
8	T
9	T

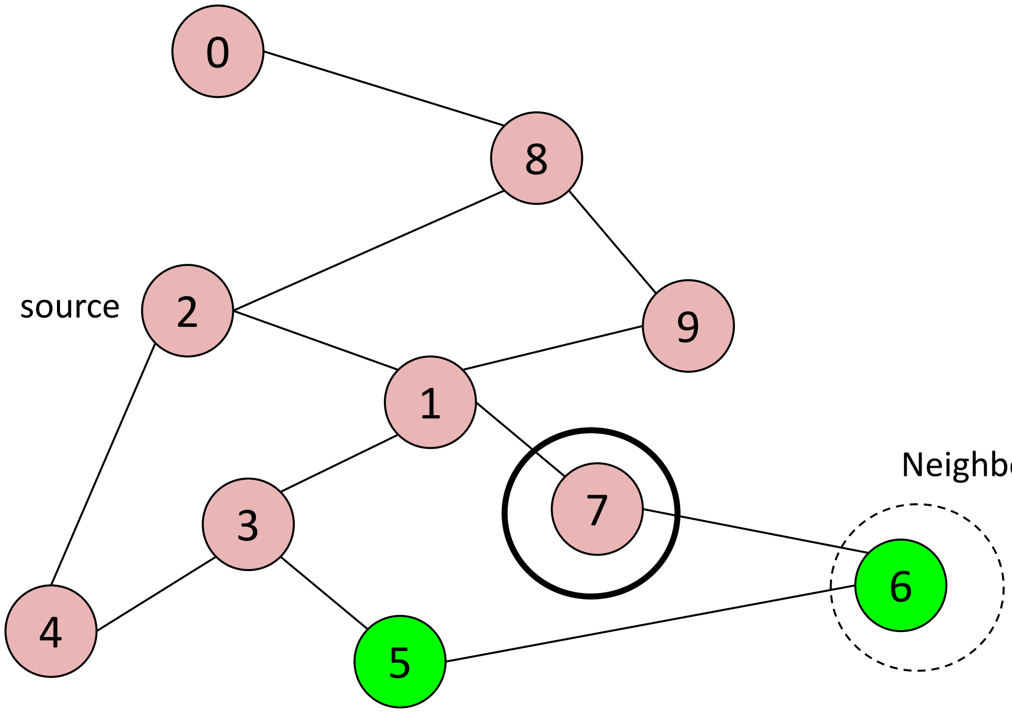
Mark new visited Vertex 5.

Q = { 3, 7 } → { 7, 5 }

Dequeue 3.

-- place neighbor 5 on the queue.

Example



Neighbors → 7

Adjacency List

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

Visited Table (T/F)

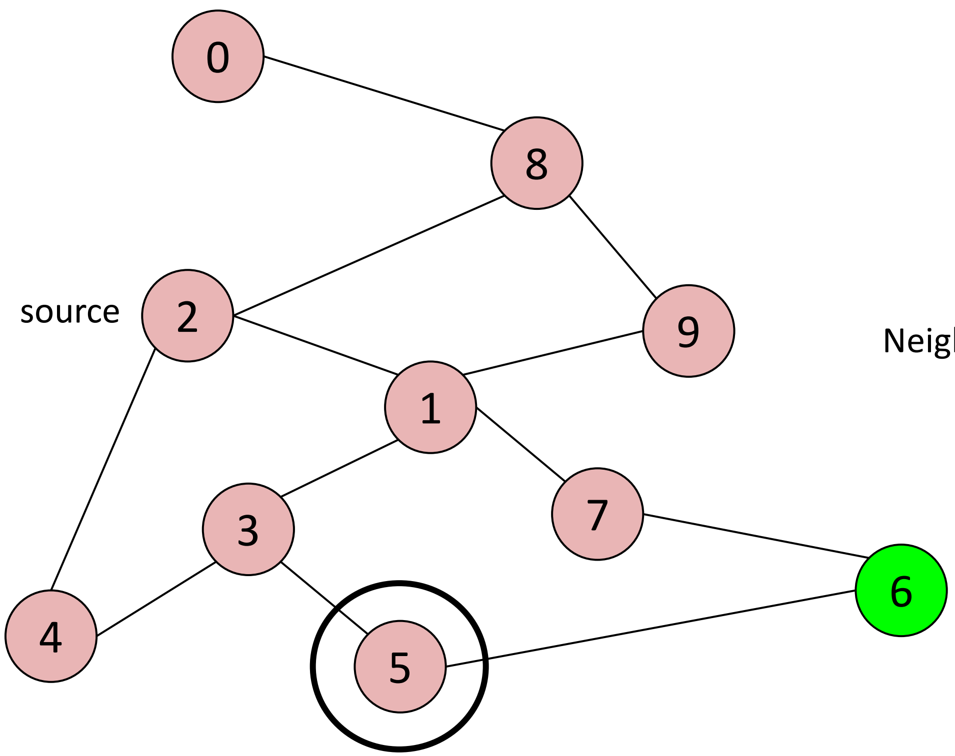
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Mark new visited Vertex 6.

Q = { 7, 5 } → { 5, 6 }

Dequeue 7. place neighbor 6 on the queue.

Example



Adjacency List

Neighbors →

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

Visited Table (T/F)

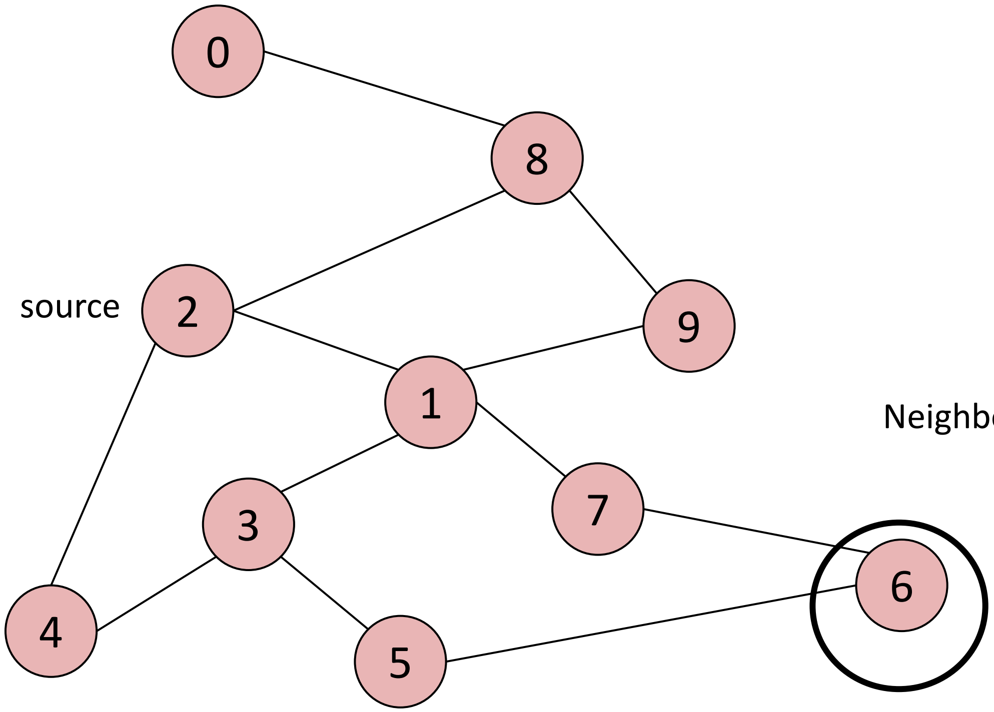
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Q = { 5, 6} → { 6 }

Dequeue 5. No unvisited neighbors of 5.

Example

JGi JAIN DEEMED-TO-BE UNIVERSITY | SCHOOL OF COMPUTER SCIENCE AND IT



Neighbors →

Adjacency List

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

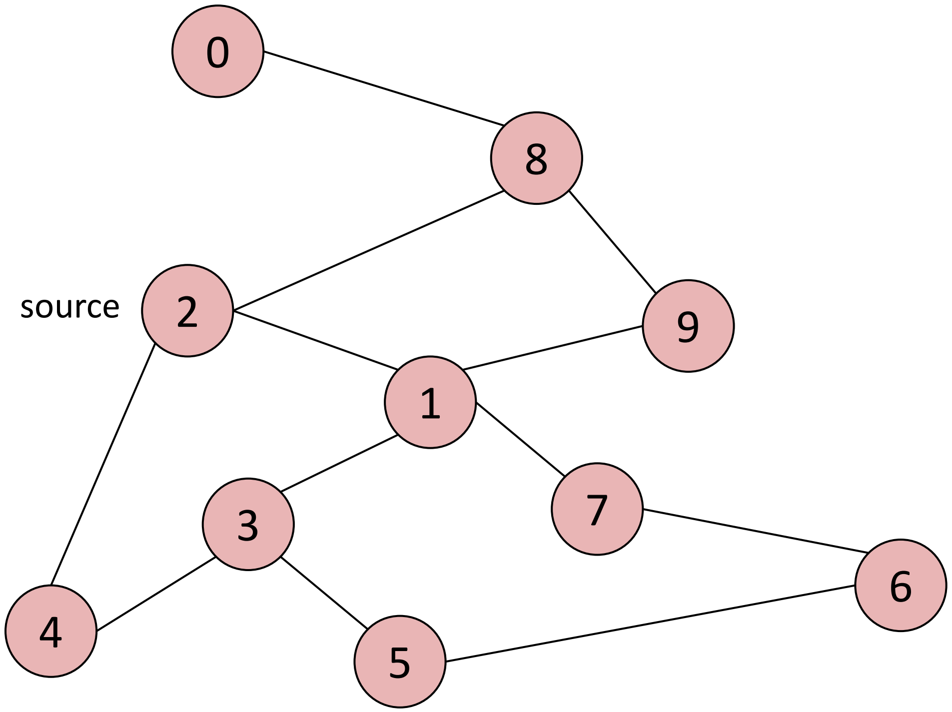
Visited Table (T/F)

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

Q = { 6 } → { }

Dequeue 6. No unvisited neighbors of 6.

Example



Adjacency List

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

Visited Table (T/F)

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

The **BFS traversal** of the given graph is **{2, 8, 1, 4, 0, 9, 3, 7, 5, 6}**

Q = { } STOP!!! Q is empty!!!

BFS Traversal

- Time Complexity:

  Assume Adjacency list

  n = number of vertices

  m = number of edges

  Time Complexity is **O(n+m)**

  If Adjacency Matrix then

  Time Complexity is **O(n*n)**

DFS Traversal

- A **recursive algorithm implicitly recording** a “backtracking” **path from the root** to **the node currently** under consideration

- ✓ **Process all of a vertex’s descendants** before **we move to an adjacent vertex.**
- ✓ **Visit all descendants of a node**, **before** **visiting sibling** nodes
- ✓ Depth first search **use a Stack or LIFO** data structure.

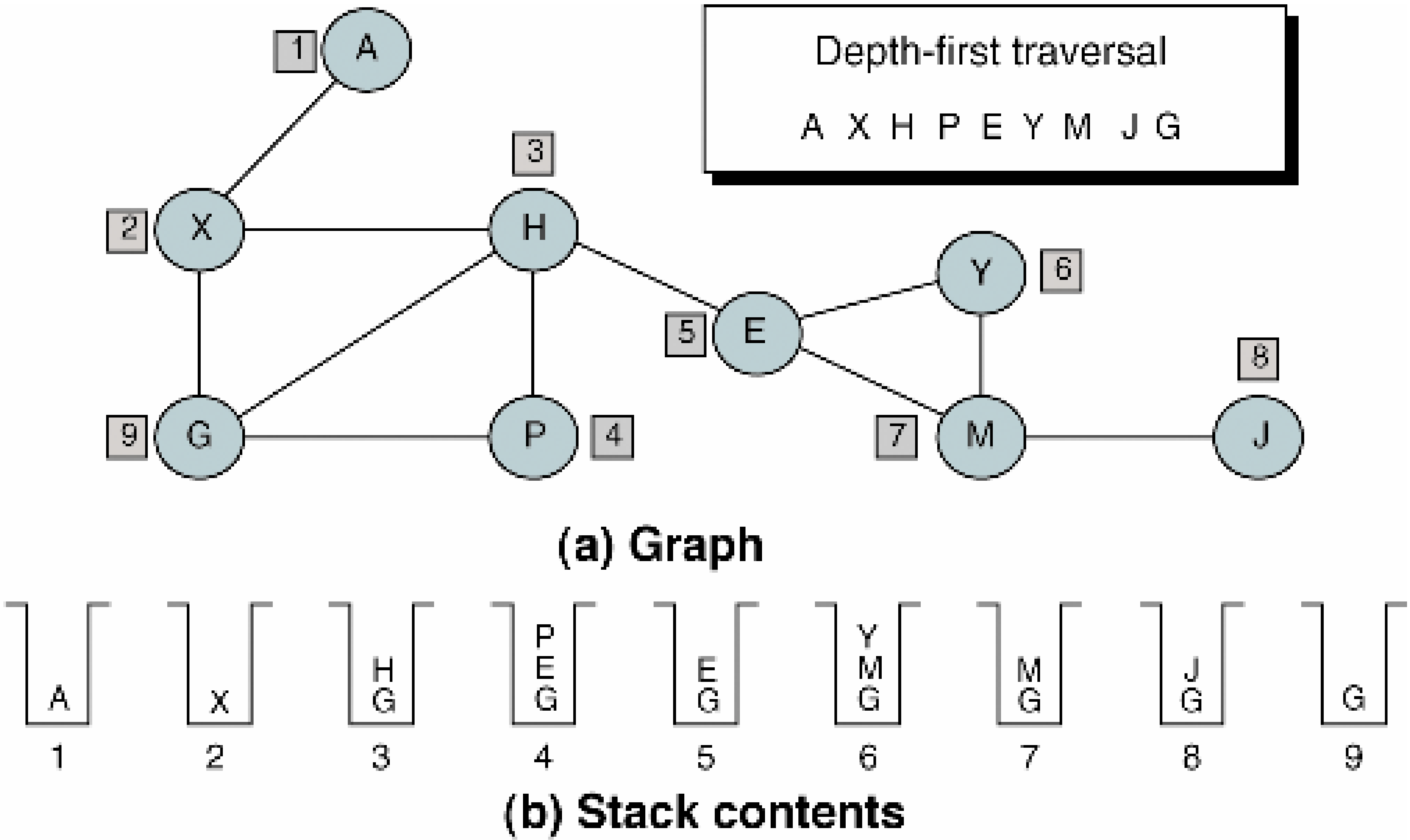
- ✓ Time complexity of is same – **O(|V|+|E|).** Or O(n+m)

DFS Traversal

PROCEDURE:

```
1.DFS(G,v)   ( v is the vertex where the search starts )
2.      Stack S := {};   ( start with an empty stack )
3.      for each vertex u, set visited[u] := false;
4.      push S, v;
5.      while (S is not empty) do
6.         u := pop S;
7.         if (not visited[u]) then
8.            visited[u] := true;
9.            for each unvisited neighbour w of u
10.              push S, w;
11.        end if
12.      end while
13.   END DFS()
```

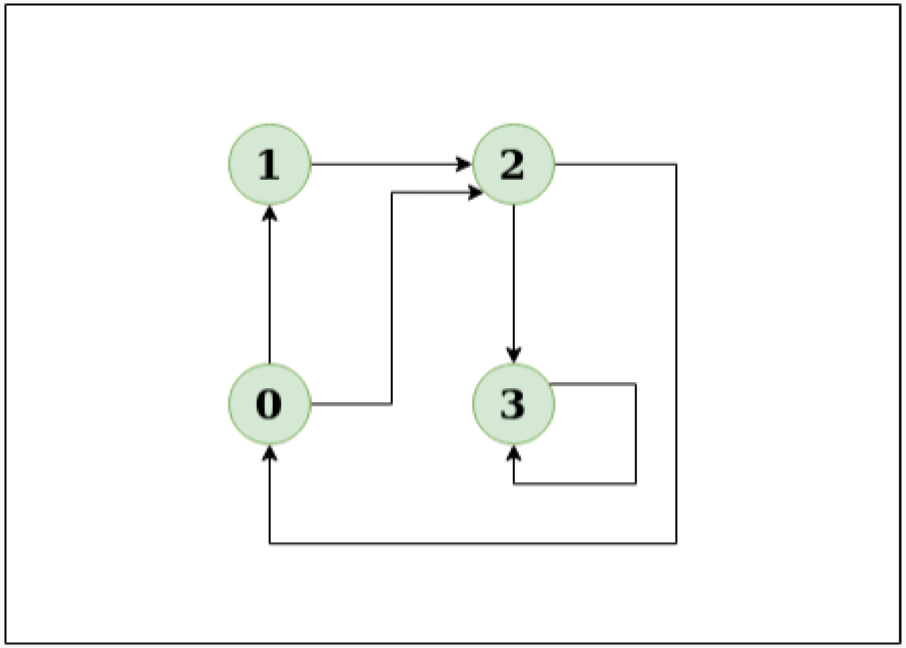
DFS Traversal



Detect Cycle in a Directed Graph

The objective is to check whether the graph contains a cycle or not.

Input: *N = 4, E = 6*



Example of graph

Output: *Yes*

Explanation: *The diagram clearly shows a cycle 0 -> 2 -> 0*

Detect Cycle in a Directed Graph

Approach to solve:

- *Use the Depth First Traversal (DFS) technique to find cycle in a directed graph.*
- *It can be found **only if there is a back edge.***
- ***The node points to one of its ancestors***

To detect **Back Edge**

- **Track of the nodes visited** till now and the **nodes** that are in the current recursion stack
- If **during recursion**, we reach a **node** that **is already in the recursion stack**, there is a cycle present in the graph.

Detect Cycle in a Directed Graph

DFS (current vertex, visited array, recursion stack)

Algorithm for Directed Graph

1.Initialize Structures:

- Use a **visited array** to track visited nodes.
- Use a **recStack array** (**recursive stack**) to **track nodes currently in the recursion stack**.

2.DFS Function:

- Mark the **current node as visited** and **add it to the recStack**.
- For **each neighbor** of **the current node**:
  - **If the neighbor is not visited**, **recursively call the DFS function**.
  - **If the neighbor is in the recStack**, a **cycle is detected and returns true.**
- **Remove the node** from the **recStack before returning**.

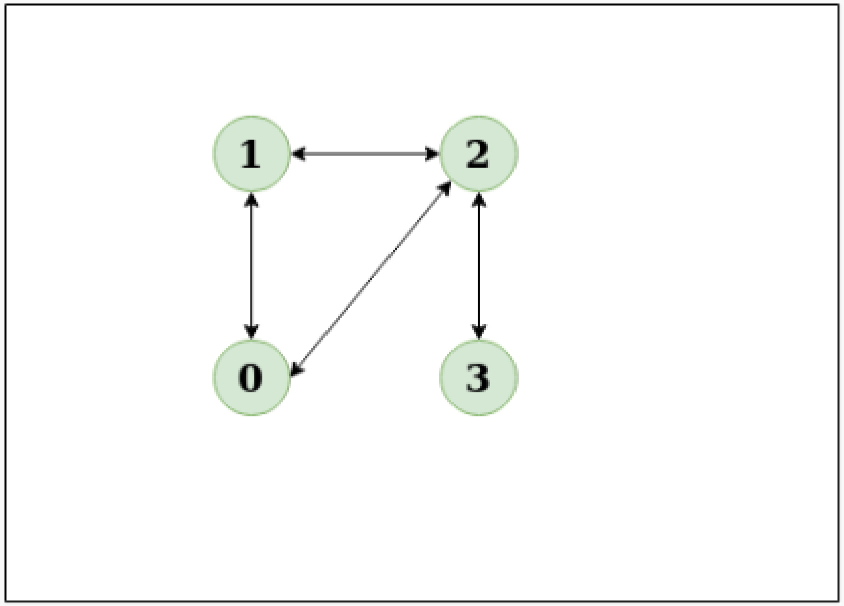
3.Cycle Detection:

- **Start DFS from every unvisited** node in the graph.

Detect Cycle in UnDirected Graph

The objective is to check whether the graph contains a cycle or not.

Input: *N = 4, E = 4*



Output: *Yes*

Explanation: *The diagram clearly shows a cycle 0 to 2 to 1 to 0*

Detect Cycle in UnDirected Graph

- *Use the Depth First Traversal (DFS) technique to find cycle in a directed graph.*
- *It can be found **only if there is a back edge.***
- ***The node points to one of its ancestors***

To **find the back edge** to any of **its ancestors keep** a visited array and if there is a **back edge to any visited node** then there **is a loop and return true**.

Detect Cycle in UnDirected Graph

Algorithm for Undirected Graph

1.Initialize Structures:

- Use a **visited array** to track **visited nodes.**

2.DFS Function:

- **Mark** the **current node as visited.**
- **For each neighbor** of the **current node**:
  - If **the neighbor is not visited**, **recursively call** the **DFS function**.
  - If the **neighbor is visited** and is **not the parent** of the **current node**, a **cycle is detected and return true.**

3.Cycle Detection:

- Start DFS from every unvisited node in the graph.

Transitive Closure of a Graph using DFS

- Given a directed graph**, find out if a vertex v is reachable from another vertex u** for all vertex pairs (u, v) in the given graph.
- Reachable means that **there is a path** from **vertex u to v.**
- The reach-ability matrix is called transitive closure of a graph

An O(V+E) algorithm is proposed to derive the objective

Transitive Closure of a Graph using DFS

Steps of the algorithm:

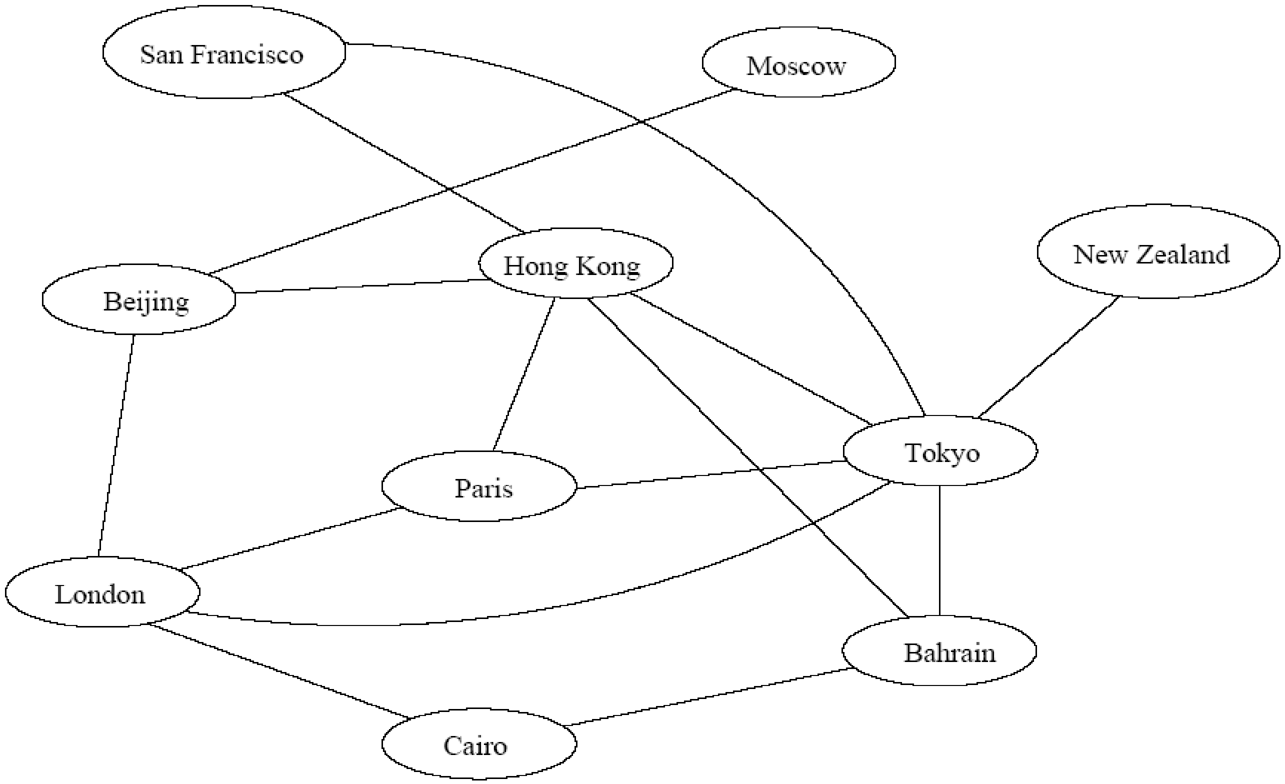
1. **Initialize a matrix of size N x N**, where **N is the number of vertices** in the graph. This matrix will **represent the transitive closure** of the graph.
2. For **each vertex v** in the graph, **perform a DFS starting at v**. During the DFS, **mark all vertices that are reachable from v.**
3. **Once the DFS is complete**, **update the matrix to include all the edges** that can be formed by following a **path from v to each of the reachable vertices**.

Specifically, if **vertex i is reachable from vertex j** during the DFS, **mark the entry (j,i)** in the matrix.

1. Repeat step 2 and 3 for all vertices in the graph.
2. The resulting matrix represents the transitive closure of the graph.

Applications

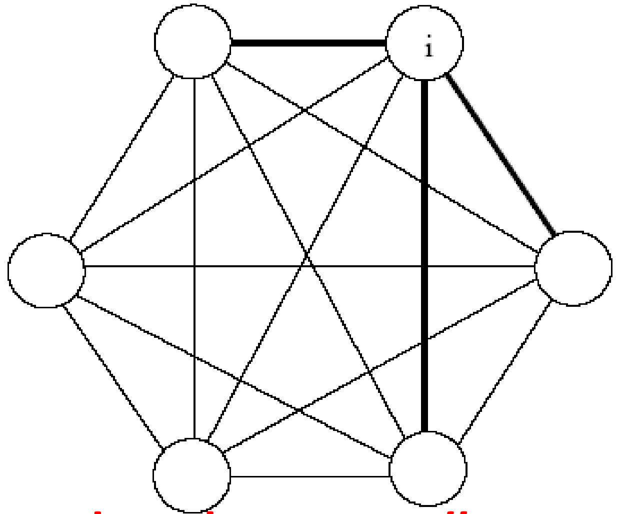
Air flight system



- Each **vertex represents a city**
- Each **edge represents a direct flight** between two cities
- **A query on direct flights** becomes a **query on whether an edge exists**
- A **query on how to get to a location** is "**does a path exist from A to B**"
- We can even associate costs to edges (weighted graphs), then ask "**what is the cheapest path** from A to B"

Module No.5
Non-Linear DS

Another application



Wireless communication

- A typical **wireless communication problem** is: **how to broadcast between *all* stations** such that **they are all connected** and **the power consumption is minimized.**

- Can be represented by a **weighted complete graph** (every two vertices are connected by an edge).

- Each **edge represents** the **Euclidean distance** *dij* between two **stations.**

- **Each station uses a certain power** i to **transmit messages**. Given **this power i**, only a **few nodes can be reached** (**bold edges**). A station reachable by i then use its own power to relay the message to other stations not reachable by i.

Applications

- ✓ Route Finding
- ✓ Game-Playing
- ✓ Critical Path Analysis
- ✓ Travel arrangement
- ✓ Communication and transportation
- ✓ Networks
- ✓ Logic circuits
- ✓ Computer aided designs
- ✓ Road network
- ✓ Pipeline network
- ✓ Activity chart.
- ✓ Maps, Schedules
- ✓ Computer networks

7.	Trees	Graphs
1.Path	Tree is special form of graph i.e. **minimally connected graph** and having only one path between any two vertices.	In graph there can be more than one path i.e. graph can have uni-directional or bi-directional paths (edges) between nodes
2.Loops	Tree is a special case of graph having no **loops**, no **circuits** and no self-loops.	Graph can have loops, circuits as well as can have **self-loops**.
3.Root Node	In tree there is exactly one root node and every **child** have only one **parent**.	In graph there is no such concept of **root** node.
4.Parent Child relationship	In trees, there is parent child relationship so flow can be there with direction top to bottom or vice versa.	In Graph there is no such parent child relationship.
5. Complexity	Trees are less complex then graphs as having no cycles, no self-loops and still connected.	Graphs are more complex in compare to trees as it can have cycles, loops etc
6. Types of Traversal	Tree traversal is a kind of special case of traversal of graph. Tree is traversed in **Pre-Order**, **In-Order** and **Post-Order** (all three in DFS or in BFS algorithm)	Graph is traversed by **DFS: Depth First Search** and in **BFS : Breadth First Search algorithm**

	Trees	Graphs
7. Connection Rules	In trees, there are many rules / restrictions for making connections between nodes through edges.	In graphs no such rules/ restrictions are there for connecting the nodes through edges.
8. DAG	Trees come in the category of **DAG : Directed Acyclic Graphs** is a kind of directed graph that have no cycles.	Graph can be **Cyclic or Acyclic**.
9. Different Types	Different types of trees are : **Binary Tree , Binary Search Tree, AVL tree, Heaps**.	There are mainly two types of Graphs : **Directed and Undirected graphs**.
10. Applications	Tree applications : sorting and searching like Tree Traversal & Binary Search.	Graph applications : Coloring of maps, in OR (**PERT & CPM**), algorithms, Graph coloring, job scheduling, etc.
11. No. of edges	Tree always has **n-1** edges.	In Graph, no. of edges depend on the graph.
12. Model	Tree is a **hierarchical model**.	Graph is a **network model**.
13. Figure