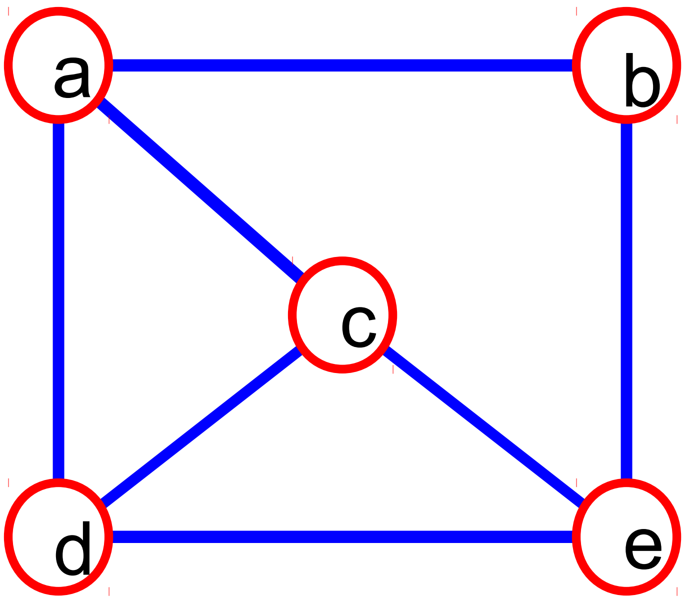
GRAPH

What is a Graph?

- A graph G = (V,E) is composed of:

  V: set of vertices

  E: set of edges connecting the vertices in V
- An edge e = (u,v) is a pair of vertices
- Example:

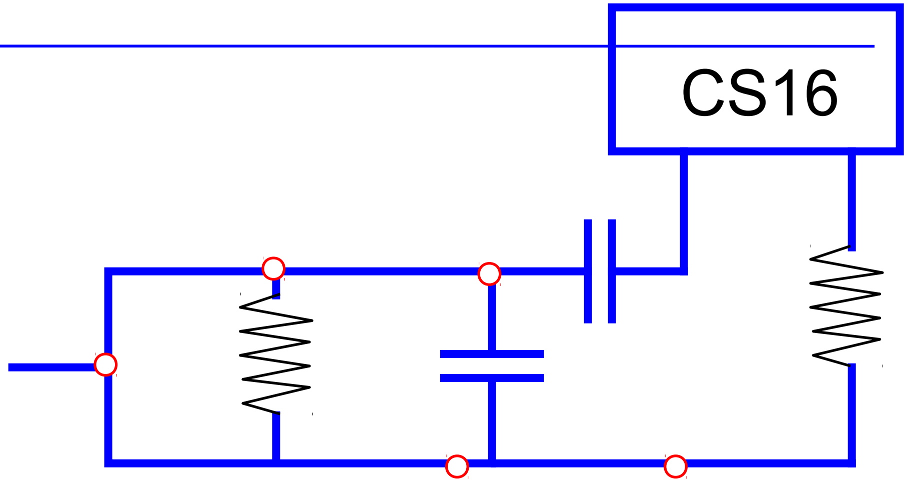


V= {a,b,c,d,e}

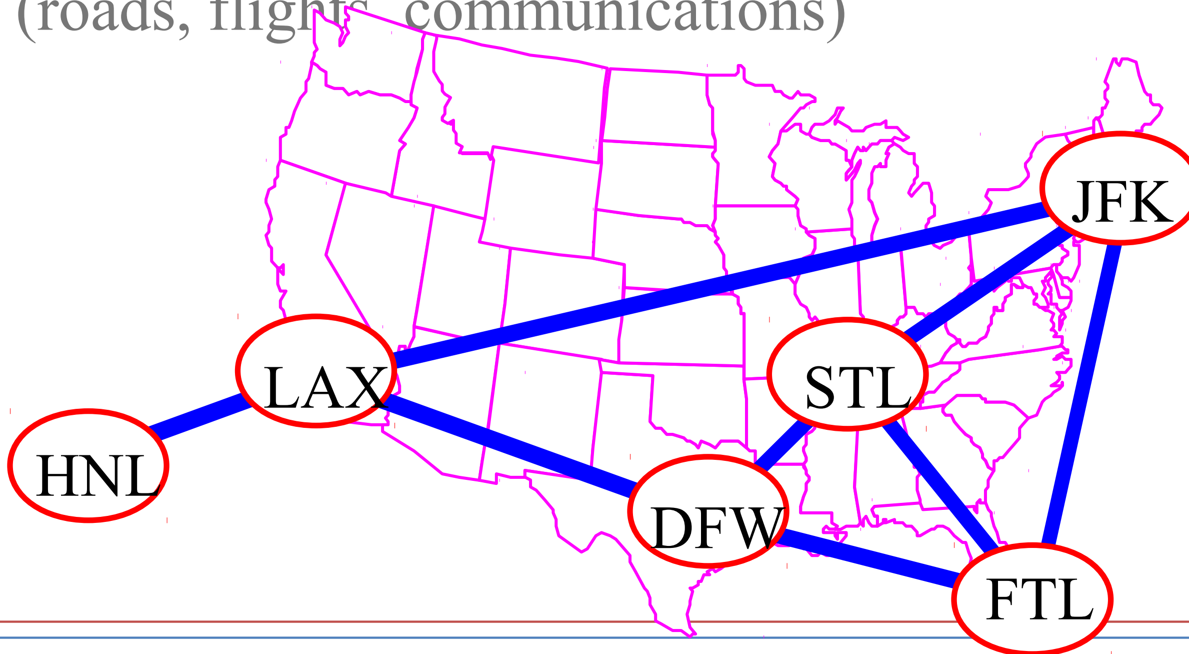
E= {(a,b),(a,c),
(a,d),
(b,e),(c,d),(c,e),
(d,e)}

Applications

- electronic circuits



- networks (roads, flights, communications)



Terminology: Adjacent and Incident

- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $<v_0, v_1>$ is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge $<v_0, v_1>$ is incident on v_0 and v_1

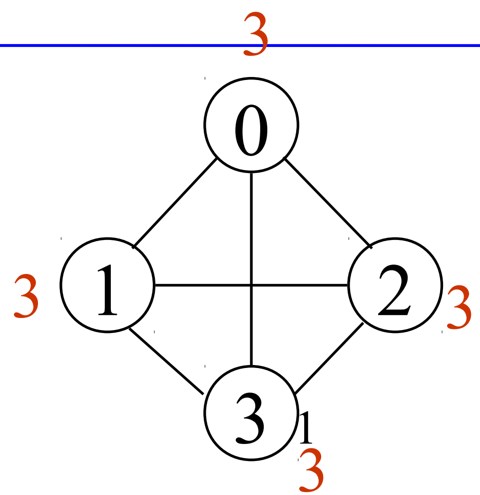
Terminology: Degree of a Vertex

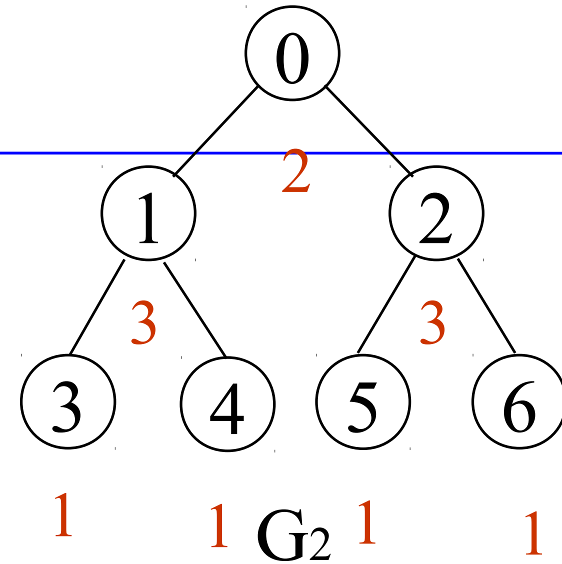
- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex v is the number of edges that have v as the head
  - the out-degree of a vertex v is the number of edges that have v as the tail
  - if d_i is the degree of a vertex i in a graph G with n vertices and e edges, the number of edges is

$$e = \left(\sum_{0}^{n-1} d_i\right) / 2$$

Why? Since adjacent vertices each count the adjoining edge, it will be counted twice

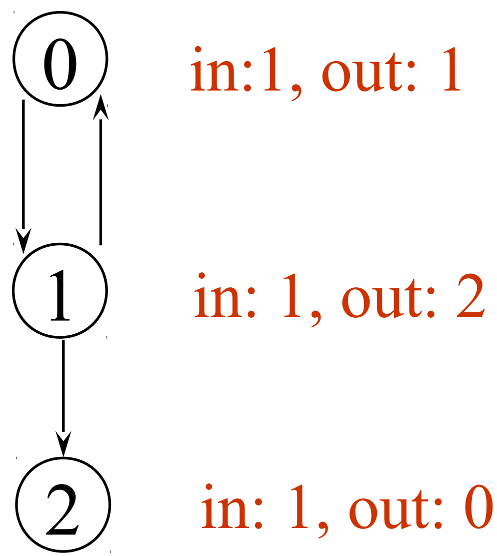
Examples





G_2

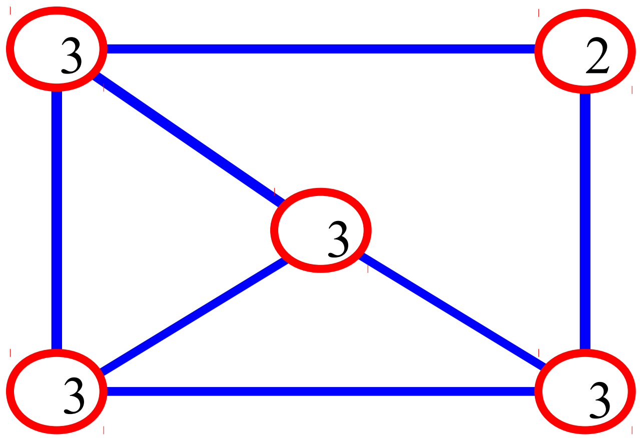
directed graph
in-degree
out-degree

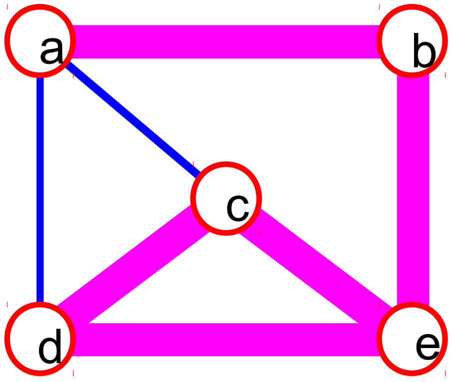


G_3

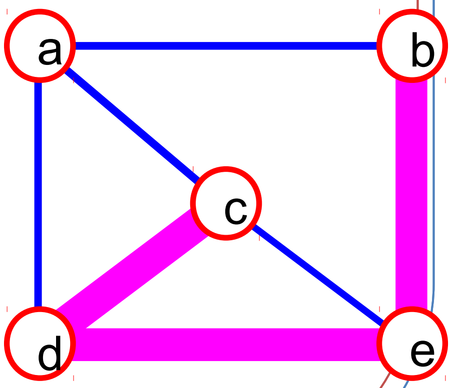
Terminology: Path

- path: sequence of vertices $v_1, v_2, \ldots v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.





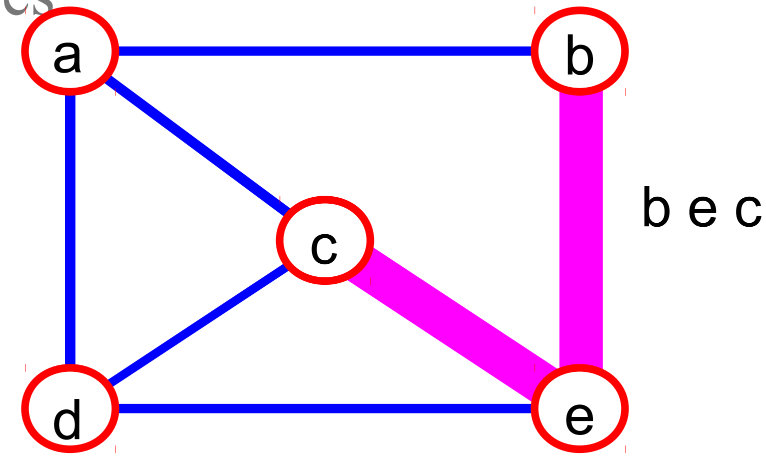
a b e d c



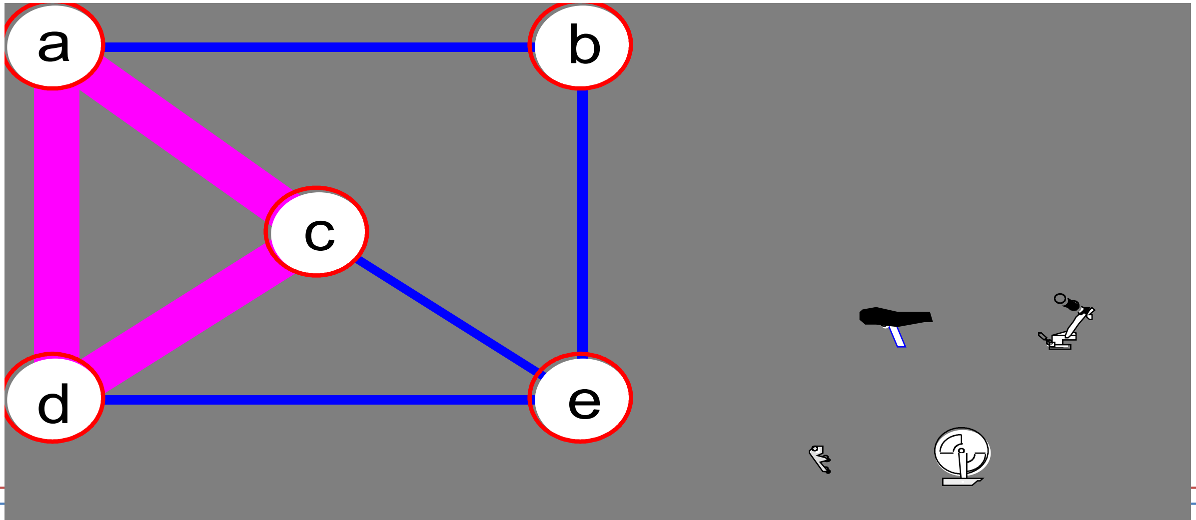
b e d c

More Terminology

- simple path: no repeated vertices

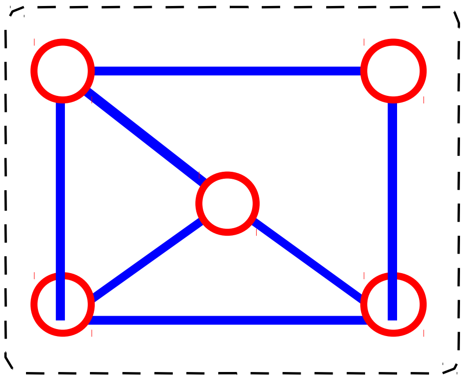


- cycle: simple path, except that the last vertex is the same as the first vertex

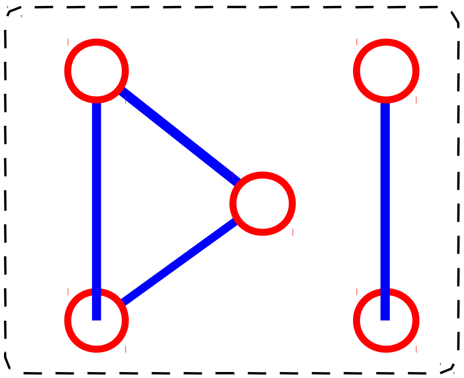


Even More Terminology

- connected graph: any two vertices are connected by some path

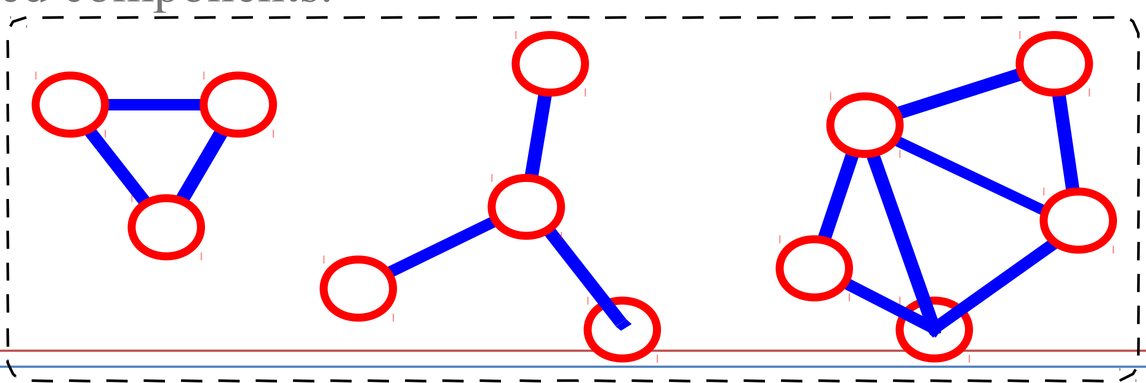


connected

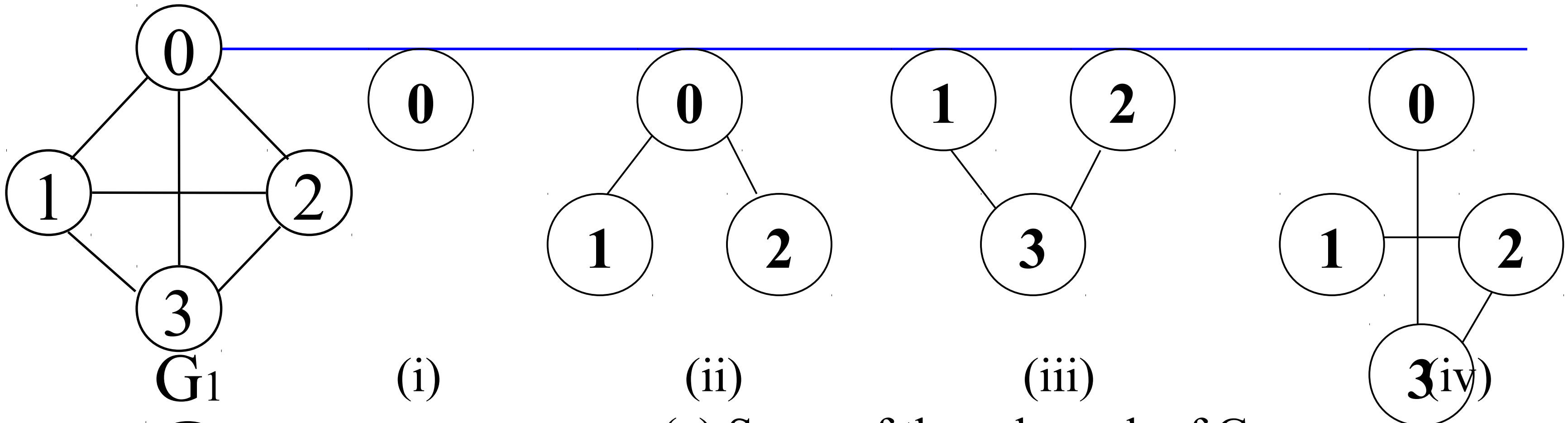


not connected

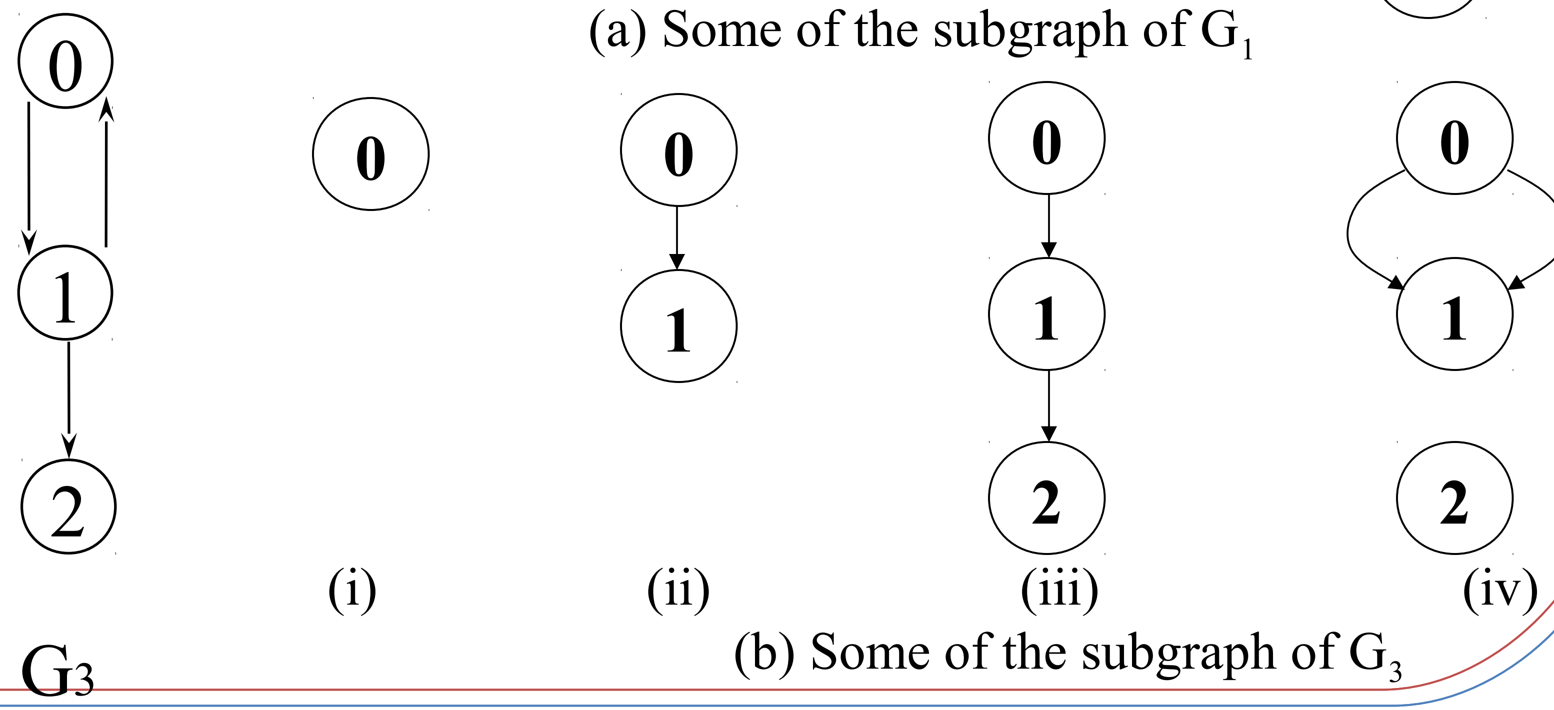
- subgraph: subset of vertices and edges forming a graph
- connected component: maximal connected subgraph. E.g., the graph below has 3 connected components.



Subgraphs Examples



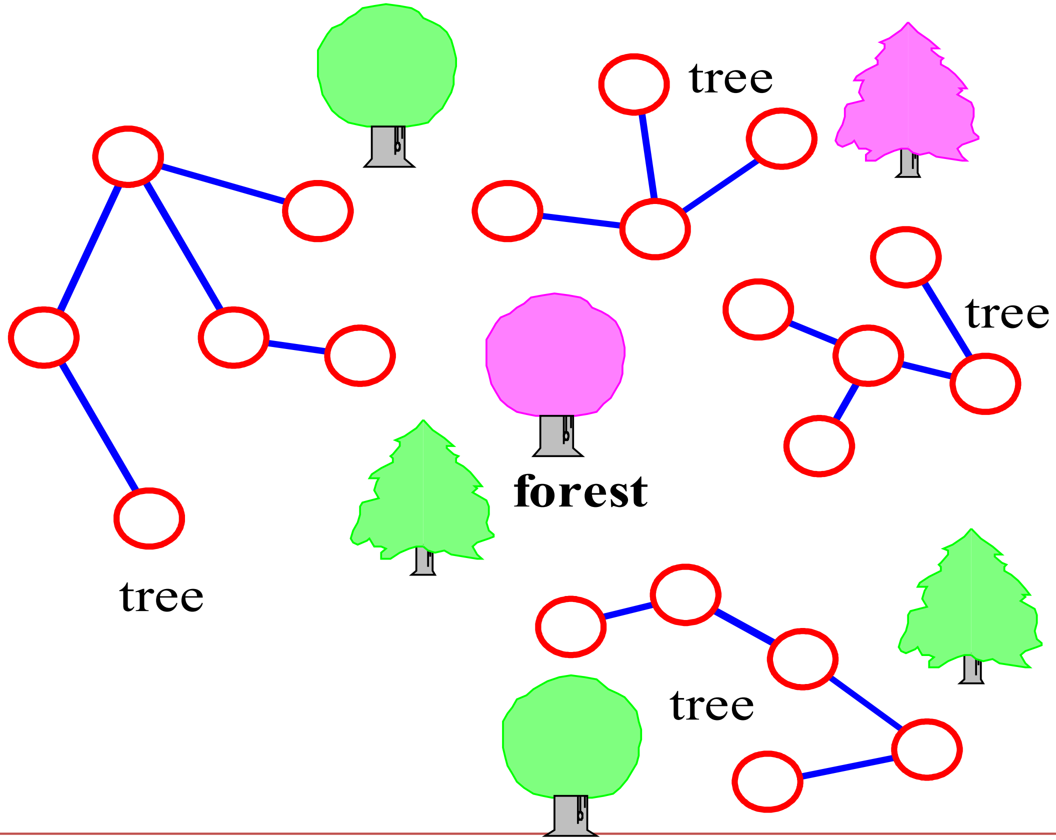
(a) Some of the subgraph of G_1



(b) Some of the subgraph of G_3

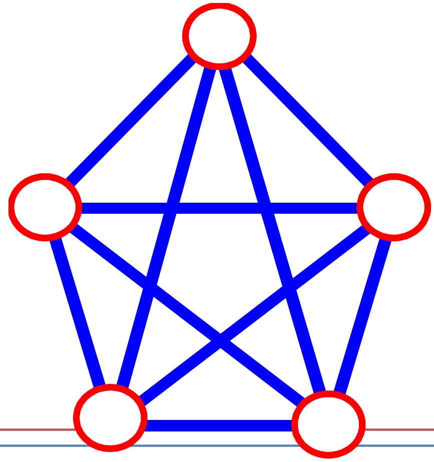
More…

- tree - connected graph without cycles
- forest - collection of trees



Connectivity

- Let **n** = #vertices, and **m** = #edges
- **A complete graph**: one in which all pairs of vertices are adjacent
- *How many total edges in a complete graph?*
  - Each of the n vertices is incident to **n**-1 edges, however, we would have counted each edge twice! Therefore, intuitively, m = **n**(**n** -1)/2.
- Therefore, if a graph is not complete, m < **n**(**n** -1)/2



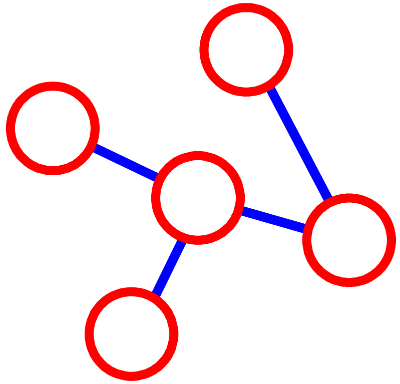
$\mathbf{n} = 5$
$\mathbf{m} = (5 * 4)/2 = 10$

More Connectivity

n = #vertices

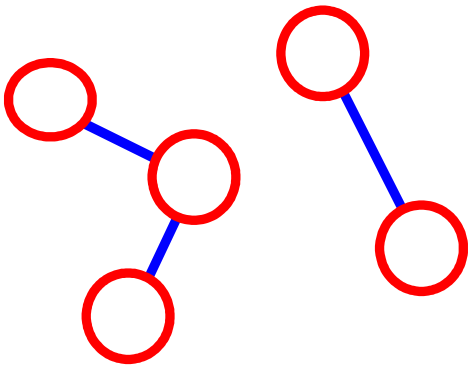
m = #edges

- For a tree **m** = **n** - 1



n = 5
m = 4

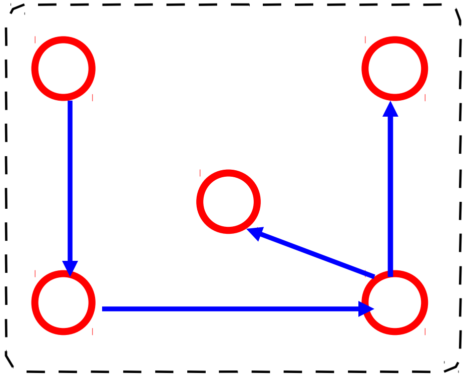
If **m** < **n** - 1, G is not connected



n = 5
m = 3

Oriented (Directed) Graph

- A graph where edges are directed



Directed vs. Undirected Graph

- An undirected graph is one in which the pair of vertices in a edge is unordered, $(v_0, v_1) = (v_1, v_0)$
- A directed graph is one in which each edge is a directed pair of vertices, $<v_0, v_1> \;!= <v_1, v_0>$

tail head

ADT for Graph

objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

functions: for all *graph* $\in$ *Graph*, v, v_1 and $v_2 \in$ *Vertices*

Graph Create()::=return an empty graph

Graph InsertVertex(*graph*, v)::= return a graph with v inserted. v has no incident edge.

Graph InsertEdge(*graph*, v_1,v_2)::= return a graph with new edge between v_1 and v_2

Graph DeleteVertex(*graph*, v)::= return a graph in which v and all edges incident to it are removed

Graph DeleteEdge(*graph*, v_1, v_2)::=return a graph in which the edge (v_1, v_2) is removed

Boolean IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE
else return FALSE

List Adjacent(*graph*,v)::= return a list of all vertices that are adjacent to v

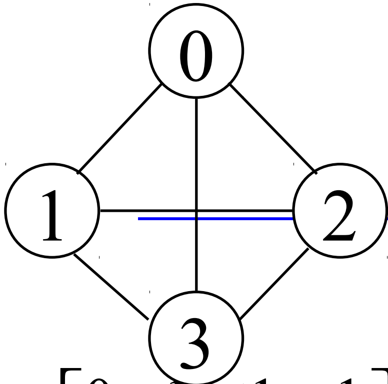
Graph Representations

- Adjacency Matrix
- Adjacency Lists

Adjacency Matrix

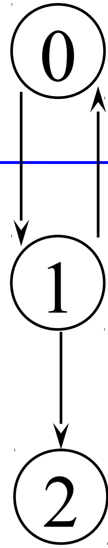
- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (v_i, v_j) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Examples for Adjacency Matrix



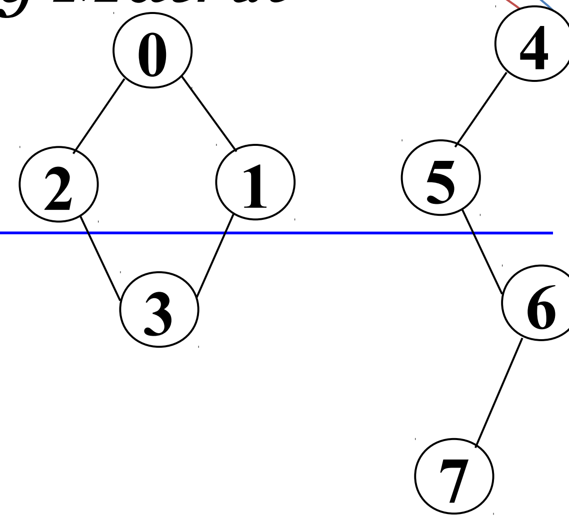
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1



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

G_2



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

symmetric

undirected: $n^2/2$

directed: n^2

G_4

Merits of Adjacency Matrix

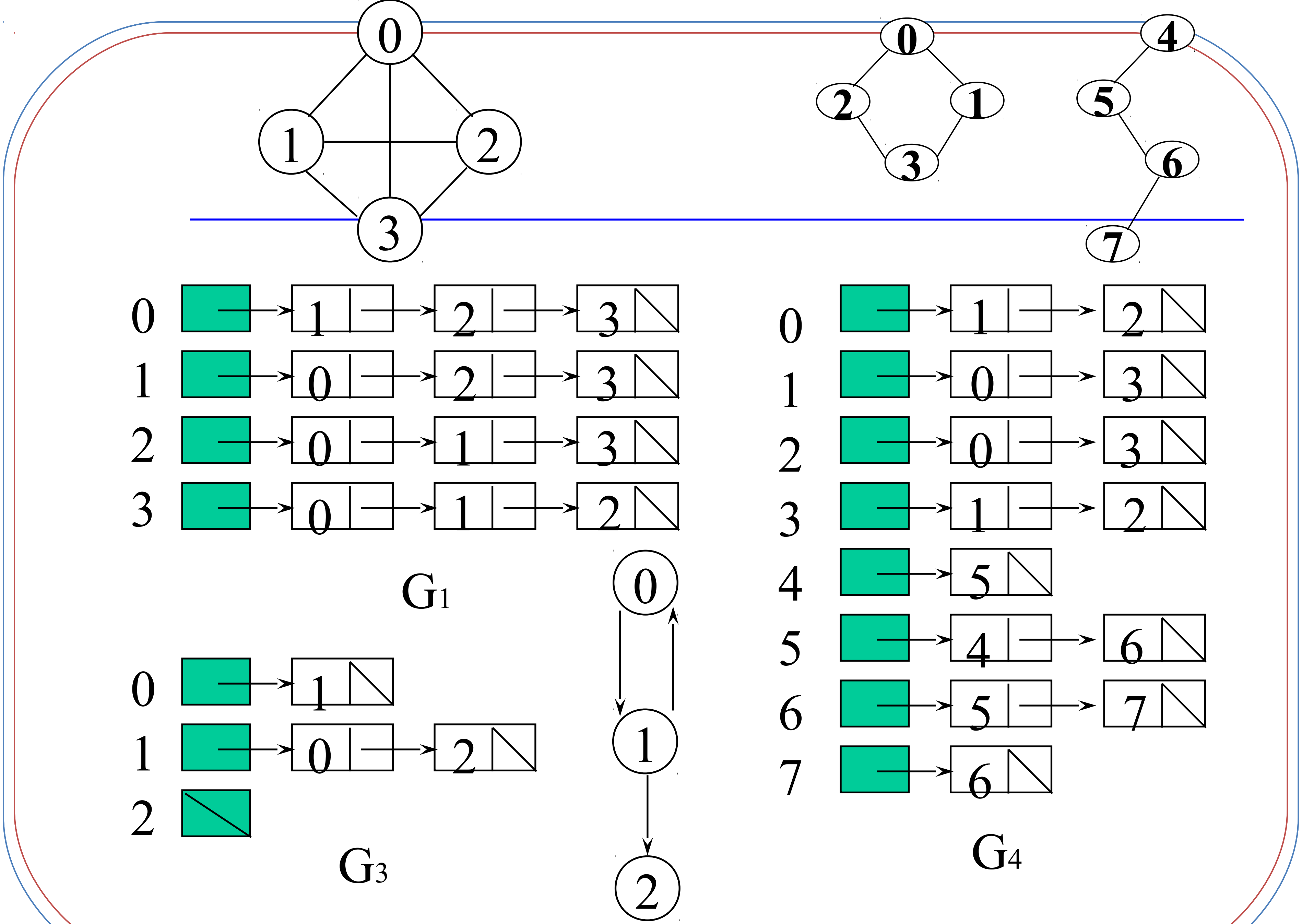
- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$
- For a digraph (= directed graph), the row sum is the out_degree, while the column sum is the in_degree

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

Adjacency Lists (data structures)

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

```
#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 */
```



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

Some Operations

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

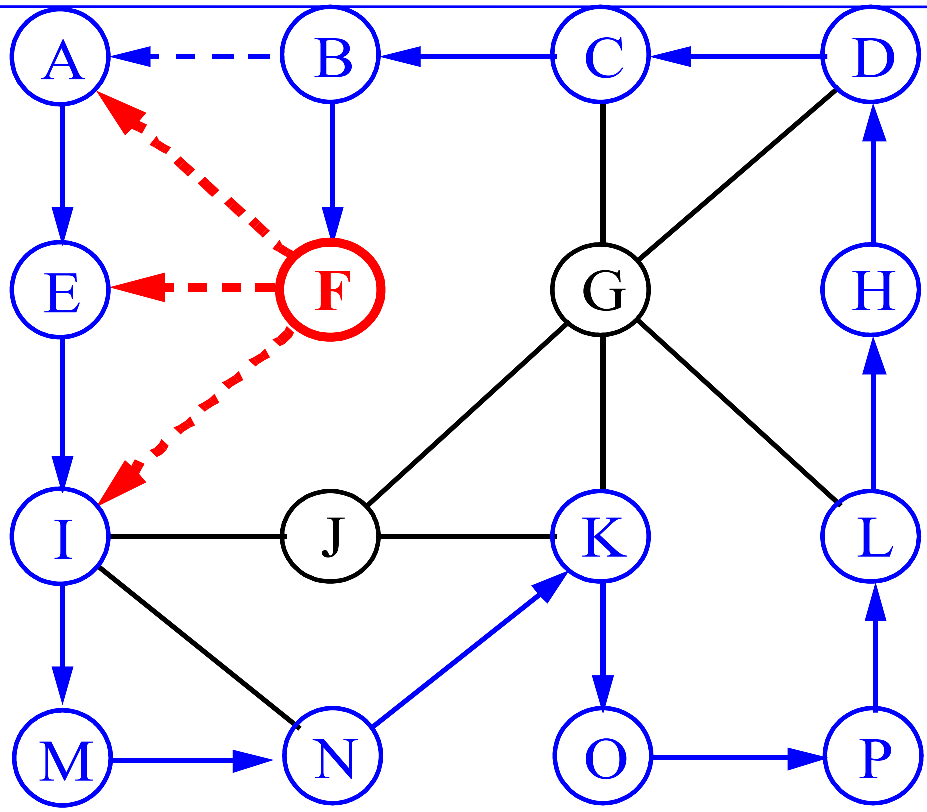
in-degree of a vertex in a directed graph

–traverse the whole data structure

Graph Traversal

- Problem: Search for a certain node or traverse all nodes in the graph
- 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

Depth-First Search



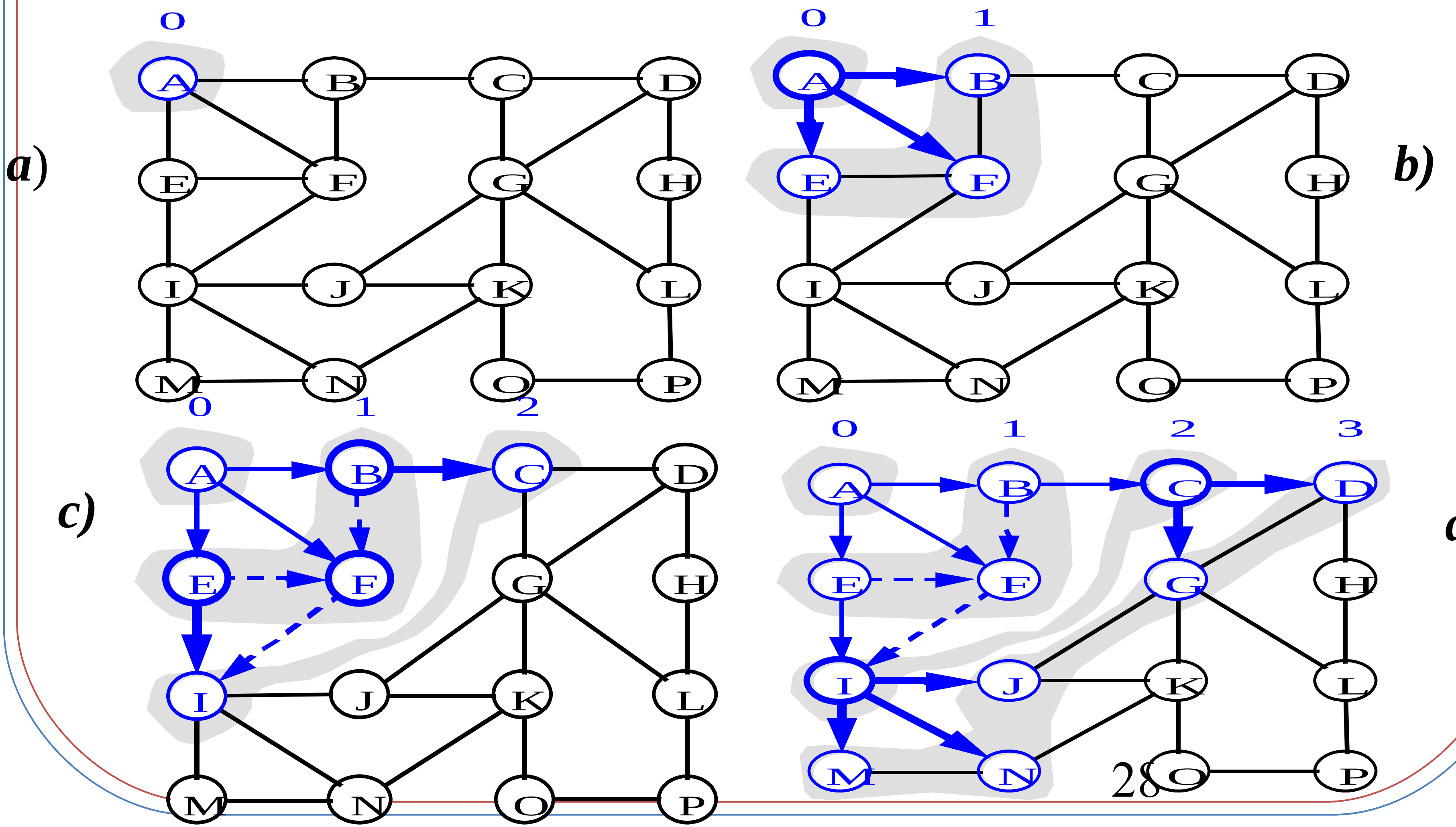
Exploring a Labyrinth Without Getting Lost

- 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.

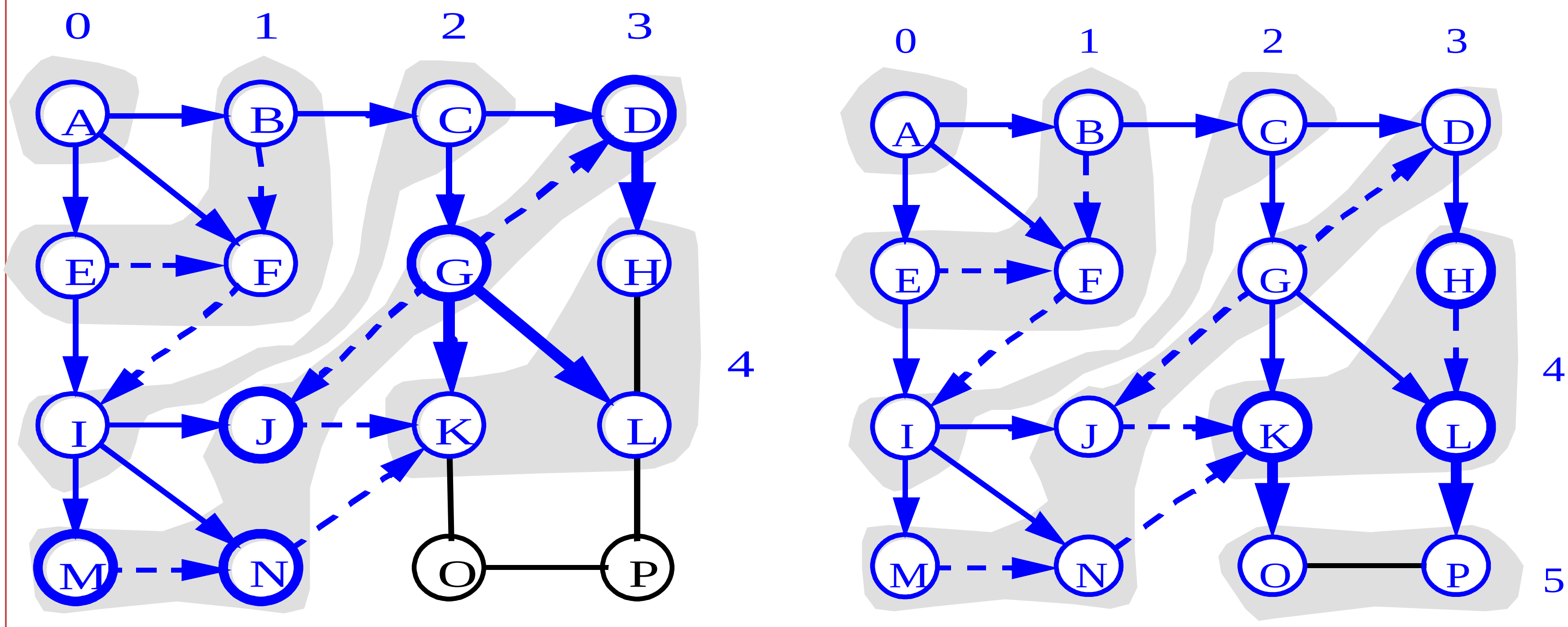
Breadth-First Search

- 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.

BFS - A Graphical Representation



More BFS

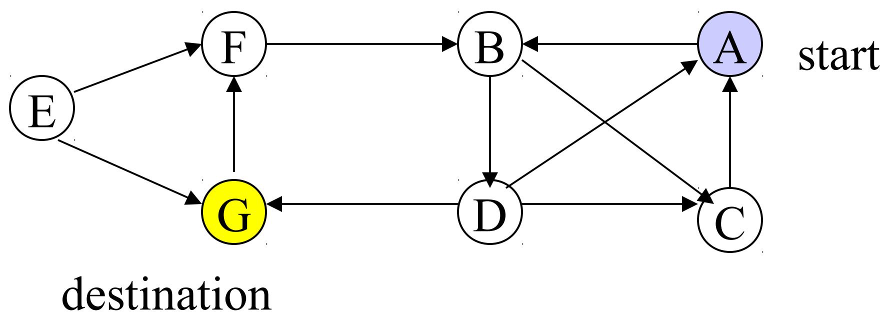


Applications: Finding a Path

- 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?

DFS vs. BFS

DFS Process



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

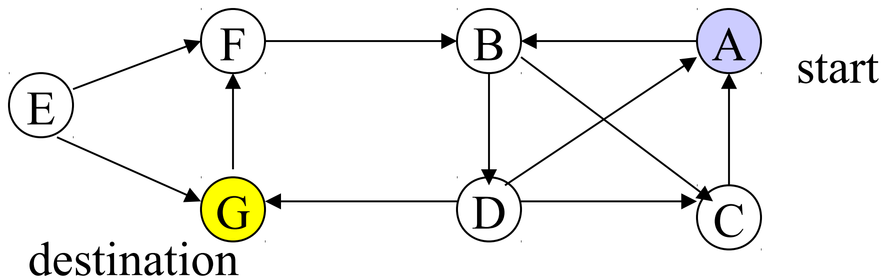
found destination - done!

Path is implicitly stored in DFS recursion

Path is: A, B, D, G

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