Graphs

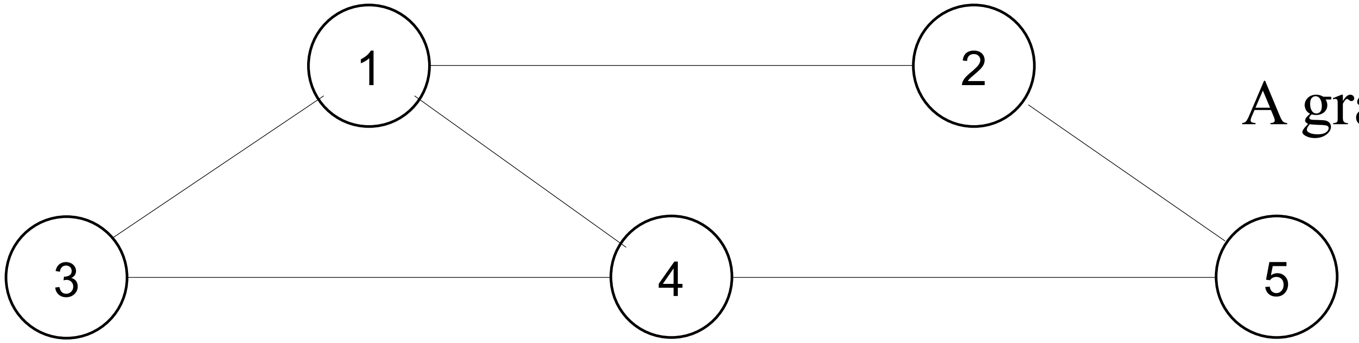
GRAPHS – Definitions

- A **graph** G = (V, E) consists of
  - a set of ***vertices***, V, and
  - a set of ***edges***, E, where each edge is a pair (v,w) s.t. v,w $\in$ V
- Vertices are sometimes called ***nodes***, edges are sometimes called ***arcs***.
- If the edge pair is ordered then the graph is called a **directed graph** **(**also called *digraphs)* .
- We also call a normal graph (which is not a directed graph) an ***undirected graph***.
  - When we say graph we mean that it is an undirected graph.

Graph – Definitions

- Two vertices of a graph are ***adjacent*** if they are joined by an edge.
- Vertex w is ***adjacent to*** v iff $(v,w) \in E$.
  - In an undirected graph with edge (v, w) and hence (w,v) w is adjacent to v and v is adjacent to w.
- A ***path*** between two vertices is a sequence of edges that begins at one vertex and ends at another vertex.
  - i.e. $w_1, w_2, \ldots, w_N$ is a path if $(w_i, w_{i+1}) \in E$ for $1 \leq i \leq$. N-1
- A ***simple path*** passes through a vertex only once.
- A ***cycle*** is a path that begins and ends at the same vertex.
- A ***simple cycle*** is a cycle that does not pass through other vertices more than once.

Graph – An Example



A graph G (undirected)

The graph G= (V,E) has 5 vertices and 6 edges:

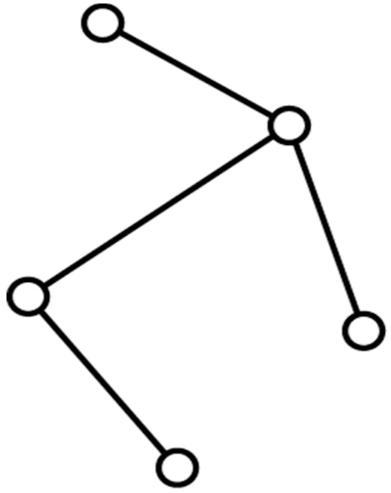
V = {1,2,3,4,5}

E = { (1,2),(1,3),(1,4),(2,5),(3,4),(4,5), (2,1),(3,1),(4,1),(5,2),(4,3),(5,4) }

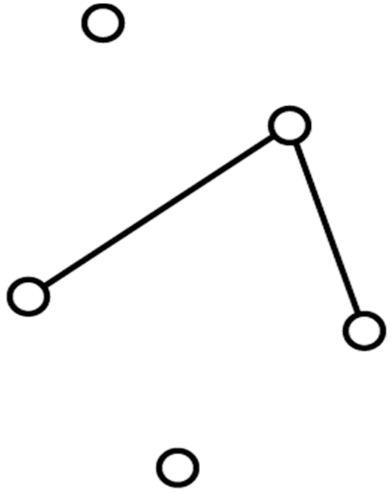
- *Adjacent:*
  1 and 2 are adjacent -- 1 is adjacent to 2 and 2 is adjacent to 1
- *Path:*
  1,2,5 (a simple path), 1,3,4,1,2,5 (a path but not a simple path)
- *Cycle:*
  1,3,4,1 (a simple cycle), 1,3,4,1,4,1 (cycle, but not simple cycle)

Graph -- Definitions

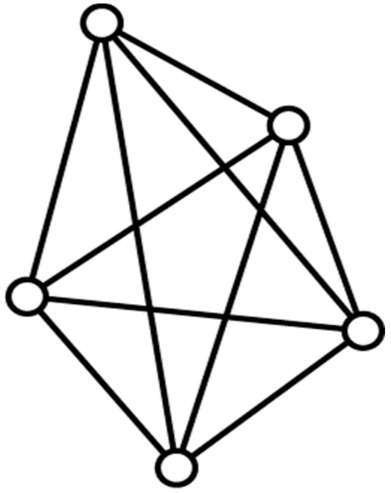
- A ***connected graph*** has a path between each pair of distinct vertices.
- A ***complete graph*** has an edge between each pair of distinct vertices.
  - A complete graph is also a connected graph. But a connected graph may not be a complete graph.



(a) **connected**



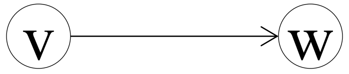
(b) **disconnected**



(c) **complete**

Directed Graphs

- If the edge pair is ordered then the graph is called a **directed graph (**also called *digraphs)* .
- Each edge in a directed graph has a direction, and each edge is called a ***directed edge***.
- Definitions given for undirected graphs apply also to directed graphs, with changes that account for direction.
- Vertex w is ***adjacent to*** v iff (v,w) ∈ E.
  - i.e. There is a direct edge from v to w
  - w is ***successor*** of v
  - v is ***predecessor*** of w

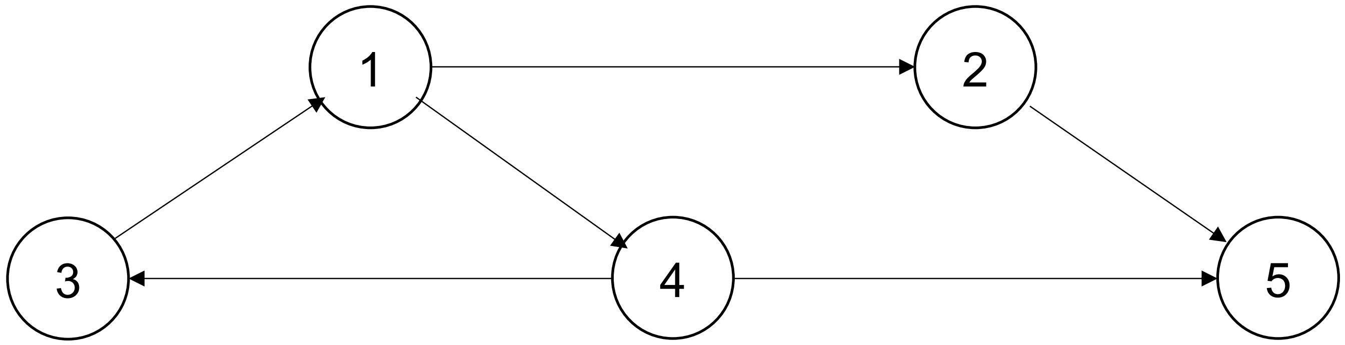


- A ***directed path*** between two vertices is a sequence of directed edges that begins at one vertex and ends at another vertex.
  - i.e. $w_1, w_2, \ldots, w_N$ is a path if $(w_i, w_{i+1}) \in E$ for $1 \leq i \leq$. N-1

Directed Graphs

- A **cycle** in a directed graph is a path of length at least 1 such that $w_1 = w_N$.
  - This cycle is simple if the path is simple.
  - For undirected graphs, the edges must be distinct
- A **directed acyclic graph** (*DAG*) is a type of directed graph having no cycles.
- An undirected graph is **connected** if there is a path from every vertex to every other vertex.
- A directed graph with this property is called **strongly connected**.
  - If a directed graph is not strongly connected, but the underlying graph (without direction to arcs) is connected then the graph is **weakly connected**.

Directed Graph – An Example



The graph G= (V,E) has 5 vertices and 6 edges:

V = {1,2,3,4,5}

E = { (1,2),(1,4),(2,5),(4,5),(3,1),(4,3) }

- *Adjacent:*

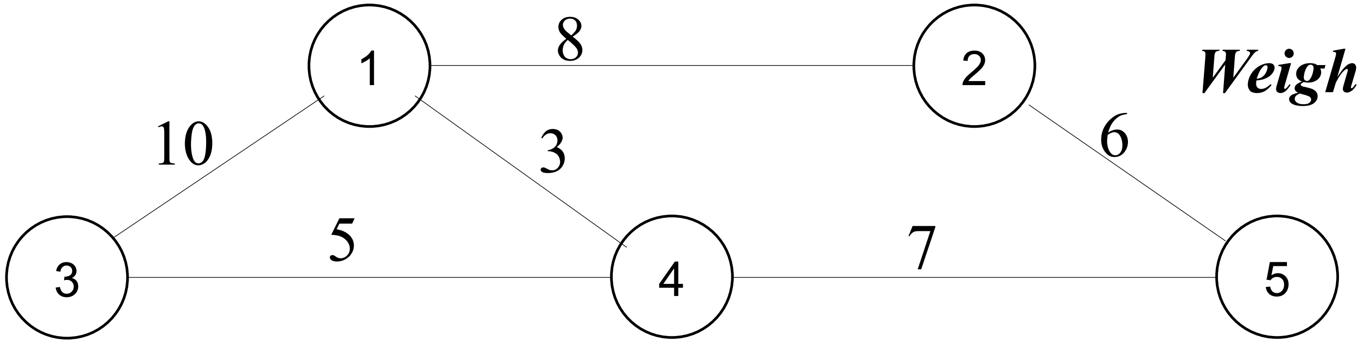
  2 is adjacent to 1, but 1 is NOT adjacent to 2
- *Path:*

  1,2,5 (a directed path),
- *Cycle:*

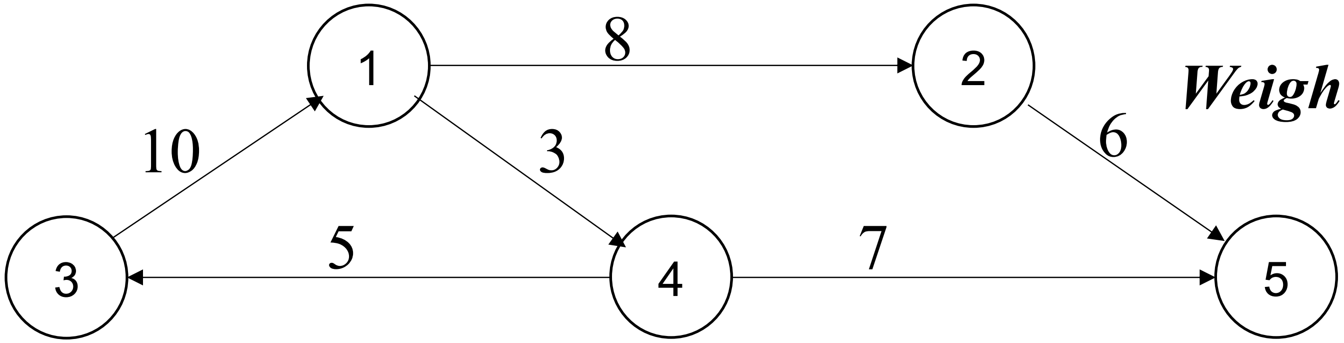
  1,4,3,1 (a directed cycle),

Weighted Graph

- We can label the edges of a graph with numeric values, the graph is called a ***weighted graph***.



Weighted (Undirected) Graph



Weighted Directed Graph

Graph Implementations

- The two most common implementations of a graph are:
  - ***Adjacency Matrix***
    - A two dimensional array
  - ***Adjacency List***
    - For each vertex we keep a list of adjacent vertices

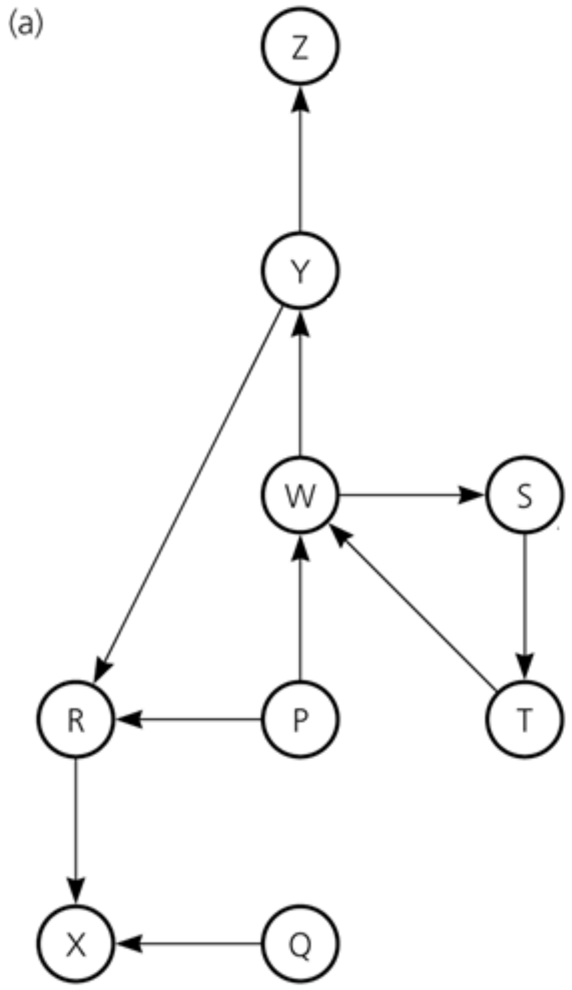
Adjacency Matrix

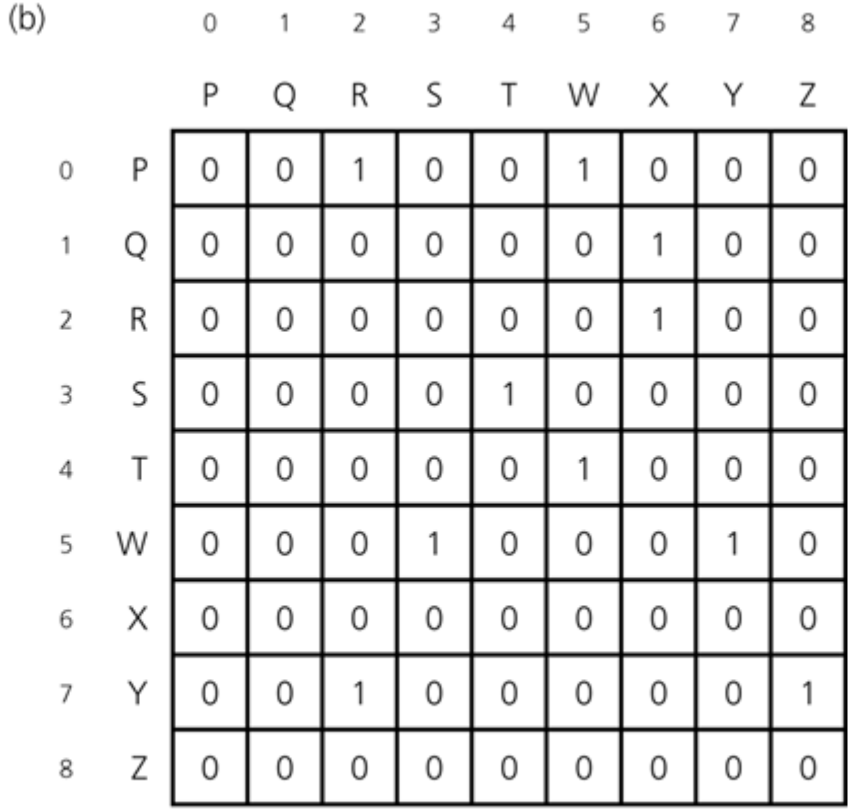
- An ***adjacency matrix*** for a graph with *n* vertices numbered 0,1,...,n-1 is an *n* by *n* array *matrix* such that *matrix[i][j]* is 1 (true) if there is an edge from vertex *i* to vertex *j*, and 0 (false) otherwise.
- When the graph is *weighted*, we can let *matrix[i][j]* be the weight that labels the edge from vertex *i* to vertex *j*, instead of simply 1, and let *matrix[i][j]* equal to ∞ instead of 0 when there is no edge from vertex *i* to vertex *j*.
- Adjacency matrix for an undirected graph is symmetrical.
  - i.e. *matrix[i][j]* is equal to *matrix[j][i]*
- Space requirement $O(|V|^2)$
- Acceptable if the graph is dense.

Adjacency Matrix – Example1

A directed graph

Its adjacency matrix



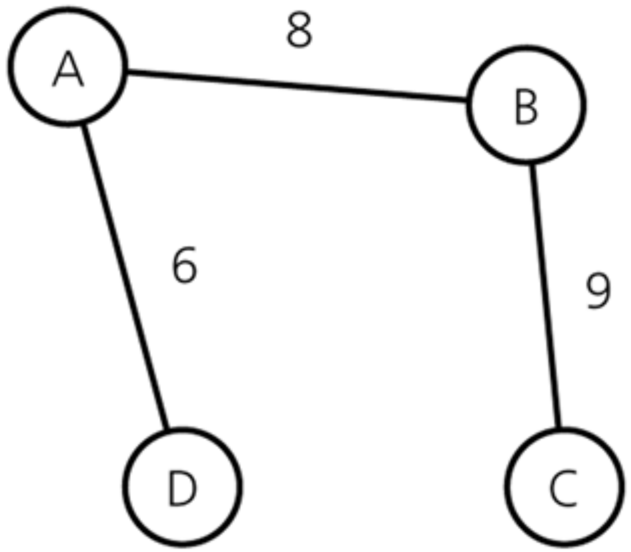


		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

```
int G[9][9]={{0,0,1,0,0,1,0,0,0},…};
```

Adjacency Matrix – Example2

An Undirected Weighted Graph



Its Adjacency Matrix

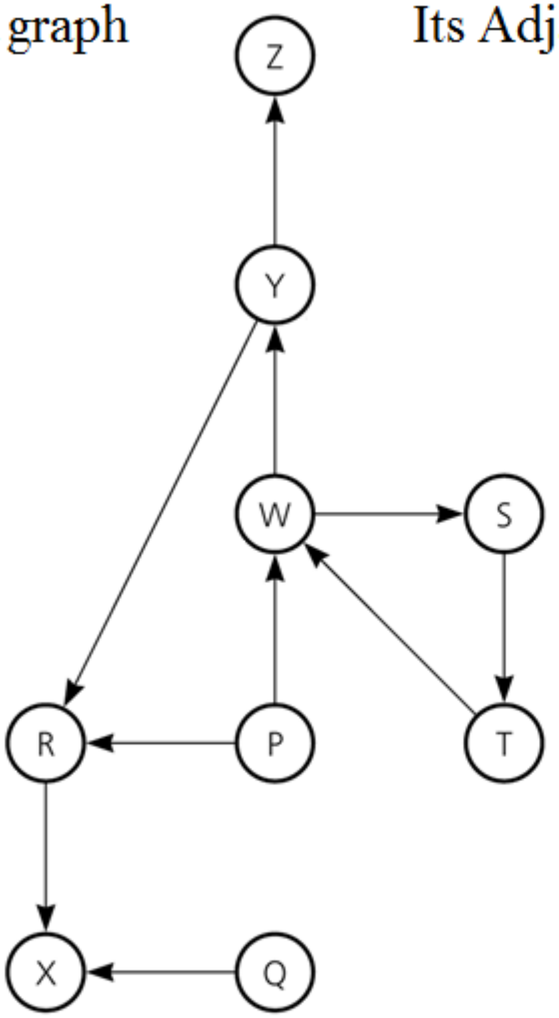
$$\begin{array}{cc|cccc}
 & & 0 & 1 & 2 & 3 \\
 & & A & B & C & D \\
\hline
0 & A & \infty & 8 & \infty & 6 \\
1 & B & 8 & \infty & 9 & \infty \\
2 & C & \infty & 9 & \infty & \infty \\
3 & D & 6 & \infty & \infty & \infty
\end{array}$$

Adjacency List

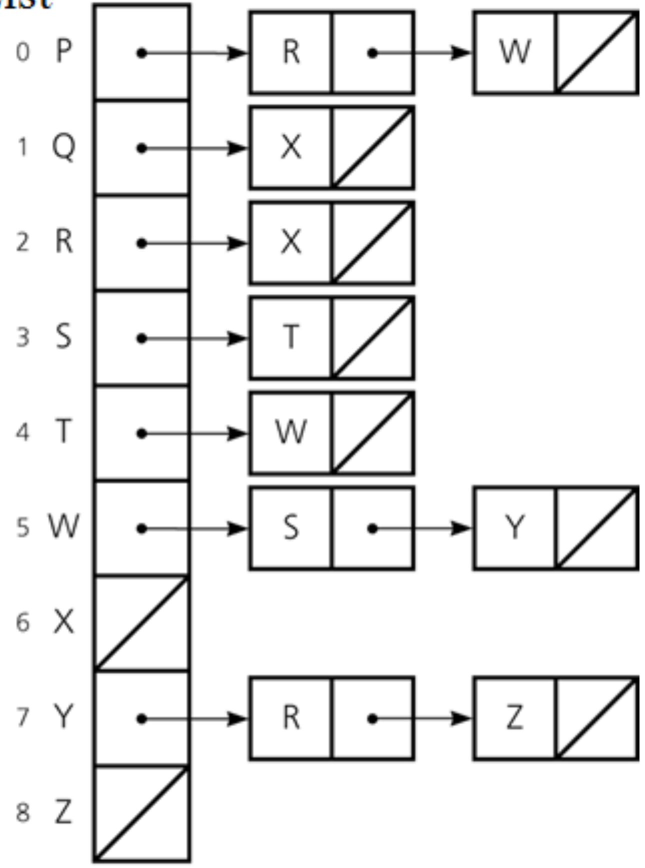
- An ***adjacency list*** for a graph with *n* vertices consists of *n* linked lists. The i^{th} linked list has a node for vertex *j* if and only if the graph contains an edge from vertex *i* to vertex *j*.
- Adjacency list is a better solution if the graph is sparse.
- Space requirement is O(|E| + |V|), which is linear in the size of the graph.
- In an undirected graph each edge (v,w) appears in two lists.
  - Space requirement is doubled.

Adjacency List – Example1

A directed graph

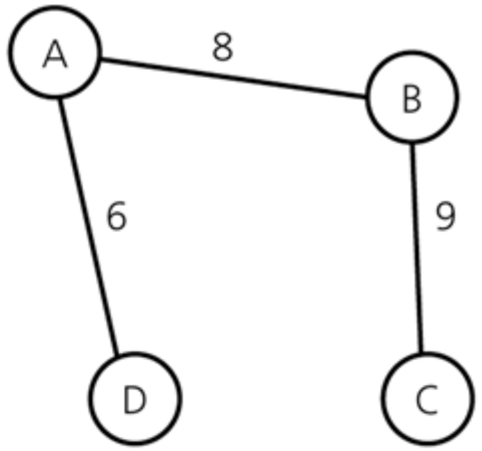


Its Adjacency List

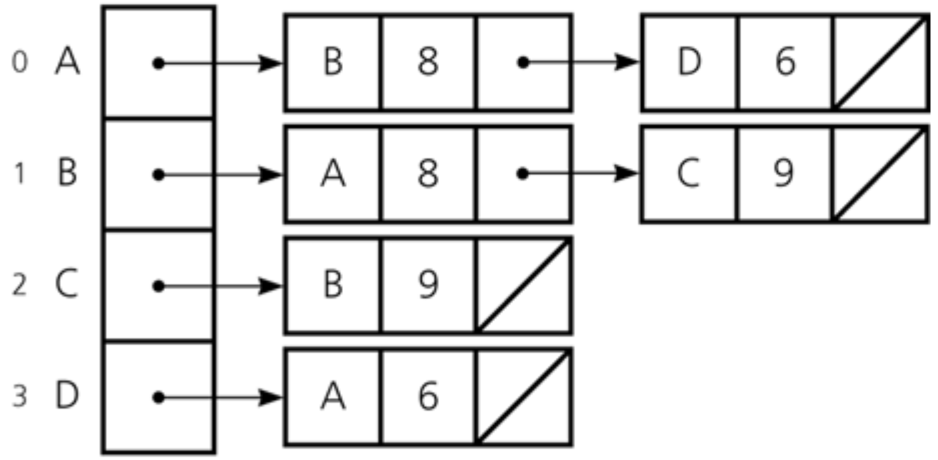


Adjacency List – Example2

An Undirected Weighted Graph



Its Adjacency List



Adjacency List Implementation: Using Array and Linked Lists

```
//unweighted graph

typedef struct node{
   char name;
   struct node *next;
}NODE;

NODE *G[9];
```

```
//weighted graph

typedef struct node{
   char name;
   int weight;
   struct node *next;
}NODE;

NODE *G[9];
```

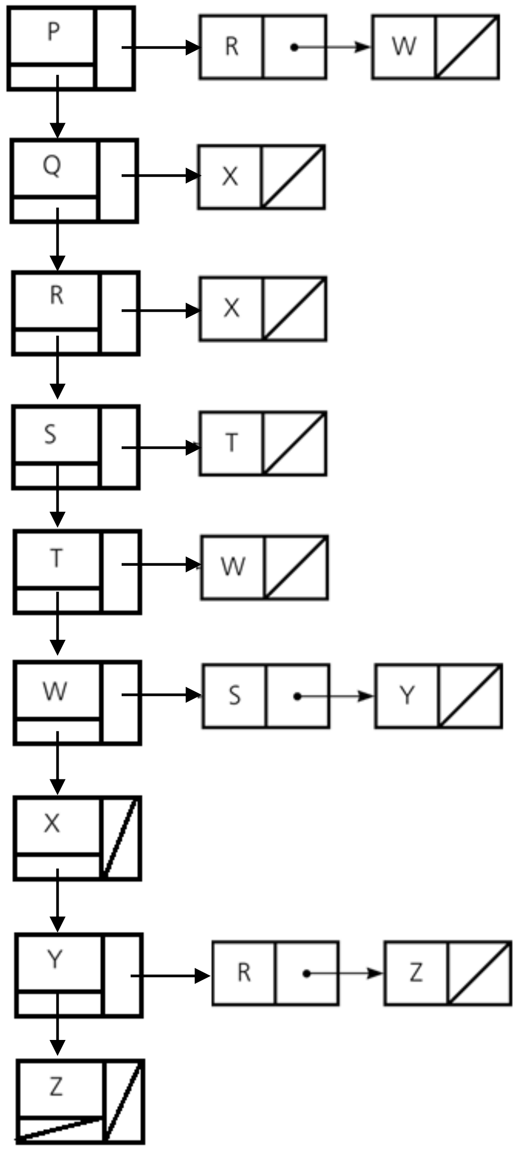
Adjacency List Implementation: Using Only Linked Lists

```
//unweighted graph

typedef struct horizontal_node{
   char name;
   struct horizontal_node *next;
}H_NODE;

typedef struct vertical_node{
   char name;
   struct horizontal_node *Hnext;
   struct vertical_node *Vnext;
}V_NODE;

V_NODE *G = NULL;
```



Adjacency Matrix vs Adjacency List

- Two common graph operations:
  1. Determine whether there is an edge from vertex i to vertex j.
  2. Find all vertices adjacent to a given vertex i.

- An adjacency matrix supports operation 1 more efficiently.
- An adjacency list supports operation 2 more efficiently.

- An adjacency list often requires less space than an adjacency matrix.
  - Adjacency Matrix: Space requirement is $O(|V|^2)$
  - Adjacency List : Space requirement is $O(|E| + |V|)$, which is linear in the size of the graph.
  - Adjacency matrix is better if the graph is dense (too many edges)
  - Adjacency list is better if the graph is sparse (few edges)

Graph Coloring

- **Graph coloring** is an assignment of different colors to all neighboring vertices in a graph.
- **Chromatic number** of a graph is the number which is equal to the minimum number of colors required to color the given graph.

- **Four colors theorem:** any planar graph G can be colored by using at most 4 colors.
- A **planar graph** can be drawn in such a way that no edges cross each other.
- Graph coloring is used to solve many real life problems:
  - Preparing exam schedule for students without any conflicts,
  - Scheduling processes to processors when there exist time and processor constraints.

Welch-Powel Algorithm for Graph Coloring

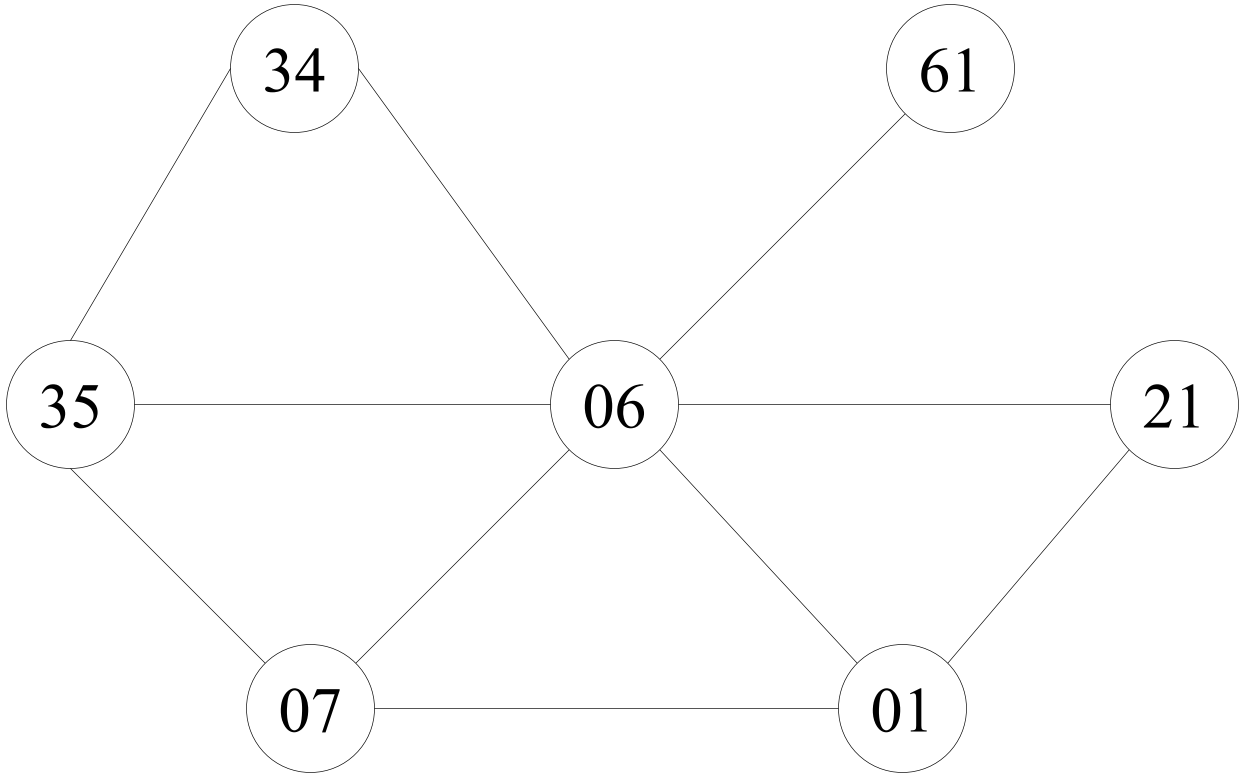
1. Compute degree of each vertex on the graph, then sort the vertices w.r.t. their degrees in descending order.

2. Assign the first color to the first vertex in the list. Then, assign this color to other vertices which are not adjacent to each other.

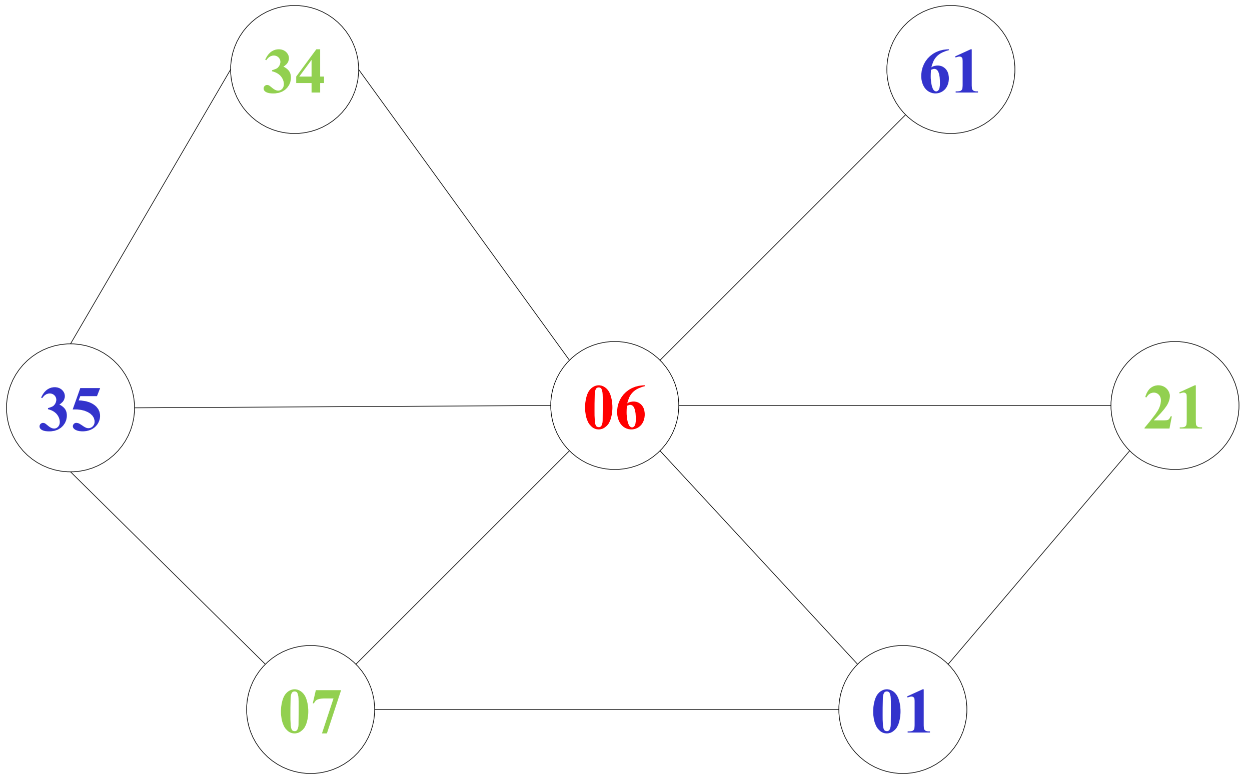
3. Continue with the second color. Assign the second color to the vertex having the highest degree among the unassigned vertices. Then, assign this color to other vertices which are not adjacent to each other.

4. Repeat step 3 until all vertices are colored.

Example 1:



Example 1 (cont.)



Node	Degree	Color
06	6	Red
01	3	Blue
07	3	Green
35	3	Blue
21	2	Green
34	2	Green
61	1	Blue

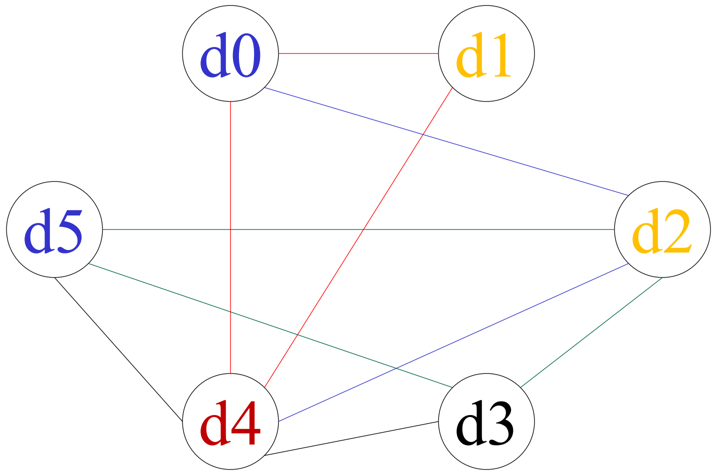
Example 2:

- Use graph coloring to solve the problem of determining groups of exams that can be made at the same time in an exam schedule without any conflict. Assume that there are 4 students and 6 courses.
- Student 1 will take exams for courses d0, d1, d4
- Student 2 will take exams for courses d0, d2, d4
- Student 3 will take exams for courses d2, d3, d5
- Student 4 will take exams for courses d3, d4, d5

- Find the list of courses whose exams can be made at the same time without any conflict between students and exams.
- What is the minimum number of sessions that the exams can be performed without any conflict?

Example 2 (cont.)

- Student 1 will take exams for courses d0, d1, d4
- Student 2 will take exams for courses d0, d2, d4
- Student 3 will take exams for courses d2, d3, d5
- Student 4 will take exams for courses d3, d4, d5



Node	Degree	Color
d4	5	Red
d2	4	Orange
d0	3	Blue
d3	3	Yellow
d5	3	Blue
d1	2	Orange

Graph Traversals

- A *graph-traversal* algorithm starts from a vertex v, visits all of the vertices that can be reachable from the vertex v.
- A graph-traversal algorithm visits all vertices if and only if the graph is connected.
- A connected component is the subset of vertices visited during a traversal algorithm that begins at a given vertex.
- A graph-traversal algorithm must mark each vertex during a visit and must never visit a vertex more than once.
  – Thus, if a graph contains a cycle, the graph-traversal algorithm can avoid infinite loop.
- We look at two graph-traversal algorithms:
  – ***Depth-First Traversal***
  – ***Breadth-First Traversal***

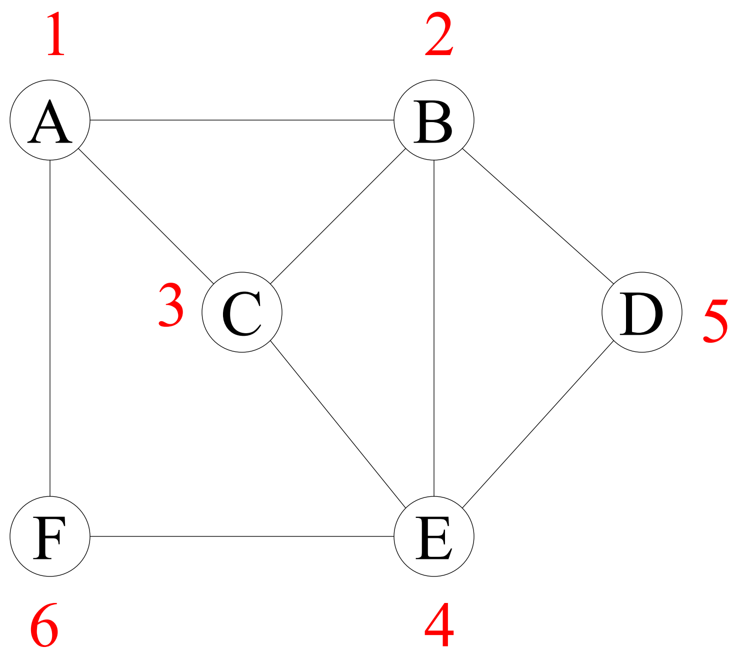
Depth-First Traversal

- For a given vertex v, the ***depth-first traversal*** algorithm proceeds along a path from v as deeply into the graph as possible before backing up.

- That is, after visiting a vertex v, the ***depth-first traversal*** algorithm visits (if possible) an unvisited adjacent vertex to vertex v.

- The depth-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to v.
  - We may visit the vertices adjacent to v in sorted order.

Depth-First Traversal – Example



• A depth-first traversal of the graph starting from vertex v.

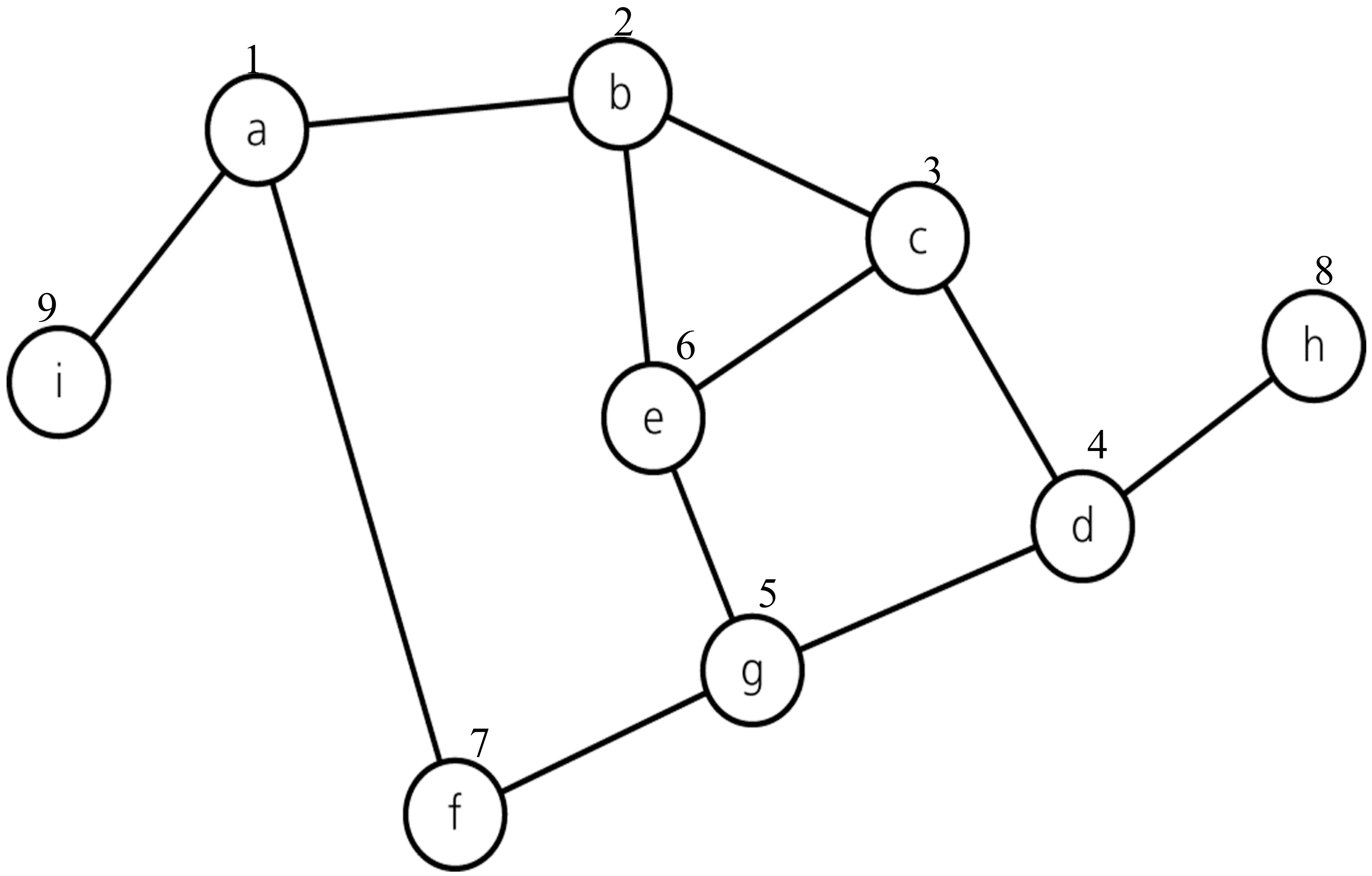
• Visit a vertex, then visit a vertex adjacent to that vertex.

• If there is no unvisited vertex adjacent to visited vertex, back up to the previous step.

Recursive Depth-First Traversal Algorithm

```
int G[6][6]={{0,1,1,0,0,1},…};
int visited[6]={0,0,0,0,0,0};
void DFT(void){
   int node;
   for (node = 0; node < 6; node++){
      if (visited[node] == 0)
         TraverseGraph(node);
   }
}
void TraverseGraph(int node){
   int i;
   visited[node]=1;
   printf("%d\n",node);
   for (i = 0; i < 6; i++){
      if ((G[node][i] != 0) && (visited[i] == 0))
         TraverseGraph(i);
   }
}
```

Trace of DFT – starting from vertex a

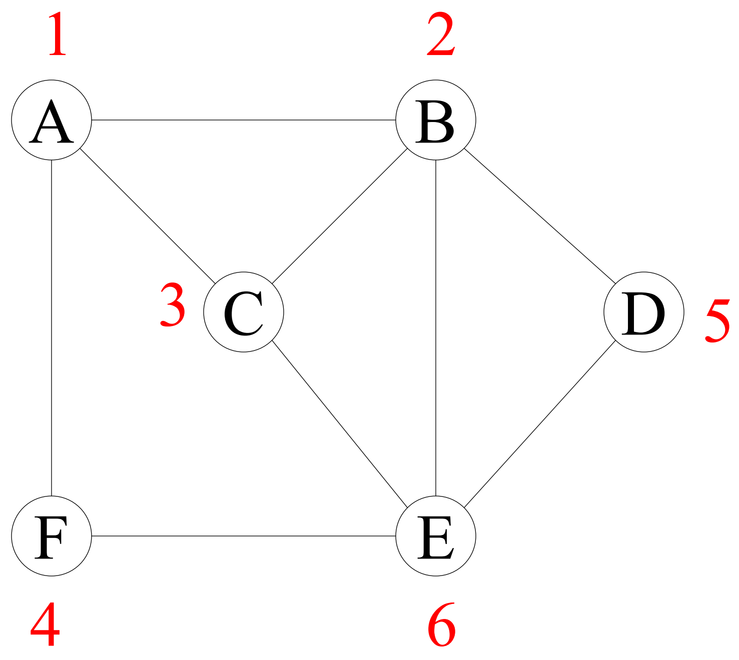


a
b
c
d
g
e
f
h
i

Breath-First Traversal

- After visiting a given vertex v, the breadth-first traversal algorithm visits every vertex adjacent to v that it can before visiting any other vertex.
- Continue with the first neighbor of v, and visit all of its’ neighbors.
- Continue this process until all the nodes in the graph are visited.

Breath-First Traversal – Example



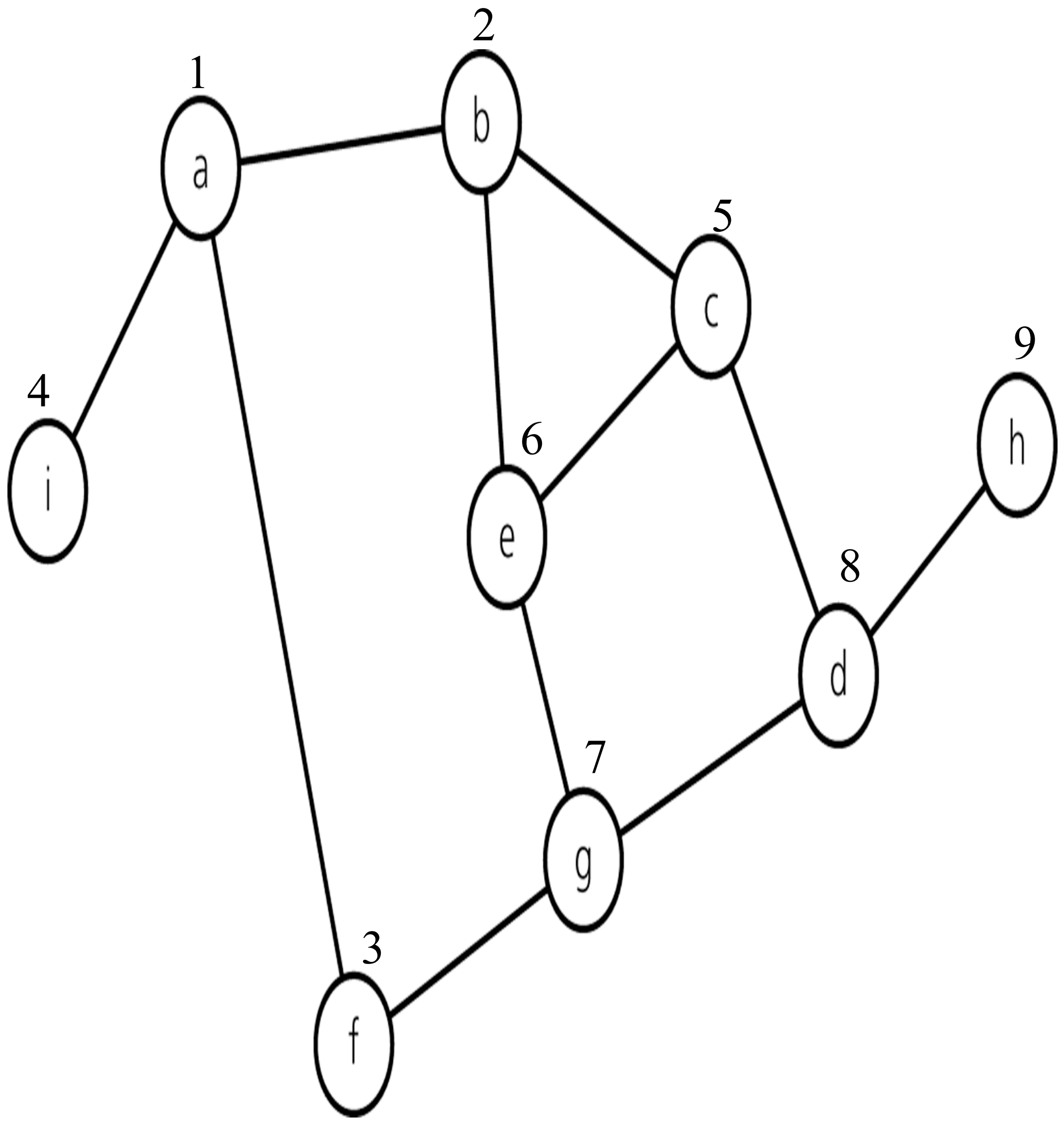
- A breath-first traversal of the graph starting from vertex A.

- Visit a vertex, then visit all vertices adjacent to that vertex.

Breath-First Traversal Algorithm

```
int visited[6]={0,0,0,0,0,0};
void BFT(void) {
   int node, n, i;
   for (node = 0; node < 6; node++){
      if (visited[node] == 0){
         enqueue(node);
         while (!isEmptyQueue()) {
            n = dequeue();
            if (visited[n] != 0)
               continue;
            visited[n] = 1;
            printf("%d\n",n);
            for (i = 0; i < 6; i++) {
               if ((G[n][i] != 0) && (visited[i] == 0))
                  enqueue(i);
            }
         }
      } } }
```

Trace of BFT – starting from vertex a



Node visited	Queue (front to back)
a	a
	(empty)
b	b
f	b f
i	b f i
	f i
c	f i c
e	f i c e
	i c e
g	i c e g
	c e g
	e g
d	e g d
	g d
	d
	(empty)
h	h
	(empty)