COM 201Data Structures and Algorithms

Graphs

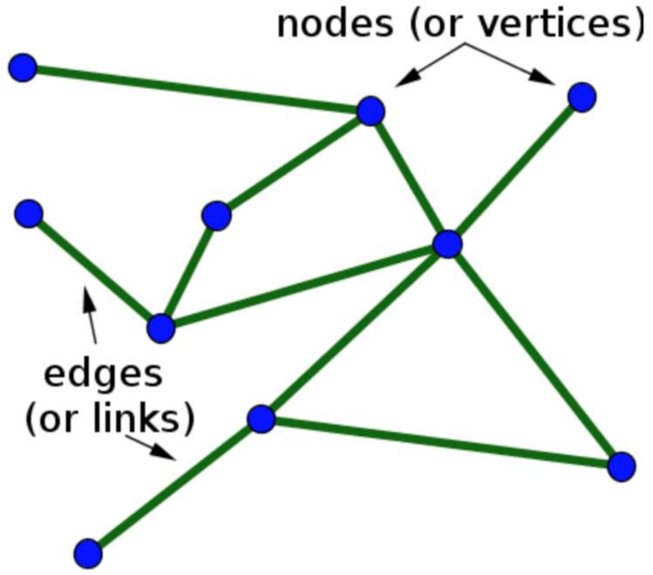
Asst. Prof. Dr. Özge Öztimur Karadağ

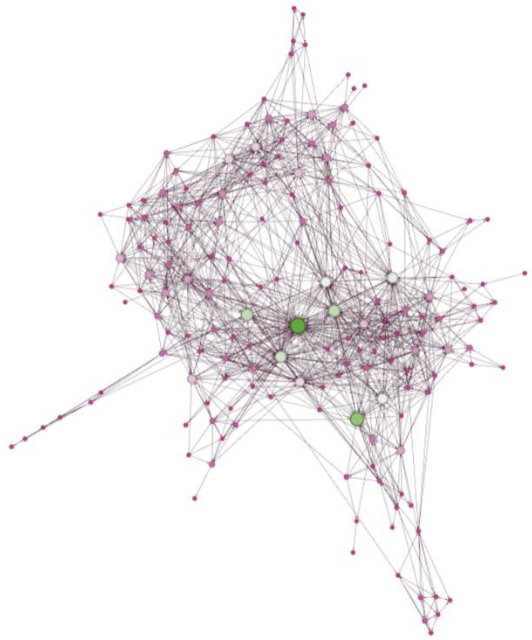
Graphs

- In the real world, many problems are represented in terms of objects and connections between them. For example, in an airline route map, we might be interested in questions like: “What’s the fastest way to go from Hyderabad to New York?” *or* “What is the cheapest way to go from Hyderabad to New York?” To answer these questions we need information about connections (airline routes) between objects (towns).
- Graphs are data structures used for solving these kinds of problems.

Graph – Definitions

- Graph: set of vertices and edges that model many problems in CS.





- Footballers (vertices) are connected (edges) if they played at the same team anytime in their careers.
- People are connected if they are friends, e.g., Facebook network.

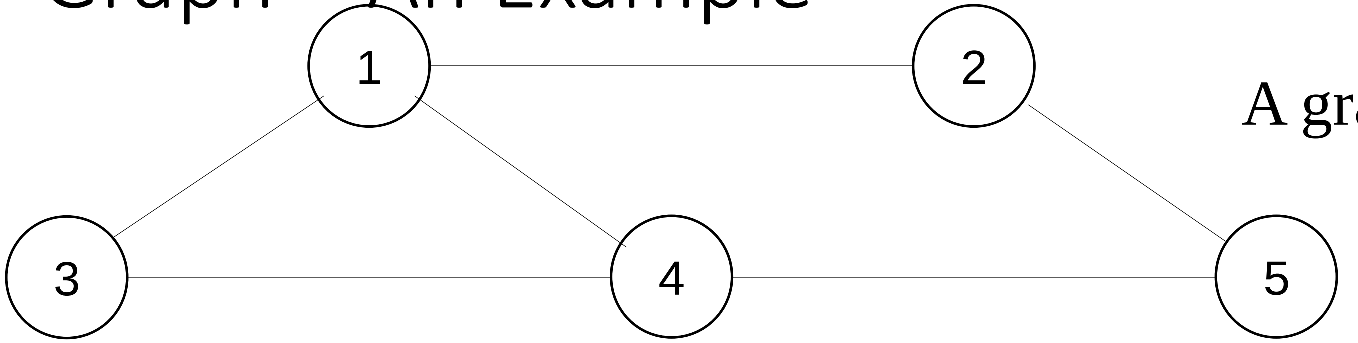
Graph – Definitions

- 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, …, 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:

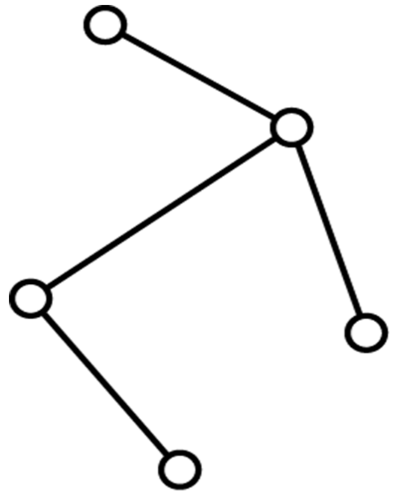
V = {1,2,3,4,5}

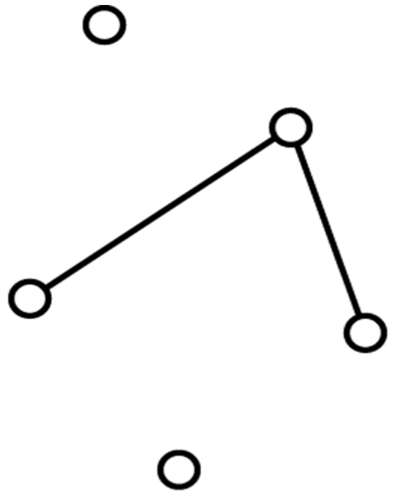
E = { (1,2),(1,3),(1,4),(2,5),(3,4),(4,5), (2,1),(3,1),(4,1),(5,2),(4,3),(5,4) }

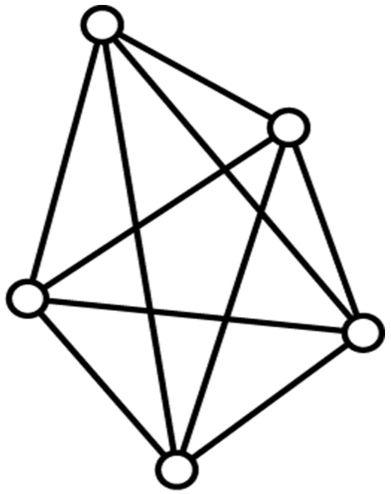
- *Adjacent:*
  1 and 2 are adjacent -- 1 is adjacent to 2 and 2 is adjacent to 1
- *Path:*
  1,2,5 (a simple path), 1,3,4,1,2,5 (a path but not a simple path)
- *Cycle:*
  1,3,4,1 (a simple cycle), 1,3,4,1,4,1 (cycle, but not simple cycle)

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**

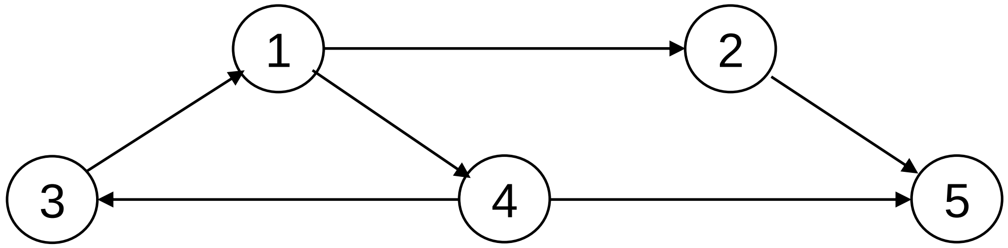

(c) **complete**

Directed Graphs

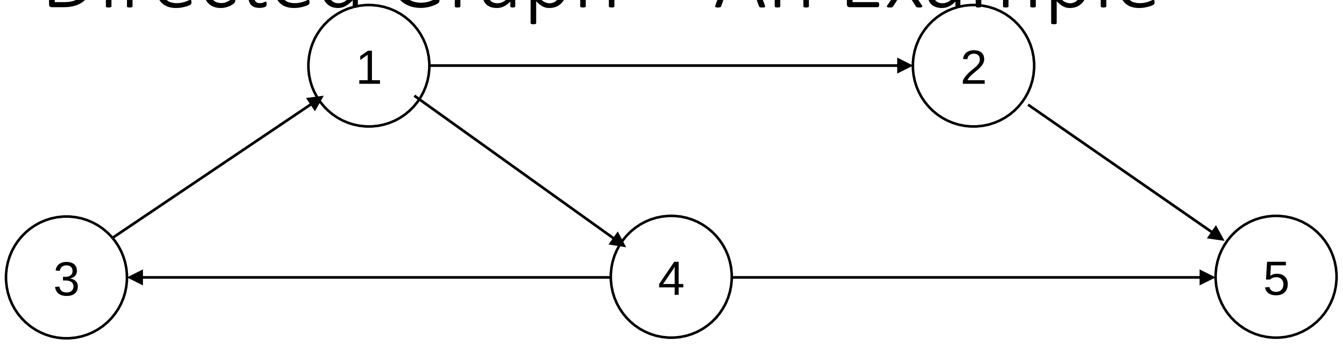
- If the edge pair is ordered then the graph is called a **directed graph (**also called *digraphs)* .
- Each edge in a directed graph has a direction, and each edge is called a ***directed edge***.
- Definitions given for undirected graphs apply also to directed graphs, with changes that account for direction.
- Vertex w is ***adjacent to*** v iff (v,w) $\in$ 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, …, 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 dir. 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:*

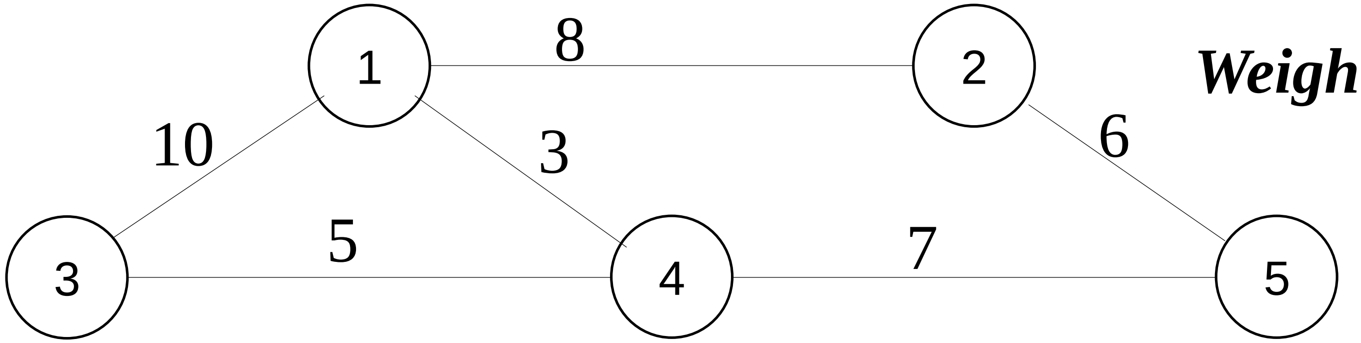
  2 is adjacent to 1, but 1 is NOT adjacent to 2
- *Path:*

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

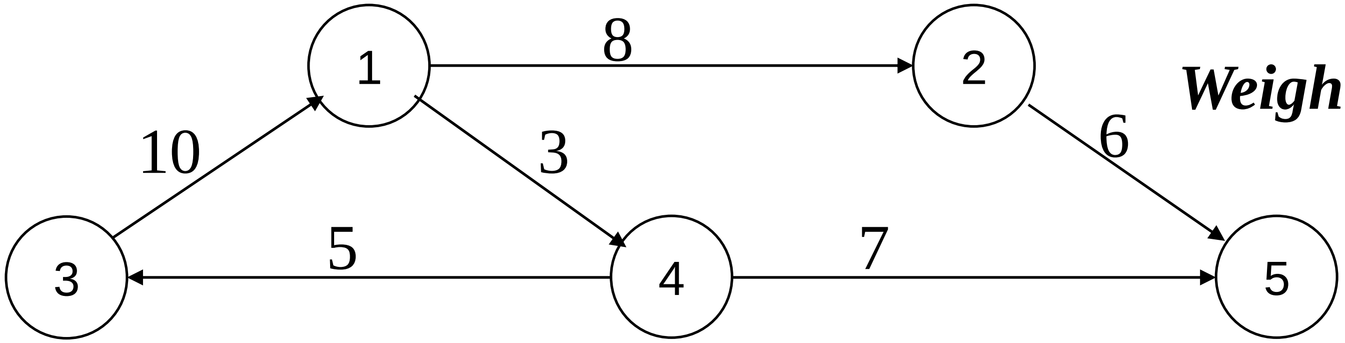
  1,4,3,1 (a directed cycle),

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

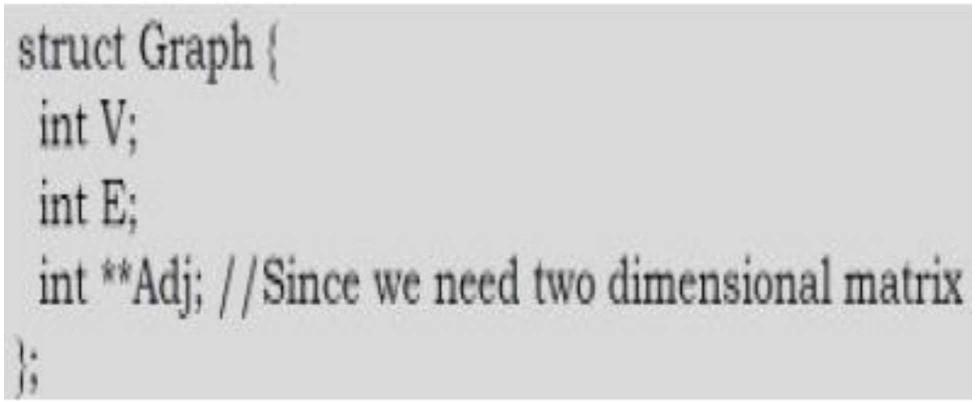
Graph Representation

As in other ADTs, to manipulate graphs we need to represent them in some useful form. Basically, there are three ways of doing this:

- *Adjacency Matrix*
- *Adjacency List*
- *Adjacency Set*

Graph Declaration for Adjacency Matrix

- First, let us look at the components of the graph data structure. To represent graphs, we need the number of vertices, the number of edges and also their interconnections. So, the graph can be declared as:



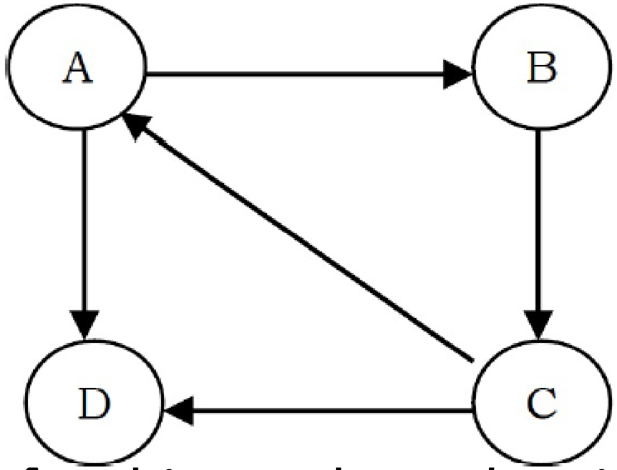
```
struct Graph {
  int V;
  int E;
  int **Adj; //Since we need two dimensional matrix
};
```

Graph Declaration for Adjacency Matrix (Continued)

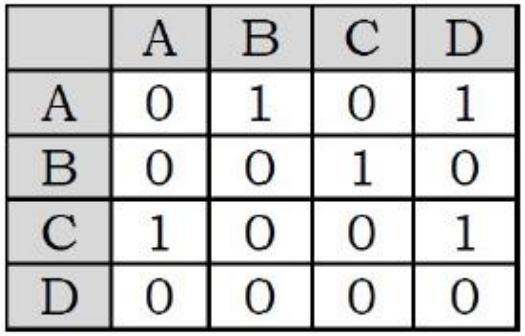
- In this method, we use a matrix with size *V* × *V*. The values of matrix are boolean. Let us assume the matrix is *Adj*. The value *Adj*[*u, v*] is set to 1 if there is an edge from vertex u to vertex v and 0 otherwise.
- In the matrix, each edge is represented by two bits for undirected graphs. That means, an edge from **u** to **v** is represented by 1 value in both *Adj*[*u,v*] and *Adj*[*v,u*]. To save time, we can process only half of this symmetric matrix.

Graph Declaration for Adjacency Matrix (Continued)

- If the graph is a directed graph then we need to mark only one entry in the adjacency matrix. As an example, consider the directed graph below.



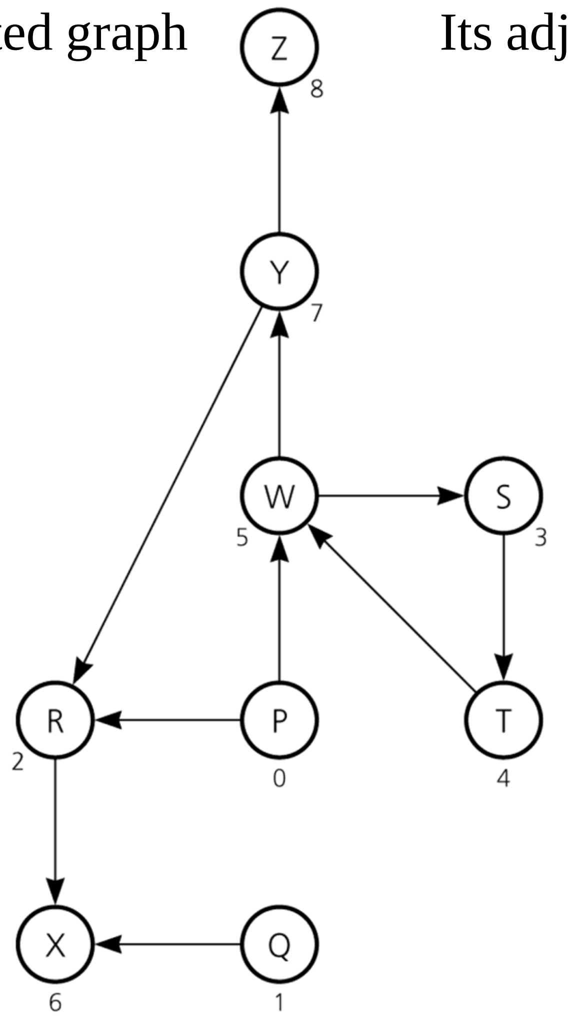
- The adjacency matrix for this graph can be given as:



	A	B	C	D
A	0	1	0	1
B	0	0	1	0
C	1	0	0	1
D	0	0	0	0

Adjacency Matrix – Example1

A directed graph

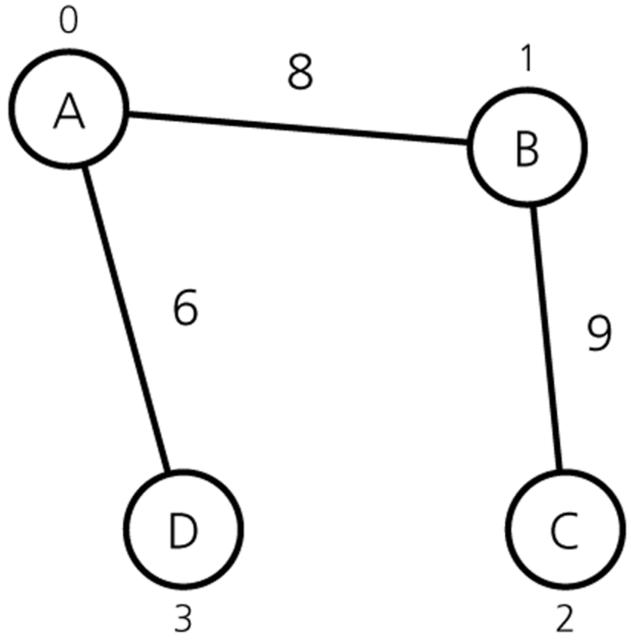


Its adjacency matrix

		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

Adjacency Matrix – Example2

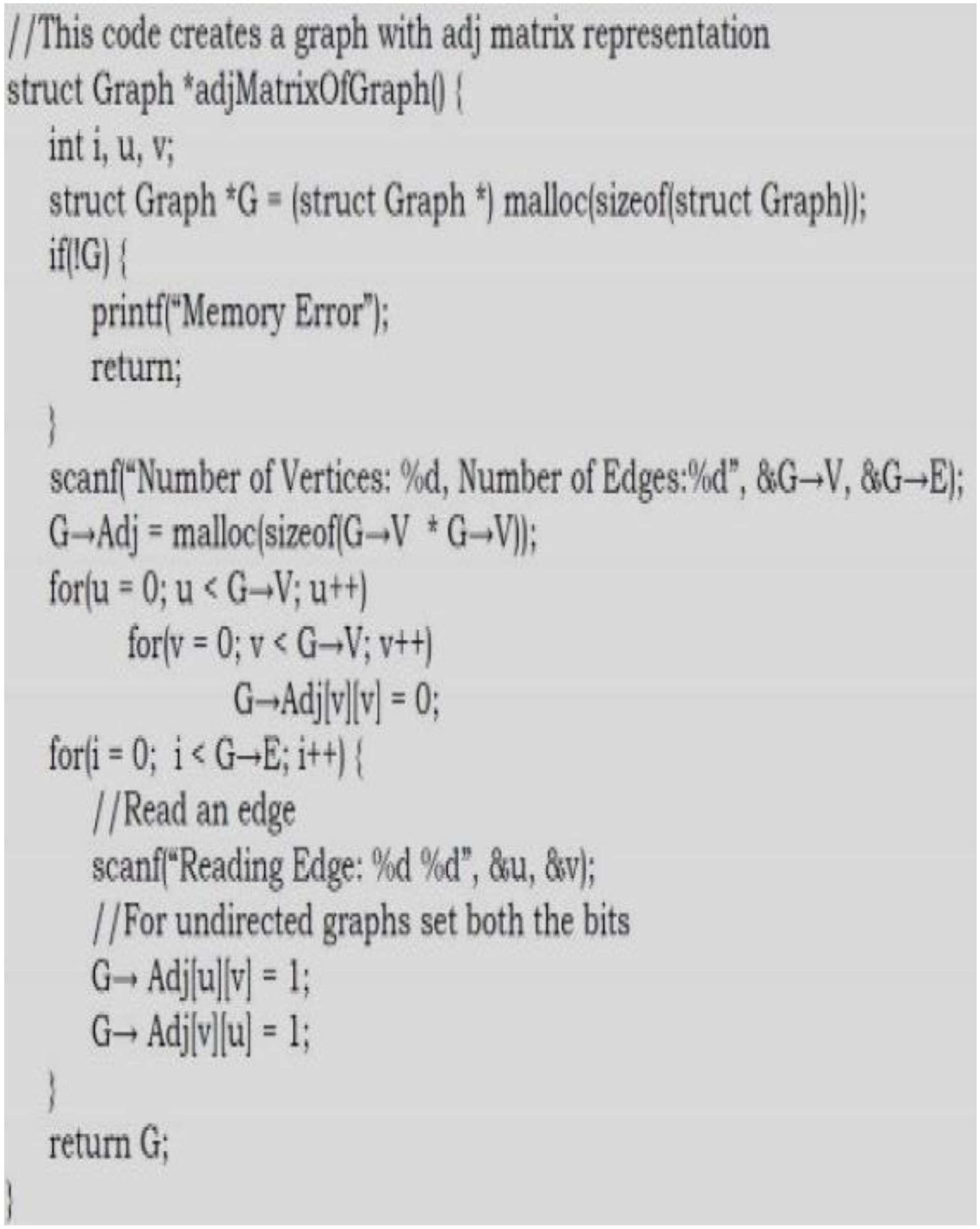
An Undirected Weighted Graph



Its Adjacency Matrix

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

Graph Declaration for Adjacency Matrix (Continued)



```
//This code creates a graph with adj matrix representation
struct Graph *adjMatrixOfGraph() {
    int i, u, v;
    struct Graph *G = (struct Graph *) malloc(sizeof(struct Graph));
    if(!G) {
        printf("Memory Error");
        return;
    }
    scanf("Number of Vertices: %d, Number of Edges:%d", &G→V, &G→E);
    G→Adj = malloc(sizeof(G→V  * G→V));
    for(u = 0; u < G→V; u++)
          for(v = 0; v < G→V; v++)
                    G→Adj[v][v] = 0;
    for(i = 0;  i < G→E; i++) {
        //Read an edge
        scanf("Reading Edge: %d %d", &u, &v);
        //For undirected graphs set both the bits
        G→ Adj[u][v] = 1;
        G→ Adj[v][u] = 1;
    }
    return G;
}
```

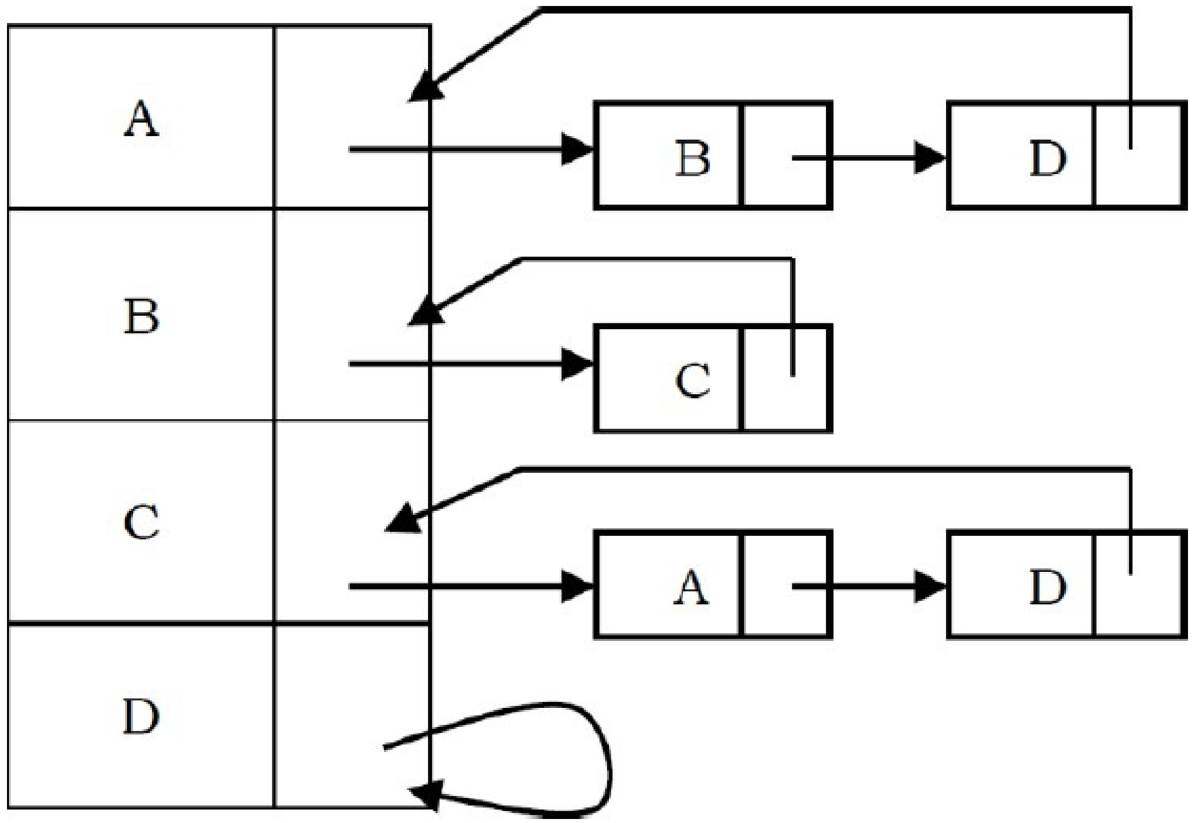
To read a graph, one way is to first read the vertex names and then read pairs of vertex names (edges). The code reads an undirected graph.

Graph Declaration for Adjacency List

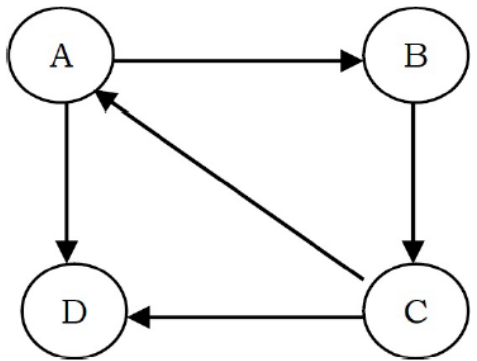
- In this representation all the vertices connected to a vertex *v* are listed on an adjacency list for that vertex *v*. This can be easily implemented with linked lists. That means, for each vertex *v* we use a linked list and list nodes represent the connection between *v* and other vertices to which *v* has an edge.
- The total number of linked lists is equal to the number of vertices in the graph.

Graph Declaration for Adjacency List (Continued)

- Considering the same example as that of the adjacency matrix, the adjacency list representation can be given as:

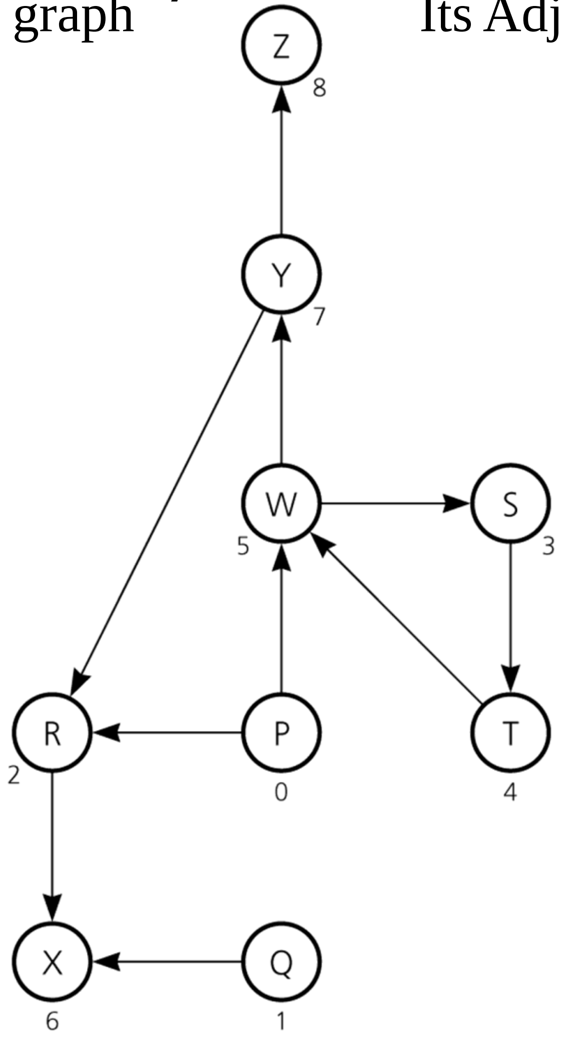


Since vertex A has an edge for B and D, we have added them in the adjacency list for A. The same is the case with other vertices as well.

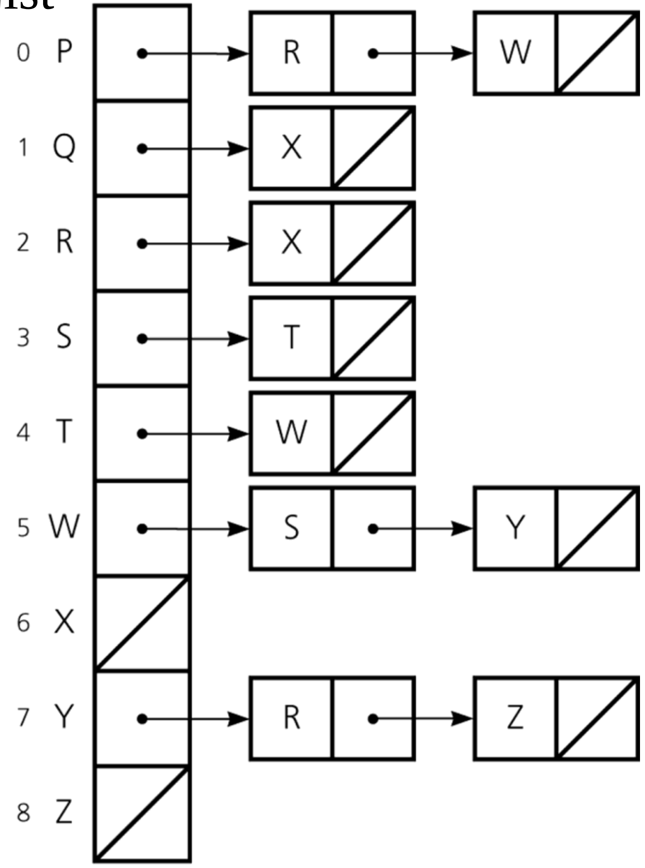


Adjacency List – Example1

A directed graph

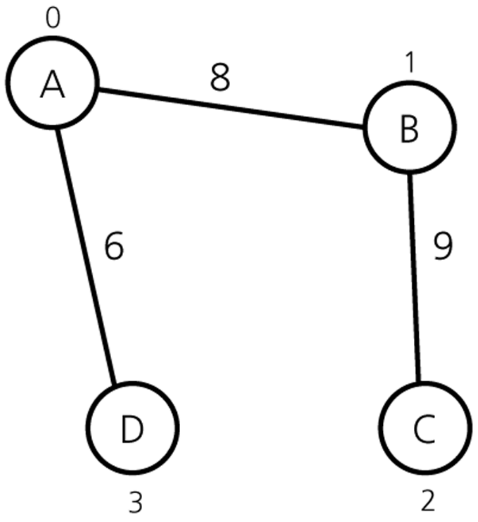


Its Adjacency List

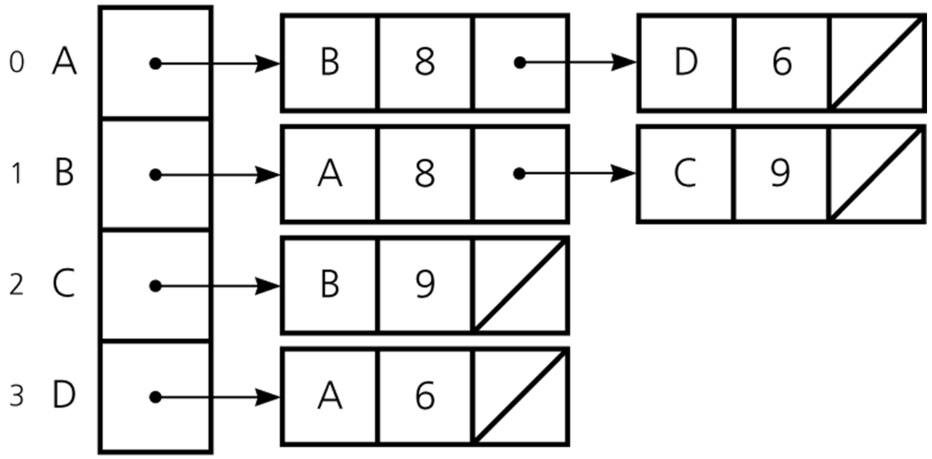


Adjacency List – Example2

An Undirected Weighted Graph



Its Adjacency List



Disadvantages of Adjacency Lists

- Using adjacency list representation we cannot perform some operations efficiently. As an example, consider the case of deleting a node. In adjacency list representation, it is not enough if we simply delete a node from the list representation. For each node on the adjacency list of that node specifies another vertex. We need to search other nodes linked list also for deleting it. This problem can be solved by linking the two list nodes that correspond to a particular edge and making the adjacency lists doubly linked. But all these extra links are risky to process.

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.

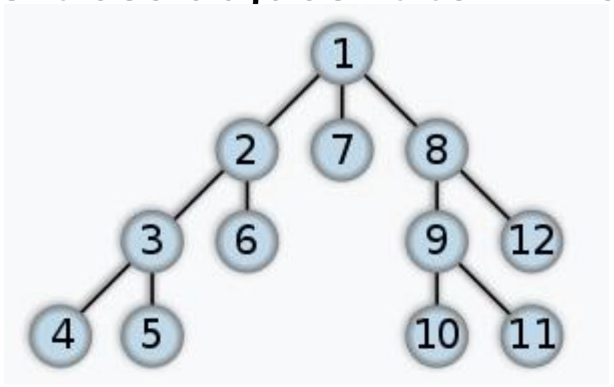
- An adjacency list often requires less space than an adjacency matrix.
  - Adjacency Matrix: Space requirement is $O(|V|^2)$
  - Adjacency List : Space requirement is $O(|E| + |V|)$.
  - Adjacency matrix is better if the graph is dense (too many edges)
  - Adjacency list is better if the graph is sparse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

Graph Traversals

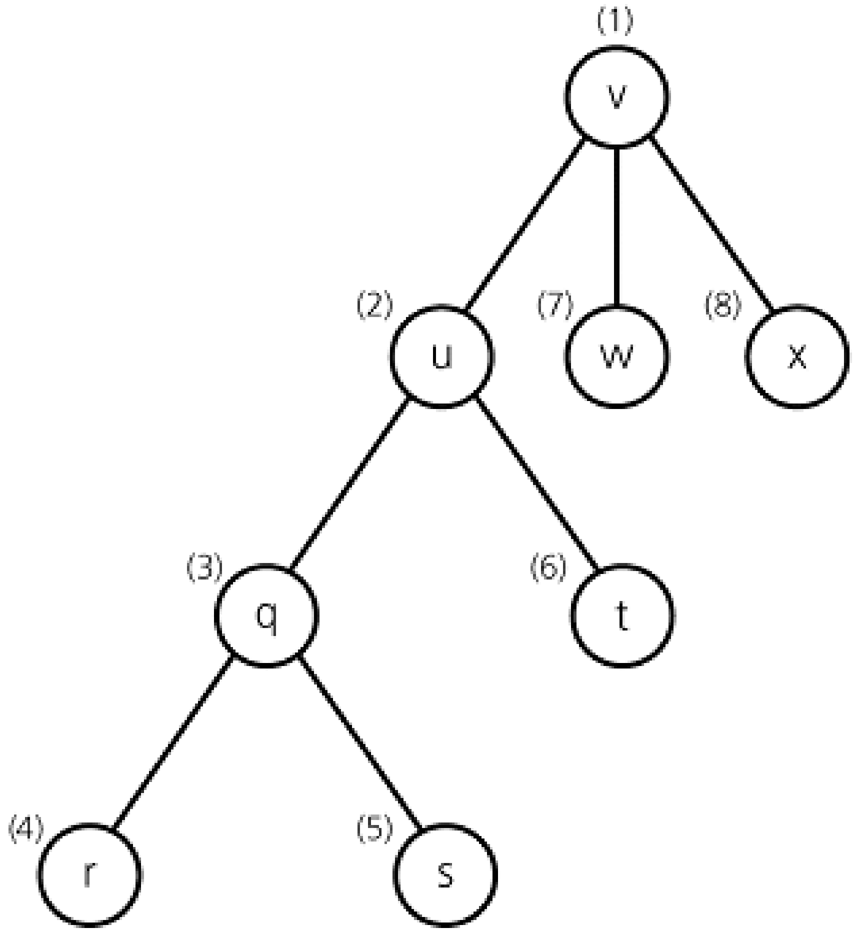
- To solve problems on graphs, we need a mechanism for traversing the graphs. Graph traversal algorithms are also called *graph search* algorithms. Like tree traversal algorithms (Inorder, Preorder, Postorder and Level-Order traversals), graph search algorithms can be thought of as starting at some source vertex in a graph and "searching" the graph by going through the edges and marking the vertices. Now, we will discuss two such algorithms for traversing the graphs.
  - *Depth First Search [DFS]*
  - *Breadth First Search [BFS]*

Depth-First Search

- For a given vertex v, the ***depth-first search*** 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 search*** 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 Search – Example



• A depth-first search 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.

Depth First Search [DFS]

- DFS algorithm works in a manner similar to preorder traversal of the trees. Like preorder traversal, internally this algorithm also uses stack.
- Let us consider the following example. Suppose a person is trapped inside a maze. To come out from that maze, the person visits each path and each intersection (in the worst case). Let us say the person uses two colors of paint to mark the intersections already passed. When discovering a new intersection, it is marked grey, and he continues to go deeper.

Depth First Search [DFS] (Continued)

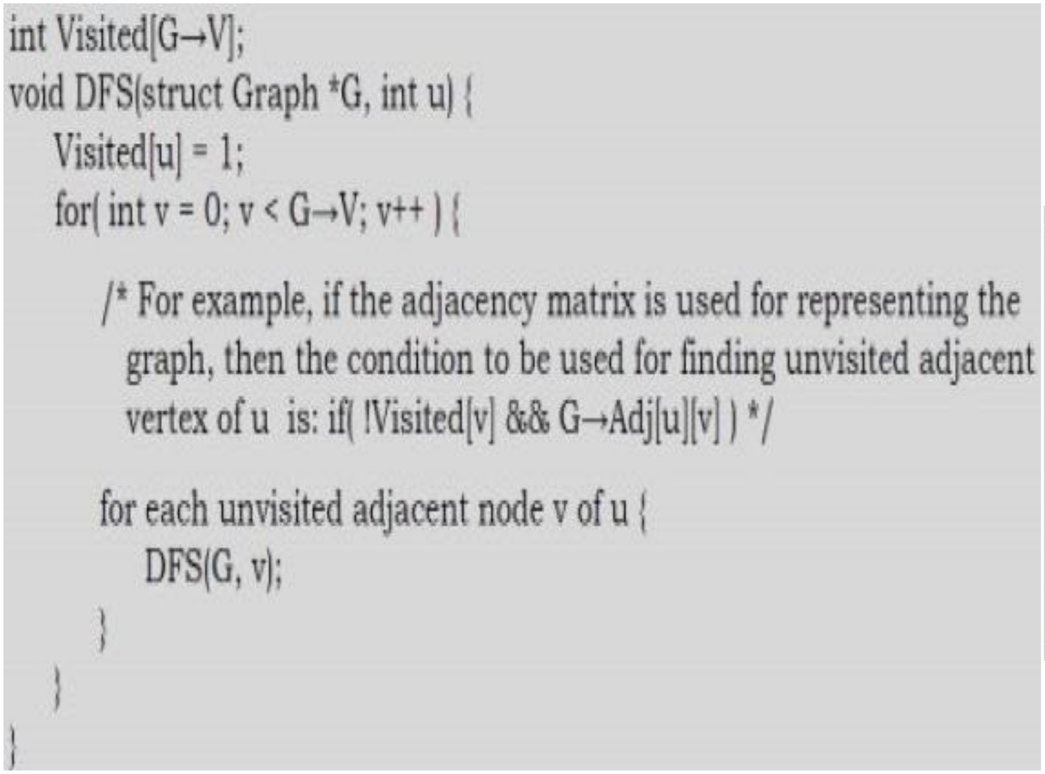
- After reaching a “dead end” the person knows that there is no more unexplored path from the grey intersection, which now is completed, and he marks it with black. This “dead end” is either an intersection which has already been marked grey or black, or simply a path that does not lead to an intersection.
- The intersections of the maze are the vertices and the paths between the intersections are the edges of the graph. The process of returning from the “dead end” is called *backtracking*. We are trying to go away from the starting vertex into the graph as deep as possible, until we have to backtrack to the preceding grey vertex.

Depth First Search [DFS] (Continued)

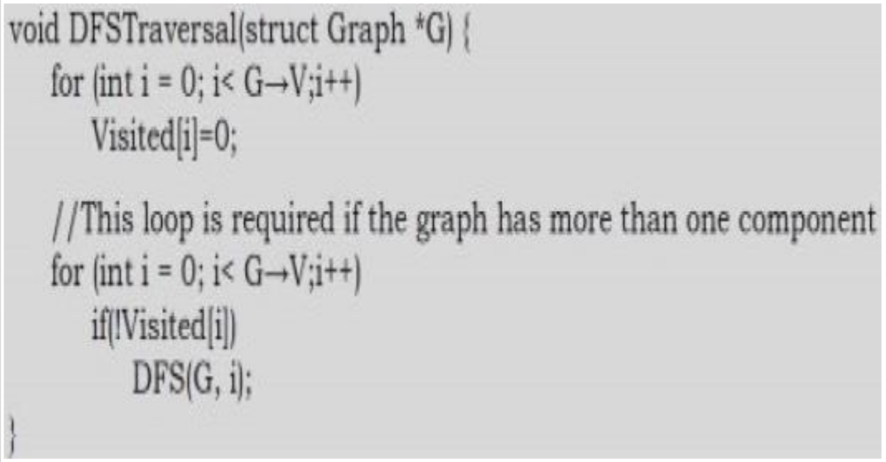
- For most algorithms boolean classification, unvisited/visited is enough.
- Initially all vertices are marked unvisited (false). The DFS algorithm starts at a vertex *u* in the graph. By starting at vertex *u* it considers the edges from *u* to other vertices. If the edge leads to an already visited vertex, then backtrack to current vertex *u*. If an edge leads to an unvisited vertex, then go to that vertex and start processing from that vertex. That means the new vertex becomes the current vertex. Follow this process until we reach the dead-end. At this point start *backtracking.*

Depth First Search [DFS] (Continued)

- The process terminates when backtracking leads back to the start vertex. The algorithm based on this mechanism is given below: assume Visited[] is a global array.



```
int Visited[G→V];
void DFS(struct Graph *G, int u) {
    Visited[u] = 1;
    for( int v = 0; v < G→V; v++ ) {

        /* For example, if the adjacency matrix is used for representing the
          graph, then the condition to be used for finding unvisited adjacent
          vertex of u  is: if( !Visited[v] && G→Adj[u][v] ) */

        for each unvisited adjacent node v of u {
            DFS(G, v);
        }
    }
}
```



```
void DFSTraversal(struct Graph *G) {
    for (int i = 0; i< G→V;i++)
        Visited[i]=0;

    //This loop is required if the graph has more than one component
    for (int i = 0; i< G→V;i++)
        if(!Visited[i])
            DFS(G, i);
}
```

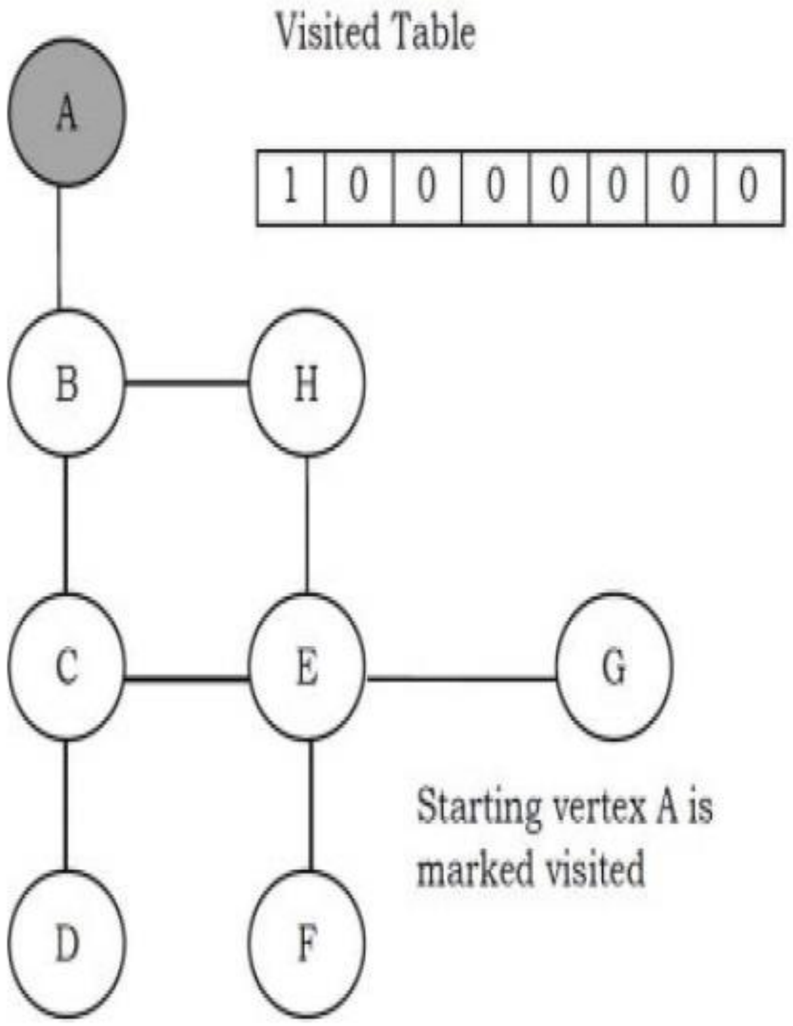
Recursive Depth-First Search Algorithm

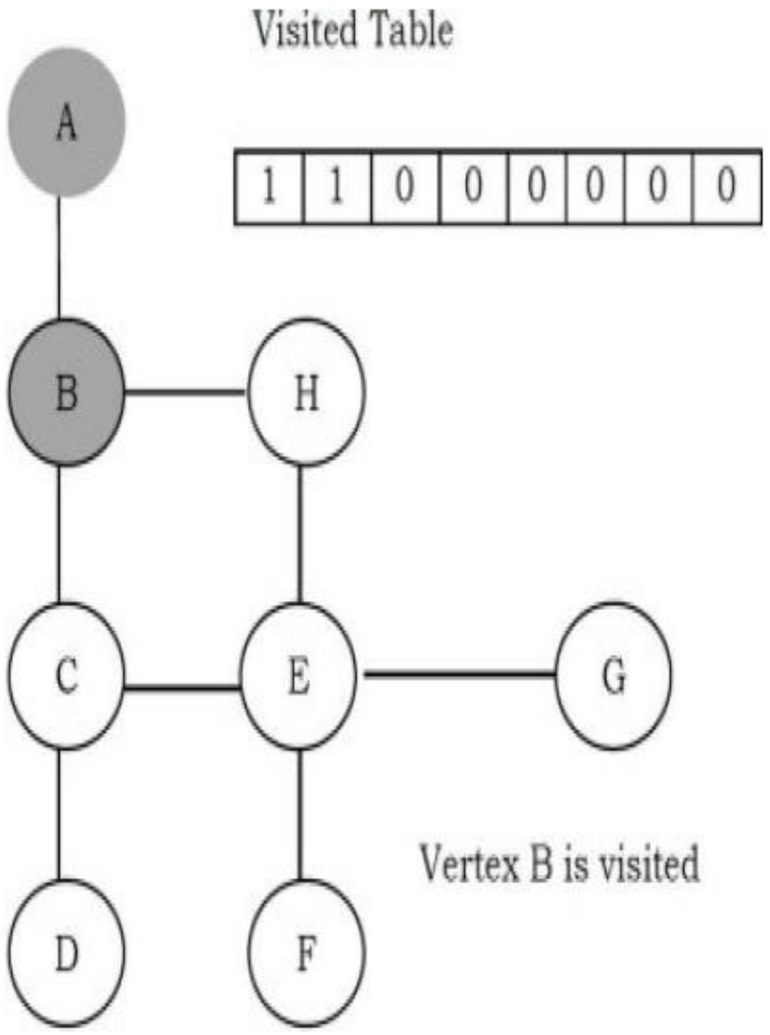
```
dfs(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using depth-first strategy
// Recursive Version
 Mark v as visited;
 for (each unvisited vertex u adjacent to v)
     dfs(u)
}
```

Iterative Depth-First Search Algorithm

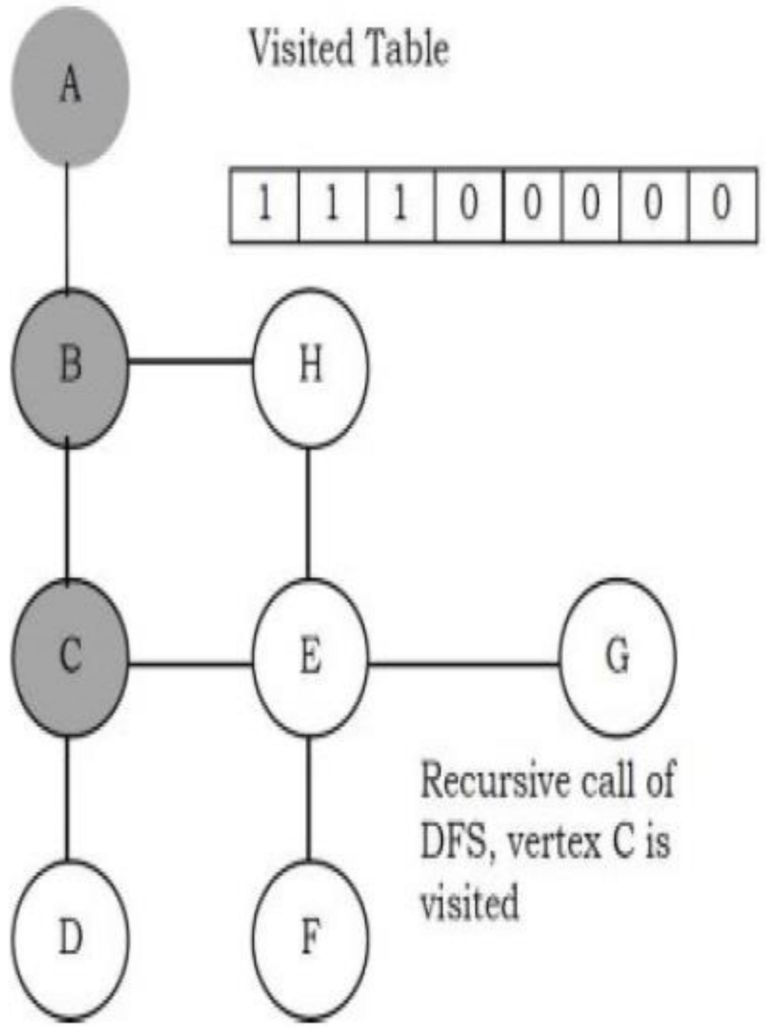
```
dfs(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using depth-first strategy: Iterative Version
  s.createStack();
  // push v into the stack and mark it
  s.push(v);
  Mark v as visited;
  while (!s.isEmpty()) {
     if (no unvisited vertices are adjacent to the vertex on
         the top of stack)
        s.pop();  // backtrack
     else {
        Select an unvisited vertex u adjacent to the vertex
           on the top of the stack;
        s.push(u);
        Mark u as visited;
     }
  }
} ➔ O(V+E) worst-case time complexity via aggregate analysis.
```

Example

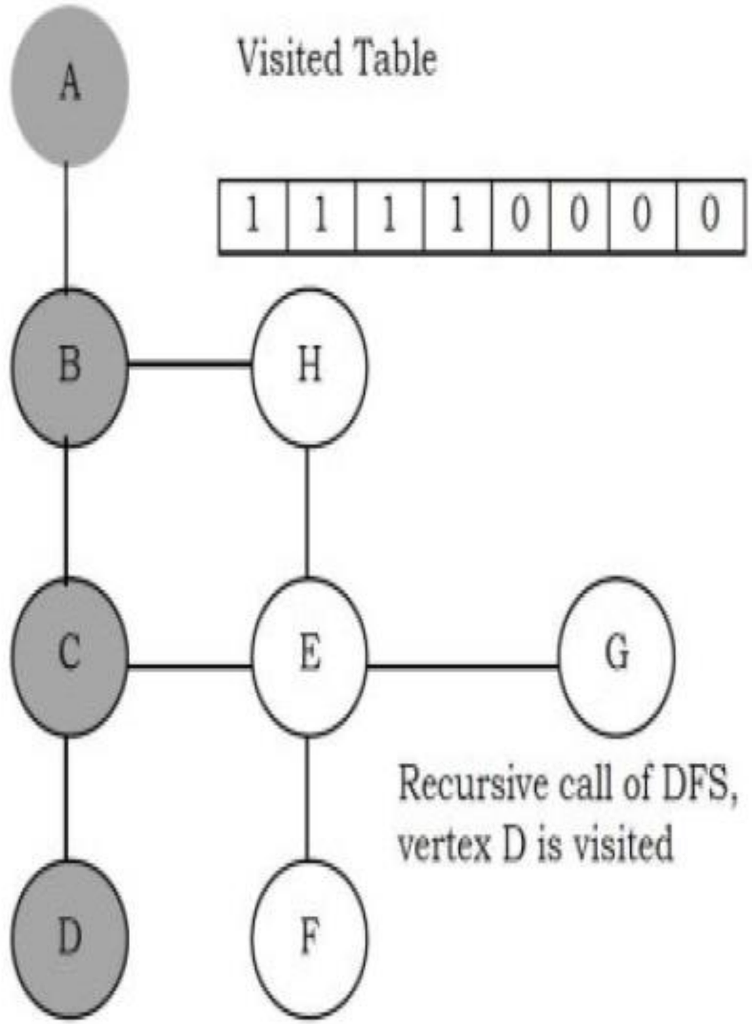

Visited Table
1 0 0 0 0 0 0 0
A
B
H
C
E
G
D
F
Starting vertex A is marked visited


Visited Table
1 1 0 0 0 0 0 0
A
B
H
C
E
G
D
F
Vertex B is visited

Example (Continued)

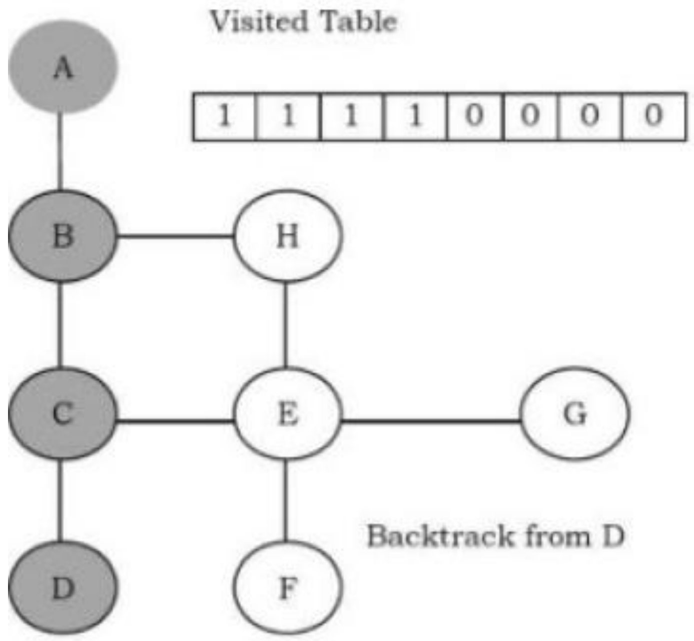


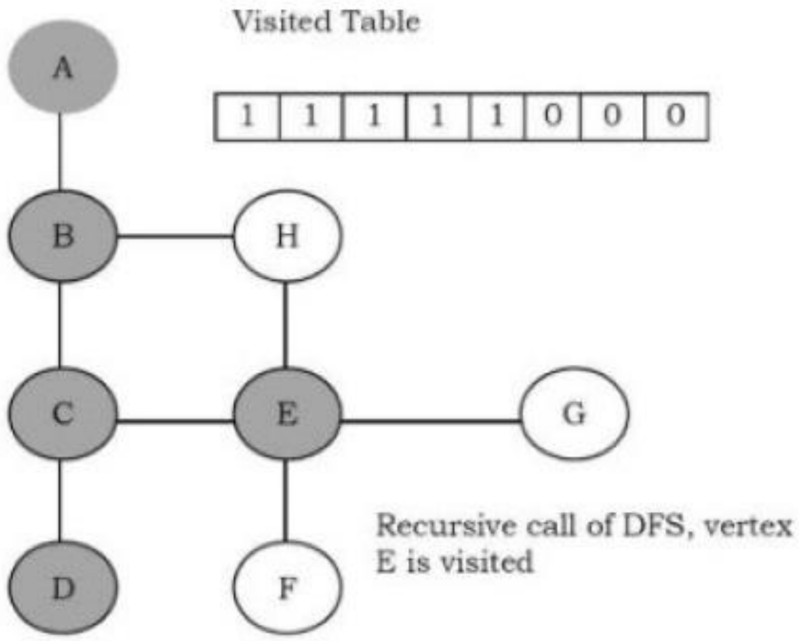
Recursive call of DFS, vertex C is visited



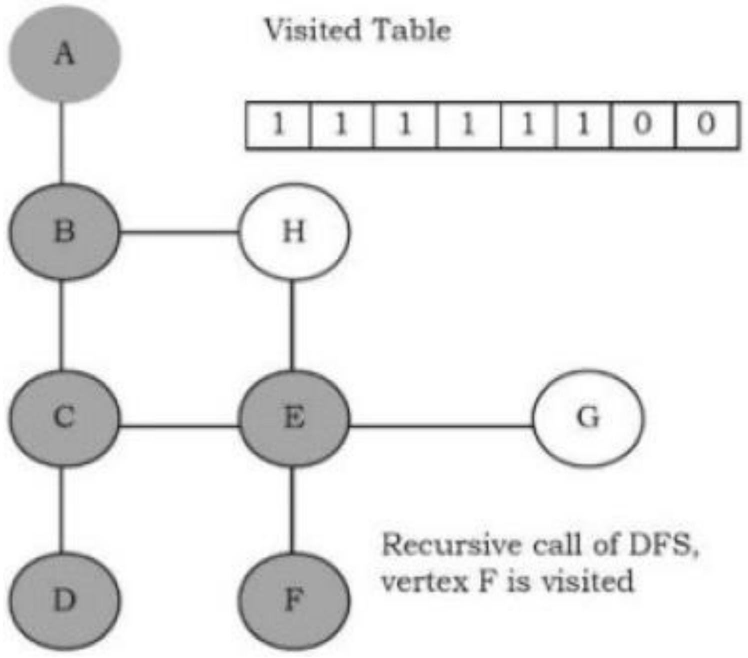
Recursive call of DFS, vertex D is visited

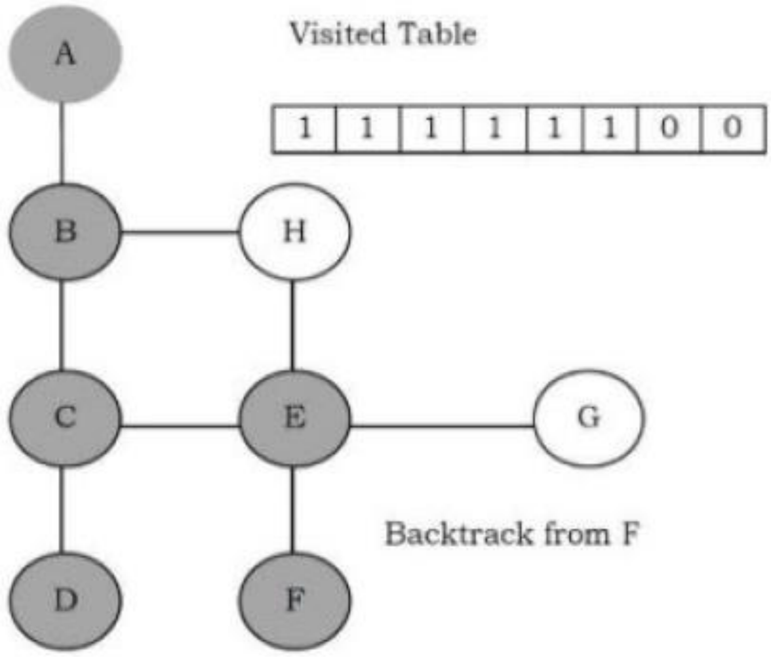
Example (Continued)



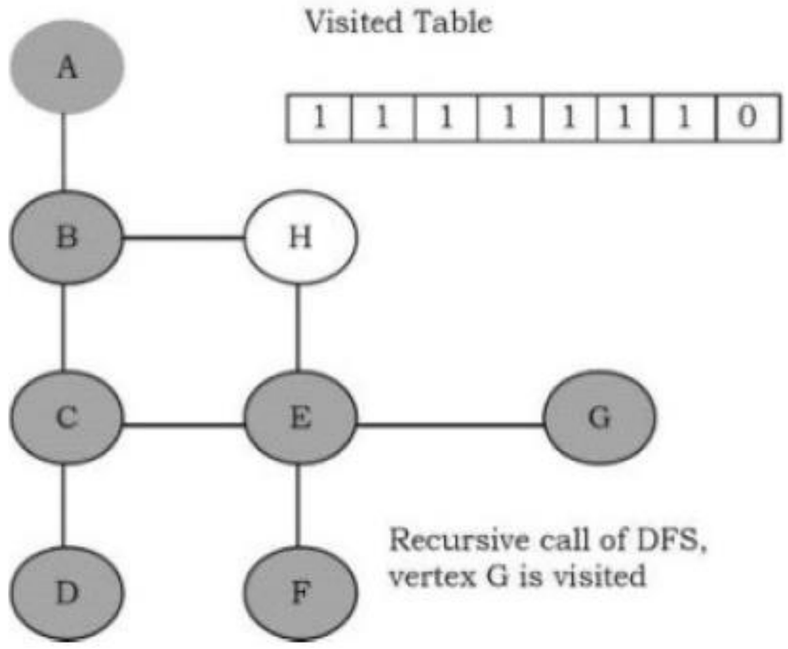


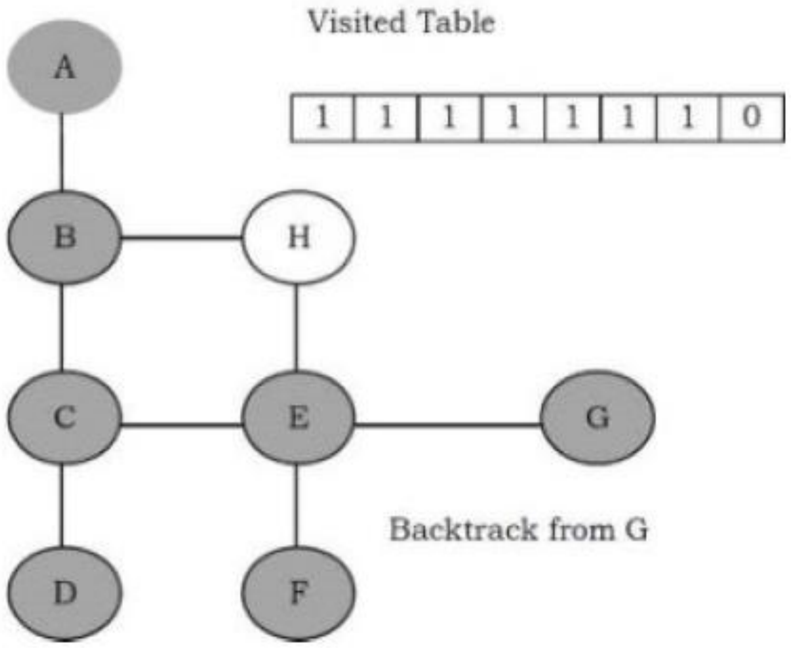
Example (Continued)



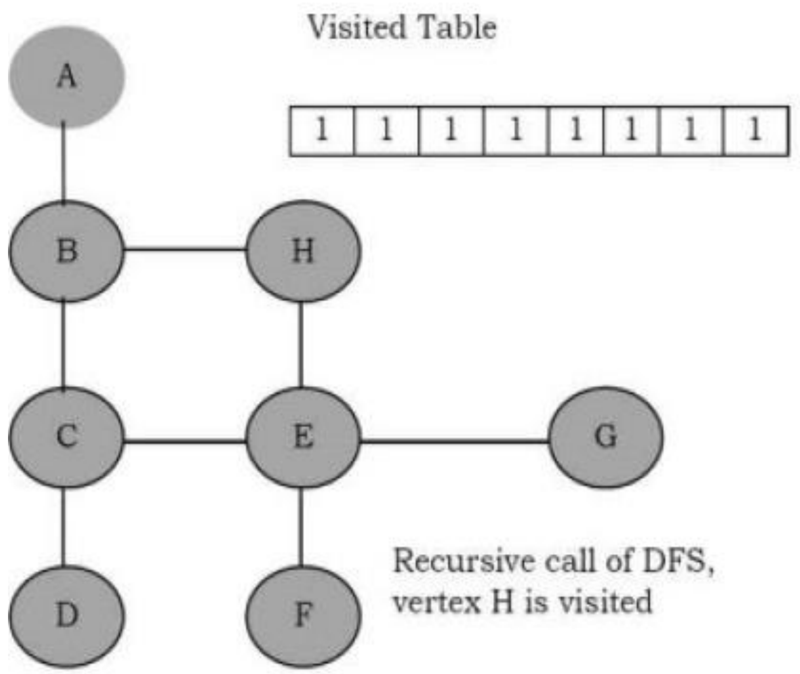


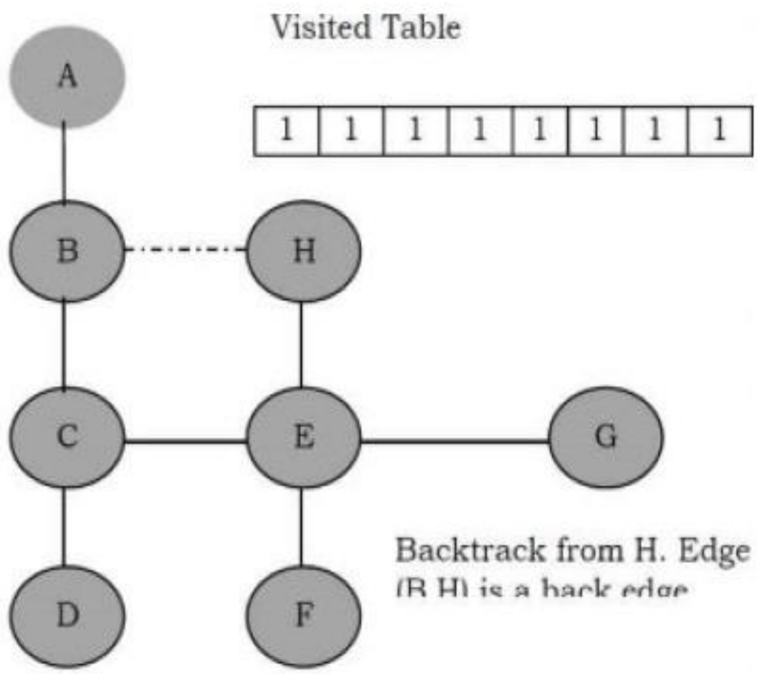
Example (Continued)



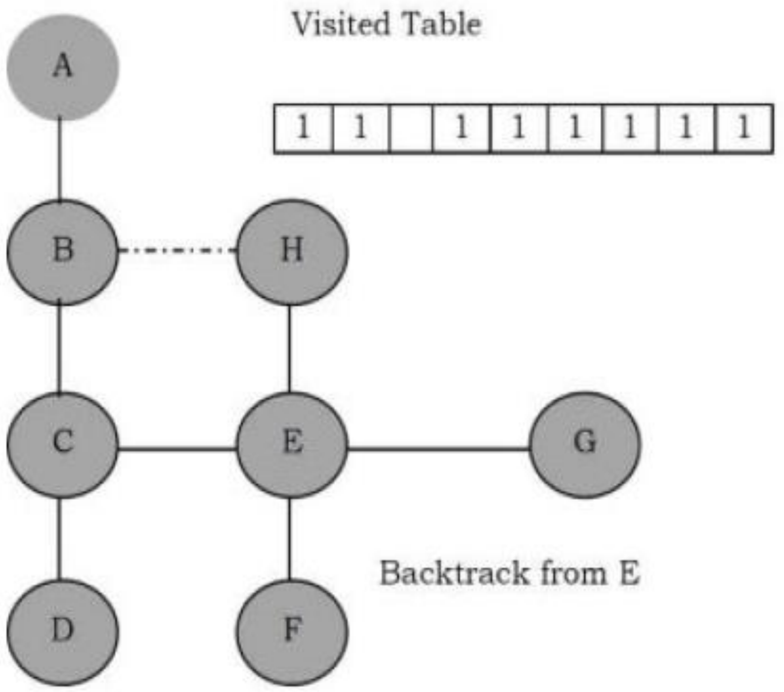


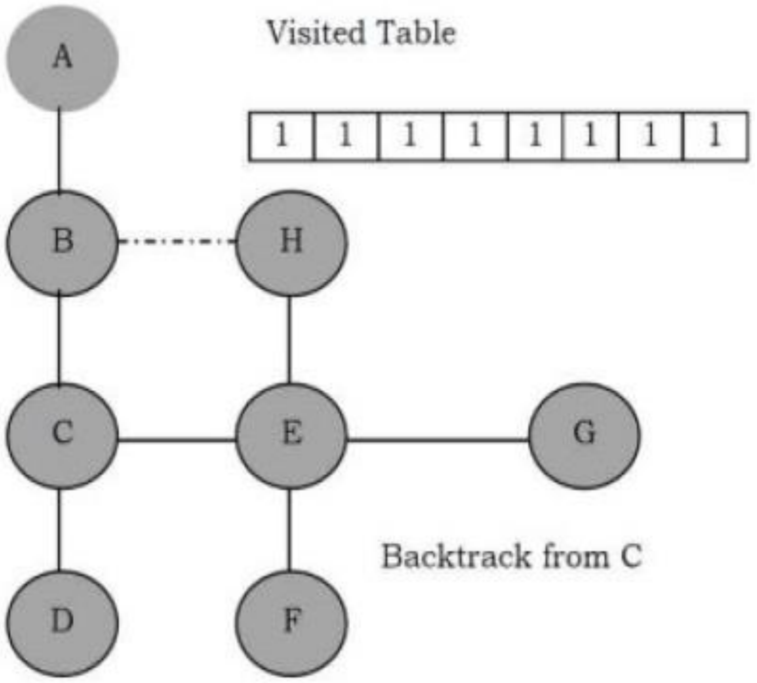
Example (Continued)



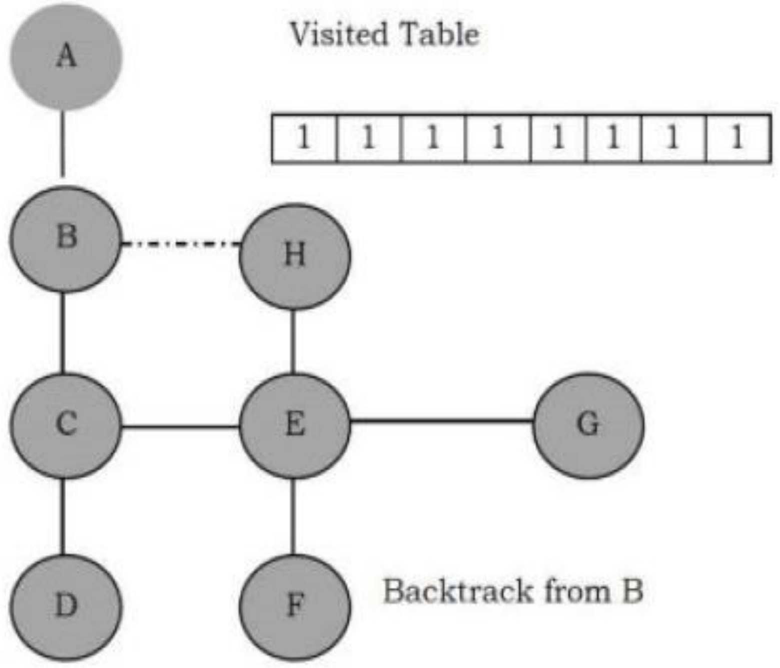


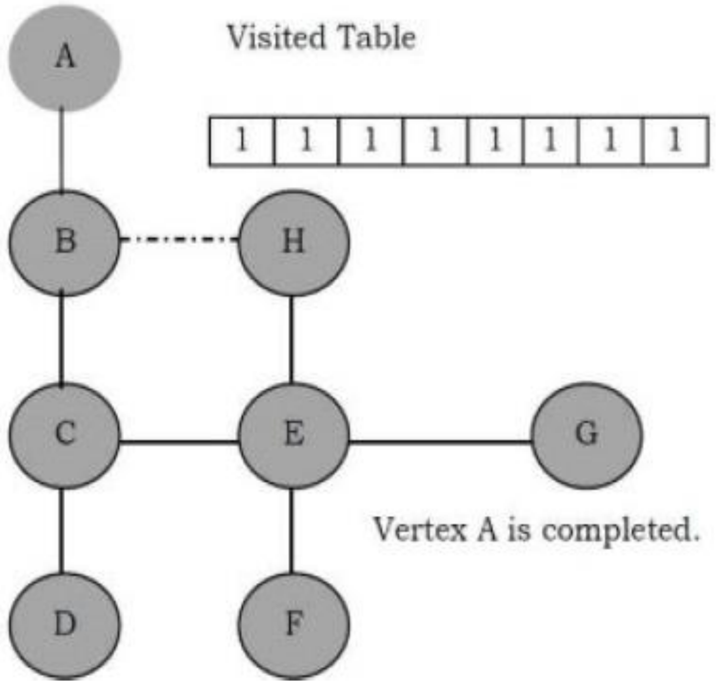
Example (Continued)



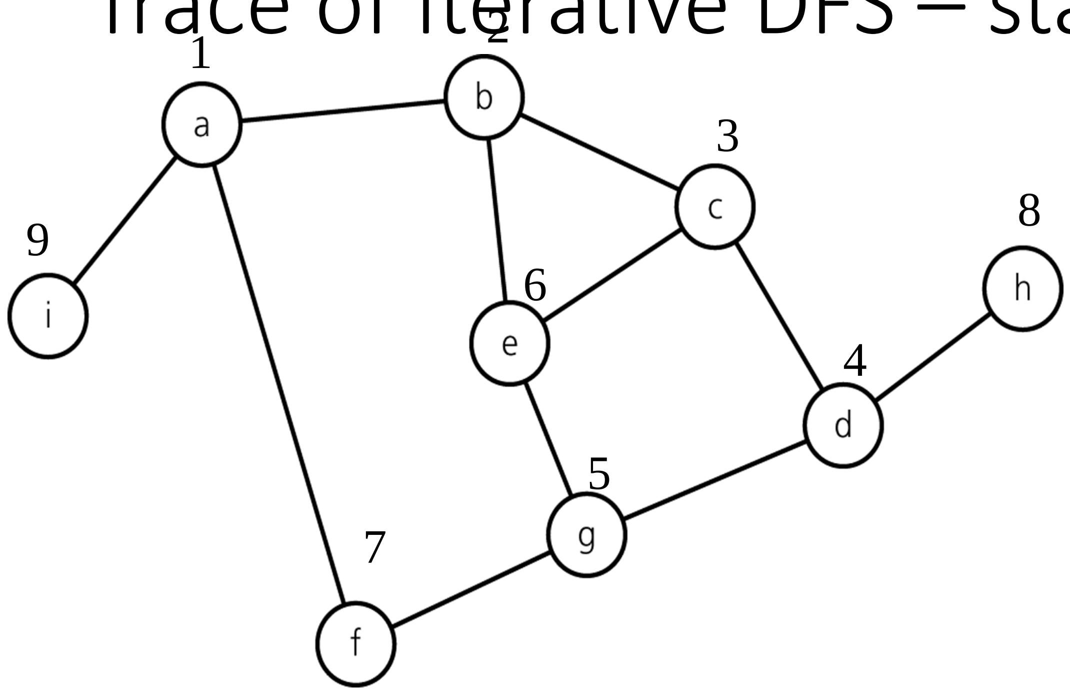


Example (Continued)





Trace of Iterative DFS – starting from vertex a



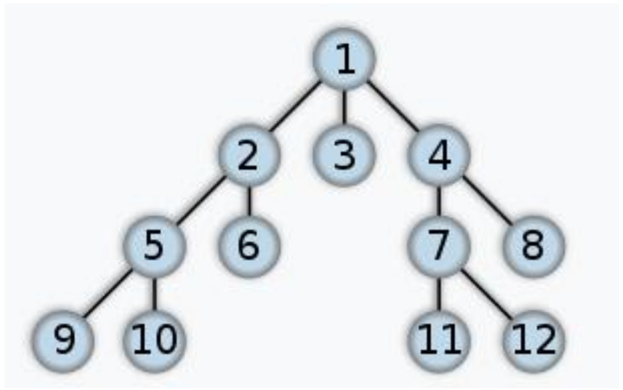
Node visited	Stack (bottom to top)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(backtrack)	a b c d g
f	a b c d g f
(backtrack)	a b c d g
(backtrack)	a b c d
h	a b c d h
(backtrack)	a b c d
(backtrack)	a b c
(backtrack)	a b
(backtrack)	a
i	a i
(backtrack)	a
(backtrack)	*(empty)*

Depth First Search [DFS] (Continued)

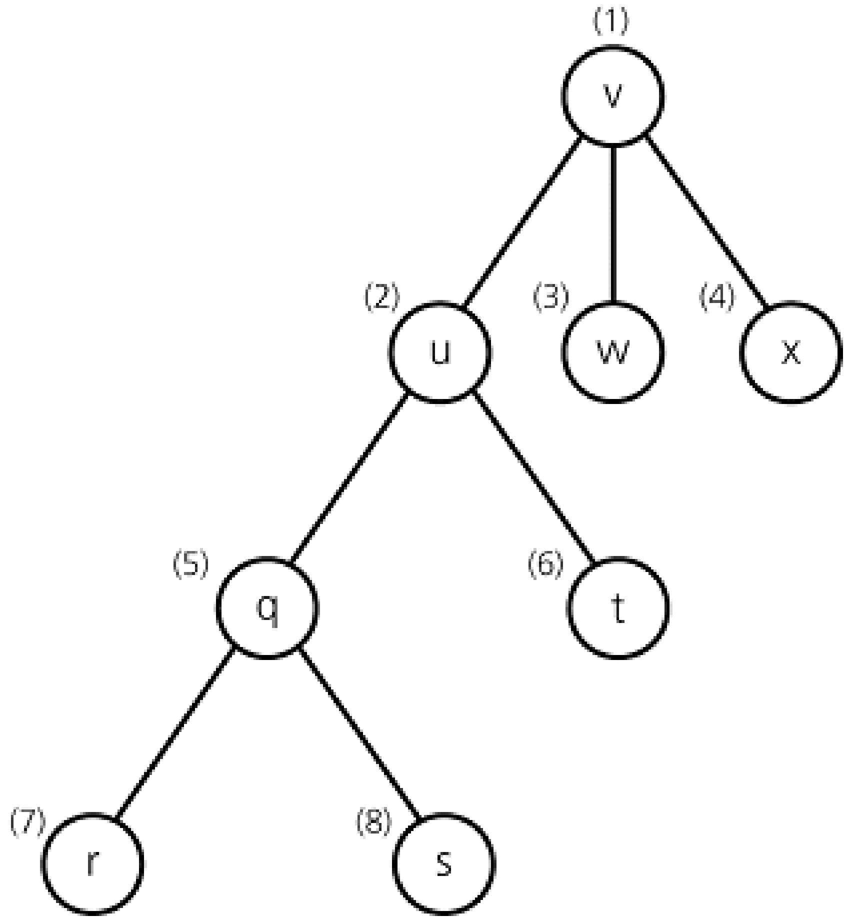
- The time complexity of DFS is O(V + E), if we use adjacency lists for representing the graphs. This is because we are starting at a vertex and processing the adjacent nodes only if they are not visited. Similarly, if an adjacency matrix is used for a graph representation, then all edges adjacent to a vertex can't be found efficiently, and this gives O(V^2) complexity.

Breadth-First Search

- After visiting a given vertex v, the breadth-first search algorithm visits every vertex adjacent to v that it can before visiting any other vertex.
- The breadth-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.



Breadth-First Search– Example



- A breadth-first search of the graph starting from vertex v.

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

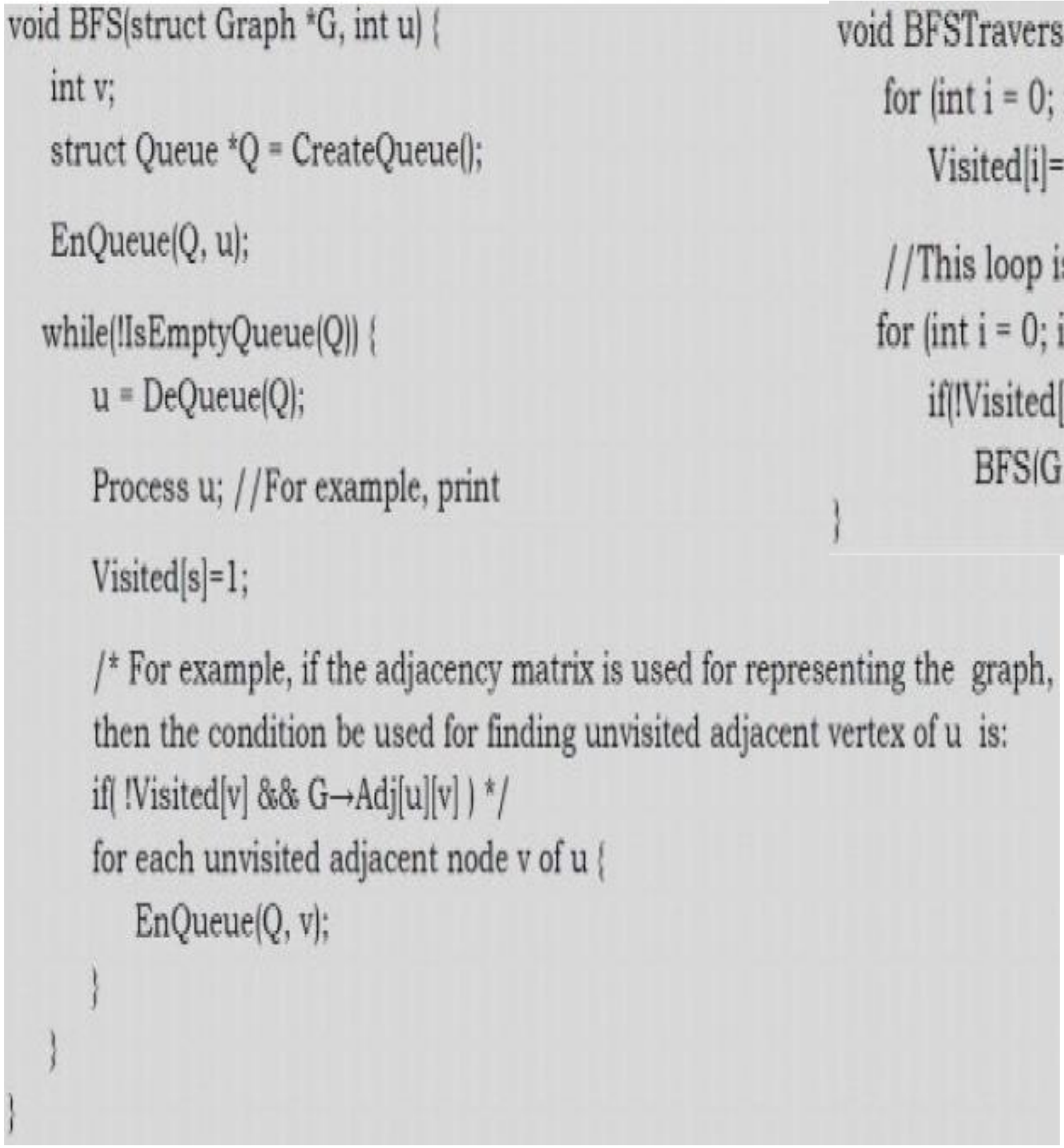
Breadth First Search [BFS]

- The BFS algorithm works similar to *level – order* traversal of the trees. Like *level – order* traversal, BFS also uses queues. In fact, *level – order* traversal got inspired from BFS. BFS works level by level. Initially, BFS starts at a given vertex, which is at level 0. In the first stage it visits all vertices at level 1 (that means, vertices whose distance is 1 from the start vertex of the graph). In the second stage, it visits all vertices at the second level. These new vertices are the ones which are adjacent to level 1 vertices.

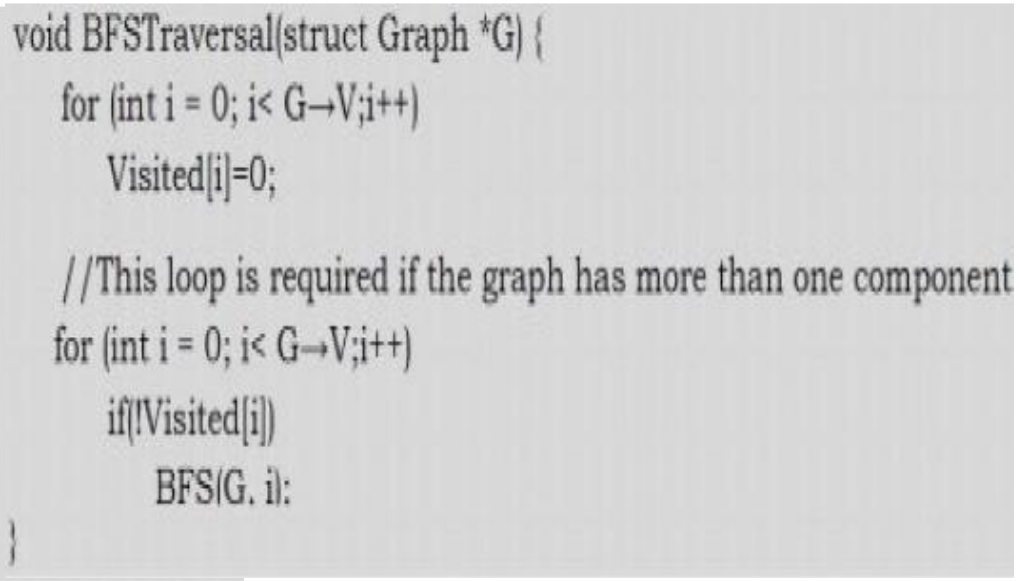
Breadth First Search [BFS] (Continued)

- BFS continues this process until all the levels of the graph are completed. Generally *queue* data structure is used for storing the vertices of a level.
- As similar to DFS, assume that initially all vertices are marked *unvisited* (*false*). Vertices that have been processed and removed from the queue are marked *visited* (*true*). We use a queue to represent the visited set as it will keep the vertices in the order of when they were first visited.

Breadth First Search [BFS] (Continued)



```
void BFS(struct Graph *G, int u) {
    int v;
    struct Queue *Q = CreateQueue();

    EnQueue(Q, u);

   while(!IsEmptyQueue(Q)) {
        u = DeQueue(Q);

        Process u; //For example, print

        Visited[s]=1;

        /* For example, if the adjacency matrix is used for representing the  graph,
        then the condition be used for finding unvisited adjacent vertex of u  is:
        if( !Visited[v] && G→Adj[u][v] ) */
        for each unvisited adjacent node v of u {
            EnQueue(Q, v);
        }
    }
}
```

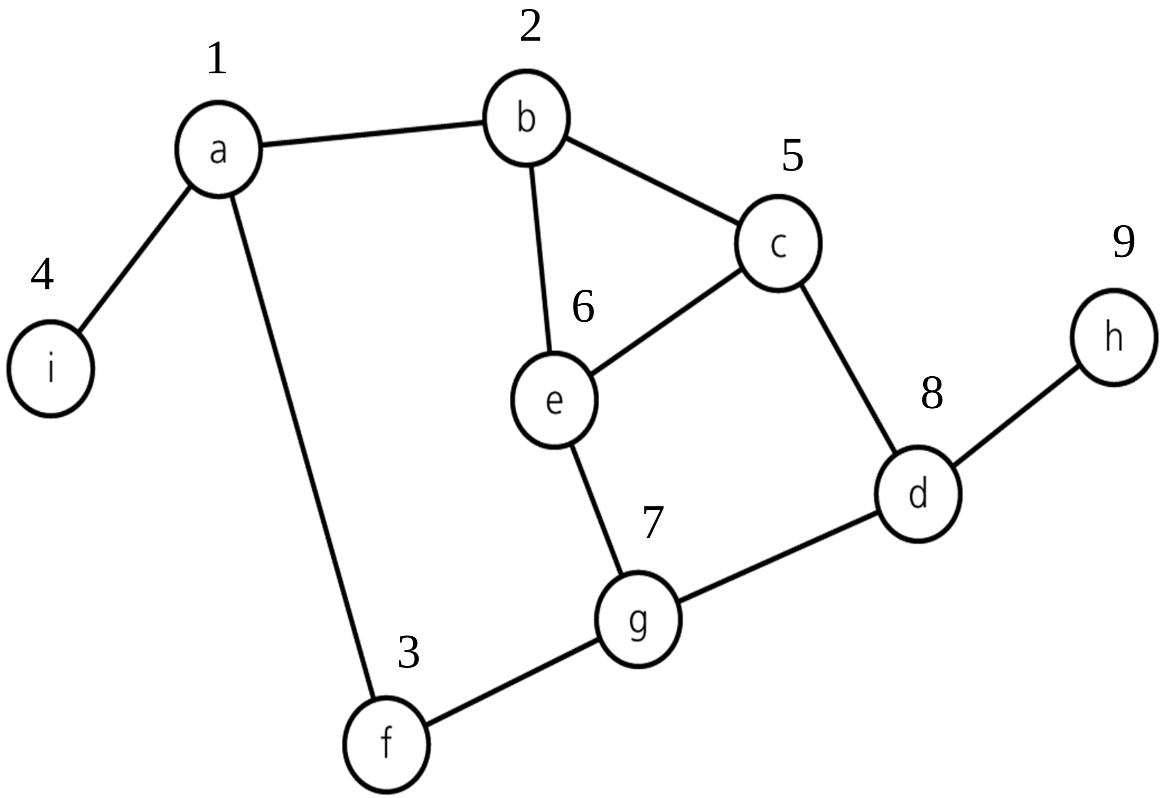


```
void BFSTraversal(struct Graph *G) {
    for (int i = 0; i< G→V;i++)
        Visited[i]=0;

    //This loop is required if the graph has more than one component
   for (int i = 0; i< G→V;i++)
        if(!Visited[i])
            BFS(G. i):
}
```

Iterative Breadth-First Search Algorithm

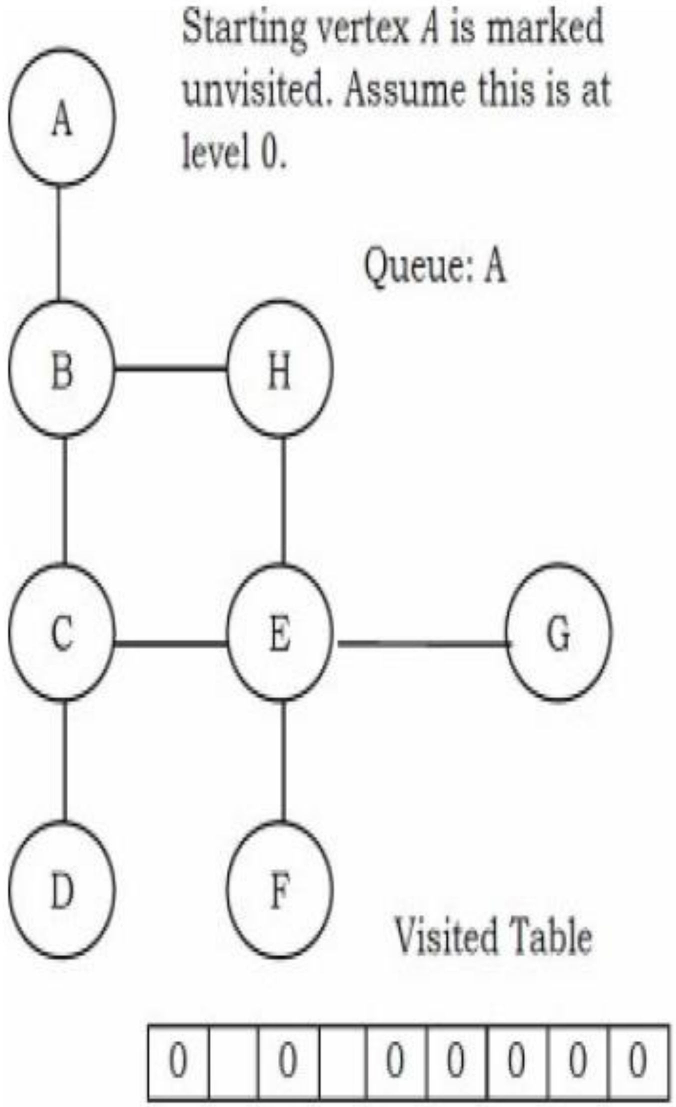
```
bfs(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using breath-first strategy: Iterative Version
  q.createQueue();
  // add v to the queue and mark it
  q.enqueue(v);
  Mark v as visited;
  while (!q.isEmpty()) {
      q.dequeue(w);
      for (each unvisited vertex u adjacent to w) {
         Mark u as visited;
         q.enqueue(u);
      }
  }
} ➔ O(V+E) worst-case time complexity via aggregate analysis.
```

Trace of Iterative 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)

Example

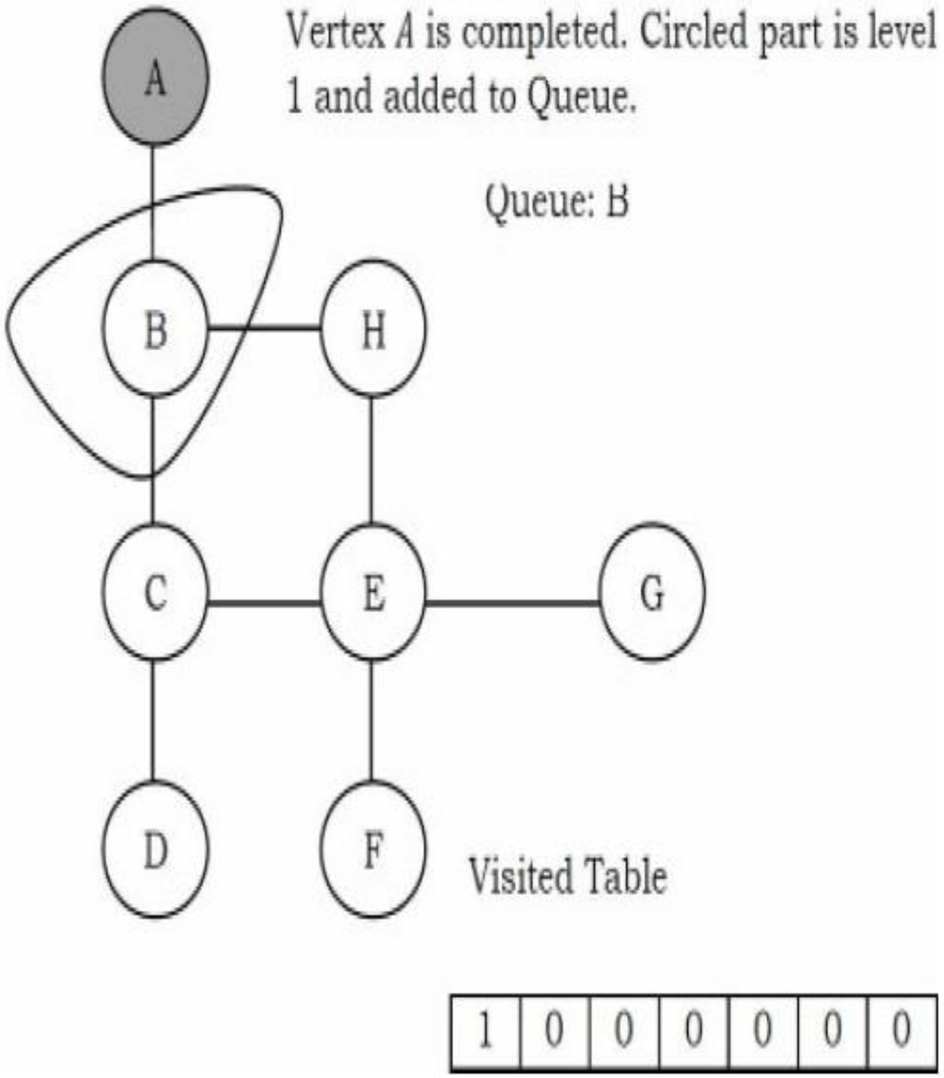


Starting vertex *A* is marked unvisited. Assume this is at level 0.

Queue: A

Visited Table

0		0		0	0	0	0	0



Vertex *A* is completed. Circled part is level 1 and added to Queue.

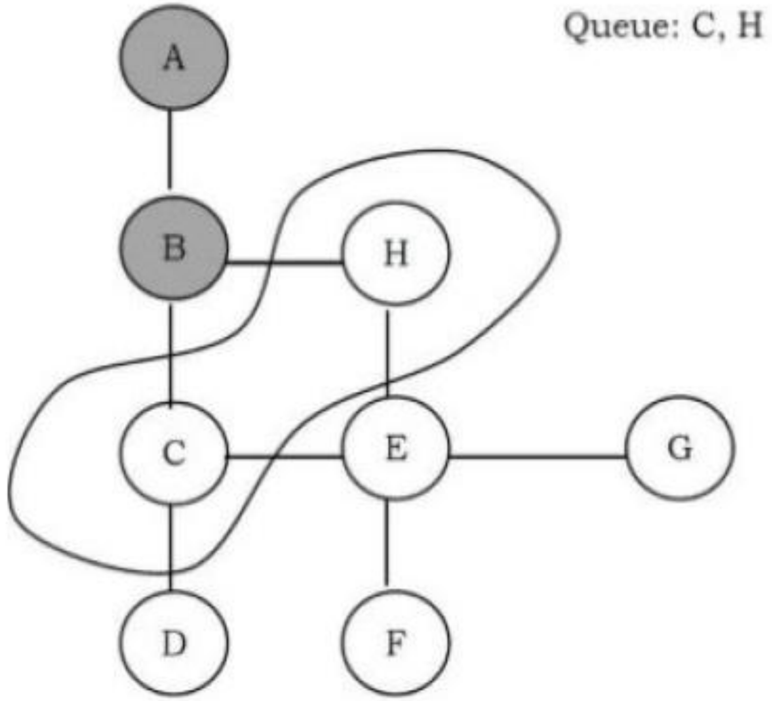
Queue: B

Visited Table

1	0	0	0	0	0	0

Example

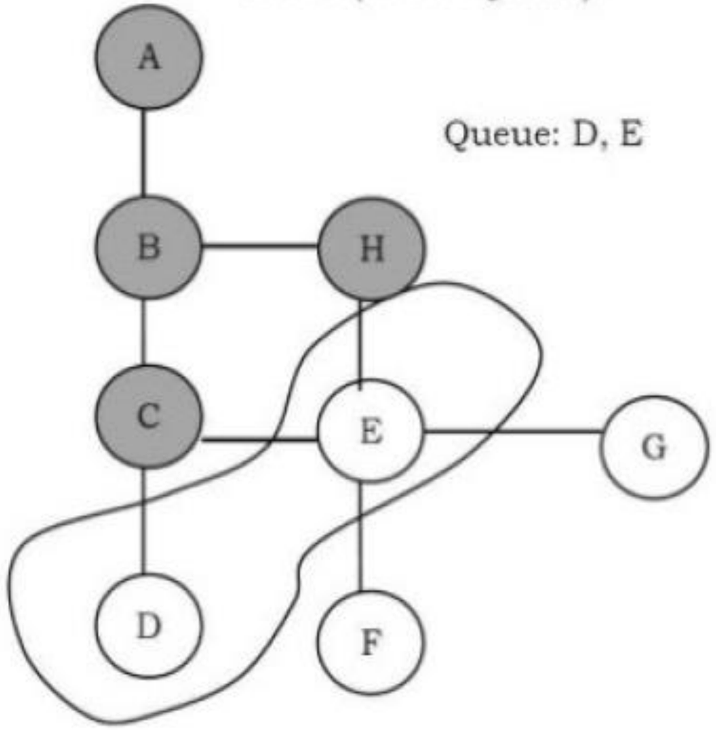
B is completed. Selected part is level 2 (add to Queue).



Visited Table

1	1	0	0	0	0	0

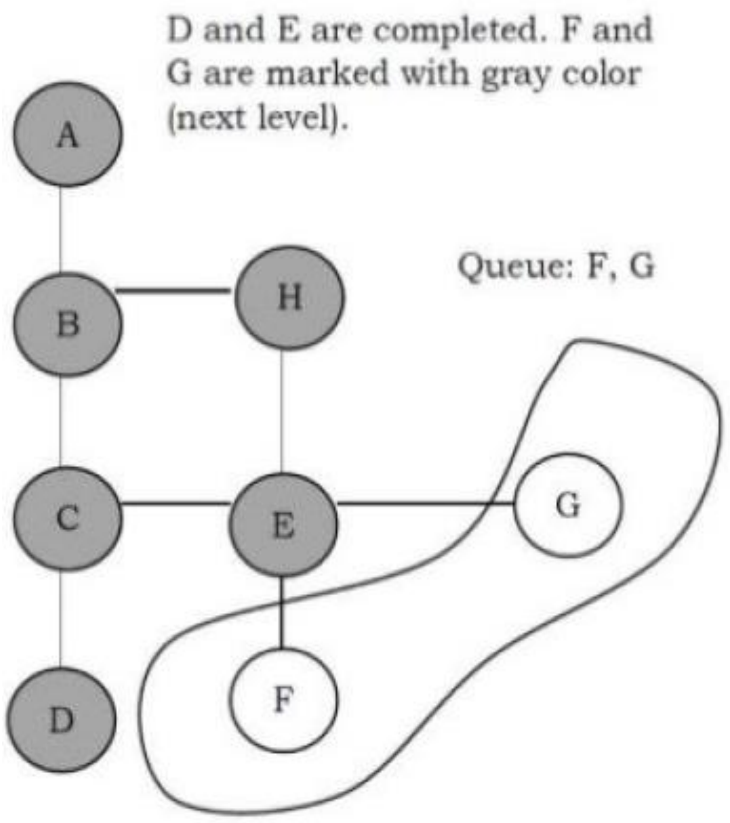
Vertices C and H are completed. Circled part is level 3 (add to Queue).



Visited Table

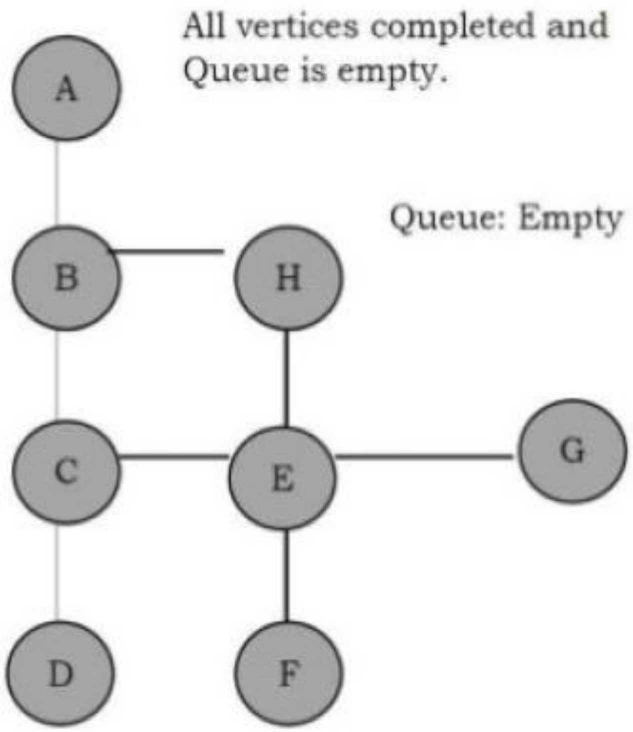
1	1	0	0	0	0	1

Example


D and E are completed. F and G are marked with gray color (next level).
Queue: F, G
A
B
H
C
E
G
D
F

Visited Table

1	1	1	1	0	0	1



Visited Table

1	1	1	1	1	1	1

Breadth First Search [BFS] (Continued)

- Time complexity of BFS is O(V + E), if we use adjacency lists for representing the graphs, and O(V^2) for adjacency matrix representation.
- Applications of BFS
  - Finding all connected components in a graph
  - Finding the shortest path between two nodes

Comparing DFS and BFS

- Comparing BFS and DFS, the big advantage of DFS is that it has much lower memory requirements than BFS because it’s not required to store all of the child pointers at each level. Depending on the data and what we are looking for, either DFS or BFS can be advantageous. For example, in a family tree if we are looking for someone who’s still alive and if we assume that person would be at the bottom of the tree, then DFS is a better choice. BFS would take a very long time to reach that last level.

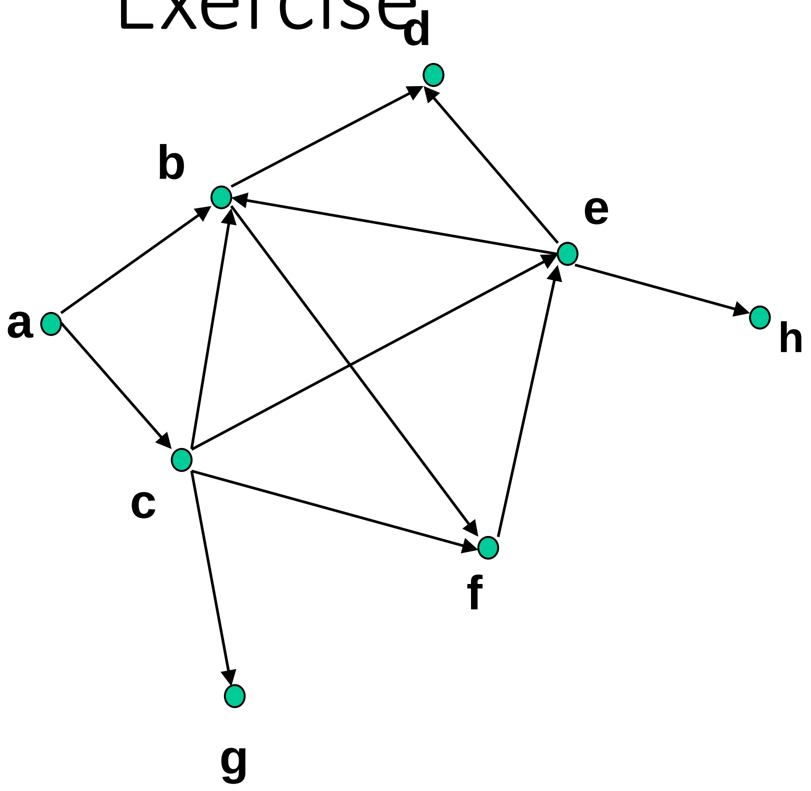
Comparing DFS and BFS (Continued)

- The DFS algorithm finds the goal faster. Now, if we were looking for a family member who died a very long time ago, then that person would be closer to the top of the tree. In this case, BFS finds faster than DFS. So, the advantages of either vary depending on the data and what we are looking for.
- DFS is related to preorder traversal of a tree. Like *preorder* traversal, DFS visits each node before its children. The BFS algorithm works similar to *level – order* traversal of the trees.

Comparing DFS and BFS (Continued)

- If someone asks whether DFS is better or BFS is better, the answer depends on the type of the problem that we are trying to solve. BFS visits each level one at a time, and if we know the solution we are searching for is at a low depth, then BFS is good. DFS is a better choice if the solution is at maximum depth.

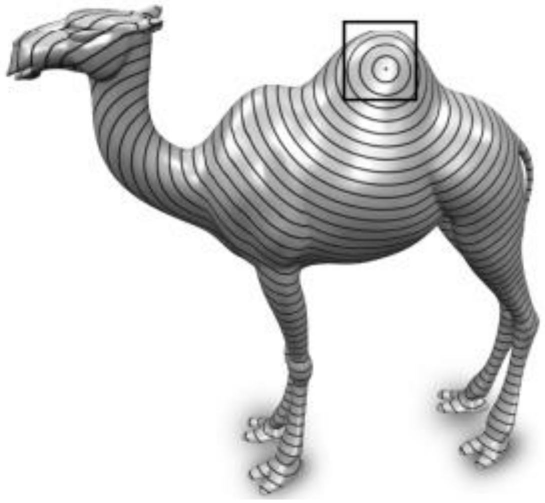
Exercise



a) Give the sequence of vertices when they are traversed starting from the vertex **a** using the breadth-first search algorithm.

b) Give the sequence of vertices when they are traversed starting from the vertex **a** using the depth-first search algorithm.

c) BFS or DFS below?

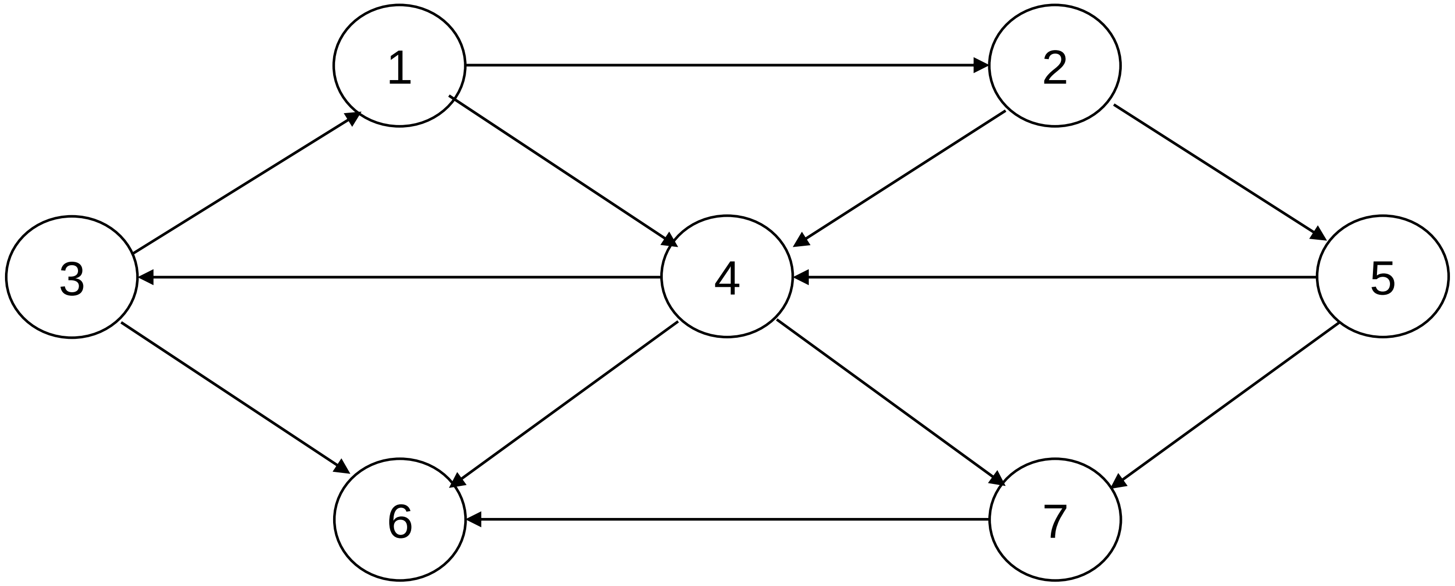


Some Graph Algorithms

- Shortest Path Algorithms
  - Unweighted shortest paths
  - Weighted shortest paths (Dijkstra's Algorithm)
- Topological sorting
- Graph Coloring
- Network Flow Problems
  - Construct a flow w/ min cost respecting capacities.
- Minimum Spanning Tree
  - Subset of edges that connects all verts w/ min cost.

Unweighted Shortest-Path problem

- *Find the shortest path (measured by number of edges) from a designated vertex S to every vertex.*



Algorithm

1. Start with an initial node s.
   - Mark the distance of s to s, D_s as 0.
   - Initially $D_i = \infty$ for all $i \neq s$.
2. Traverse all nodes starting from s as follows:
   1. If the node we are currently visiting is v, for all *w* that are adjacent to v:
      - Set $D_w = D_v + 1$ if $D_w = \infty$.
   2. Repeat step 2.1 with another vertex u that has not been visited yet, such that $D_u = D_v$ (if any).
   3. Repeat step 2.1 with another unvisited vertex u that satisfies $D_u = D_v +1$.(if any)

Figure 14.21A

Searching the graph in the unweighted shortest-path computation. The darkest-shaded vertices have already been completely processed, the lightest-shaded vertices have not yet been used as *v*, and the medium-shaded vertex is the current vertex, *v*. The stages proceed left to right, top to bottom, as numbered *(continued).*

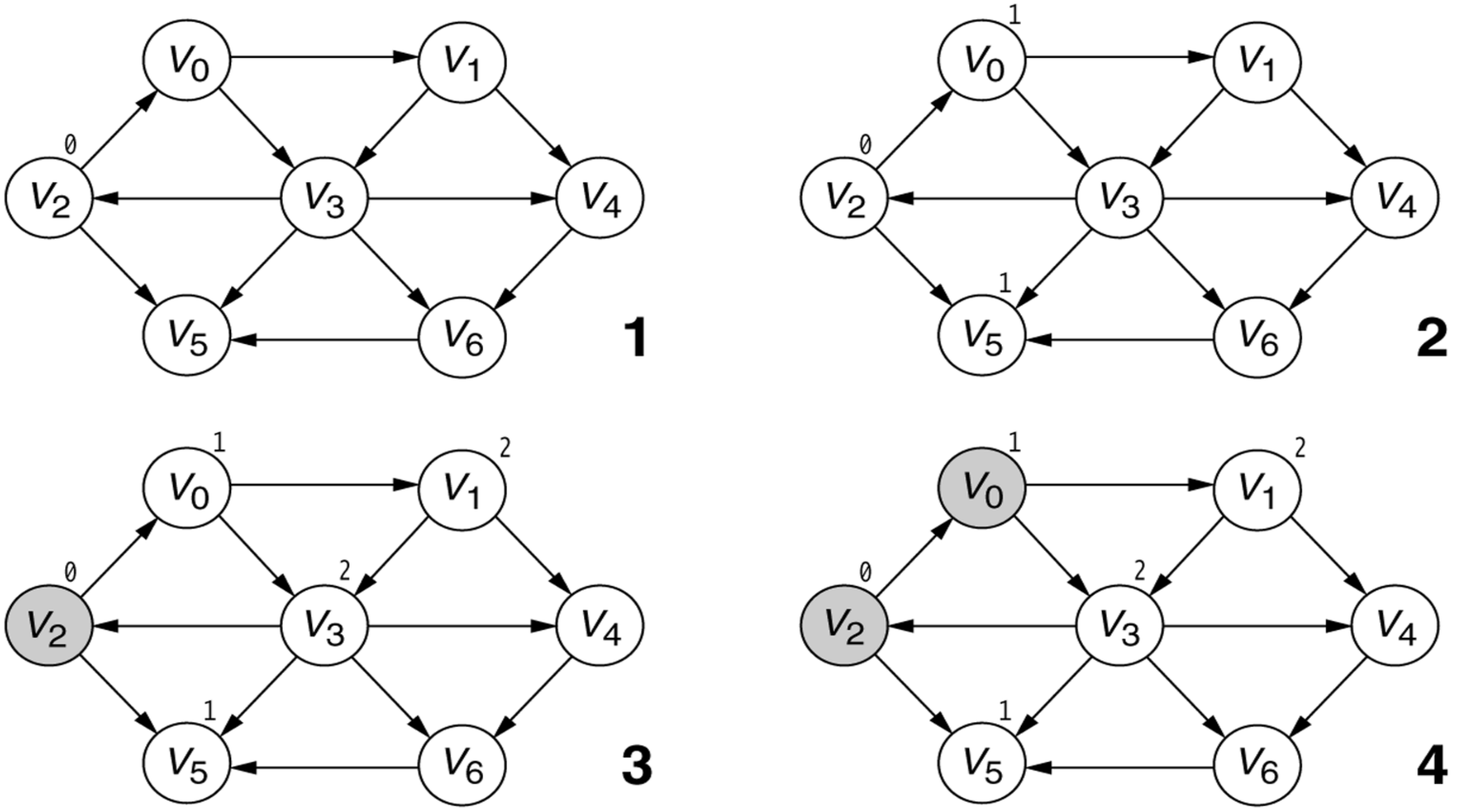
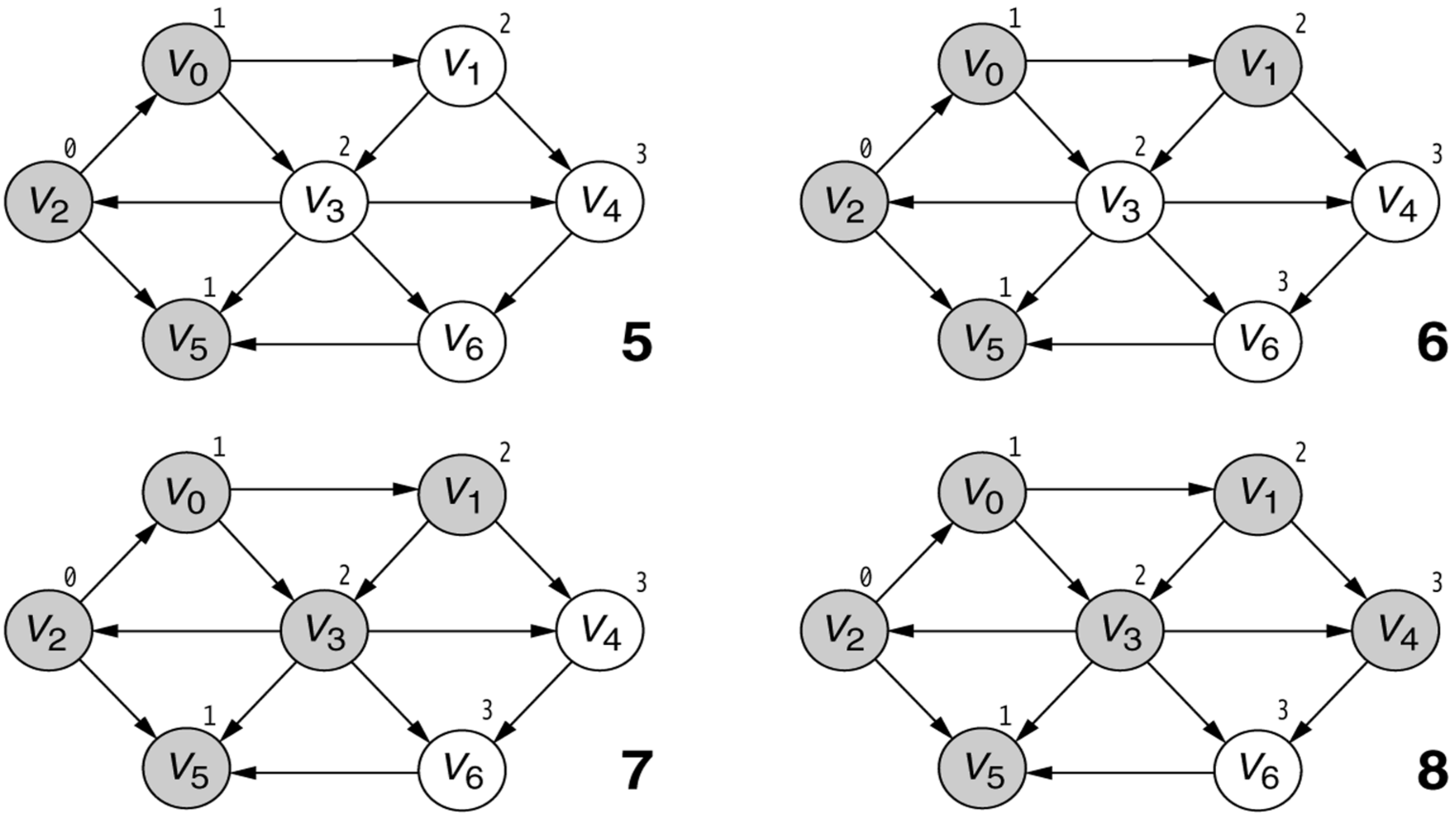


Figure 14.21B

Searching the graph in the unweighted shortest-path computation. The darkest-shaded vertices have already been completely processed, the lightest-shaded vertices have not yet been used as *v*, and the medium-shaded vertex is the current vertex, *v*. The stages proceed left to right, top to bottom, as numbered.



Unweighted shortest path algorithm

```
void Graph::unweighted_shortest_paths(vertex s)
{
   //An application of BFS: Finding the shortest path between
  //two nodes u and v, with path length measured by # of edges
  Queue<Vertex> q;
  Vertex v,w;
  for each Vertex v
       v.dist = INFINITY;

  s.dist = 0;
  q.enqueue(s);

  while (!q.isEmpty())
  {
      v= q.dequeue();
      v.known = true; // not needed anymore
      for each w adjacent to v
           if (w.dist == INFINITY)
           {
                w.dist = v.dist + 1;
                w.path = v;
                q.enqueue(w);
           }
  } }
```

References

- Assist. Prof. Dr. Fatih Soygazi, Lecture Notes on Data Structures, ADU.
- Narasimha Karumanchi; “Data Structures and Algorithms Made Easy: Data Structure and Algorithmic Puzzles”, 5th Ed., CareerMonk Publications, 2016.
- Assist. Prof. Dr. Yusuf Sahillioğlu, Lecture Notes, METU.