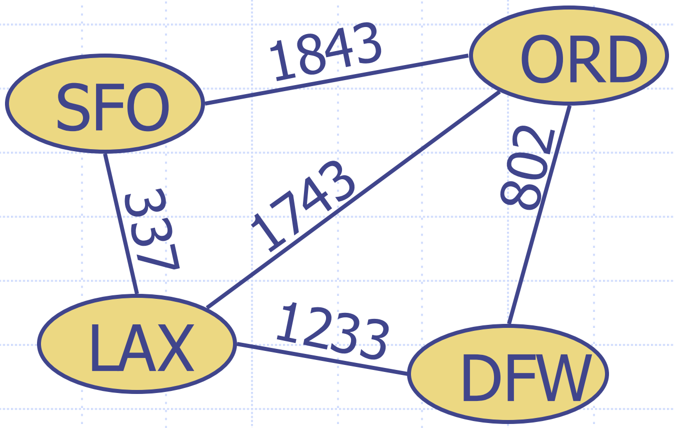
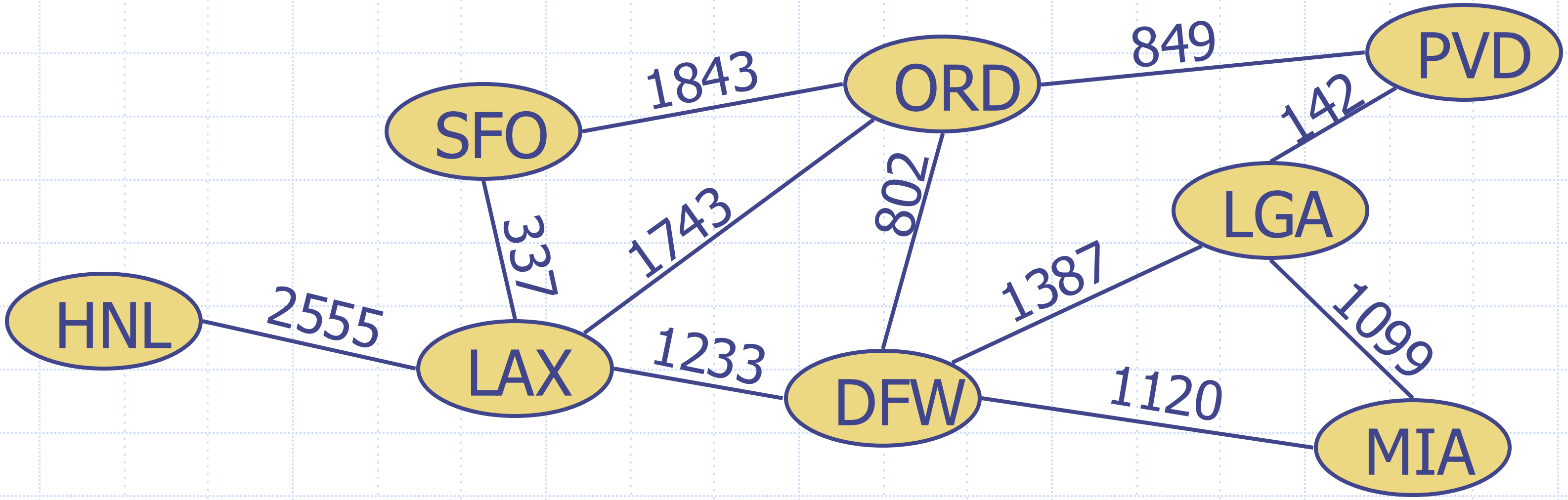
Graphs



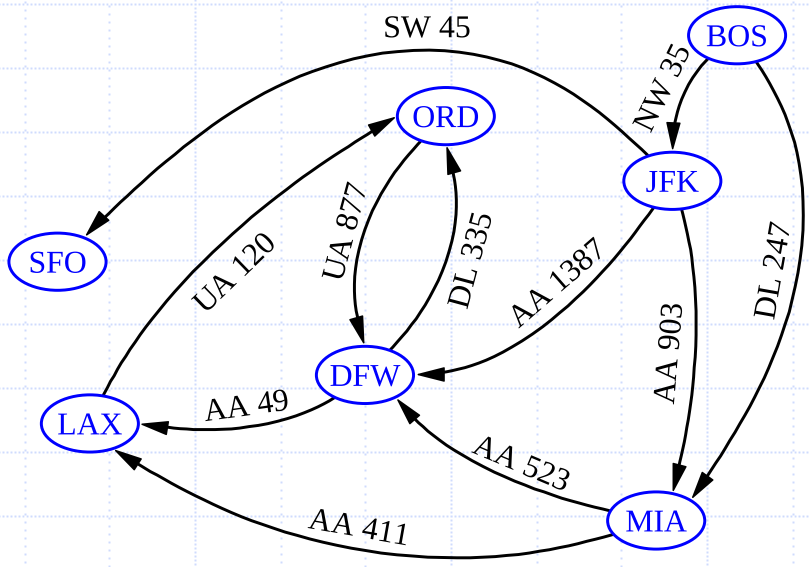
Graphs

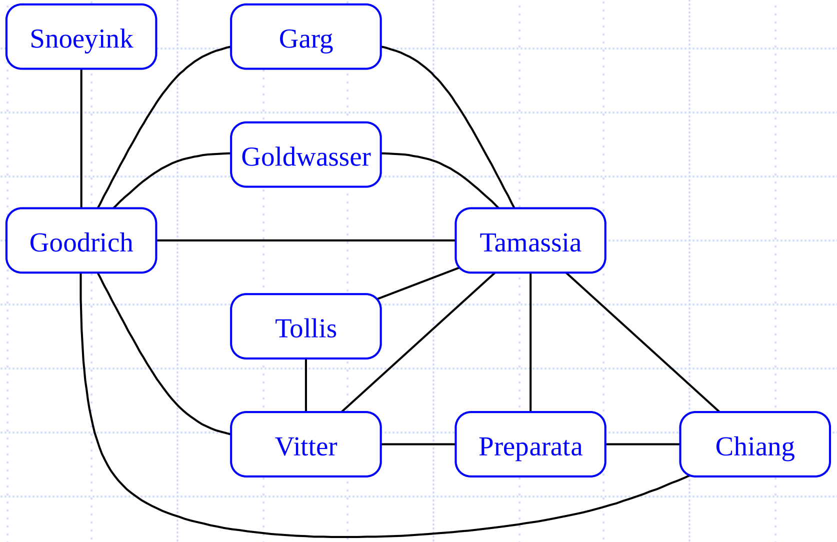
- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



Edge Types

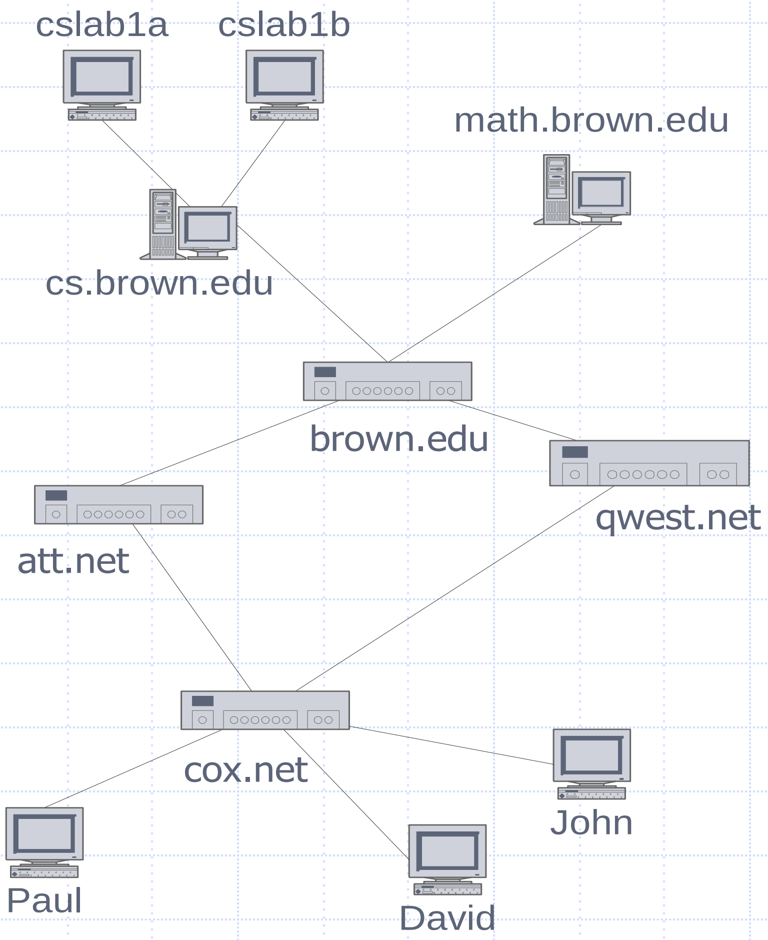
- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Directed graph
  - all the edges are directed
  - e.g., route network
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- Undirected graph
  - all the edges are undirected
  - e.g., flight network





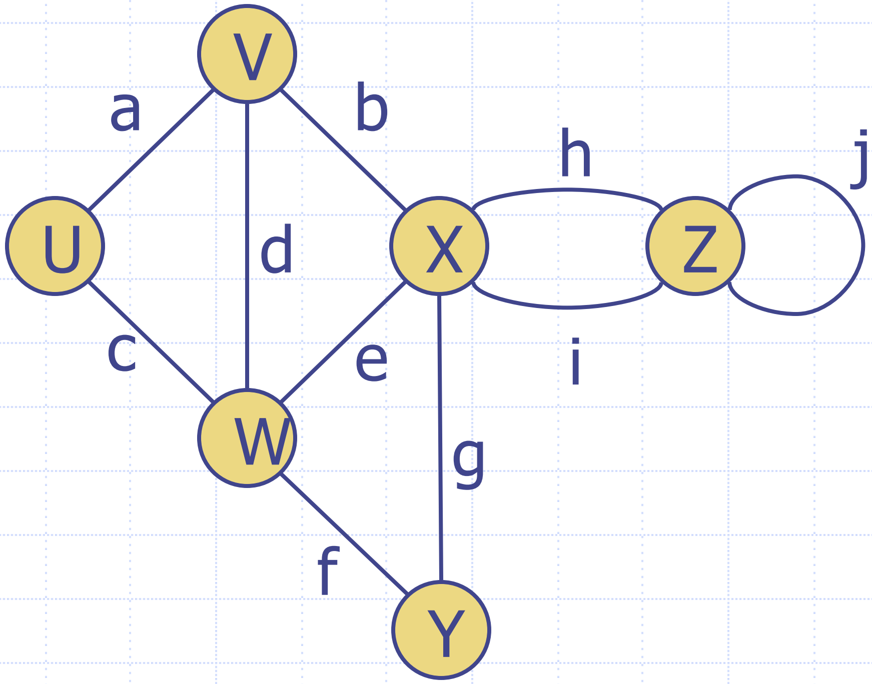
Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram



Terminology

- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
  - outgoing edges
  - incoming edges
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
    - X has degree 5
  - in-degree
  - out-degree
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop

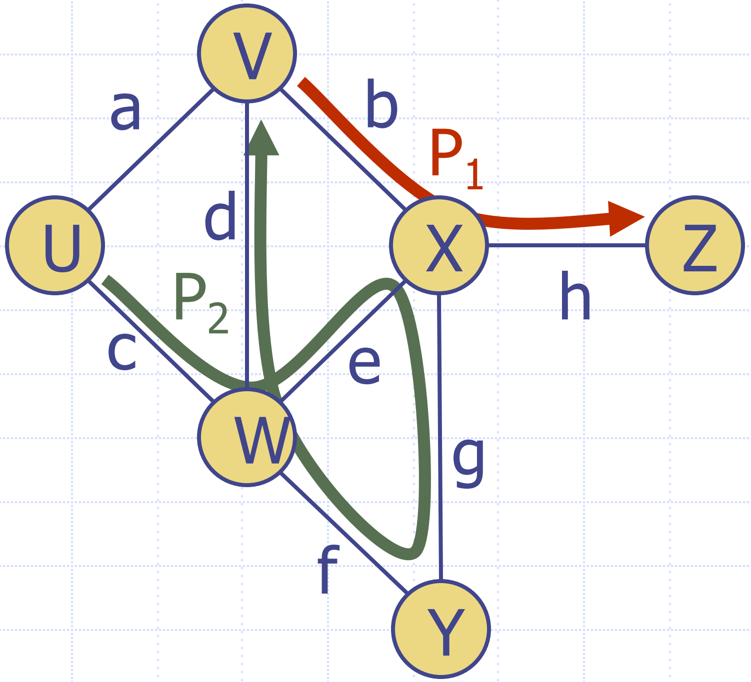


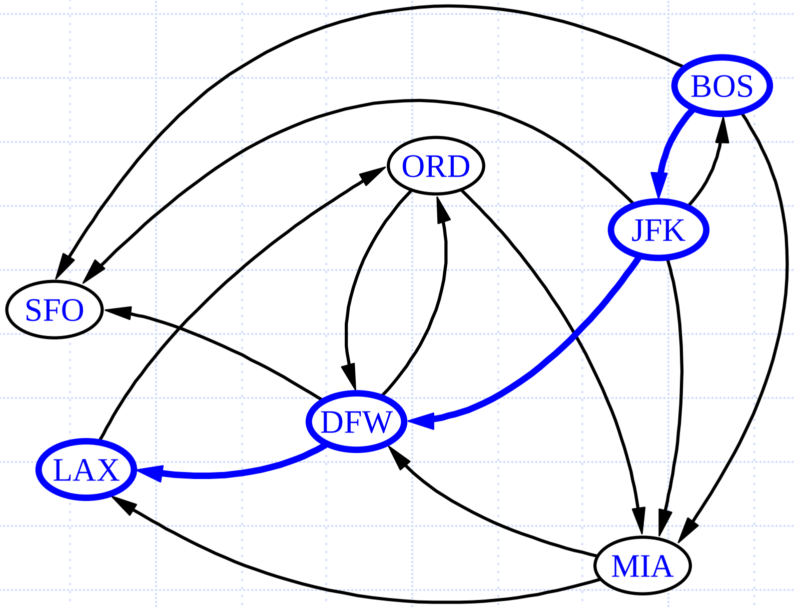
Question

- What is the sum of degrees of all vertices?

Terminology (2)

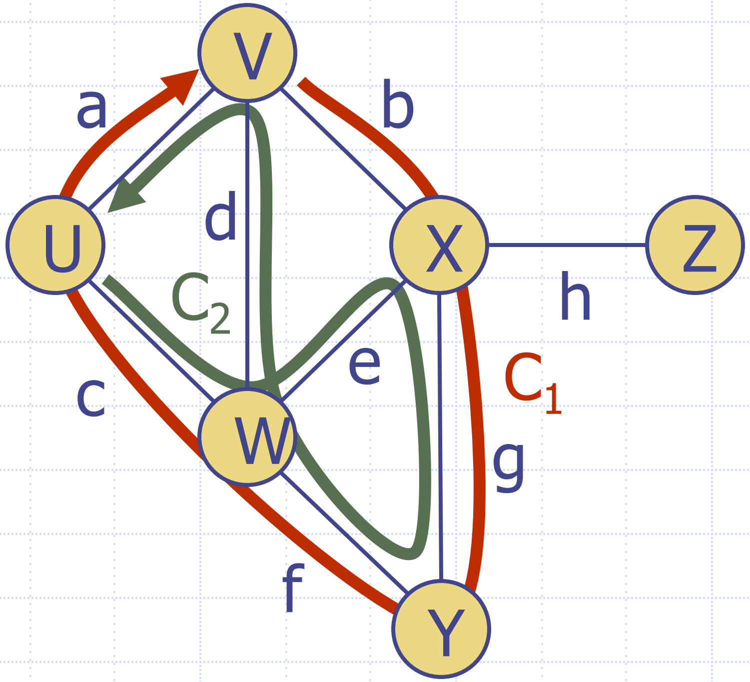
- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple





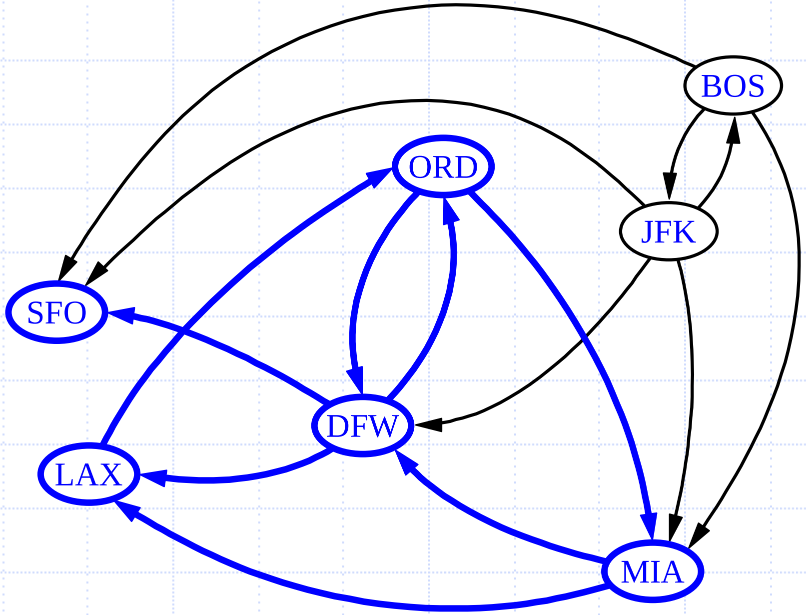
Terminology (3)

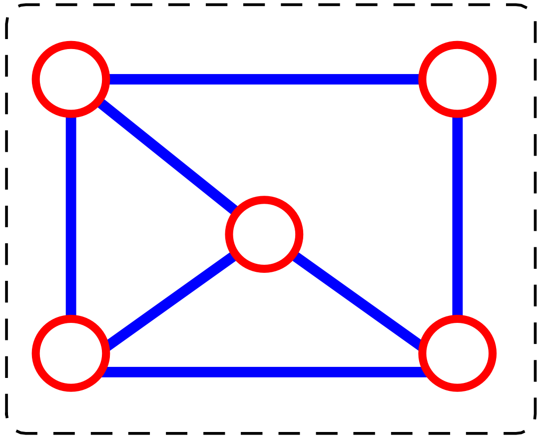
- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is not simple



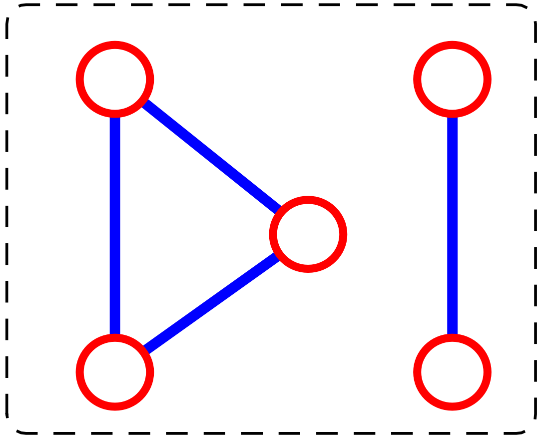
Terminology (4)

- Reachable
  - V is reachable from U if there is a path from U to V.
  - Undirected graph – reachability is symmetric
- Connected
  - a graph is connected if, for any two vertices, there is a path between them
- subgraph
  - subset of vertices and edges forming a graph





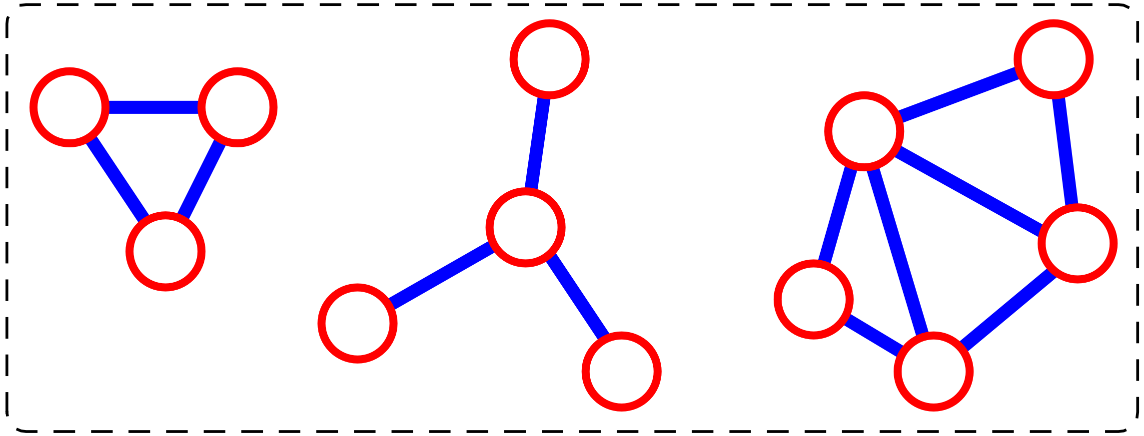
connected



not connected

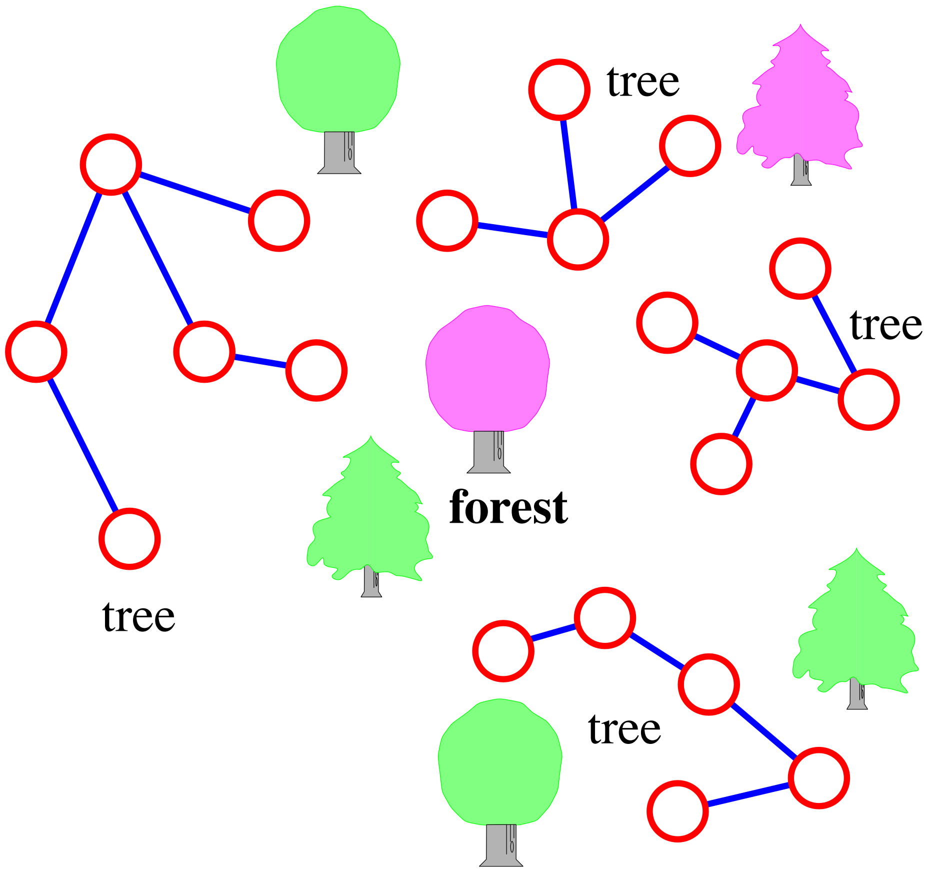
Terminology (5)

- Connected Component: maximal connected subgraph, e.g., the graph below has 3 connected components.



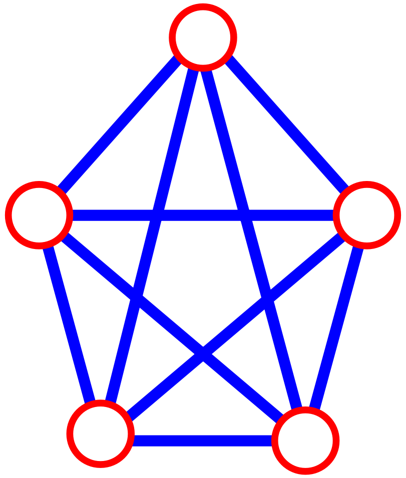
Terminology (6)

- Tree – connected graph without cycles
  - free tree
- Forest – collection of trees



Connectivity

- Let
  - n = # of vertices
  - m = # of edges

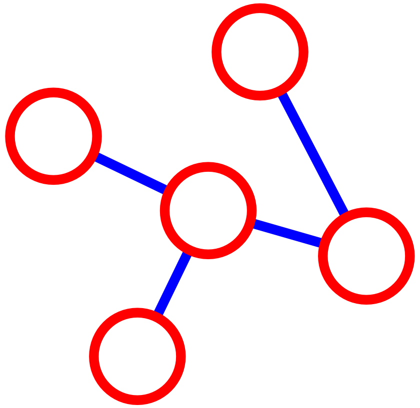


$\mathbf{n} = 5$
$\mathbf{m} = (5 * 4)/2 = 10$

- Complete graph – all pairs of vertices are adjacent
  - There are n(n-1)/2 pairs of vertices and so m=n(n-1)/2
- If a graph is not complete, m<n(n-1)/2

Connectivity (2)

- n = #vertices
- m= #edges
- For a tree m=n-1

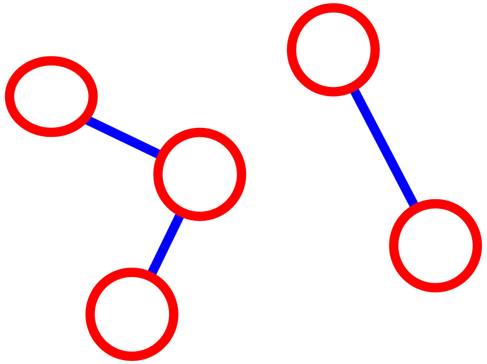


n = 5
m = 4

- Prove!
  - Tree – connected graph without cycles

Connectivity (3)

- n = #vertices
- m= #edges
- For a tree m=n-1
- If m<n-1, G is not connected

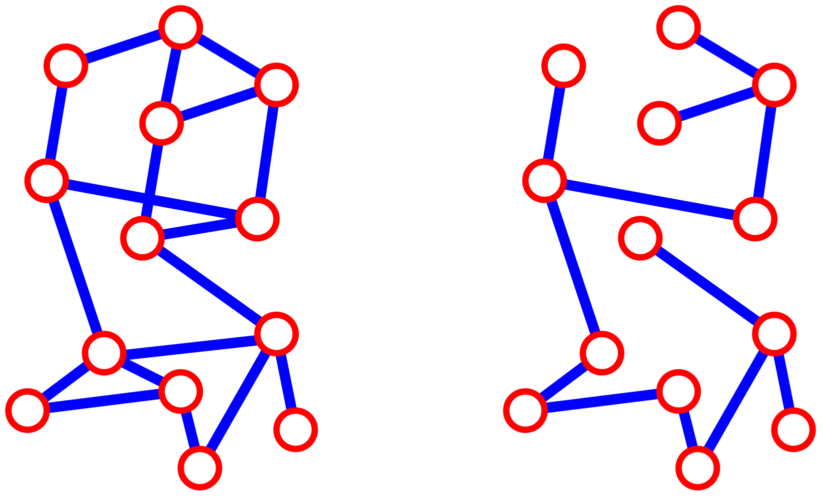


$\mathbf{n} = 5$
$\mathbf{m} = 3$

- Prove

Spanning Tree

- A spanning tree of G is a subgraph which
  - is a tree
  - contains all vertices of G

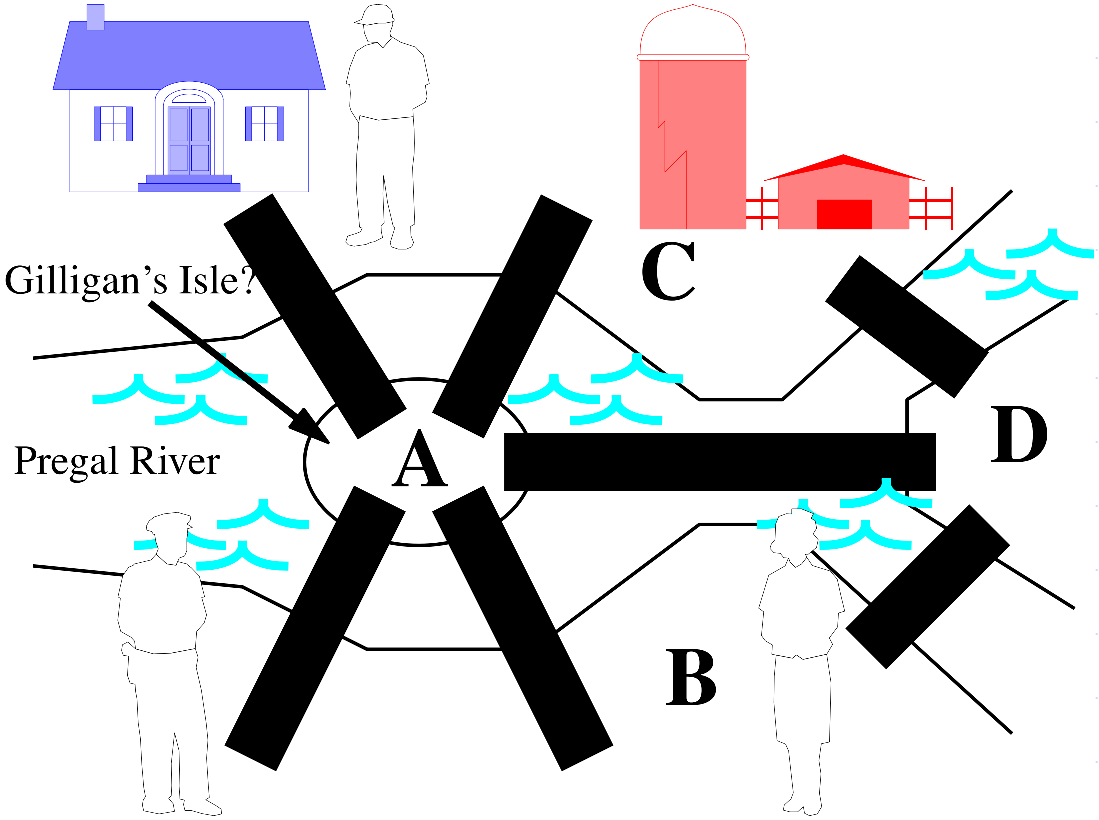


G

spanning tree of G

- Failure of any edge disconnects system

Euler and the Koenigsberg Bridges

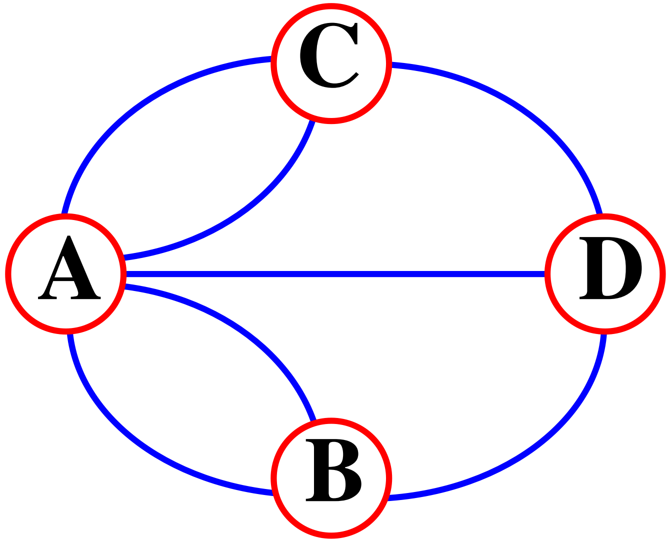


Can one walk across each bridge exactly once and return at the starting point?

In 1736, Euler proved that this is not possible.

Graph Model (with parallel edges)

- Eulerian Tour- path that traverses every edge exactly once and returns to the first vertex
- Euler's Theorem – A graph has a Eulerian tour if and only if all vertices have even degree

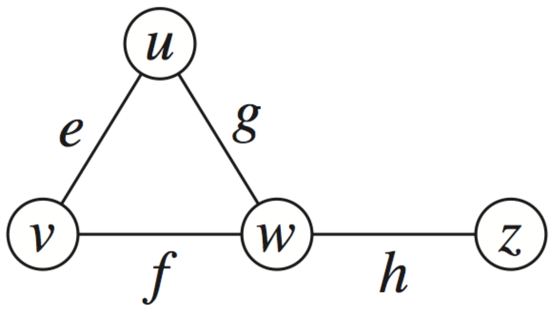


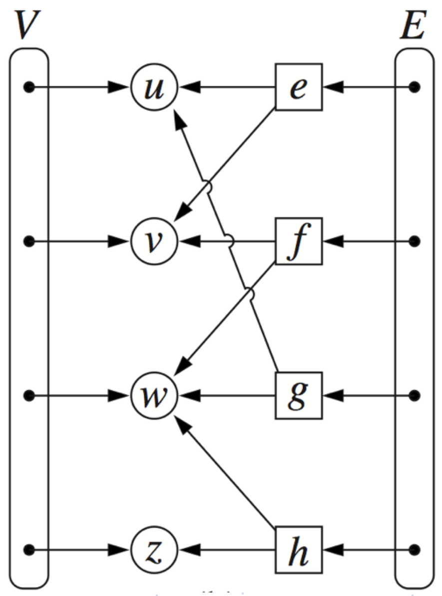
Vertices and Edges

- A **graph** is a collection of **vertices** and **edges**.
- We model the abstraction as a combination of three data types: Vertex, Edge, and Graph.
- A **Vertex** is a lightweight object that stores an arbitrary element provided by the user (e.g., an airport code)
  - We assume it supports a method, element(), to retrieve the stored element.
- An **Edge** stores an associated object (e.g., a flight number, travel distance, cost), retrieved with the element( ) method.

Edge List Structure

- Vertex object
  - element
  - reference to position in vertex sequence
- Edge object
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- Vertex sequence
  - sequence of vertex objects
- Edge sequence
  - sequence of edge objects



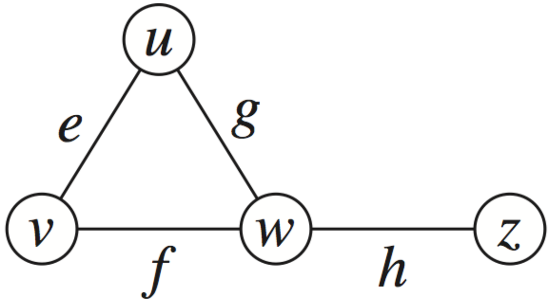


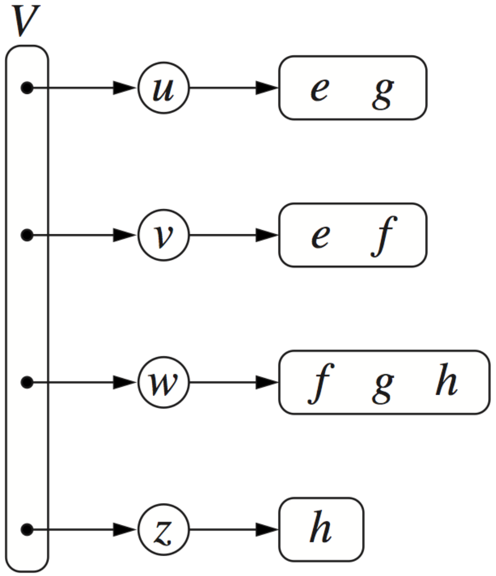
Performance

- n vertices, m edges - no parallel edges - no self-loops	Edge List		
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m		
areAdjacent (v, w)	m		
insertVertex(o)	1		
insertEdge(v, w, o)	1		
removeVertex(v)	m		
removeEdge(e)	1		

Adjacency List Structure

- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices



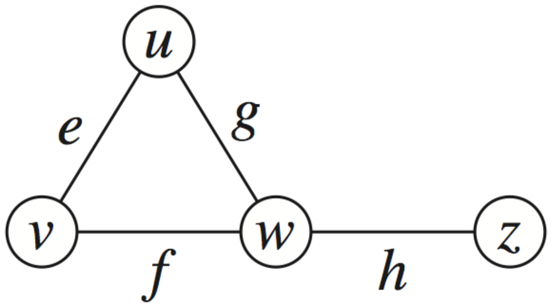


Performance

• n vertices, m edges • no parallel edges • no self-loops	Edge List	Adjacency List	
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	$\deg(v)$	
areAdjacent (v, w)	m	$\min(\deg(v), \deg(w))$	
insertVertex(o)	1	1	
insertEdge(v, w, o)	1	1	
removeVertex(v)	m	$\deg(v)$	
removeEdge(e)	1	1	

Adjacency Matrix Structure

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- The “old fashioned” version just has 0 for no edge and 1 for edge



		0	1	2	3
$u \longrightarrow$	0		e	g	
$v \longrightarrow$	1	e		f	
$w \longrightarrow$	2	g	f		h
$z \longrightarrow$	3			h	

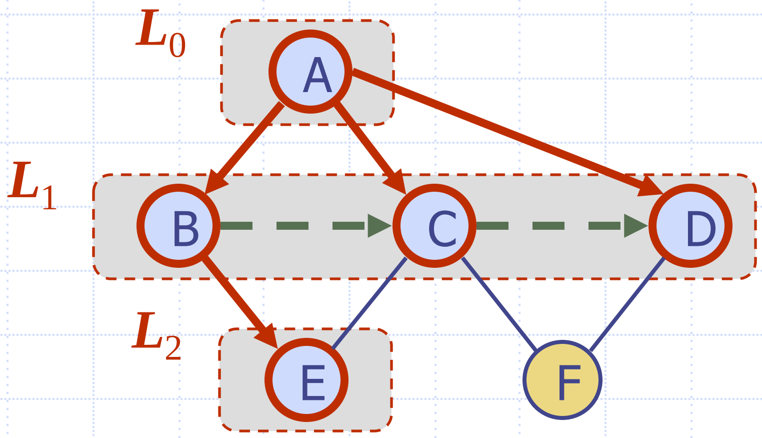
Performance

n vertices, m edges - no parallel edges - no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	$\deg(v)$	n
areAdjacent (v, w)	m	$\min(\deg(v), \deg(w))$	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	$\deg(v)$	n^2
removeEdge(e)	1	1	1

Graph Search Algorithms

- Systematic search of every edge and vertex of the graph
- Graph $G = (V, E)$ is either directed or undirected
- Applications
  - Compilers
  - Networks
  - Graphics
  - Gaming

Breadth-First Search



Breadth-First Search

- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- BFS on a graph with n vertices and m edges takes $O(n + m)$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices

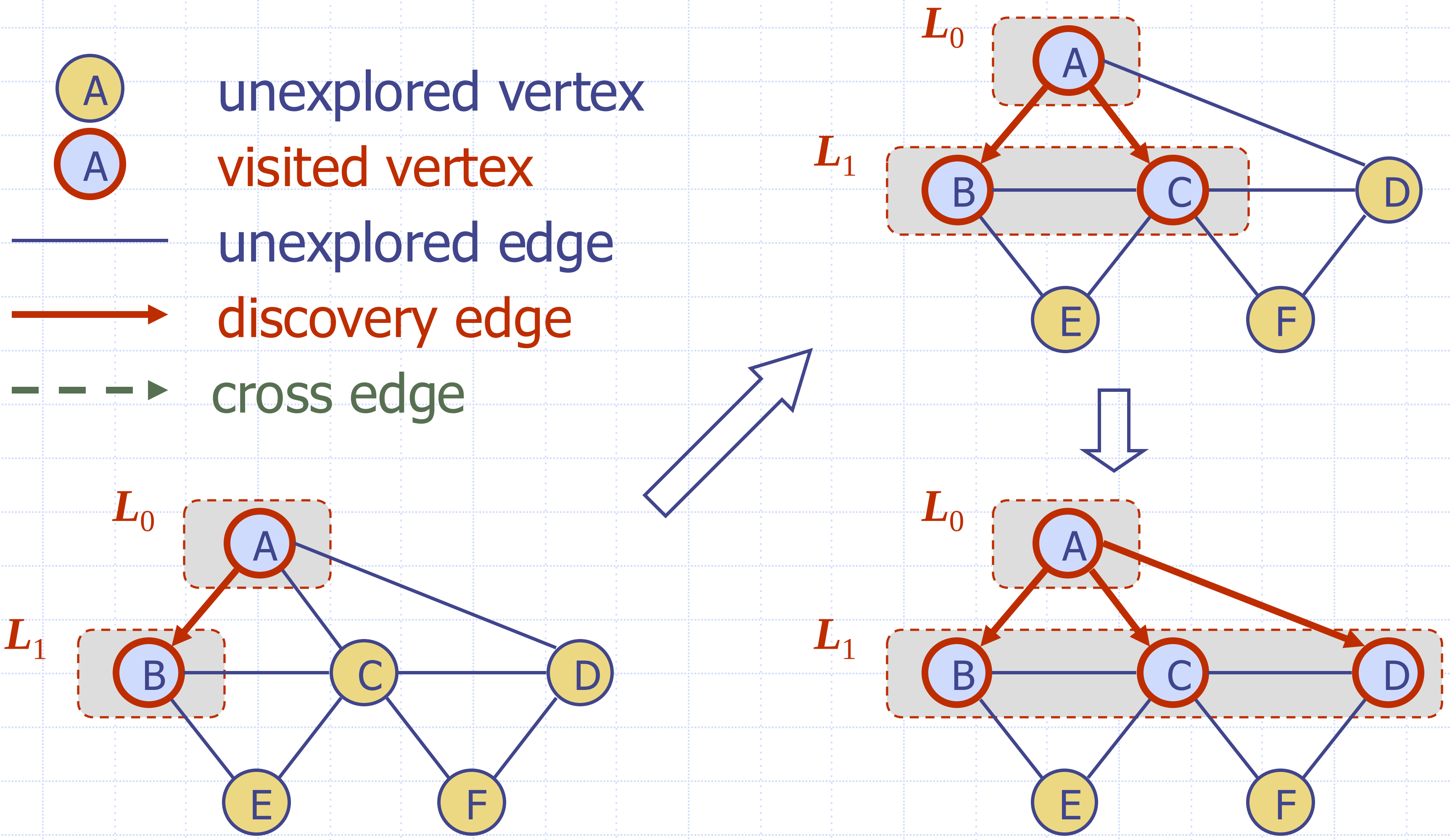
BFS Algorithm

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

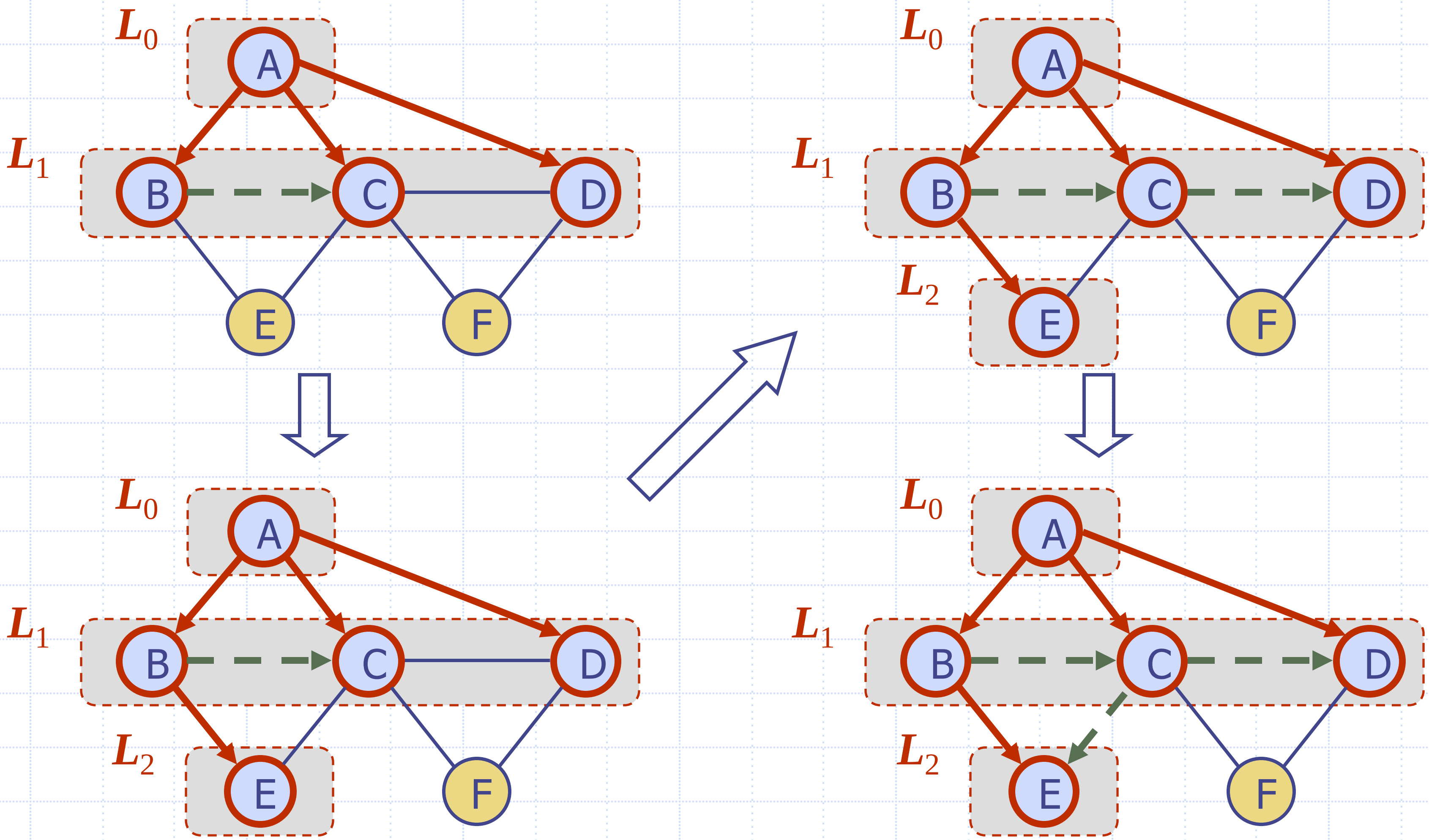
```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      BFS(G, v)
```

```
Algorithm BFS(G, s)
  L_0 ← new empty sequence
  L_0.addLast(s)
  setLabel(s, VISITED)
  i ← 0
  while ¬L_i.isEmpty()
    L_{i+1} ← new empty sequence
    for all v ∈ L_i.elements()
      for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            L_{i+1}.addLast(w)
          else
            setLabel(e, CROSS)
    i ← i + 1
```

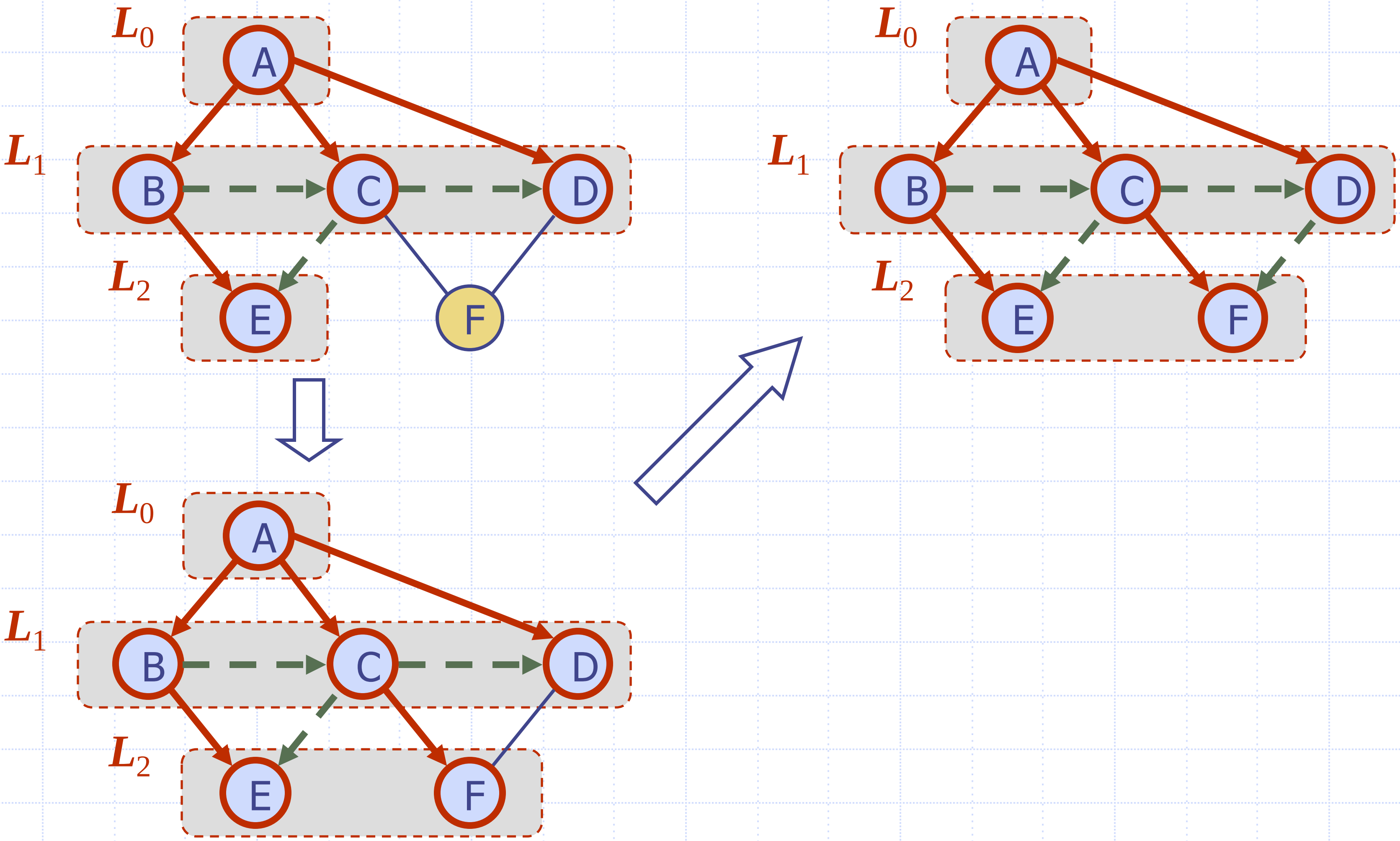
Example


A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
cross edge
L_0
L_1
A
B
C
D
E
F
L_0
L_1
A
B
C
D
E
F
L_0
L_1
A
B
C
D
E
F

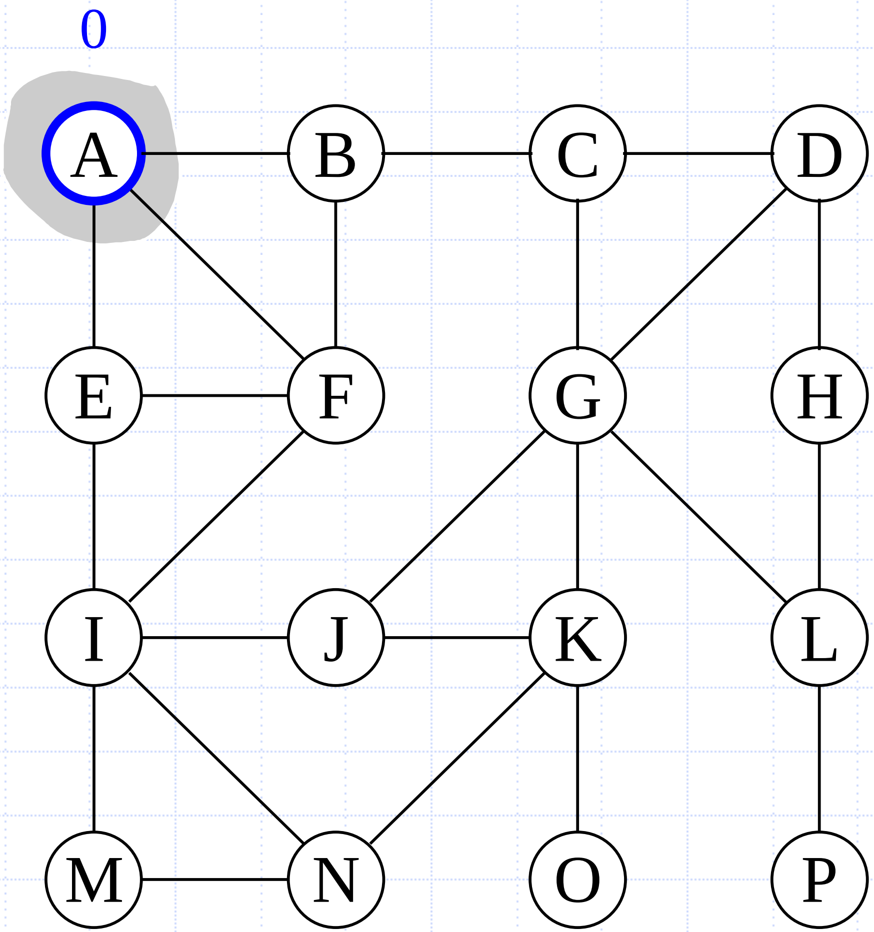
Example (cont.)



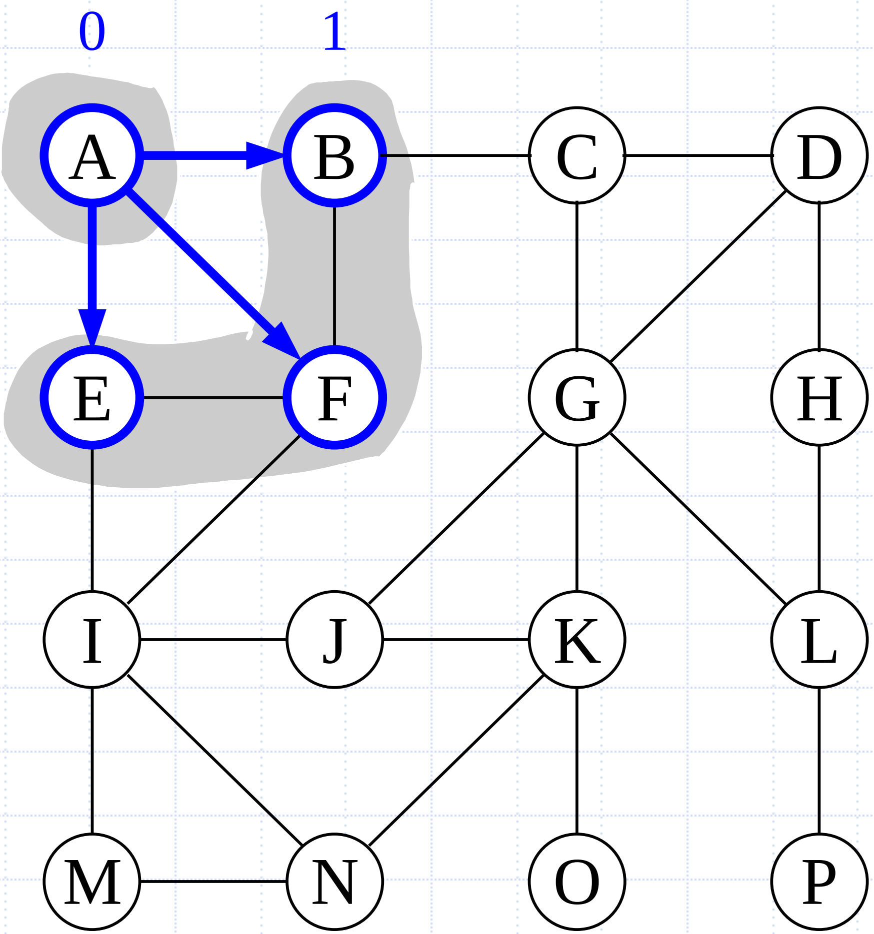
Example (cont.)



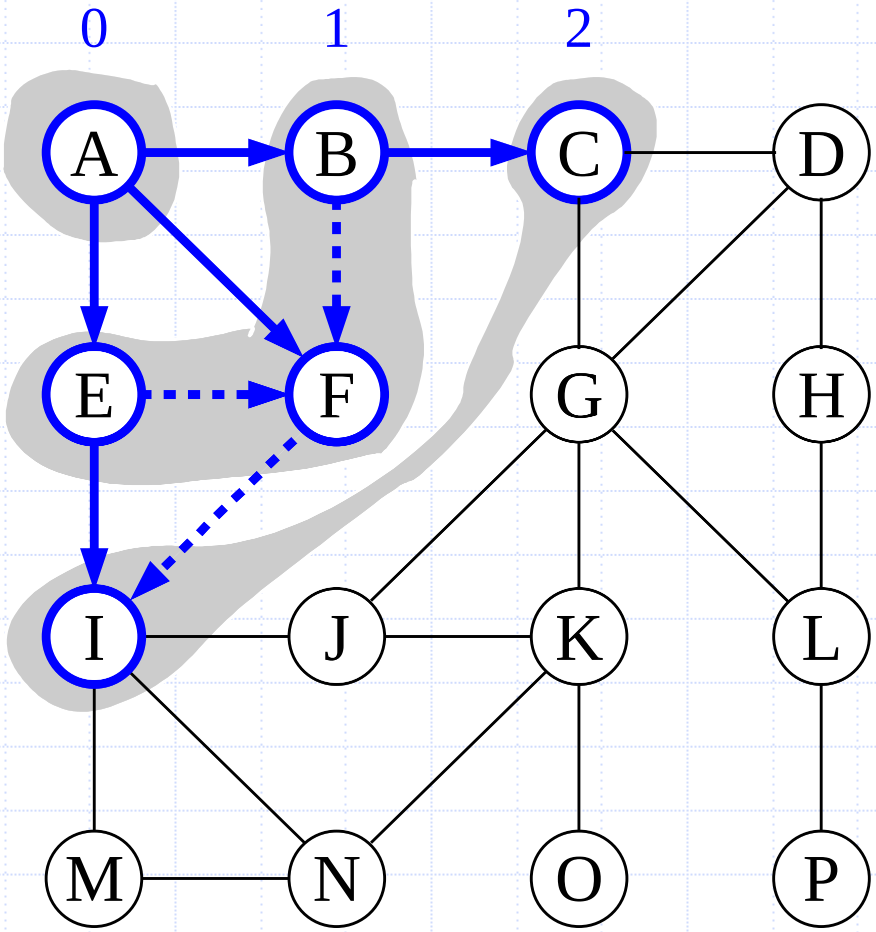
BFS - Example



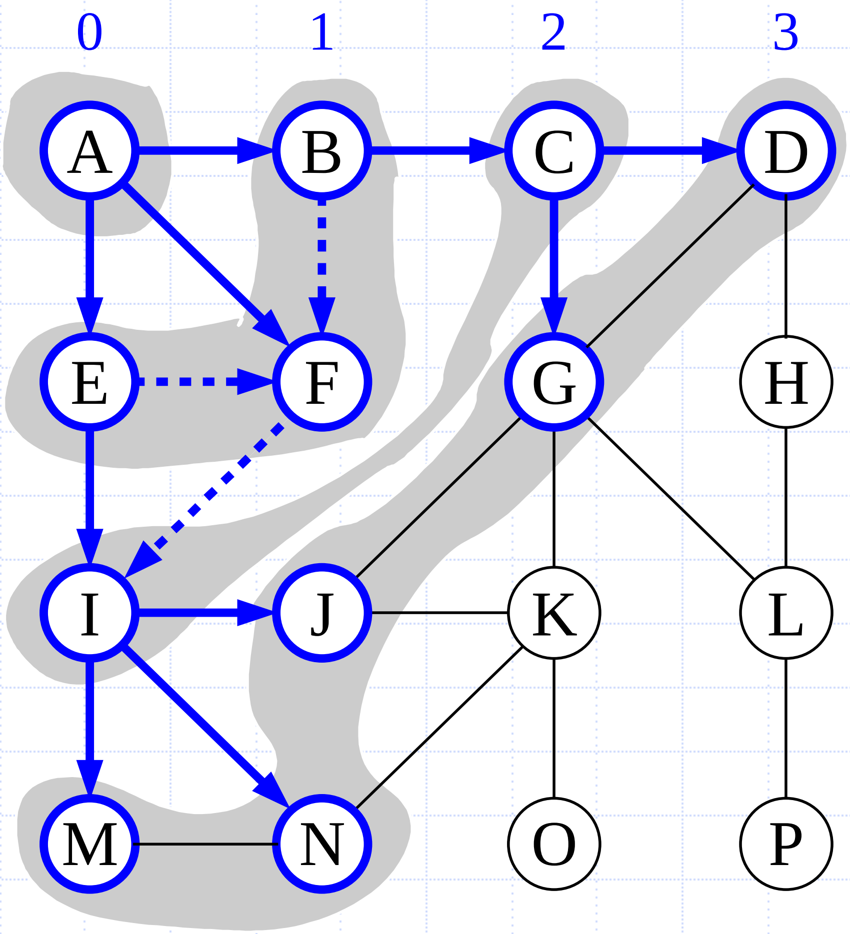
BFS - Example



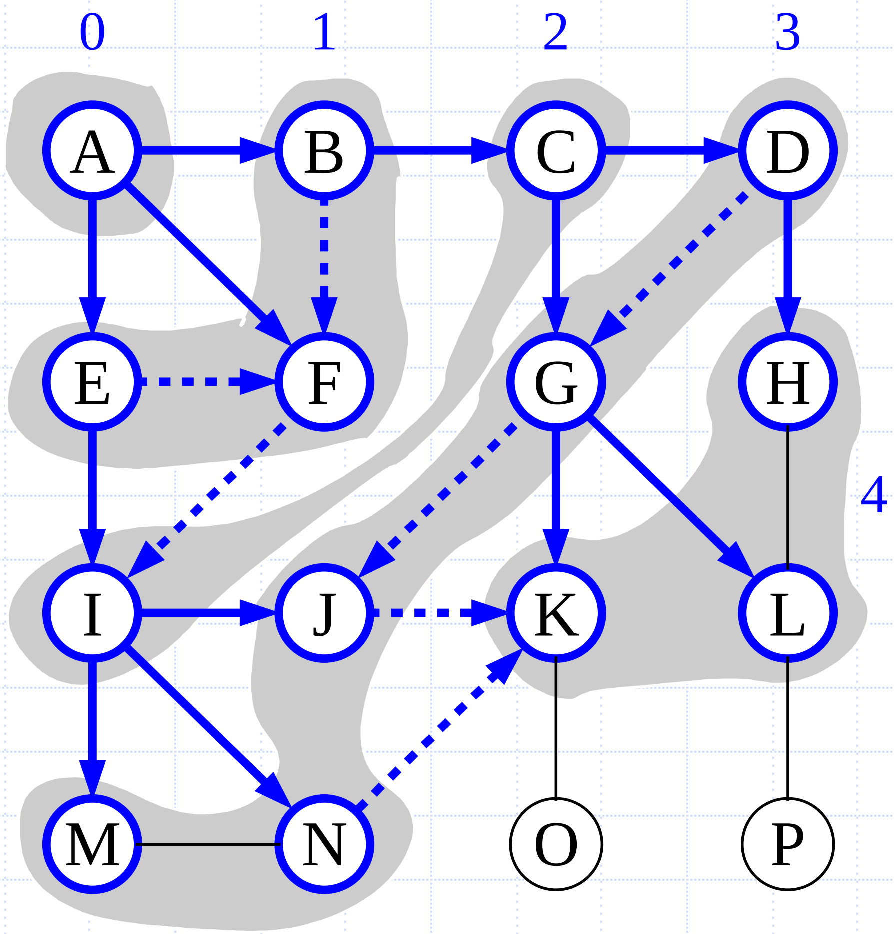
BFS - Example



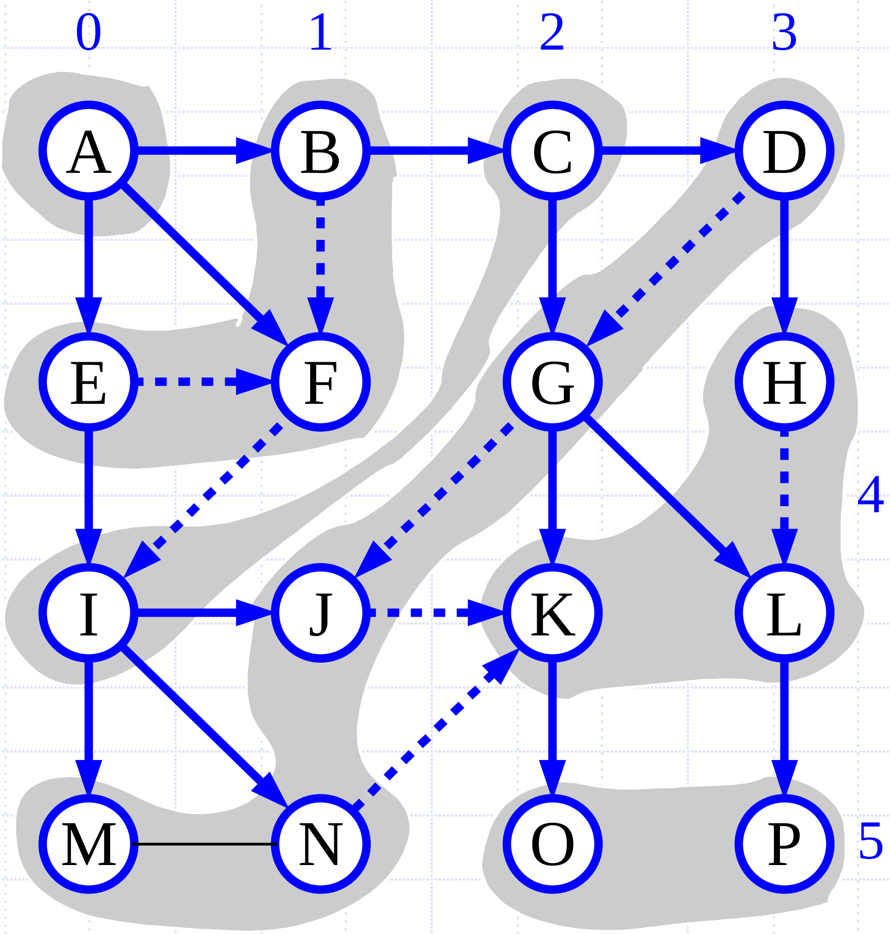
BFS - Example



BFS - Example



BFS - Example



Analysis

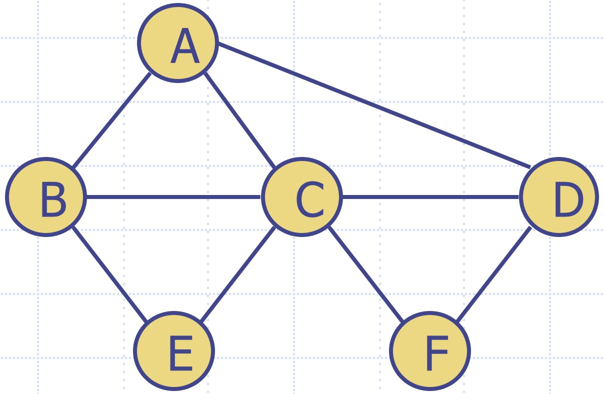
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges is called once for each vertex
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

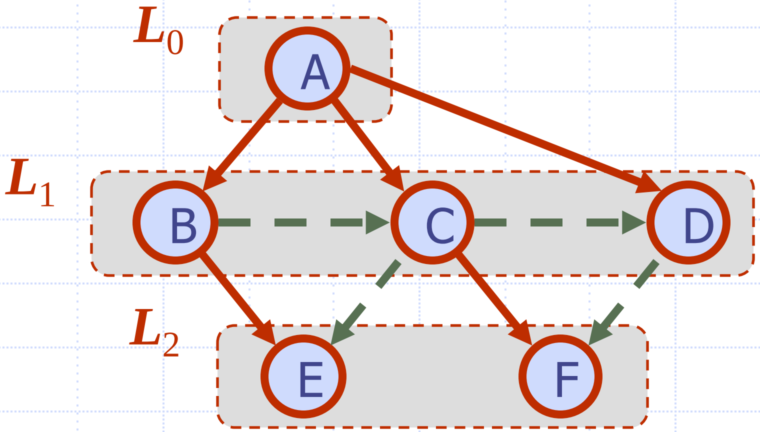
Properties

Notation

G_s: connected component of s

- G_s is a breadth first tree
  - V_s consists of the vertices reachable from s, and
  - for all v in V_s there is a unique simple path from s to v in G_s that is also a shortest path from s to v in G
- The edges in G_s are called tree edges
- For every vertex v reachable from s, the path in the breadth first tree from s to v, corresponds to a shortest path in G





Properties (2)

Notation

G_s: connected component of s

Property 1

$BFS(G, s)$ visits all the vertices and edges of G_s

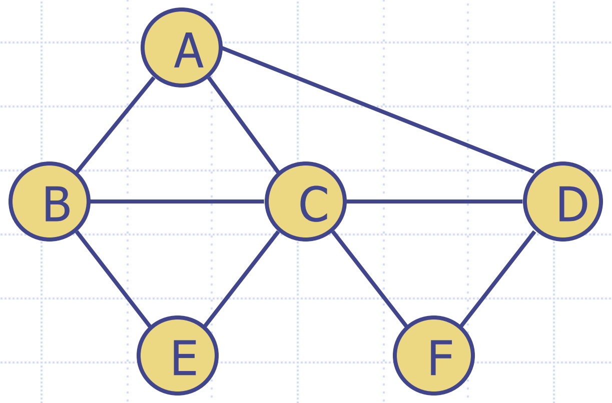
Property 2

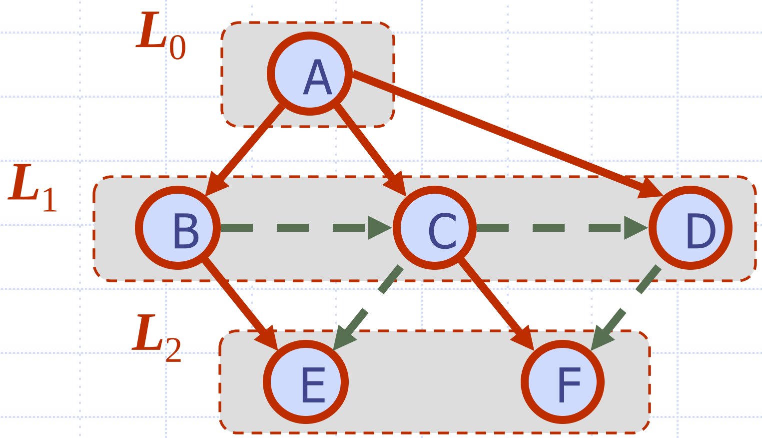
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

Property 3

For each vertex v in L_i

- The path of T_s from s to v has i edges
- Every path from s to v in G_s has at least i edges

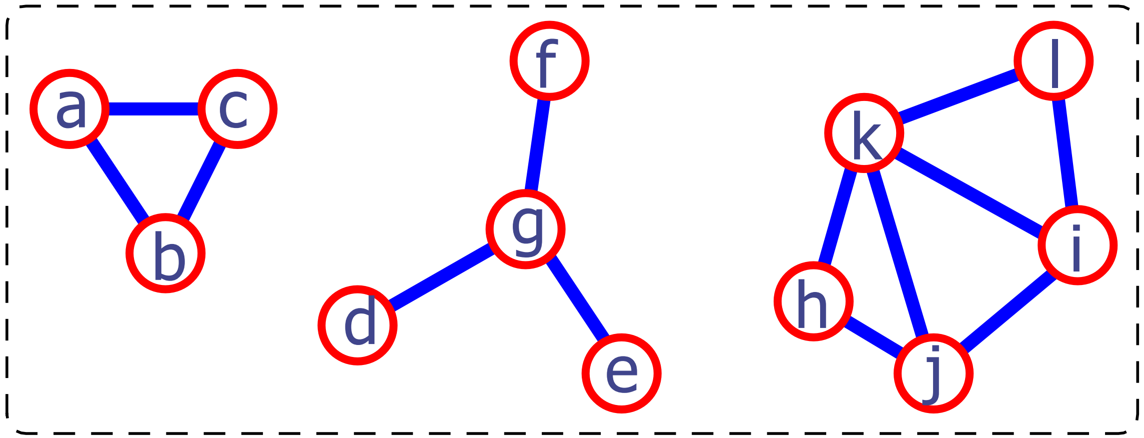




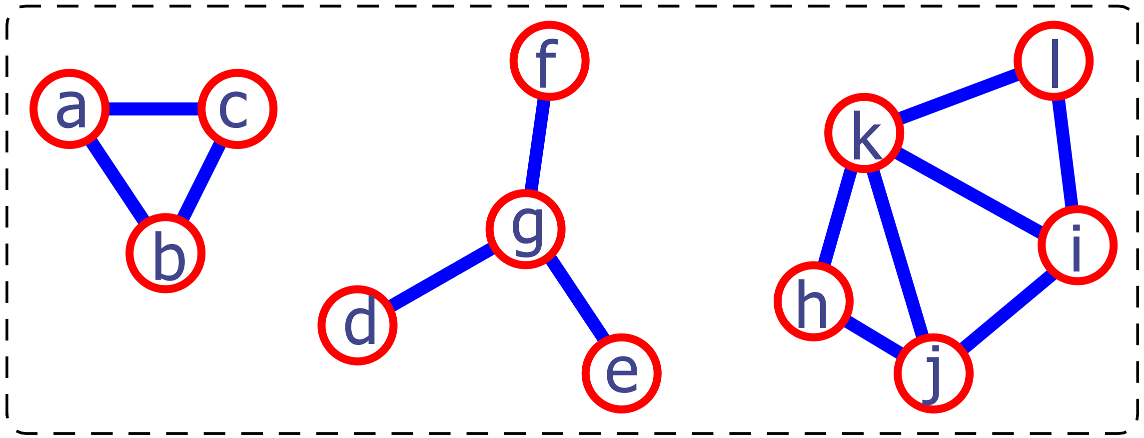
Applications

- We can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Given two vertices of G, find a path in G between them with the minimum number of edges, or report that no such path exists
  - Check if a connected graph G is bipartite.

Applications – Computing Connected Components



Applications – Computing Spanning Forest

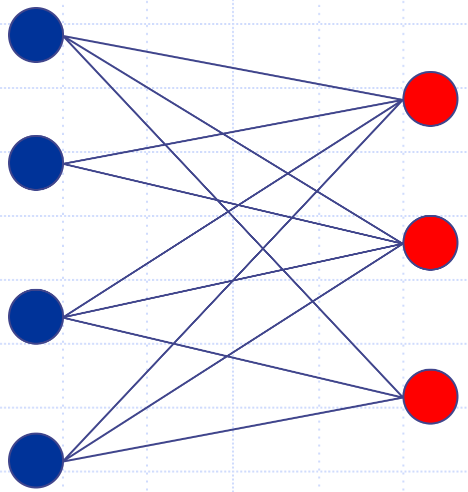


Applications – Shortest Path between two nodes

- In a BFS starting from a vertex v, the level number of vertex u is the length of the shortest path from v to u.
- Proof
  - show there is a path from u to v.
  - a path of smaller length will jump a level
    - violating the BFS.

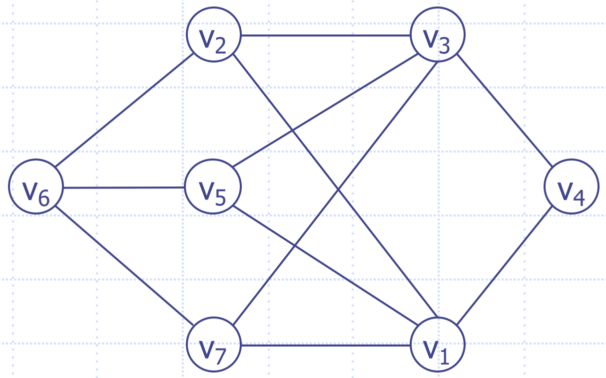
Applications – Check if a graph is bipartite

- An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.

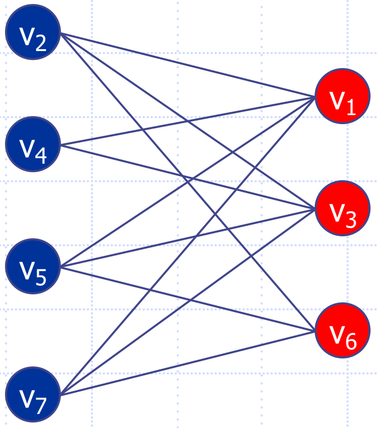


Testing Bipartiteness

- Given a graph G, is it bipartite?
- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.



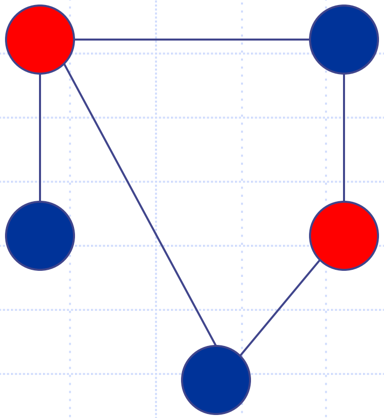
a bipartite graph G



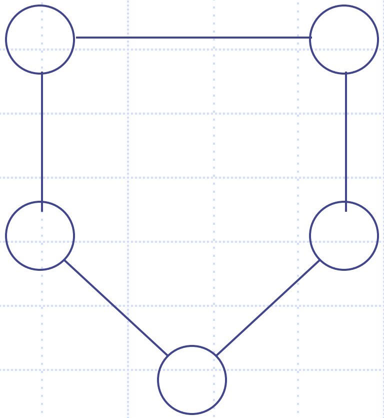
another drawing of G

Preliminary

- Lemma. If G has an odd cycle, then it cannot be bipartite.
- Proof: Not possible to 2-color the odd cycle, let alone G.



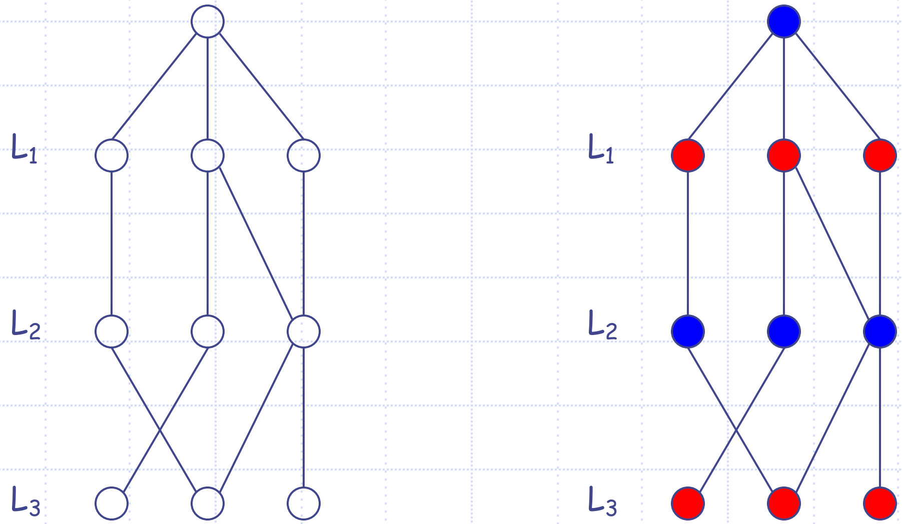
bipartite
(2-colorable)



not bipartite
(not 2-colorable)

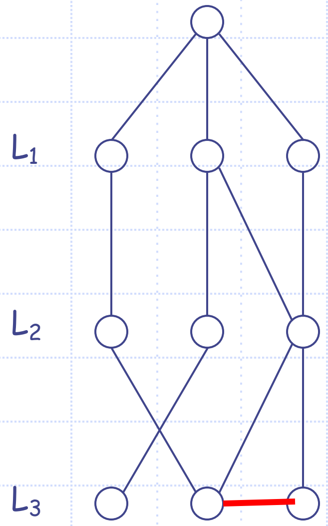
Bipartiteness and BFS

- Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.
  - case 1: No edge of G joins two nodes of the same layer, and G is bipartite.



Bipartiteness and BFS

- Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.
  - case 1: No edge of G joins two nodes of the same layer, and G is bipartite.
  - case 2: An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



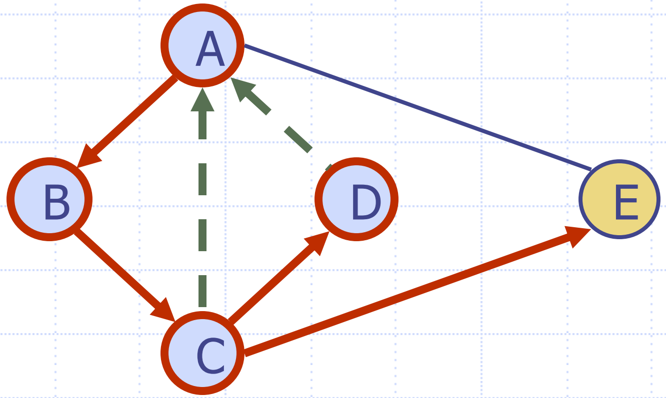
Bipartiteness and BFS

- Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.
  - case 1: No edge of G joins two nodes of the same layer, and G is bipartite.
  - case 2: An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).
- Proof case 2:
  - Suppose (x,y) is an edge with x, y in the same level L_j.
  - Let z = LCA(x, y) – lowest common ancestor
  - Let L_i be the level containing z
  - Consider the cycle that takes edge from x to y, then the path from y to x, then z to x
  - The length of this path is 1+(j-i)+(j-i), which is odd

Question

- If G has only even length cycles, then G is bipartite.

Depth-First Search

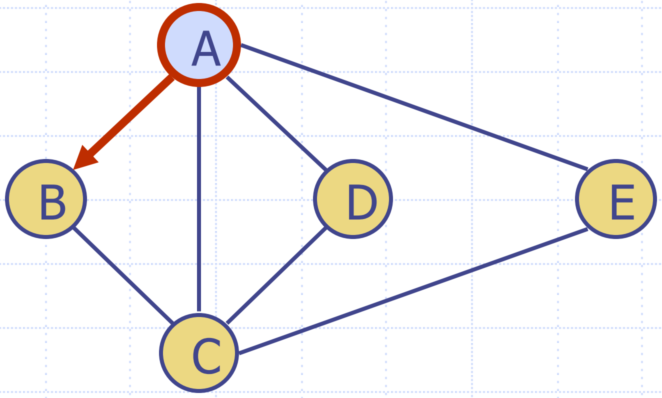


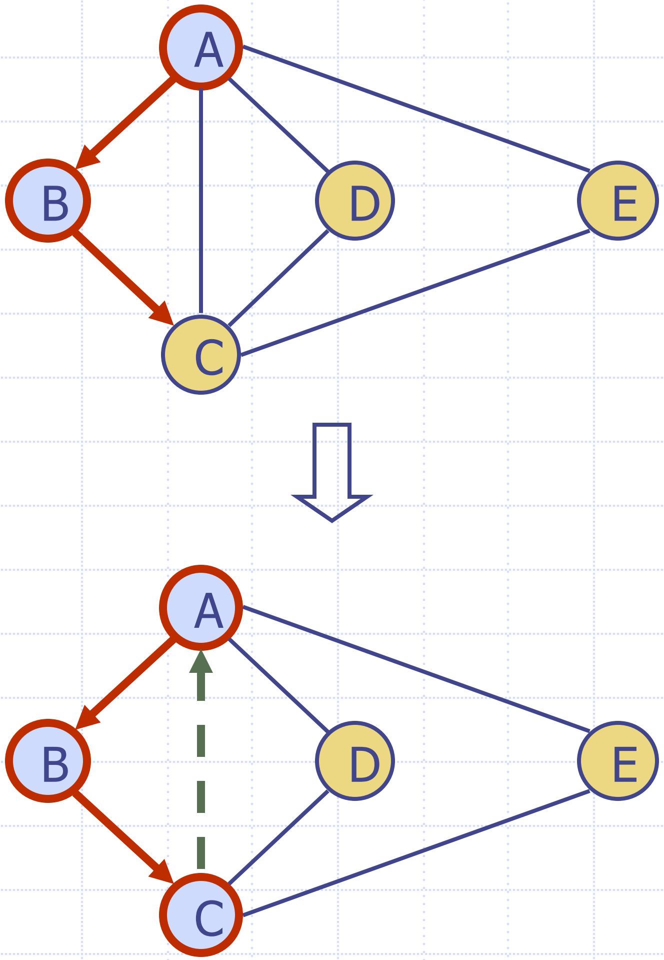
Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph
- Depth-first search is to graphs what Euler tour is to binary trees

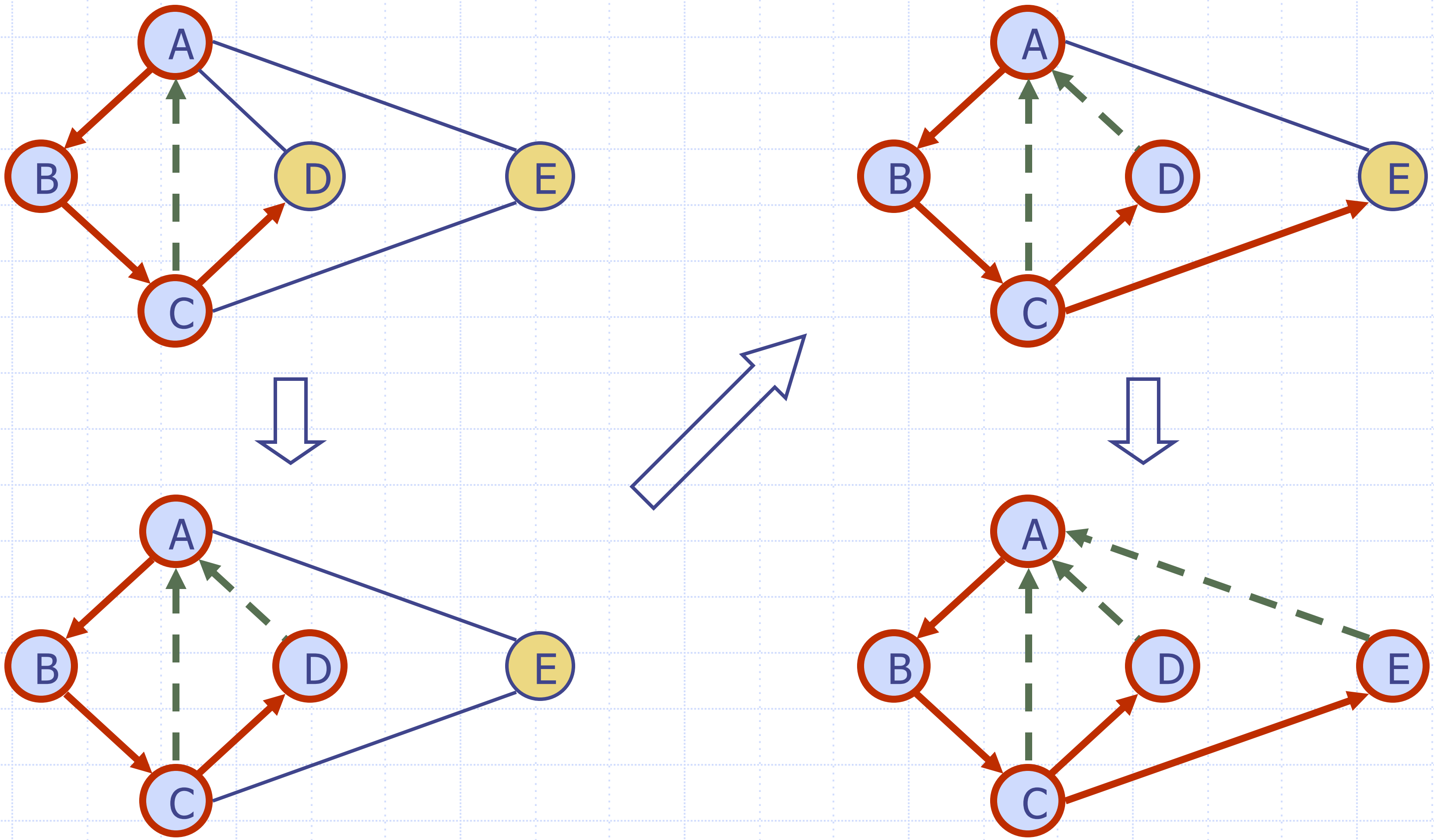
Example

- unexplored vertex
- visited vertex
- unexplored edge
- discovery edge
- back edge

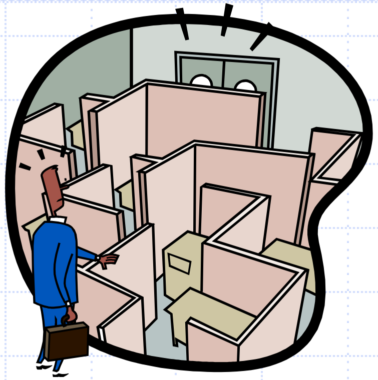




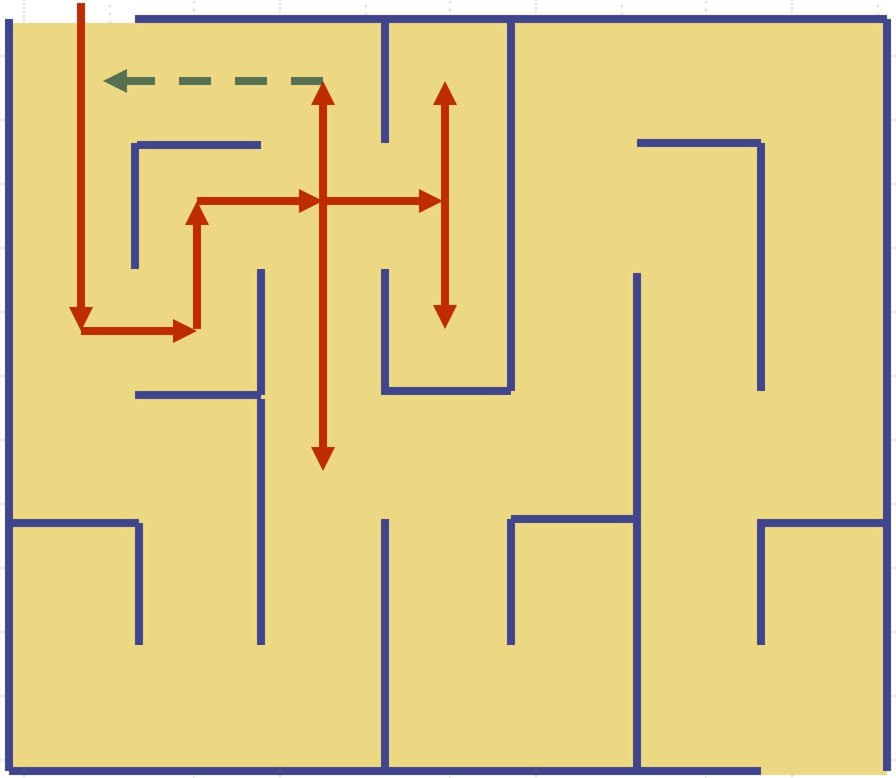
Example (cont.)



DFS and Maze Traversal



- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



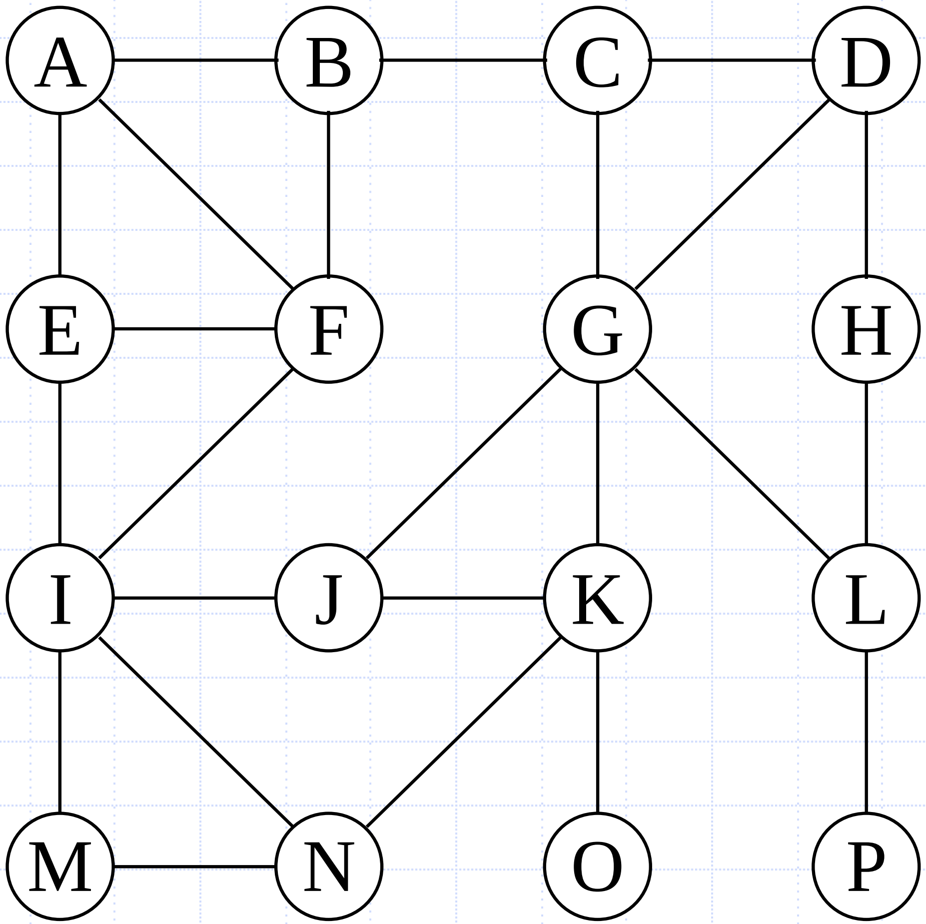
DFS for an Entire Graph

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

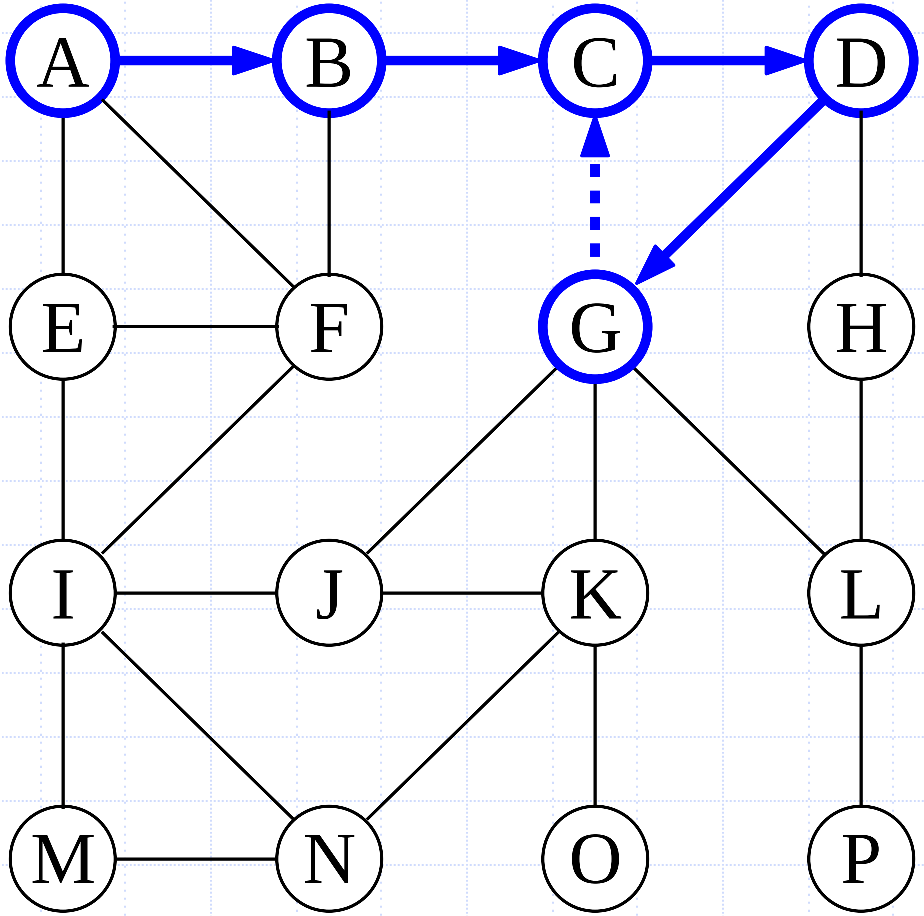
```
Algorithm DFS(G)
  Input graph G
  Output labeling of the edges of G
      as discovery edges and
      back edges
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      DFS(G, v)
```

```
Algorithm DFS(G, v)
  Input graph G and a start vertex v of G
  Output labeling of the edges of G
      in the connected component of v
      as discovery edges and back edges
  setLabel(v, VISITED)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        DFS(G, w)
      else
        setLabel(e, BACK)
```

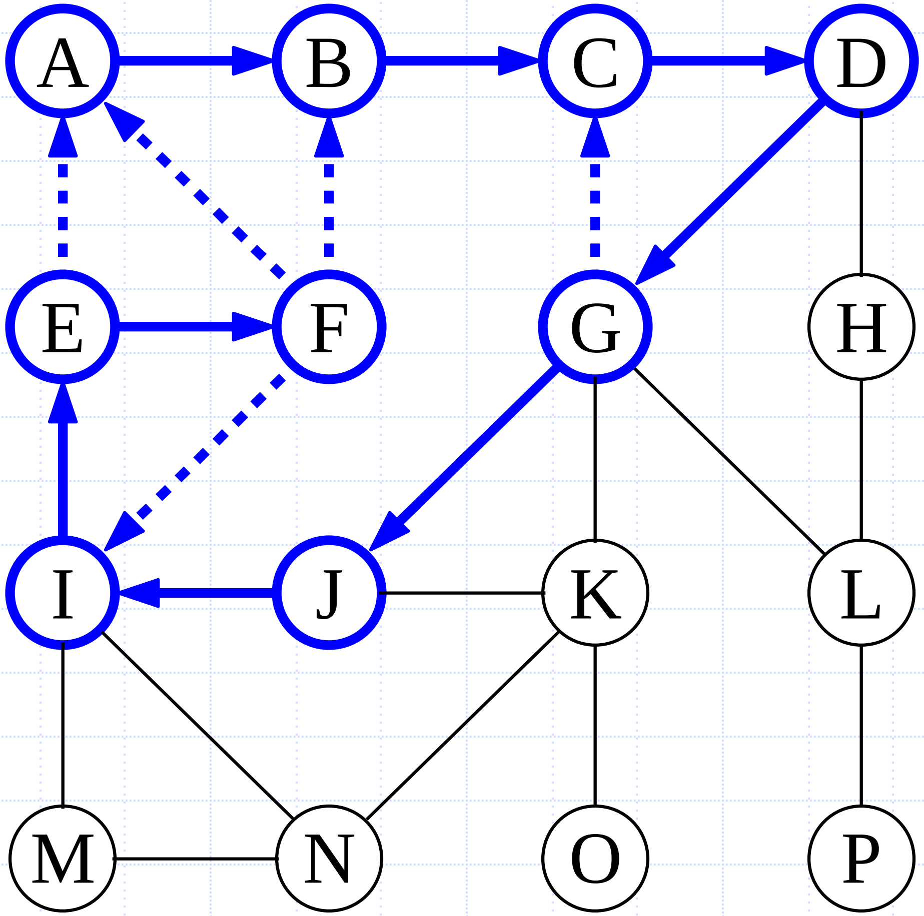
Example



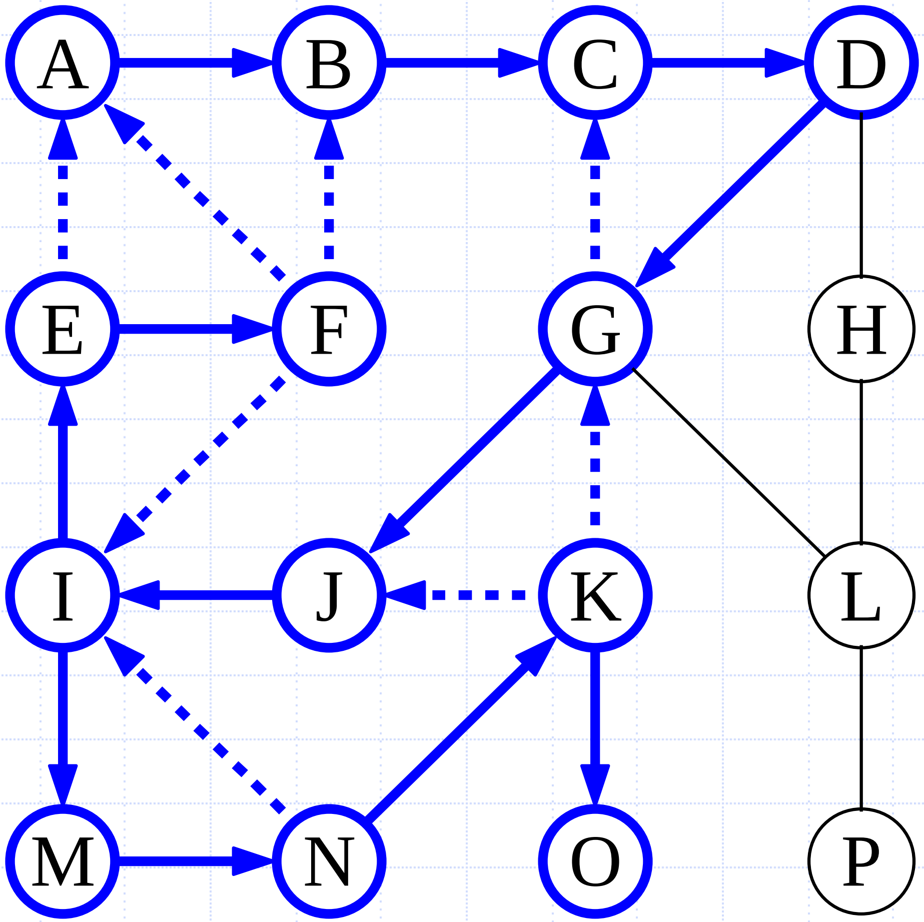
Example



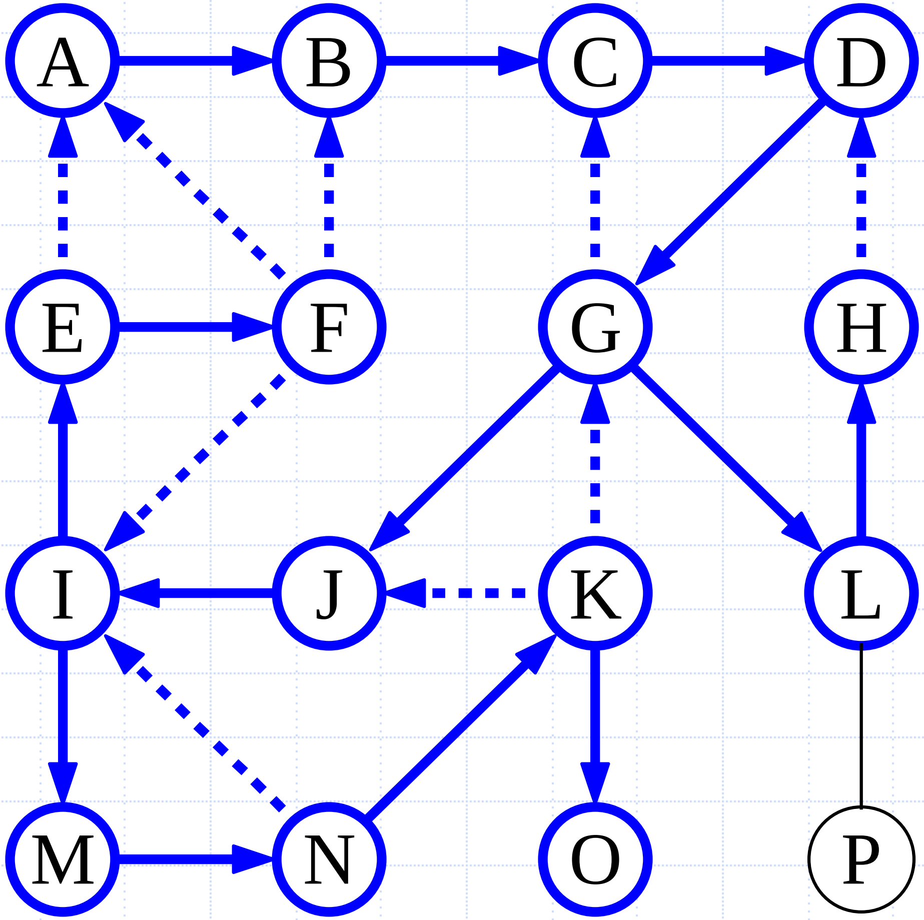
Example



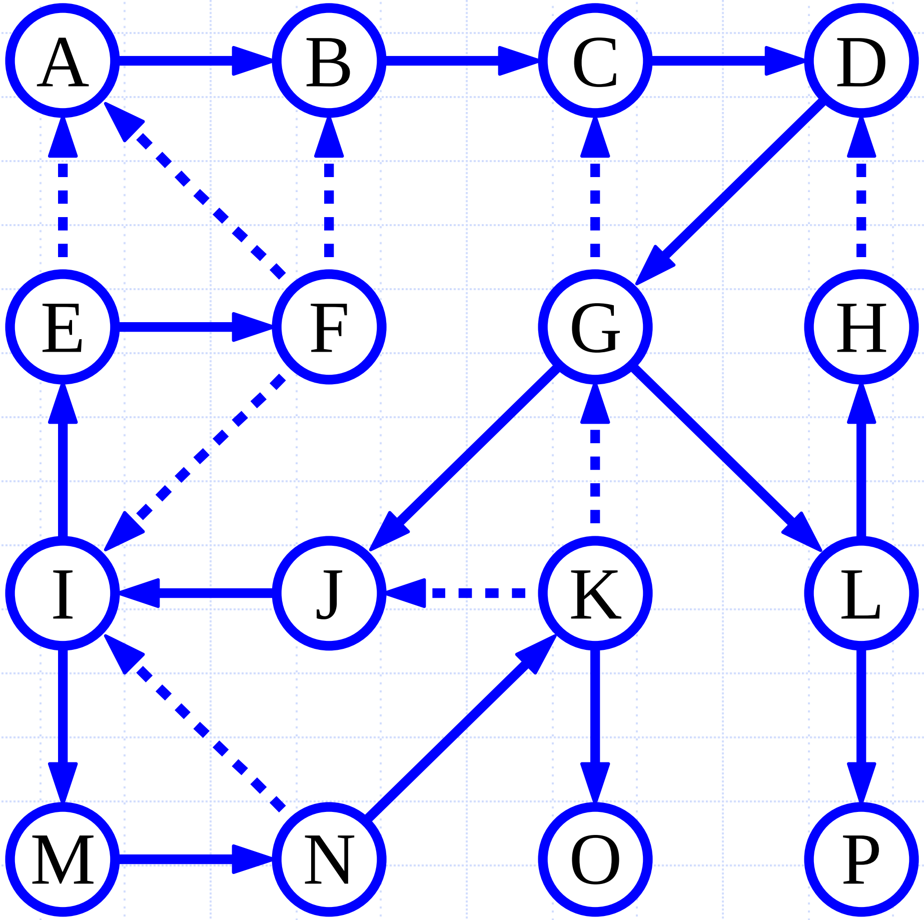
Example



Example



Example



Analysis of DFS

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

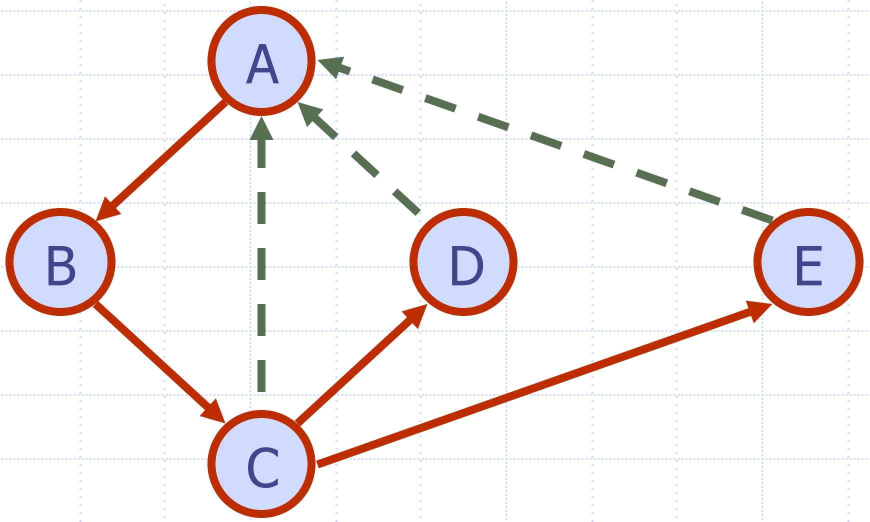
Properties of DFS

Property 1

$DFS(G, v)$ visits all the vertices and edges in the connected component of v

Property 2

The discovery edges labeled by $DFS(G, v)$ form a spanning tree of the connected component of v



Applications - Path Finding

- We can specialize the DFS algorithm to find a path between two given vertices u and z
- We call $DFS(G, u)$ with u as the start vertex
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex z is encountered, we return the path as the contents of the stack

```
Algorithm pathDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
    if v = z
        return S.elements()
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                S.push(e)
                pathDFS(G, w, z)
                S.pop(e)
            else
                setLabel(e, BACK)
    S.pop(v)
```

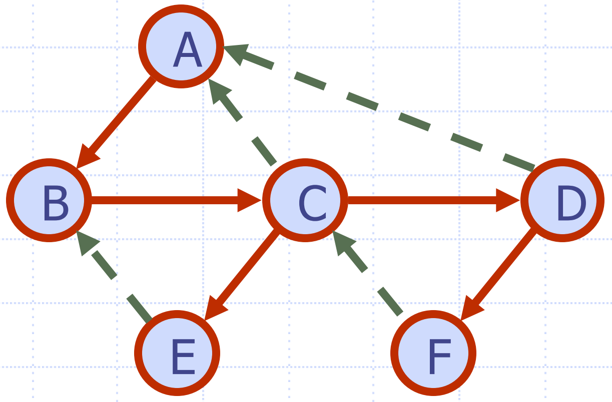
Applications - Cycle Finding

- We can specialize the DFS algorithm to find a simple cycle.
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

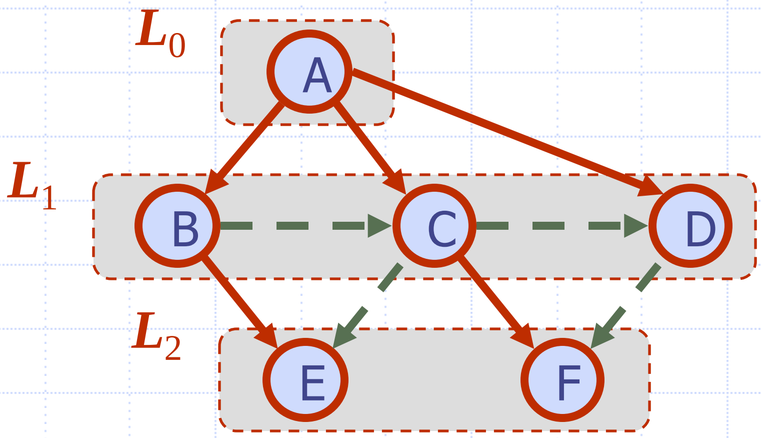
```
Algorithm cycleDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  for all  e ∈ G.incidentEdges(v)
    if  getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      S.push(e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        pathDFS(G, w, z)
        S.pop(e)
      else
        T ← new empty stack
        repeat
          o ← S.pop()
          T.push(o)
        until o = w
        return T.elements()
  S.pop(v)
```

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	



DFS

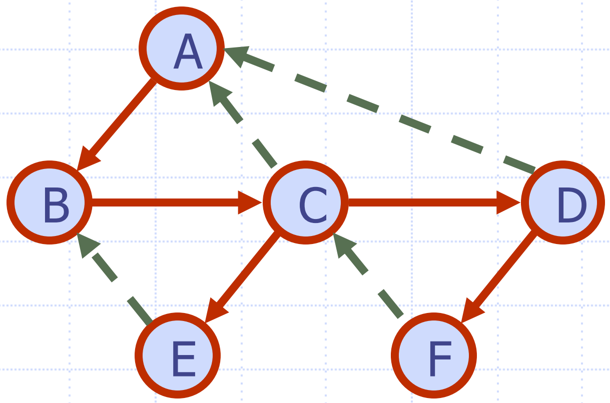


BFS

DFS vs. BFS (cont.)

Back edge (v,w)

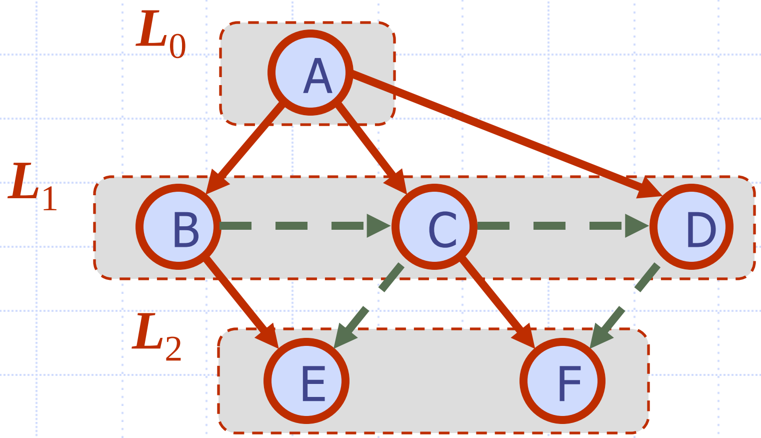
- w is an ancestor of v in the tree of discovery edges



DFS

Cross edge (v,w)

- w is in the same level as v or in the next level



BFS