CSC-391: Data Structures

Lecture: 15

Graph-2: Graph Representation

Lecture-15:
Graph-2: Graph Representation

Objectives of this Lecture:

- Definition of graphs and related concepts
  - Vertices/nodes, edges, adjacency, incidence
  - Degree, in-degree, out-degree
  - Subgraphs, unions, isomorphism
  - Adjacency matrices
- Types of Graphs
  - Trees
  - Undirected graphs
  - Simple graphs, Multigraphs, Pseudographs
  - Digraphs, Directed multigraph
  - Bipartite
  - Complete graphs, cycles, wheels, cubes, complete bipartite
- Representation of Graphs
- Graph Traversal
- Graph Applications

Prepared by: K M Akkas Ali, Assistant Professor, IIT, JU

Graph Representations:

For graphs to be computationally useful, they have to be conveniently represented in programs.

There are two computer representations of graphs:

- Adjacency Matrix representation
- Adjacency Lists representation

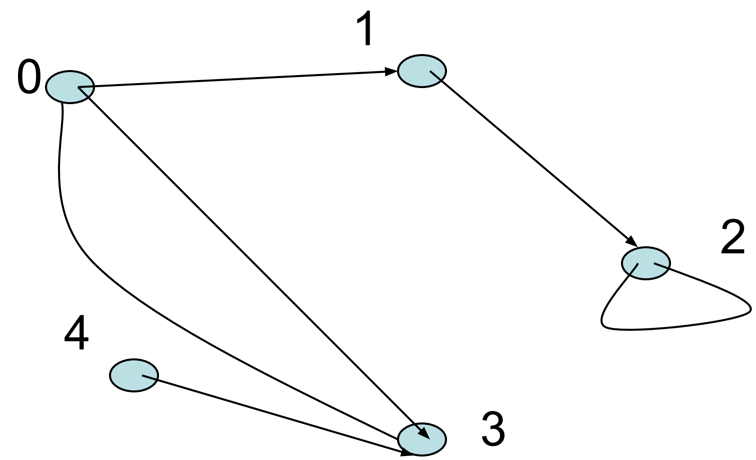
Adjacency Matrix Representation

- In this representation, each graph of n nodes is represented by an n x n matrix A, that is, a two-dimensional array A
- The nodes are (re)-labeled 1,2,…,n
- A[i][j] = 1 if (i,j) is an edge
- A[i][j] = 0 if (i,j) is not an edge
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Prepared by: K M Akkas Ali, Assistant Professor, IIT, JU

Example of Adjacency Matrix

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

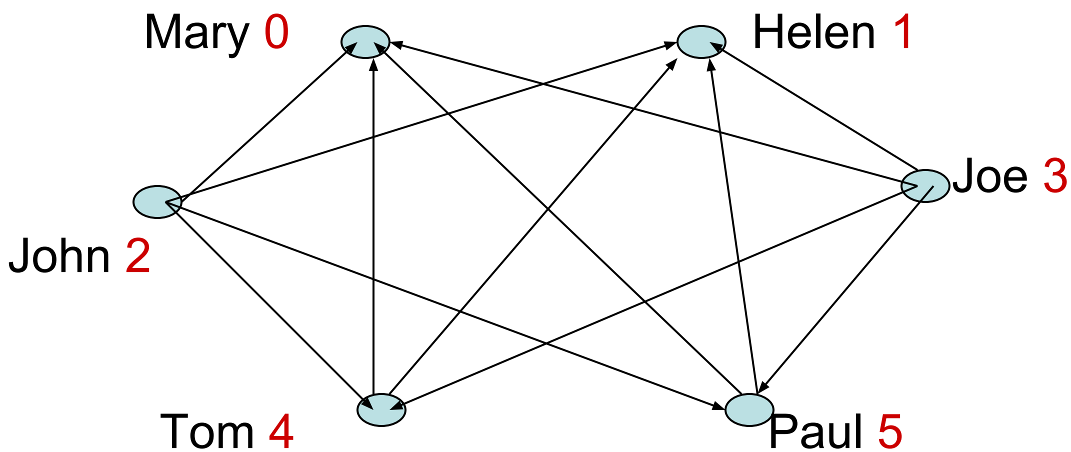


Prepared by: K M Akkas Ali, Assistant Professor, IIT, JU

Another Example of Adj. Matrix

- Re-label the nodes with numerical labels

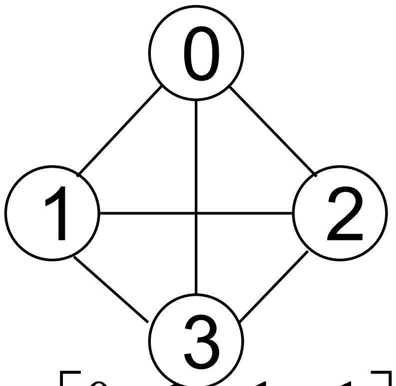
$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$



Prepared by: K M Akkas Ali, Assistant Professor, IIT, JU

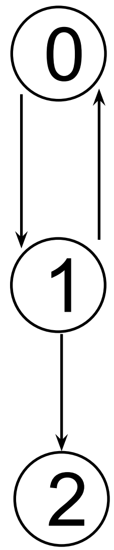
Graph Representations:

Examples for Adjacency Matrix



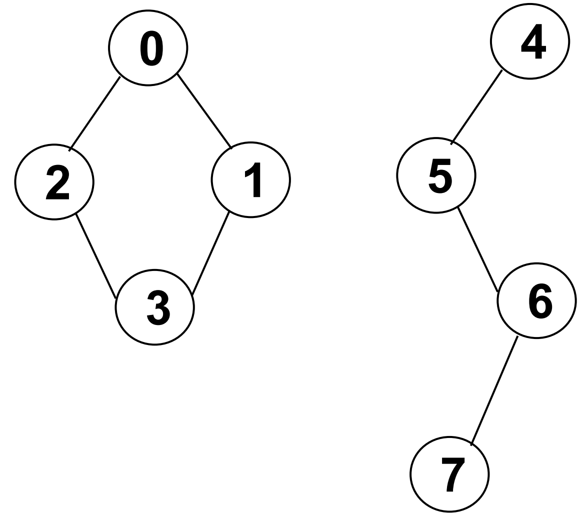
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1



$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

G_2



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G

symmetric

undirected: $n^2/2$
directed: n^2

Pros and Cons of Adjacency Matrices

- Pros:
  - From the adjacency matrix, to determine the connection of vertices is easy
  - Simple to implement
  - Easy and fast to tell if a pair (i,j) is an edge: simply check if A[i][j] is 1 or 0
  - For a digraph (= directed graph), the row sum is the out_degree, while the column sum is the in_degree

$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \qquad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

- Cons:
  - No matter how few edges the graph has, the matrix takes $O(n^2)$ in memory

Prepared by: K M Akkas Ali, Assistant Professor, IIT, JU

Adjacency Lists Representation

Each row in adjacency matrix is represented as an adjacency list.

- A graph of n nodes is represented by a one-dimensional array L of linked lists, where
  - L[i] is the linked list containing all the nodes adjacent from node i.
  - The nodes in the list L[i] are in no particular order

```
#define MAX_VERTICES 50
typedef struct node *node_pointer;
typedef struct node {
    int vertex;
    struct node *link;
};
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use */
```

Example of Linked Representation

L[0]: empty

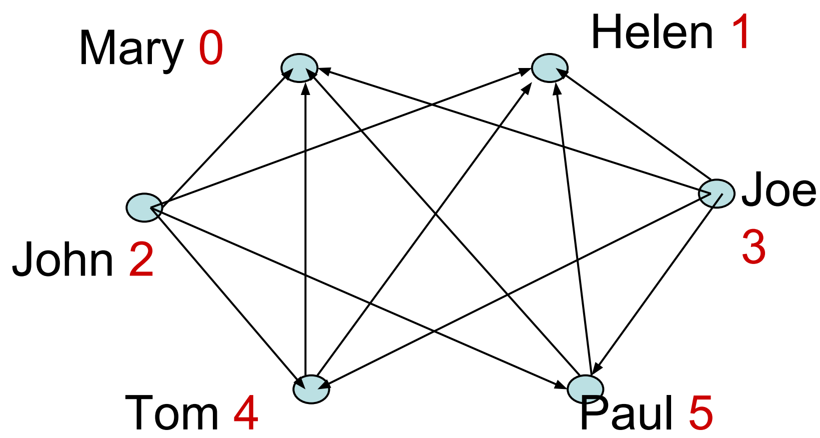
L[1]: empty

L[2]: 0, 1, 4, 5

L[3]: 0, 1, 4, 5

L[4]: 0, 1

L[5]: 0, 1

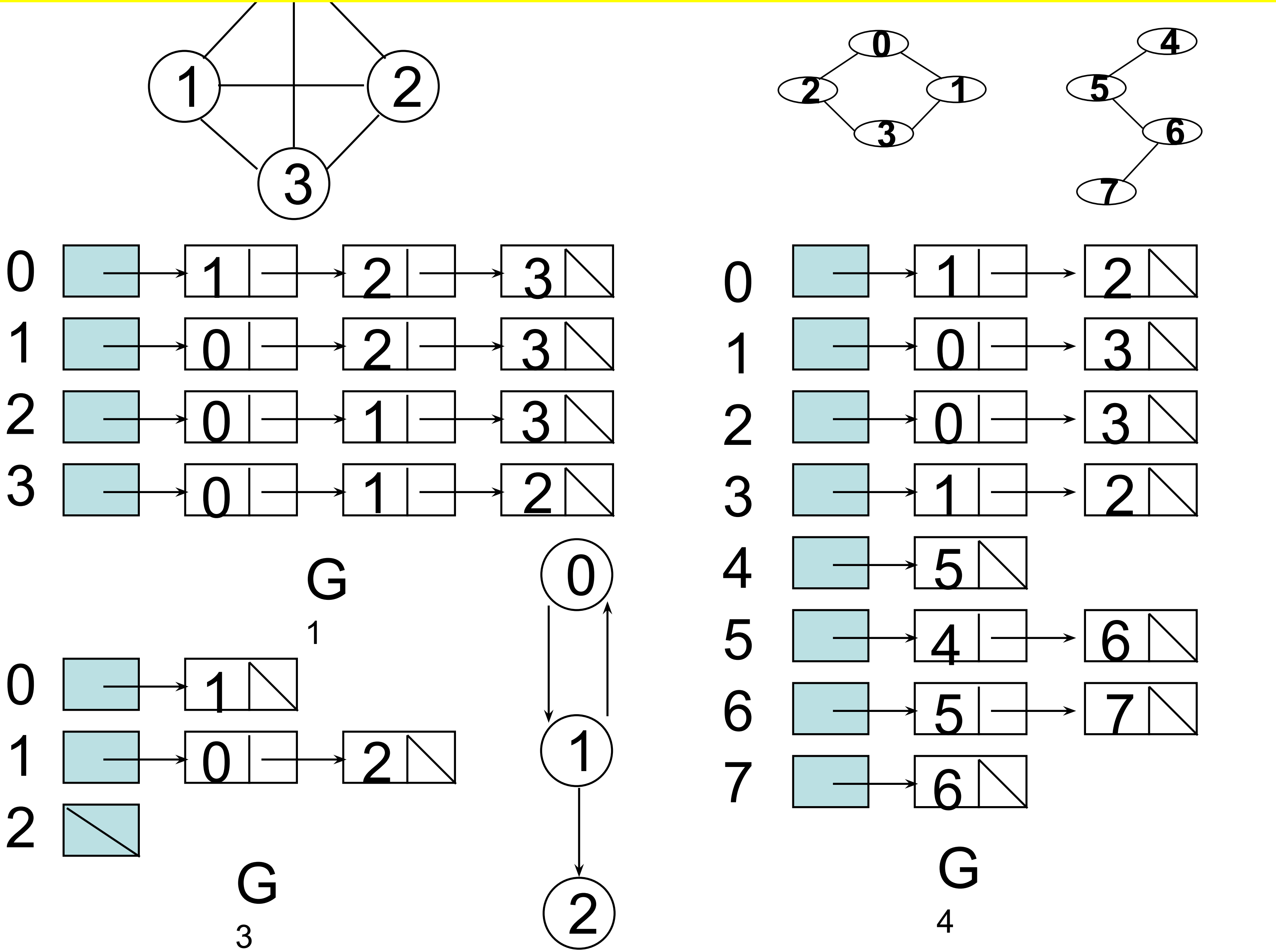


Prepared by: K M Akkas Ali, Assistant Professor, IIT, JU

Pros and Cons of Adjacency Lists

- Pros:
  - Saves on space (memory): the representation takes as many memory words as there are nodes and edge.
- Cons:
  - It can take up to O(n) time to determine if a pair of nodes (i,j) is an edge: one would have to search the linked list L[i], which takes time proportional to the length of L[i].

Graph Representations:



An undirected graph with n vertices and e edges ==> n head nodes and 2e list nodes

Some Operations on Graph:

- degree of a vertex in an undirected graph
  - # of nodes in adjacency list
- # of edges in a graph
  - determined in O(n+e)
- out-degree of a vertex in a directed graph
  - # of nodes in its adjacency list
- in-degree of a vertex in a directed graph
  - traverse the whole data structure

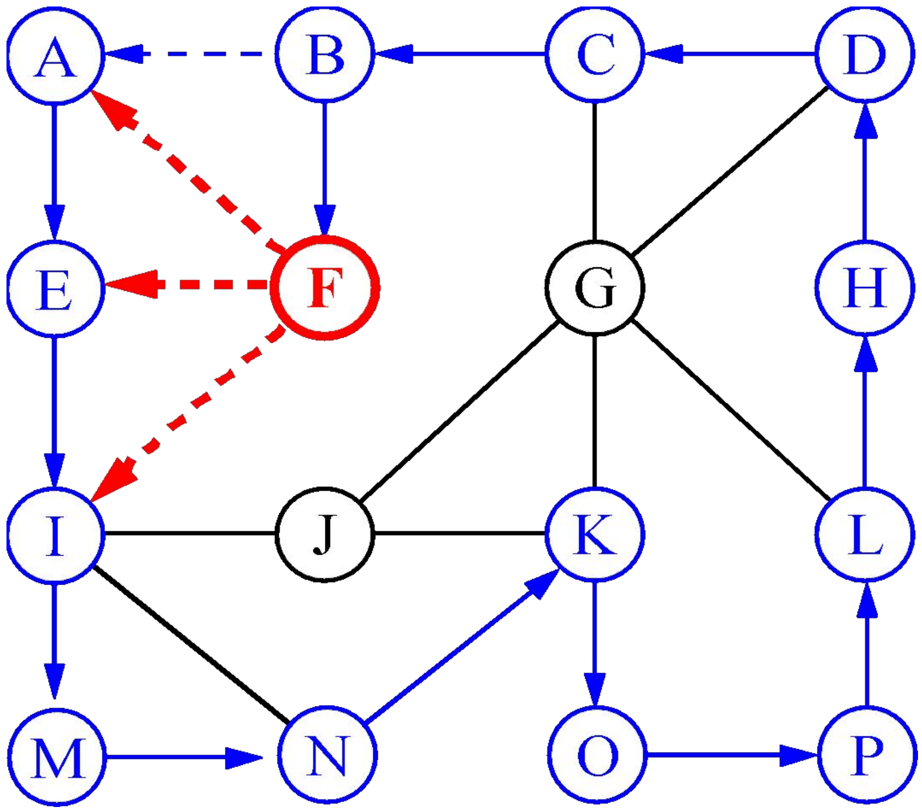
Graph Traversal Techniques:

- <u>Problem:</u> Search for a certain node or traverse all nodes in the graph
- The previous connectivity problem, as well as many other graph problems, can be solved using graph traversal techniques
- There are two standard graph traversal techniques:
  - *Depth-First Search* (DFS)
  - *Breadth-First Search* (BFS)

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

- In both DFS and BFS, the nodes of the undirected graph are visited in a systematic manner so that every node is visited exactly one.
- Both BFS and DFS give rise to a tree:
  - When a node x is visited, it is labeled as visited, and it is added to the tree
  - If the traversal got to node x from node y, y is viewed as the parent of x, and x a child of y

Depth-First Search:



Depth-First Search:

Exploring a Labyrinth Without Getting Lost

- A **depth-first search (DFS)** in an undirected graph G is like wandering in a labyrinth with a string and a can of red paint without getting lost.
- We start at vertex ***s***, tying the end of our string to the point and painting ***s*** “visited”. Next we label ***s*** as our current vertex called ***u***.
- Now we travel along an arbitrary edge (***u***, ***v***).
- If edge (***u***, ***v***) leads us to an already visited vertex ***v*** we return to ***u***.
- If vertex ***v*** is unvisited, we unroll our string and move to ***v***, paint ***v*** “visited”, set ***v*** as our current vertex, and repeat the previous steps.

Depth-First Search

- DFS follows the following rules:
  1. Select an unvisited node x, visit it, and treat as the current node
  2. Find an unvisited neighbor of the current node, visit it, and make it the new current node;
  3. If the current node has no unvisited neighbors, backtrack to the its parent, and make that parent the new current node;
  4. Repeat steps 3 and 4 until no more nodes can be visited.
  5. If there are still unvisited nodes, repeat from step 1.

Depth-First Search:

Algorithm DFS(v); **Input**: A vertex v in a graph

Output: A labeling of the edges as “discovery” edges and “backedges”

for each edge *e* incident on *v* **do**

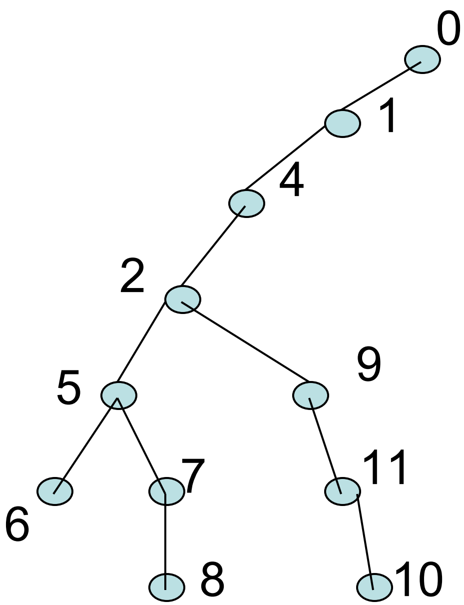
if edge *e* is unexplored **then** let *w* be the other endpoint of *e*

if vertex w is unexplored **then** label *e* as a discovery edge

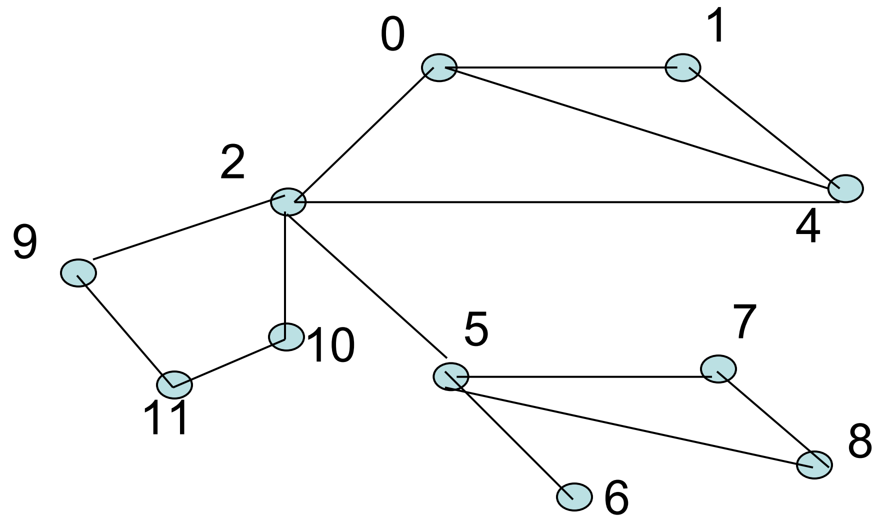
recursively call **DFS(*w*)**

else label *e* as a backedge

Illustration of DFS



DFS Tree



Graph G

Prepared by: K M Akkas Ali, Assistant Professor, IIT, JU

Implementation of DFS

- Observations:
  - the last node visited is the first node from which to proceed.
  - Also, the backtracking proceeds on the basis of "last visited, first to backtrack too".
  - This suggests that a stack is the proper data structure to remember the current node and how to backtrack.

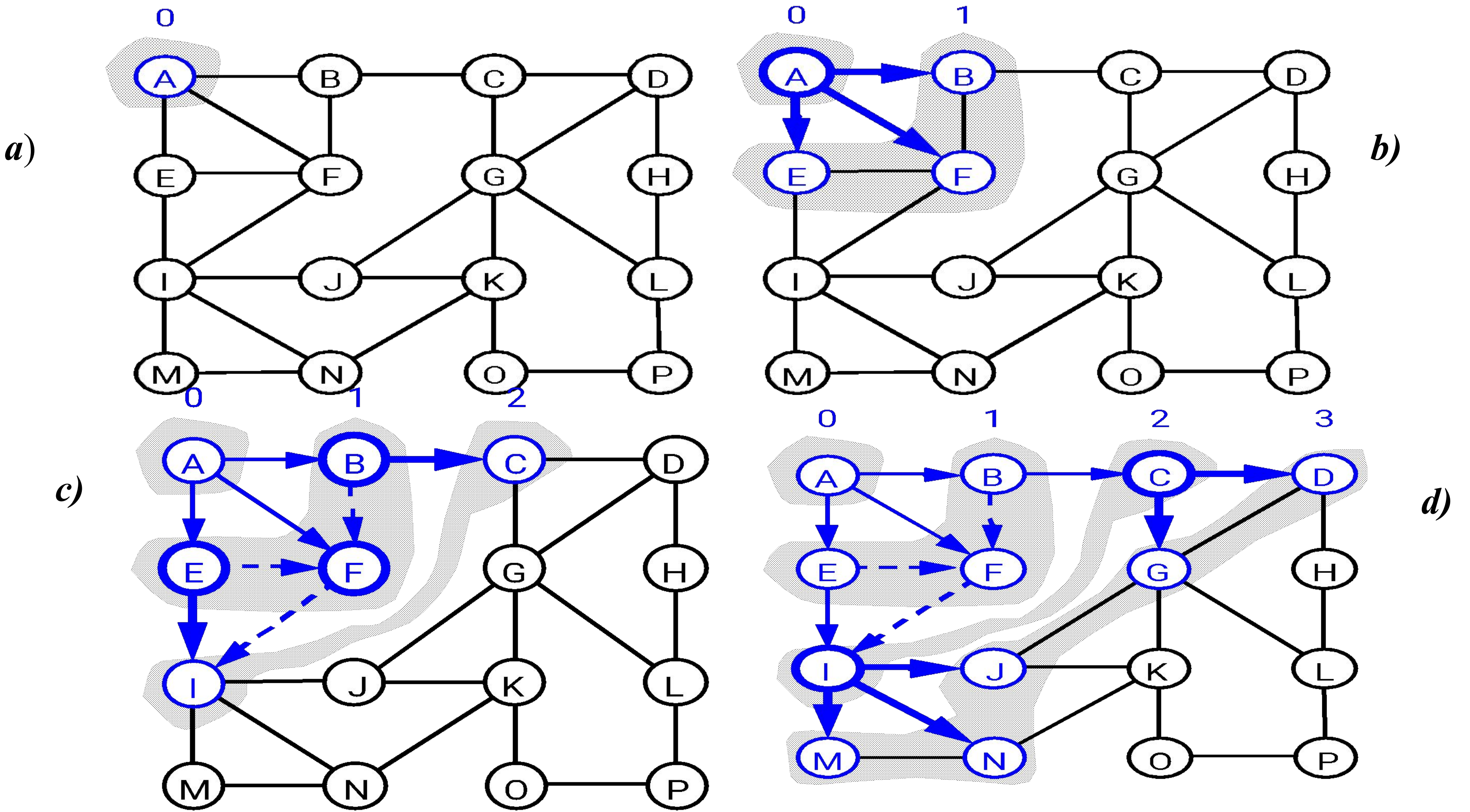
Prepared by: K M Akkas Ali, Assistant Professor, IIT, JU

Breadth-First Search:

- Like DFS, a Breadth-First Search (BFS) traverses a connected component of a graph, and in doing so defines a spanning tree with several useful properties.
- The starting vertex *s* has level 0, and, as in DFS, defines that point as an "anchor."
- In the first round, the string is unrolled the length of one edge, and all of the edges that are only one edge away from the anchor are visited.
- These edges are placed into level 1
- In the second round, all the new edges that can be reached by unrolling the string 2 edges are visited and placed in level 2.
- This continues until every vertex has been assigned a level.
- The label of any vertex ***v*** corresponds to the length of the shortest path from ***s*** to ***v***.

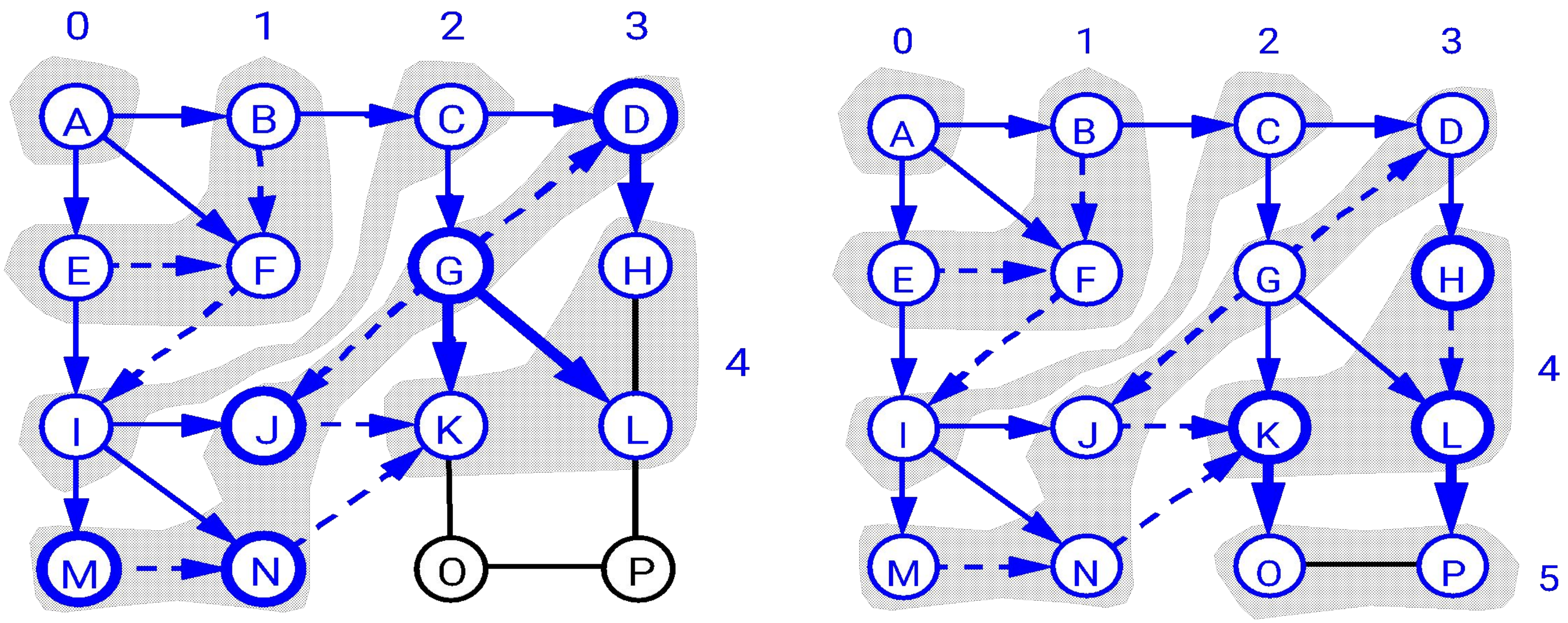
Breadth-First Search:

BFS - A Graphical Representation



Breadth-First Search:

More BFS

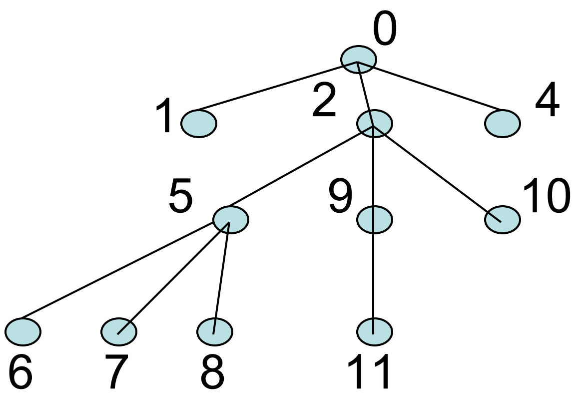


Breadth-First Search

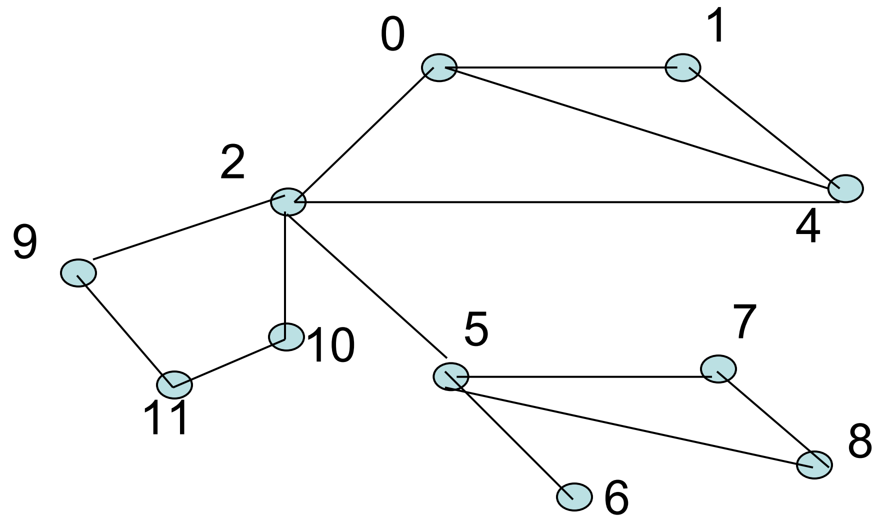
- BFS follows the following rules:
  1. Select an unvisited node x, visit it, have it be the root in a BFS tree being formed. Its level is called the current level.
  2. From each node z in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbors of z. The newly visited nodes from this level form a new level that becomes the next current level.
  3. Repeat step 2 until no more nodes can be visited.
  4. If there are still unvisited nodes, repeat from Step 1.

Prepared by: K M Akkas Ali, Assistant Professor, IIT, JU

Illustration of BFS



BFS Tree



Graph G

BFS Pseudo-Code

```
Algorithm BFS(s):  Input: A vertex s in a graph
Output: A labeling of the edges as “discovery” edges and “cross edges”
  initialize container L_0 to contain vertex s
  i ← 0
  while L_i is not empty do
      create container L_i+1 to initially be empty
      for each vertex v in L_i do
        if edge e incident on v do
          let w be the other endpoint of e
          if vertex w is unexplored then
            label e as a discovery edge
              insert w into L_i+1
           else label e as a cross edge
              i ← i + 1
```

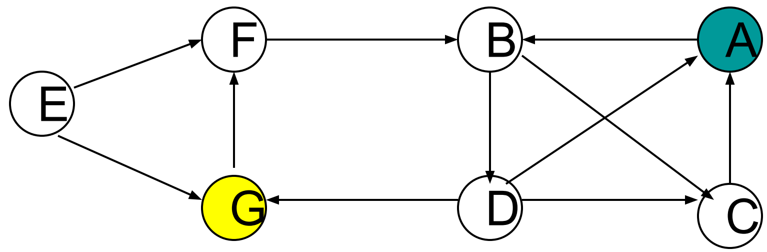
Applications: Finding a Path

- Find path from source vertex s to destination vertex d
- Use graph search starting at s and terminating as soon as we reach d
  - Need to remember edges traversed
- Use depth – first search ?
- Use breath – first search?

Prepared by: K M Akkas Ali, Assistant Professor, IIT, JU

DFS vs. BFS

DFS Process



start

destination

A

DFS on A

B
A

DFS on B

C
B
A

DFS on C

D
B
A

Call DFS on D

Return to call on B

G
D
B
A

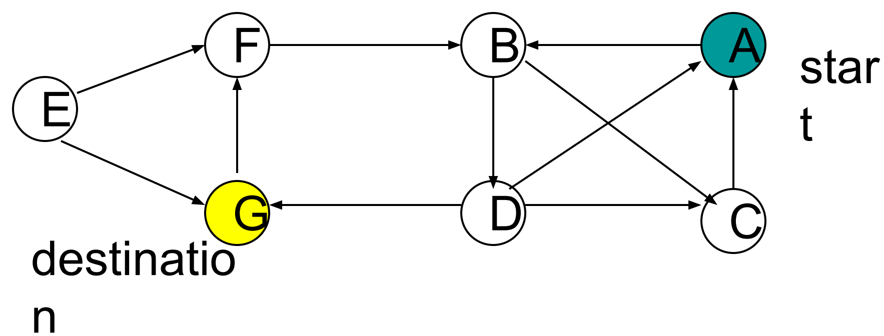
Call DFS on G

found destination - done!
Path is implicitly stored in DFS recursion
Path is: A, B, D, G

Prepared by: K M Akkas Ali, Assistant Professor, IIT, JU

DFS vs. BFS

BFS Process



rear front

A

Initial call to BFS on A
Add A to queue

rear front

B

Dequeue A
Add B

rear front

D C

Dequeue B
Add C, D

rear front

D

Dequeue C
Nothing to add

rear front

G

Dequeue D
Add G

found destination - done!
Path must be stored separately