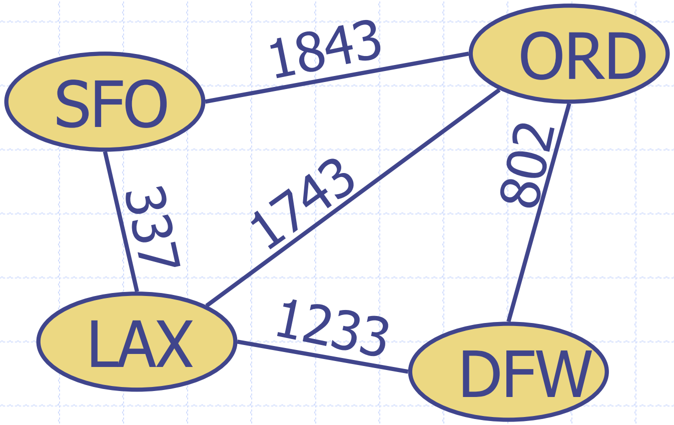
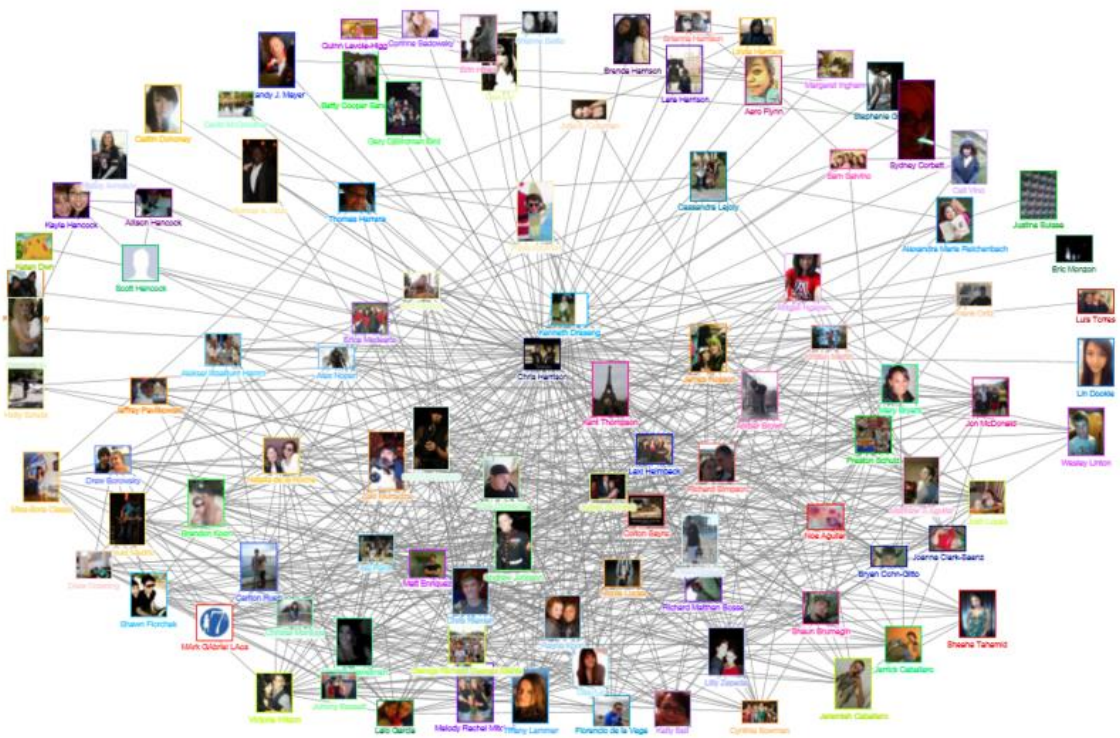
Graphs: Basics



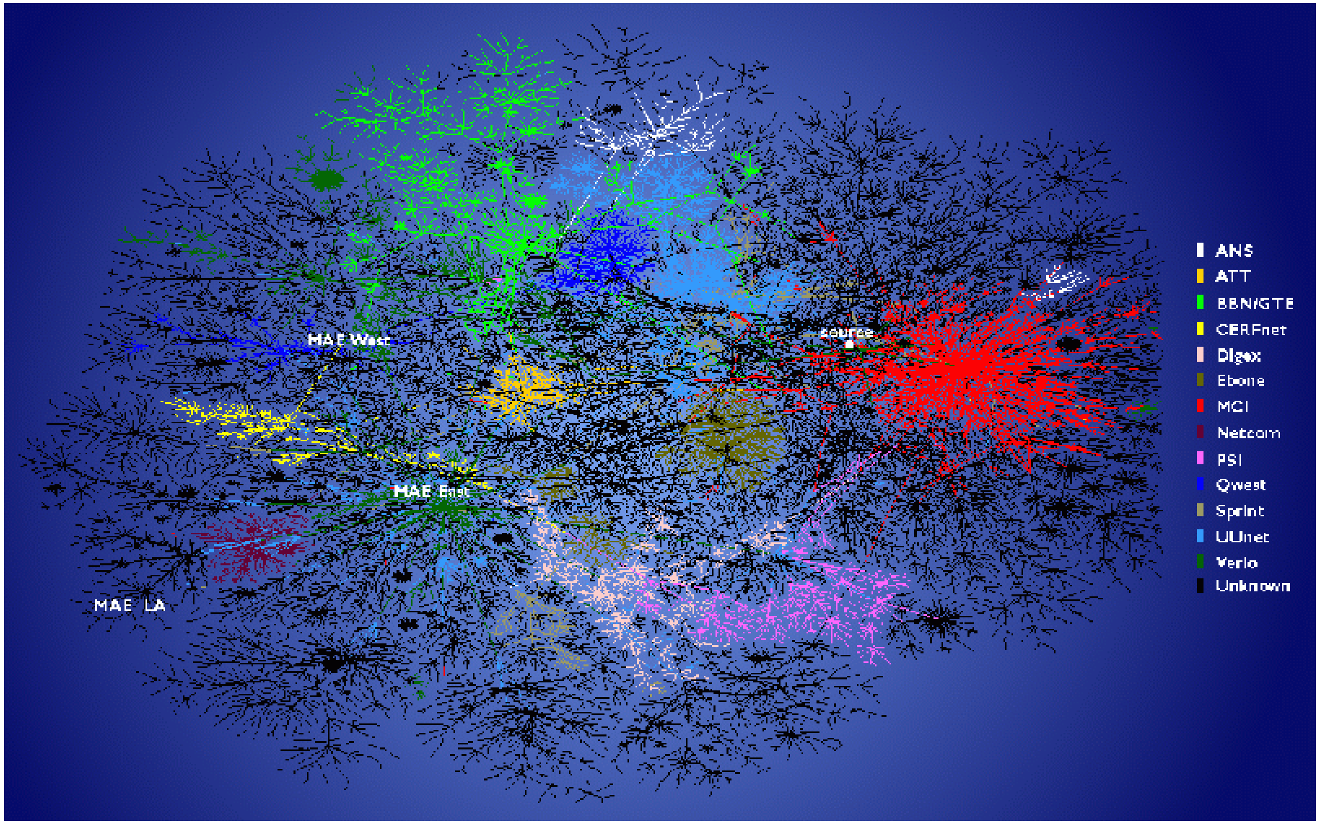
Real Life Examples

- On-line/Off-line Social Network



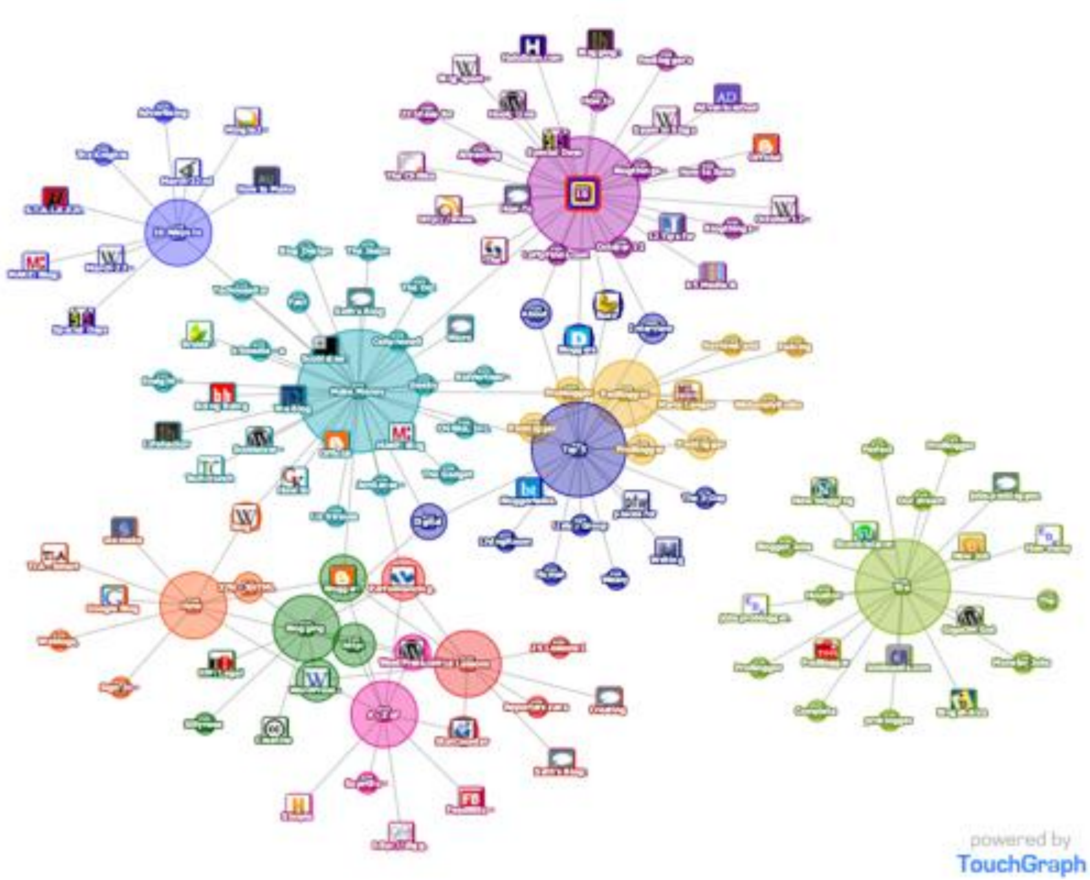
Real Life Examples

- Internet Connectivity



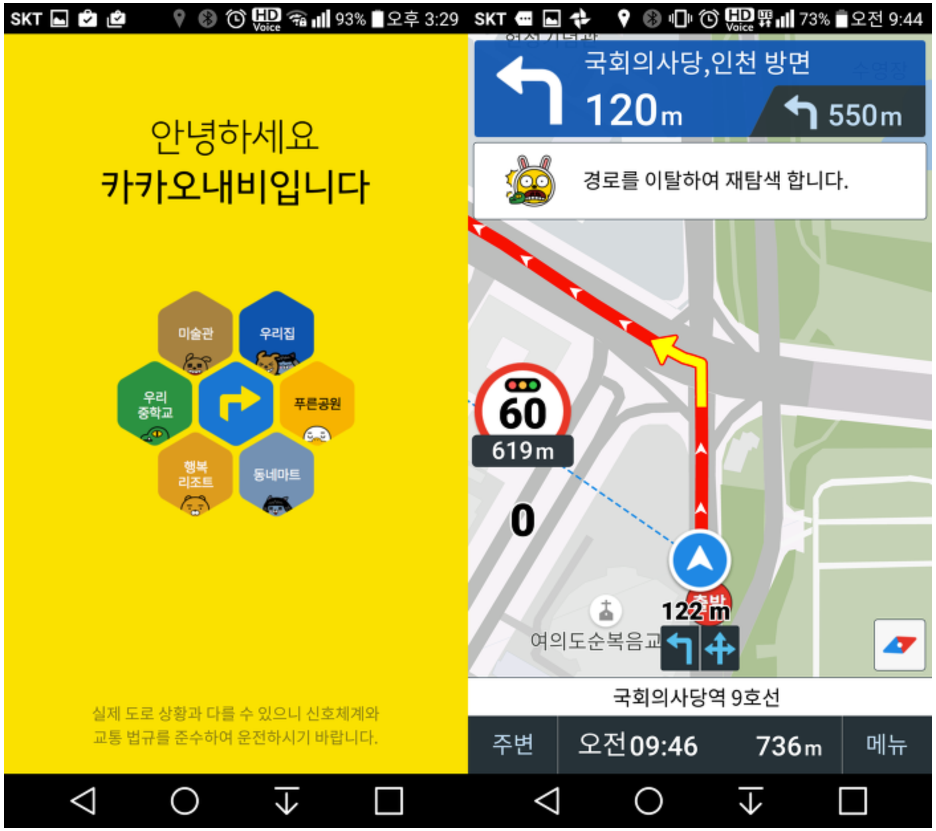
Real Life Examples

- WebBlog Connections



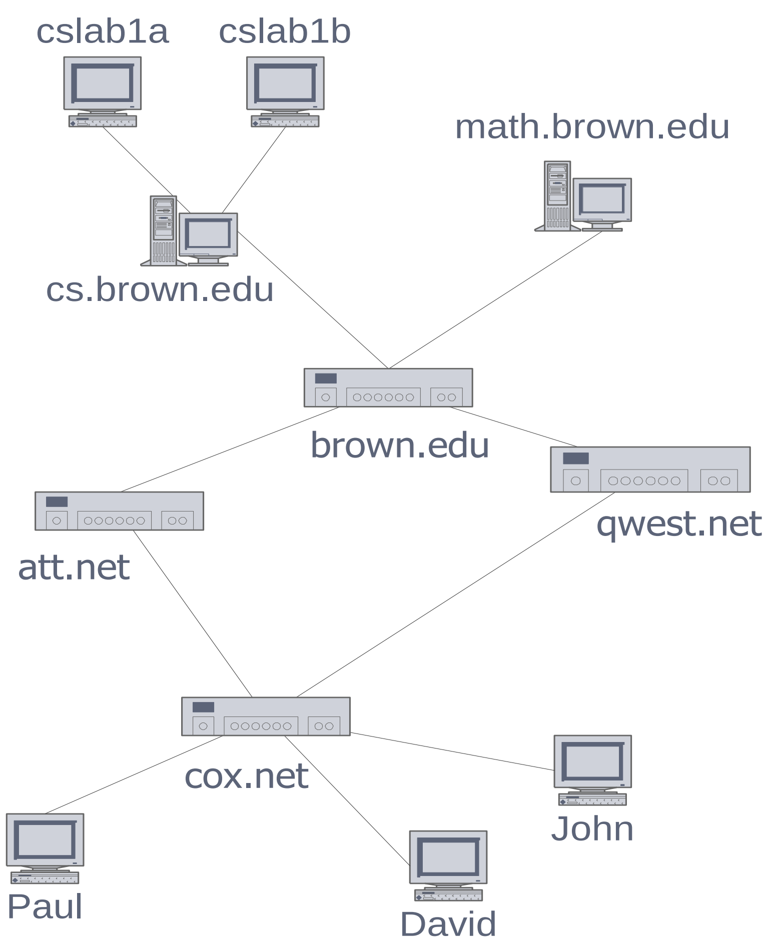
Real Life Examples

◆ Navigator



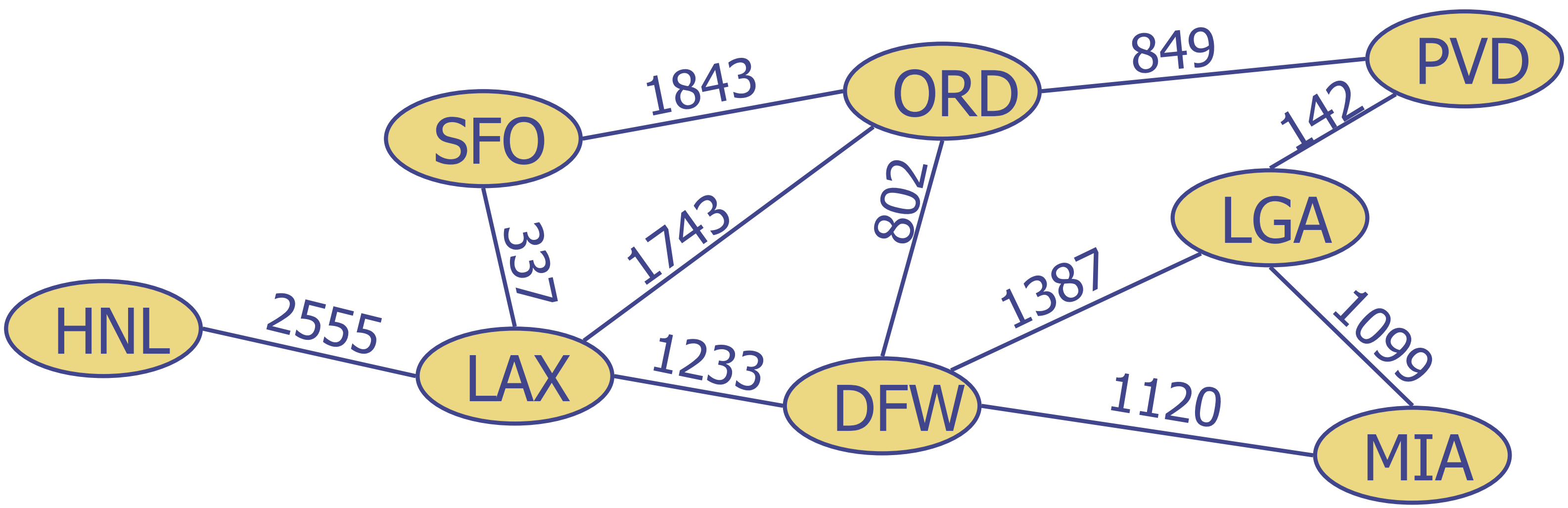
Other Applications

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



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

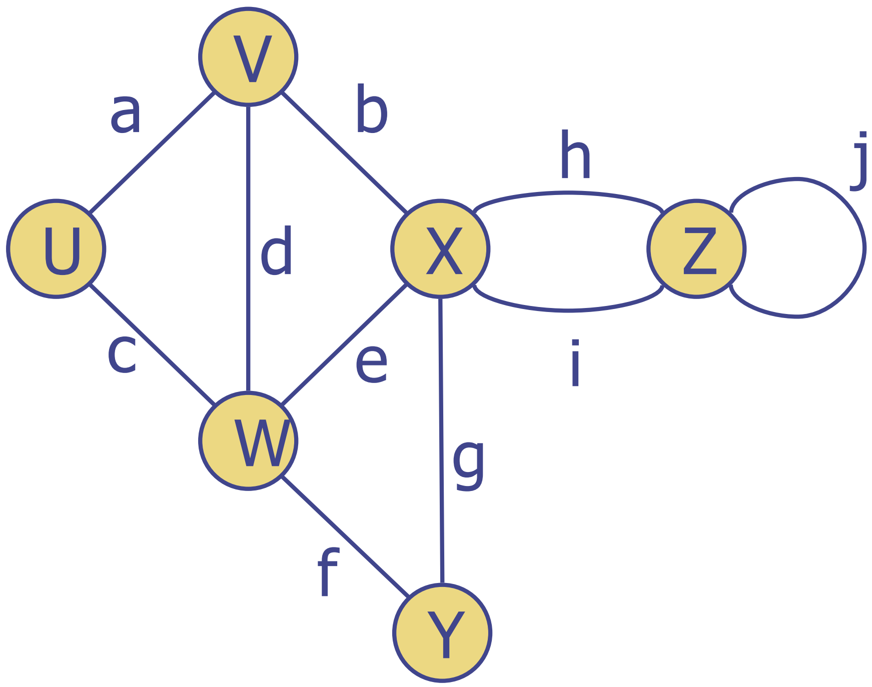


Edge Types

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

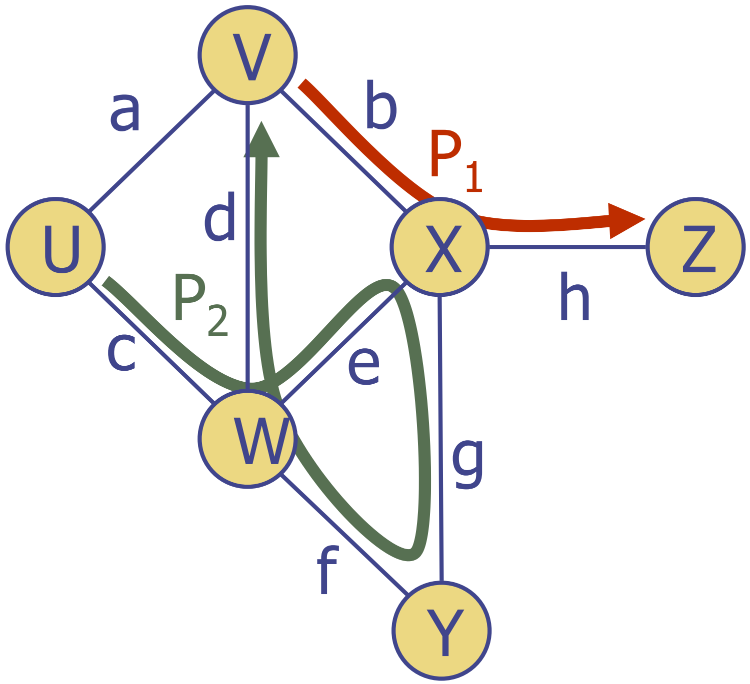
Terminology

- 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
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree 5
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop



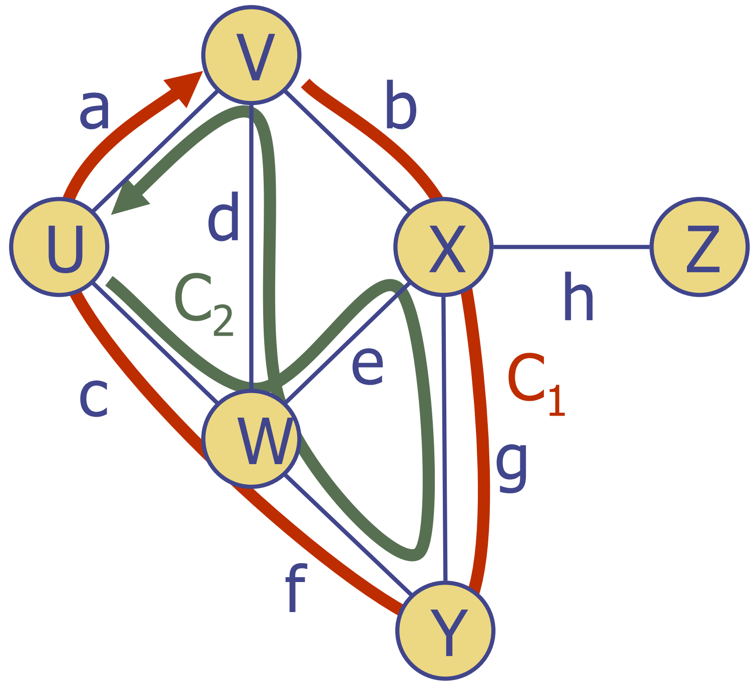
Terminology (cont.)

- 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 (cont.)

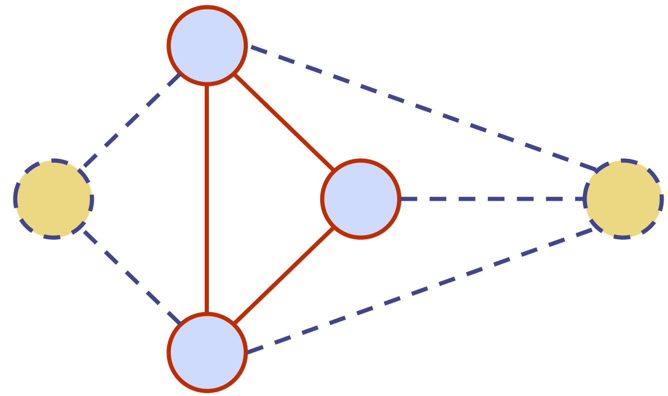
- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle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



- Note) Tree is a graph without cycles

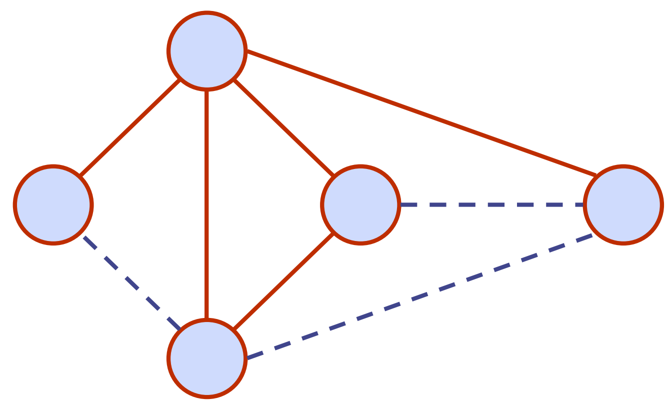
Subgraphs

- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G



Subgraph

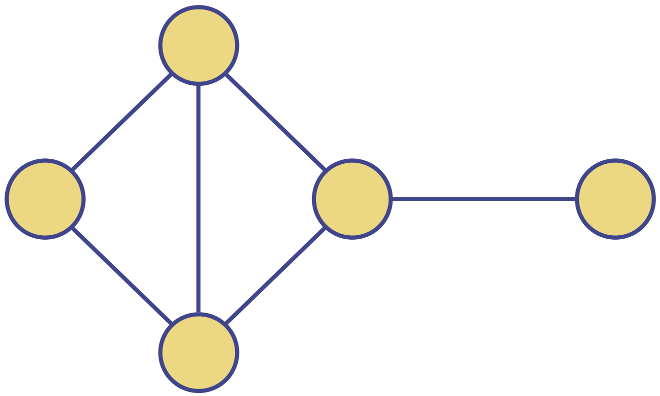
- A spanning subgraph of G is a subgraph that contains all the vertices of G



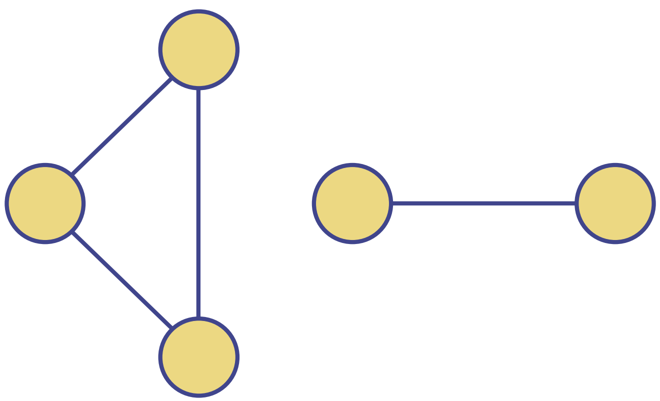
Spanning subgraph

Connectivity

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G
- “Maximal”?



Connected graph

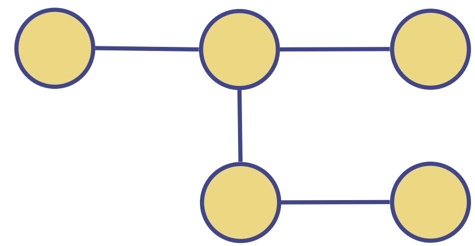


Non connected graph with two connected components

Trees and Forests

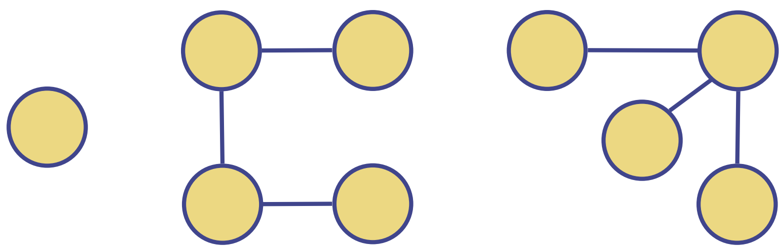
- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles

  This definition of tree is different from the one of a rooted tree



Tree

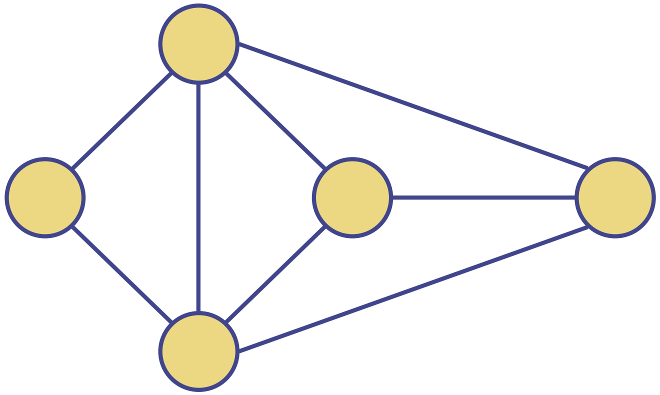
- A forest is an undirected graph without cycles
- The connected components of a forest are trees



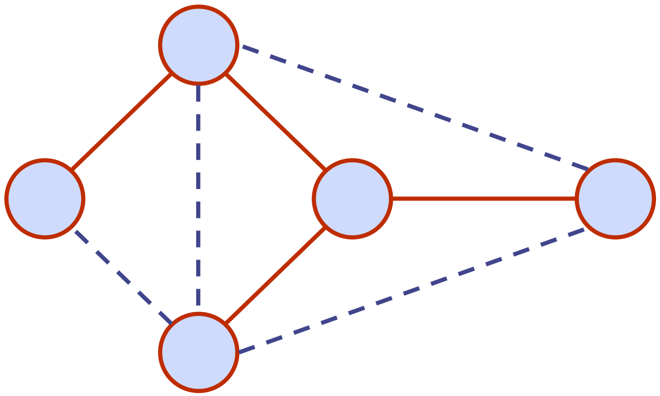
Forest

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest



Graph



Spanning tree

Some Properties for Undirected Graphs

Property 1

$\Sigma_v \deg(v) = 2m$

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges

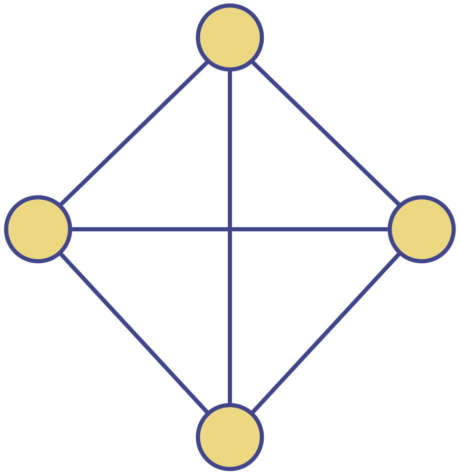
$m \leq n\,(n - 1)/2$

Proof: each vertex has degree at most $(n - 1)$

What is the bound for a directed graph?

Notation

n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v



Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

Main Methods of the Graph ADT

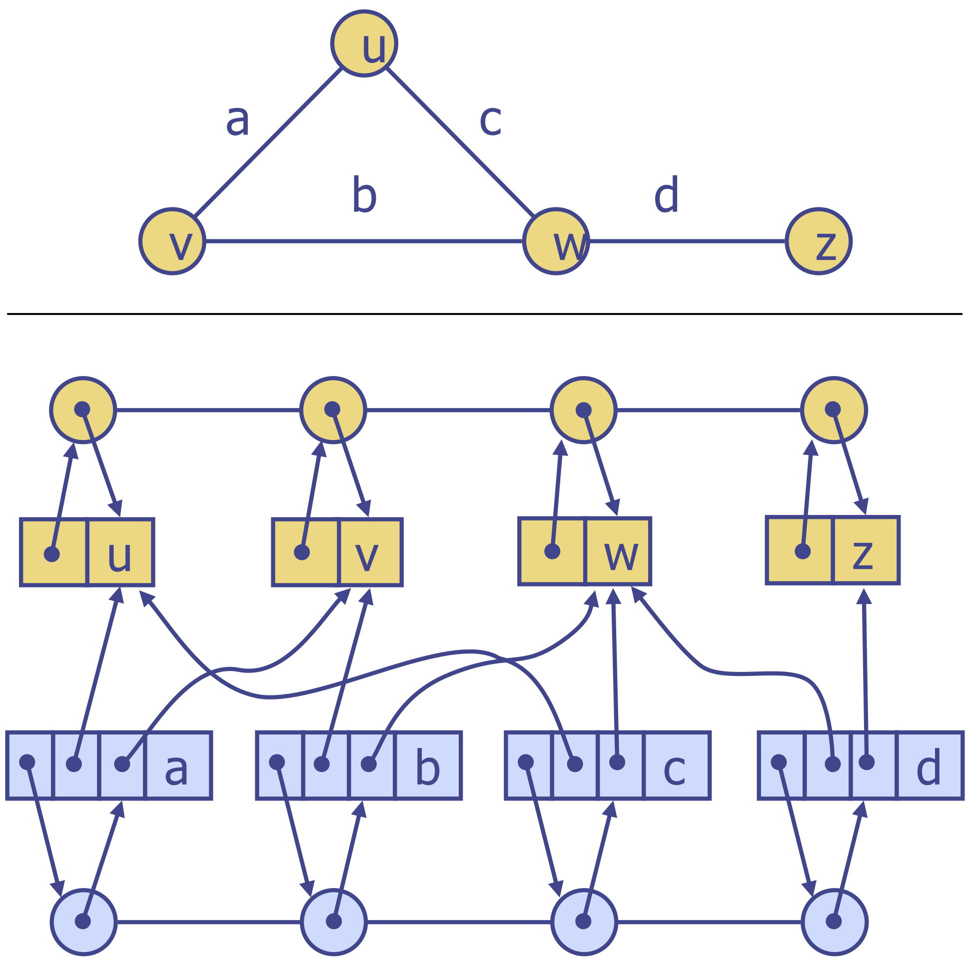
- Vertices and edges
  - are positions
  - store elements
- Accessor methods
  - e.endVertices(): a list of the two endvertices of e
  - e.opposite(v): the vertex opposite of v on e
  - u.isAdjacentTo(v): true iff u and v are adjacent
  - *v: reference to element associated with vertex v
  - *e: reference to element associated with edge e
- Update methods
  - insertVertex(o): insert a vertex storing element o
  - insertEdge(v, w, o): insert an edge (v,w) storing element o
  - eraseVertex(v): remove vertex v (and its incident edges)
  - eraseEdge(e): remove edge e
- Iterable collection methods
  - incidentEdges(v): list of edges incident to v
  - vertices(): list of all vertices in the graph
  - edges(): list of all edges in the graph

What is a data structure to represent a graph?

We will discuss three ways

1. 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 (e.g., list)
  - sequence of vertex objects
- Edge sequence (e.g., list)
  - sequence of edge objects



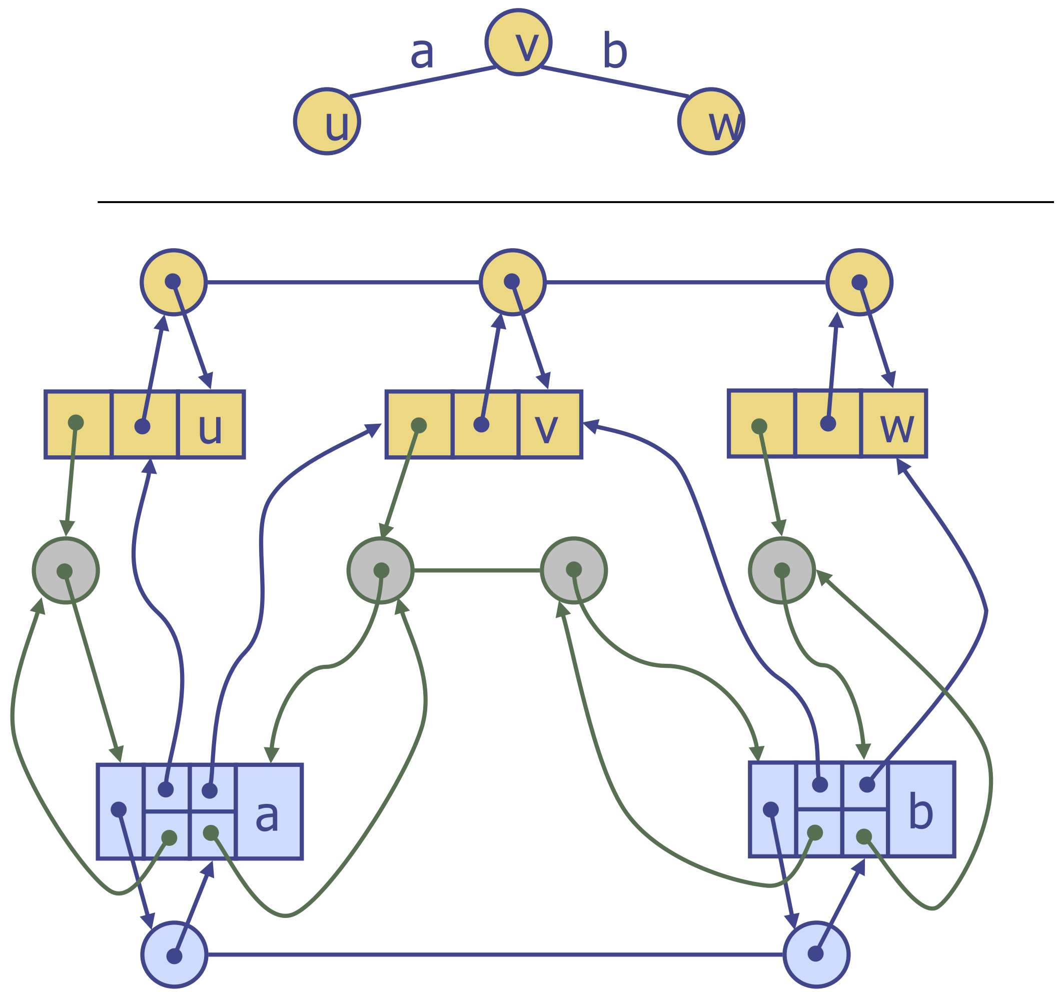
Performance

▪ *n* vertices, *m* edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
v.incidentEdges()	m	deg(*v*)	n
u.isAdjacentTo (*v*)	m	min(deg(*v*), deg(*w*))	1
insertVertex(*o*)	1	1	n^2
insertEdge(*v*, *w*, *o*)	1	1	1
eraseVertex(*v*)	m	deg(*v*)	n^2
eraseEdge(*e*)	1	1	1

- v.incidentEdges() and u.isAdjacneTo(v)
  - Need to check all the edges

2. Adjacency List Structure

- Basic: Edge list structure
- Supports direct access to the incident edges from a node
  - Incidence edge sequence for each vertex
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices
- Provides direct access
  - From the edges to the vertices
  - From the vertices to their incident edges



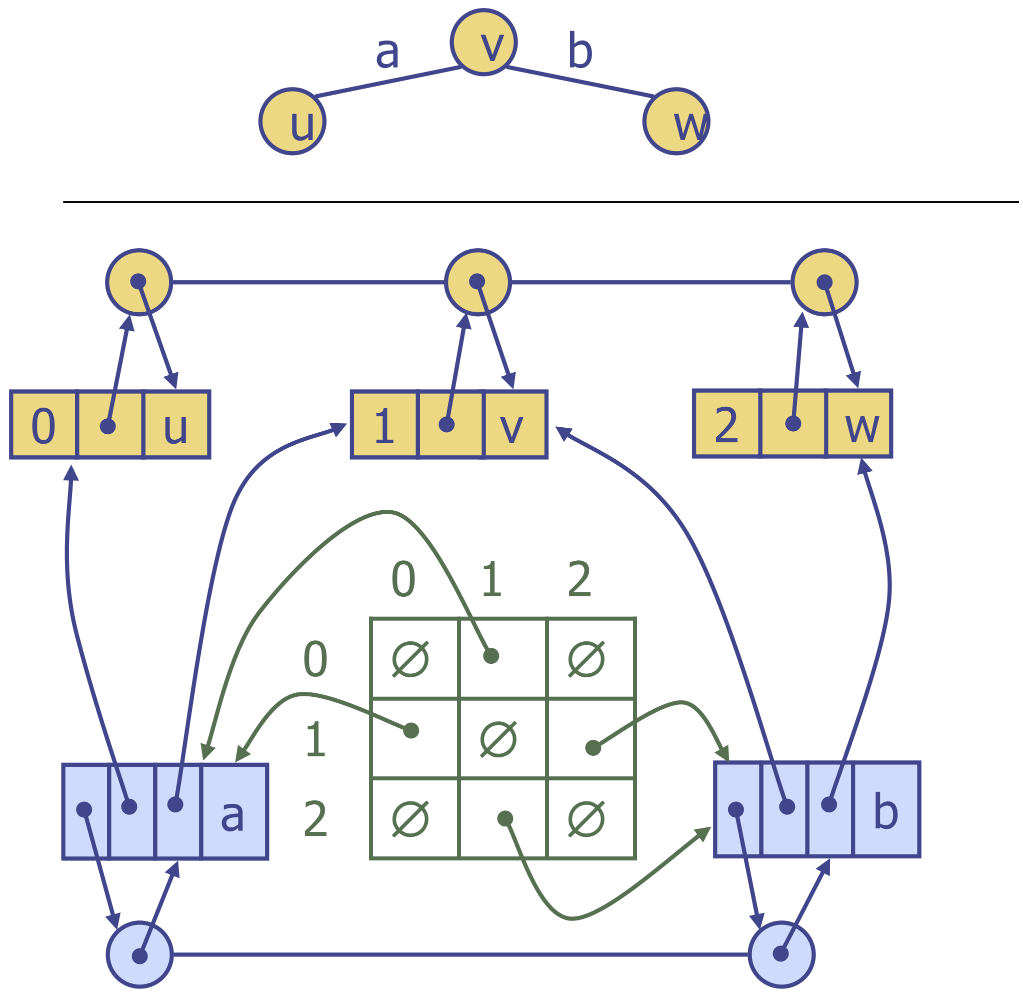
Performance

▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
v.incidentEdges()	m	deg(v)	n
u.isAdjacentTo (v)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
eraseVertex(v)	m	size of adjacency list deg(v)	n^2
eraseEdge(e)	1	1	1

- v.incidentEdges(): direct access to incident edges
- u.isAdjacentTo(v):

3. 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



Performance

▪ *n* vertices, *m* edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
v.incidentEdges()	m	deg(*v*)	n
u.isAdjacentTo (*v*)	m	min(deg(*v*), deg(*w*))	1
insertVertex(*o*)	1	1	n^2
insertEdge(*v, w, o*)	1	1	1
eraseVertex(*v*)	m	deg(*v*)	n^2
eraseEdge(*e*)	1	1	1

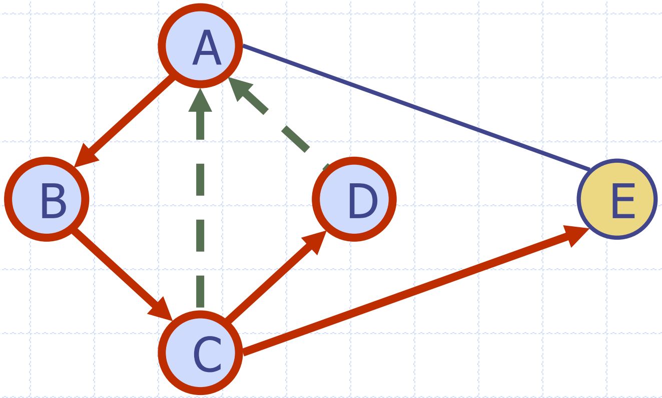
- v.incidentEdges(): matrix row check
- u.isAdjacentTo(v): using v’s key

Performance

▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
v.incidentEdges()	m	deg(v)	n
u.isAdjacentTo (***v***)	m	min(deg(v), deg(w))	1
insertVertex(***o***)	1	1	n^2
insertEdge(***v, w, o***)	1	1	1
eraseVertex(***v***)	m	deg(v)	n^2
eraseEdge(***e***)	1	1	1

- v.incidentEdges(): direct access to incident edges
- u.isAdjacentTo(v):

Depth-First Search

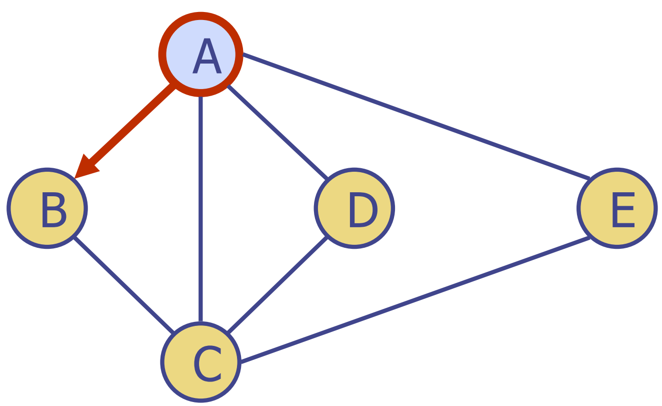


Depth-First Search

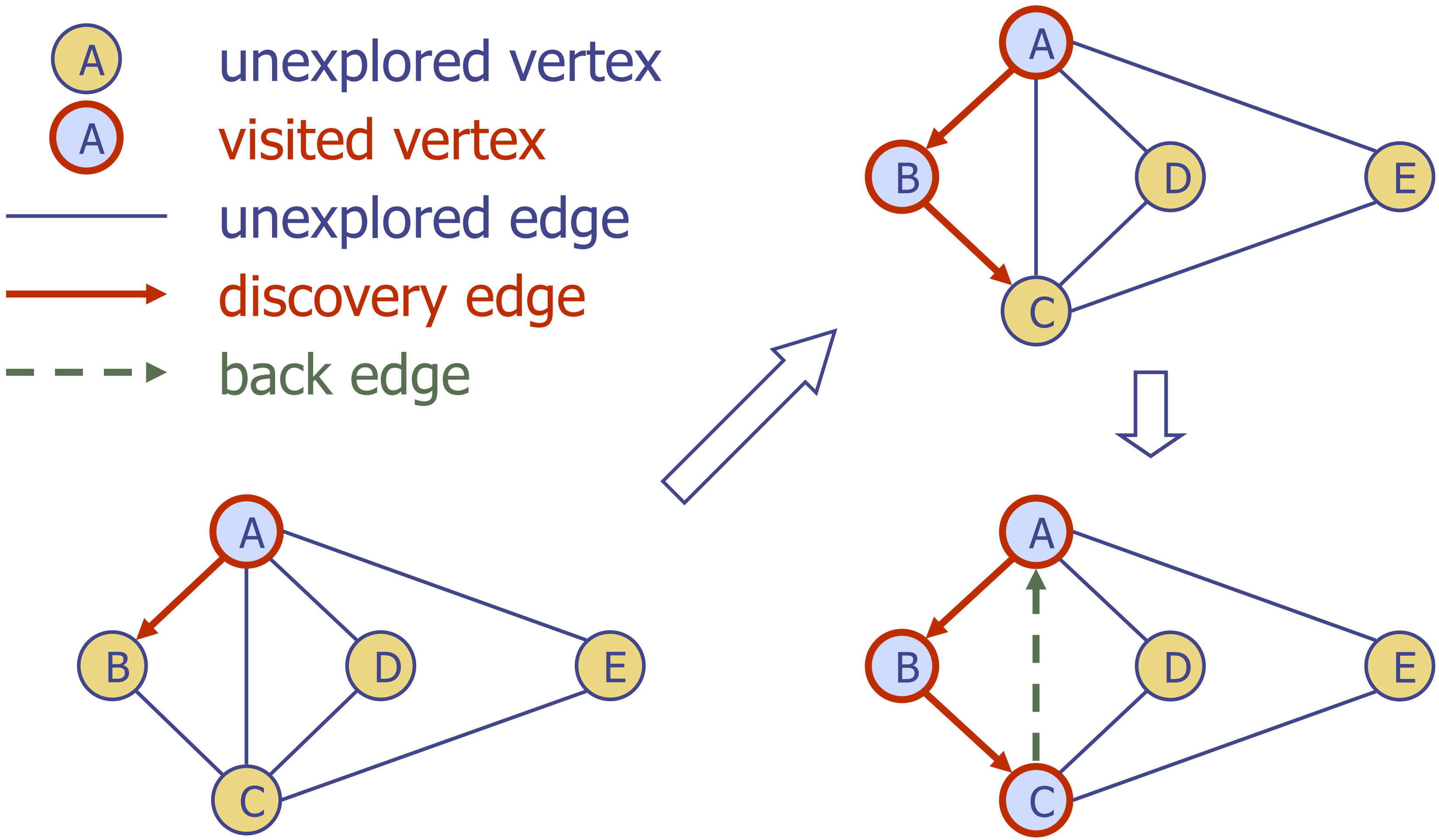
- Depth-first search (DFS) is a general technique for traversing a graph

- Why is this traversal important?

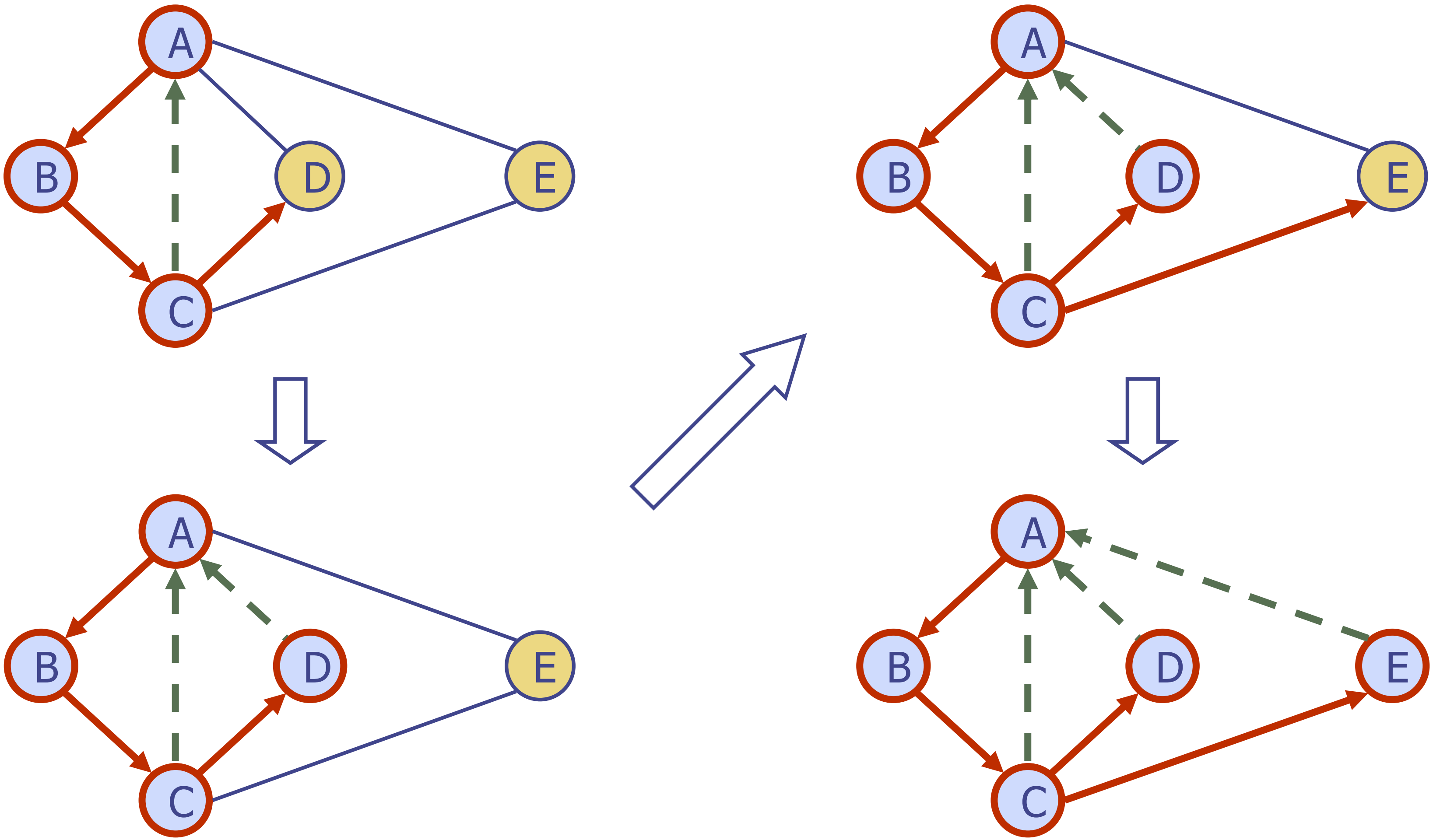
- Let’s first see the example



Example


A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
back edge
A
B
C
D
E

Example (cont.)



One implication: discovery edges form a spanning tree.

Depth-First Search

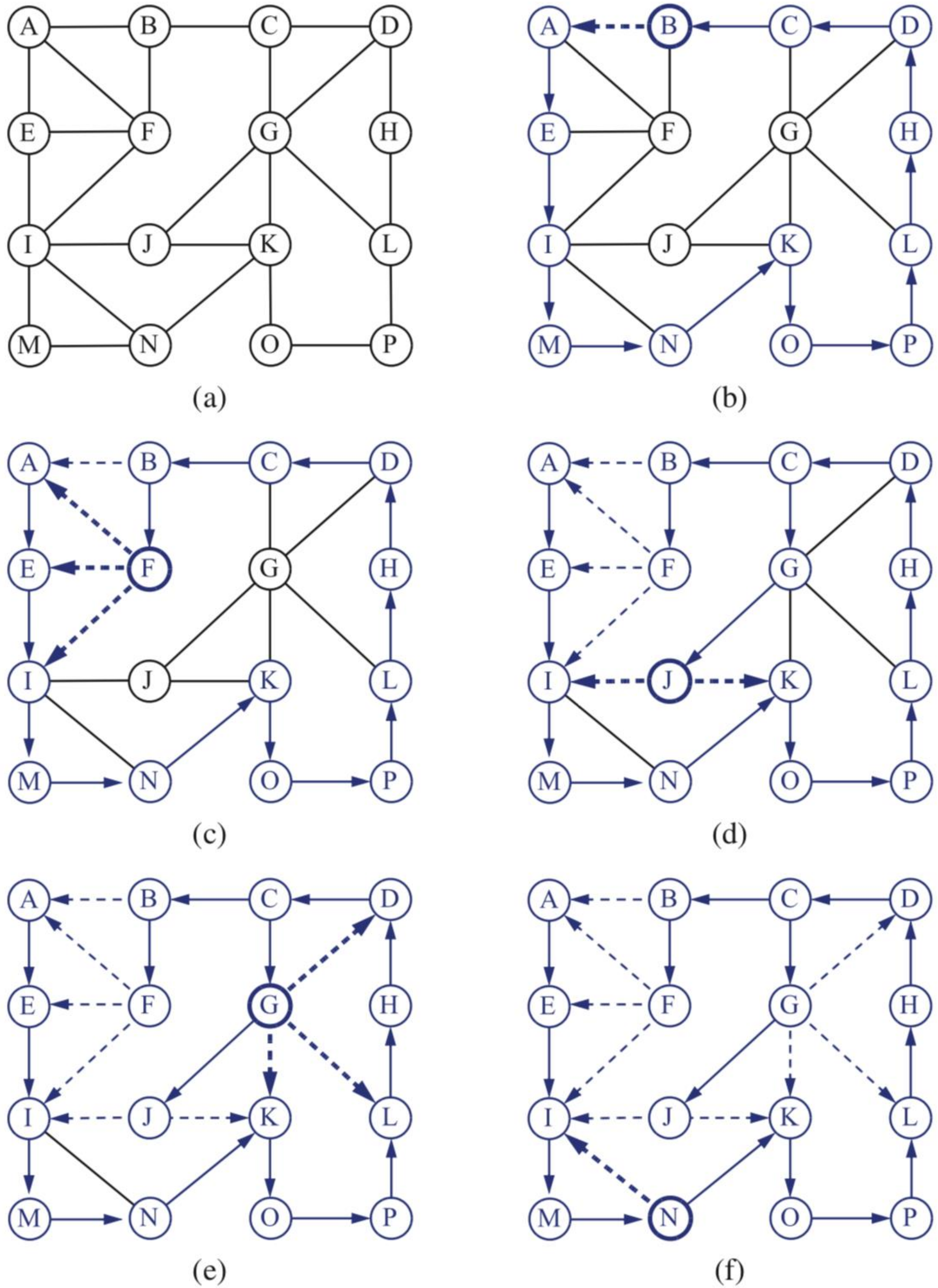
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected (how?)
  - Computes the connected components of G (how?)
  - Computes a spanning forest of G
- DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph

DFS Algorithm

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

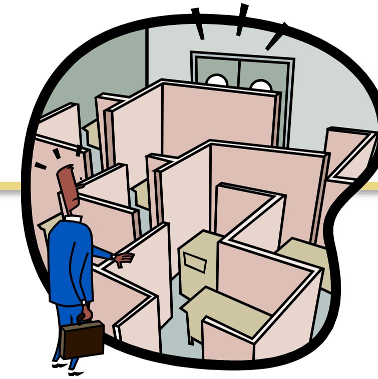
```
Algorithm DFS(G)
    Input graph G
    Output labeling of the edges of G
        as discovery edges and
        back edges
  for all u ∈ G.vertices()
    u.setLabel(UNEXPLORED)
  for all e ∈ G.edges()
    e.setLabel(UNEXPLORED)
  for all v ∈ G.vertices()
    if v.getLabel() = UNEXPLORED
        DFS(G,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

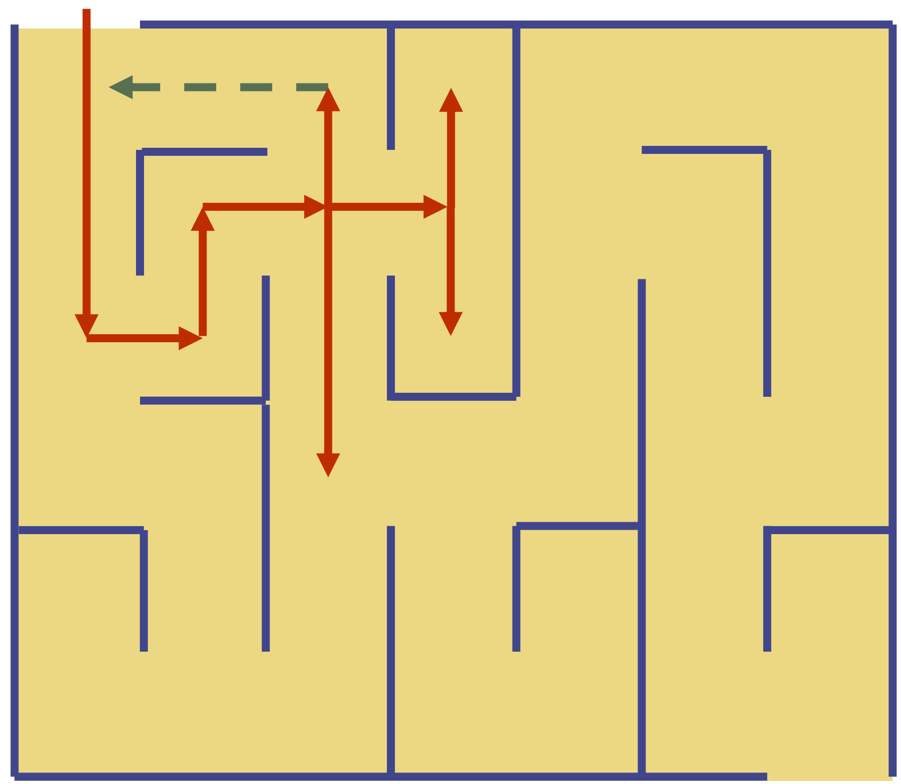


(a) (b) (c) (d) (e) (f)

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)



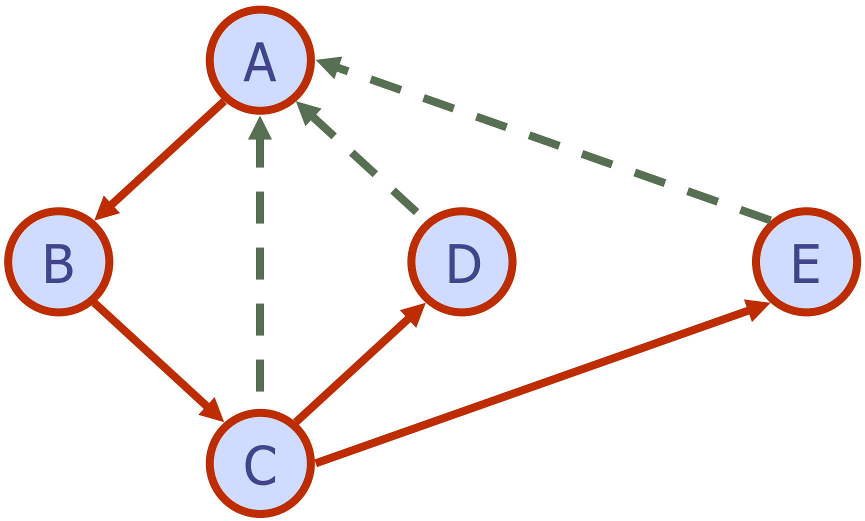
Properties of DFS

Property 1

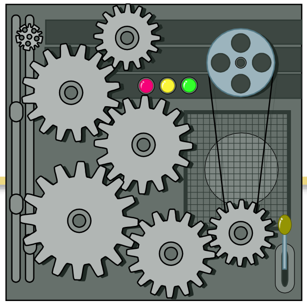
DFS(***G***, ***v***) visits all the vertices and edges in the connected component of ***v***

Property 2

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



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
  - Complexity of v.incidentEdges: deg(v)
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

Path Finding

- We can specialize the DFS algorithm to find a path between two given vertices u and z using the template method pattern
- 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

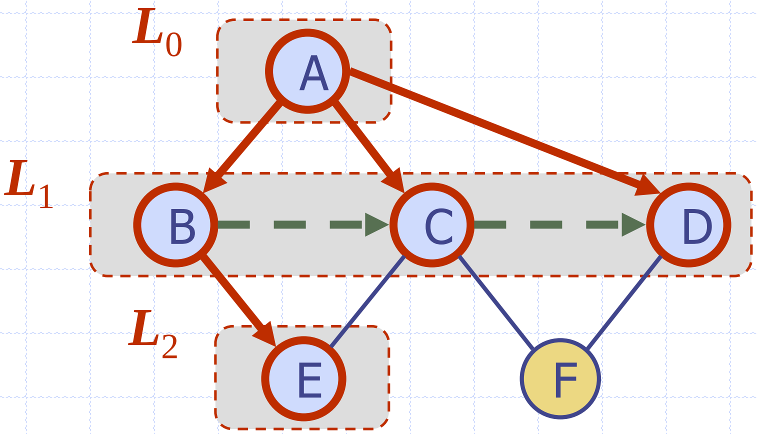
```
Algorithm pathDFS(G, v, z)
    v.setLabel(VISITED)
    S.push(v)
    if v = z
        return S.elements()
    for all e ∈ v.incidentEdges()
        if e.getLabel() = UNEXPLORED
            w ← e.opposite(v)
            if w.getLabel() = UNEXPLORED
                e.setLabel(DISCOVERY)
          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

Cycle Finding

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- 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)
    v.setLabel(VISITED)
    S.push(v)
    for all  e ∈ v.incidentEdges()
        if  e.getLabel() = UNEXPLORED
            w ← e.opposite(v)
            S.push(e)
            if w.getLabel() = UNEXPLORED
                e.setLabel(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)
```

Breadth-First Search



Breadth-First Search

- Breadth-first search (BFS) is another general technique for traversing a graph

- Let's look at the example

Example

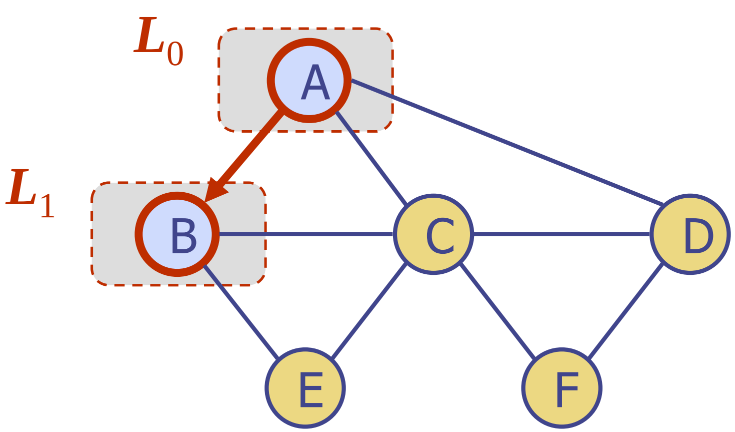
A unexplored vertex

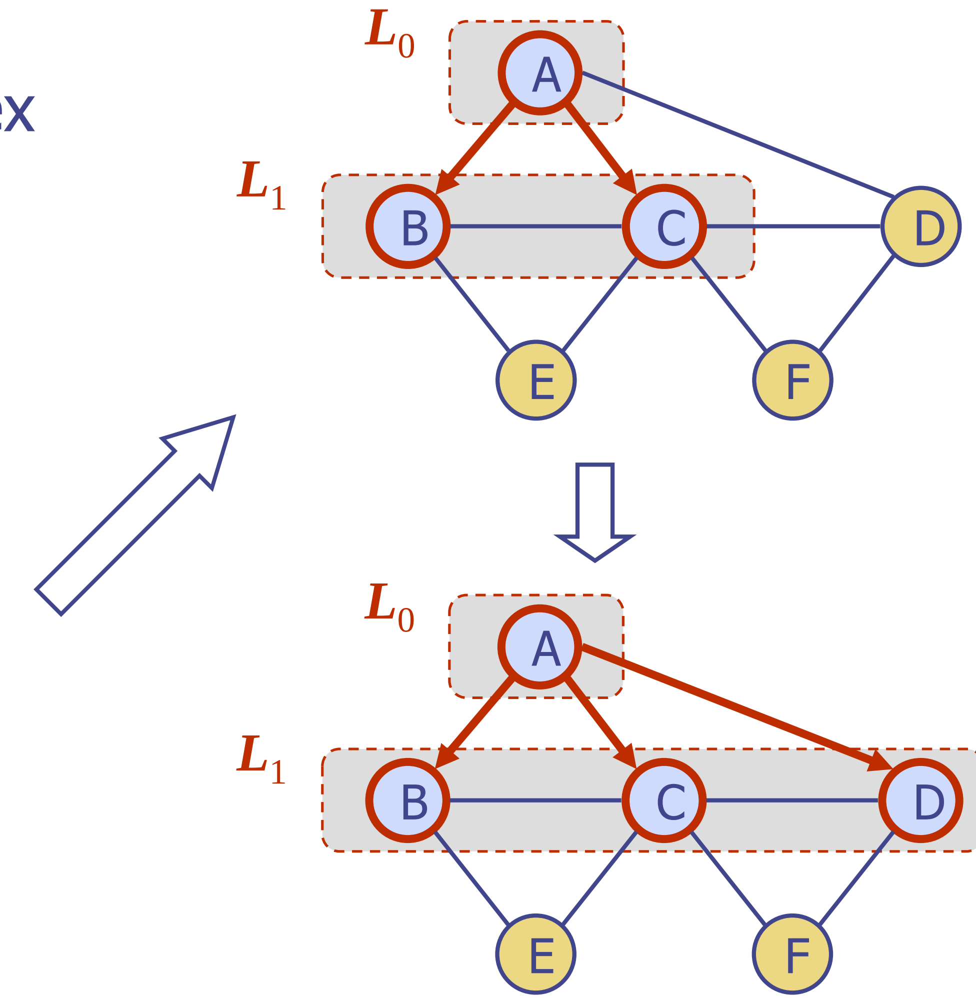
A visited vertex

unexplored edge

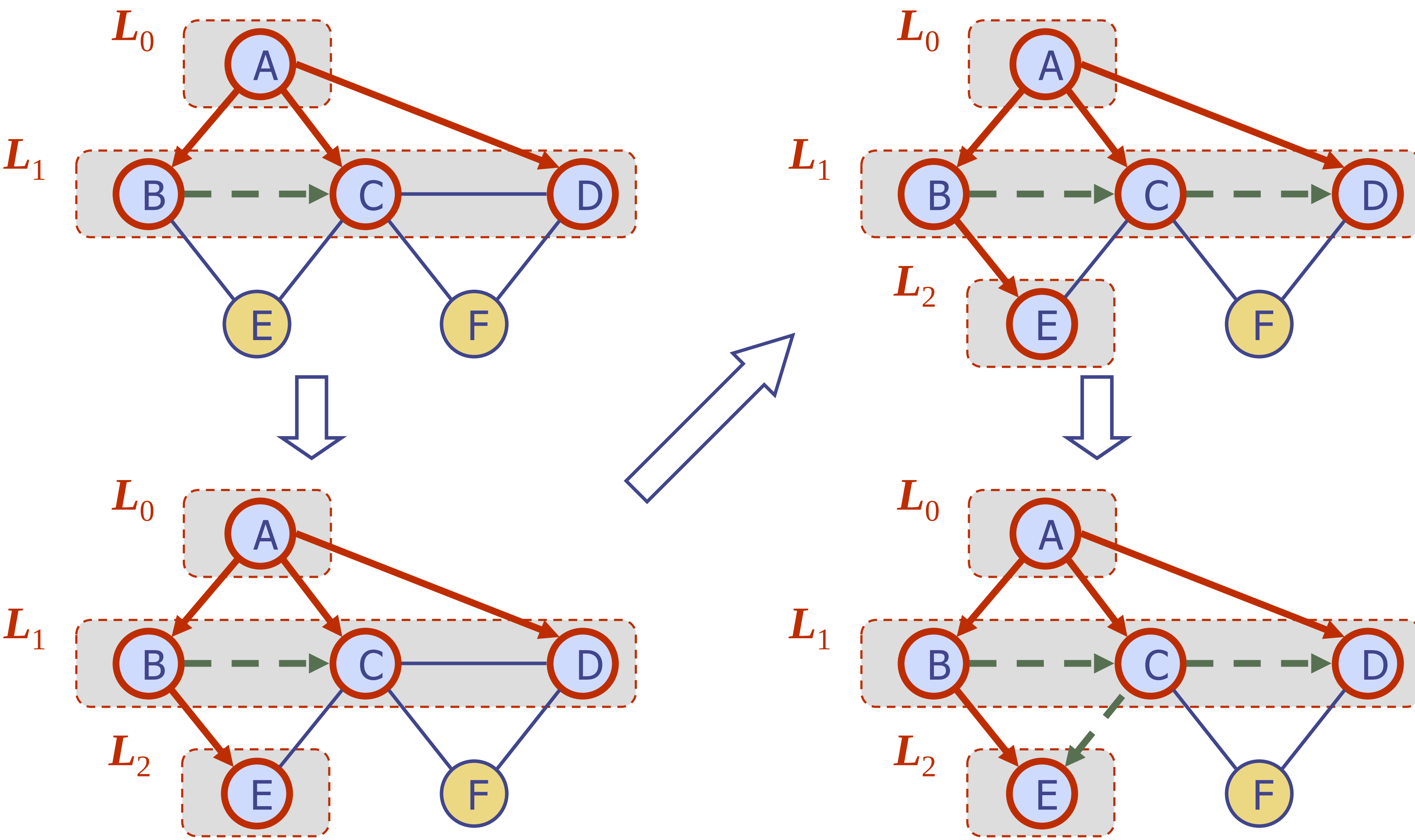
discovery edge

cross edge

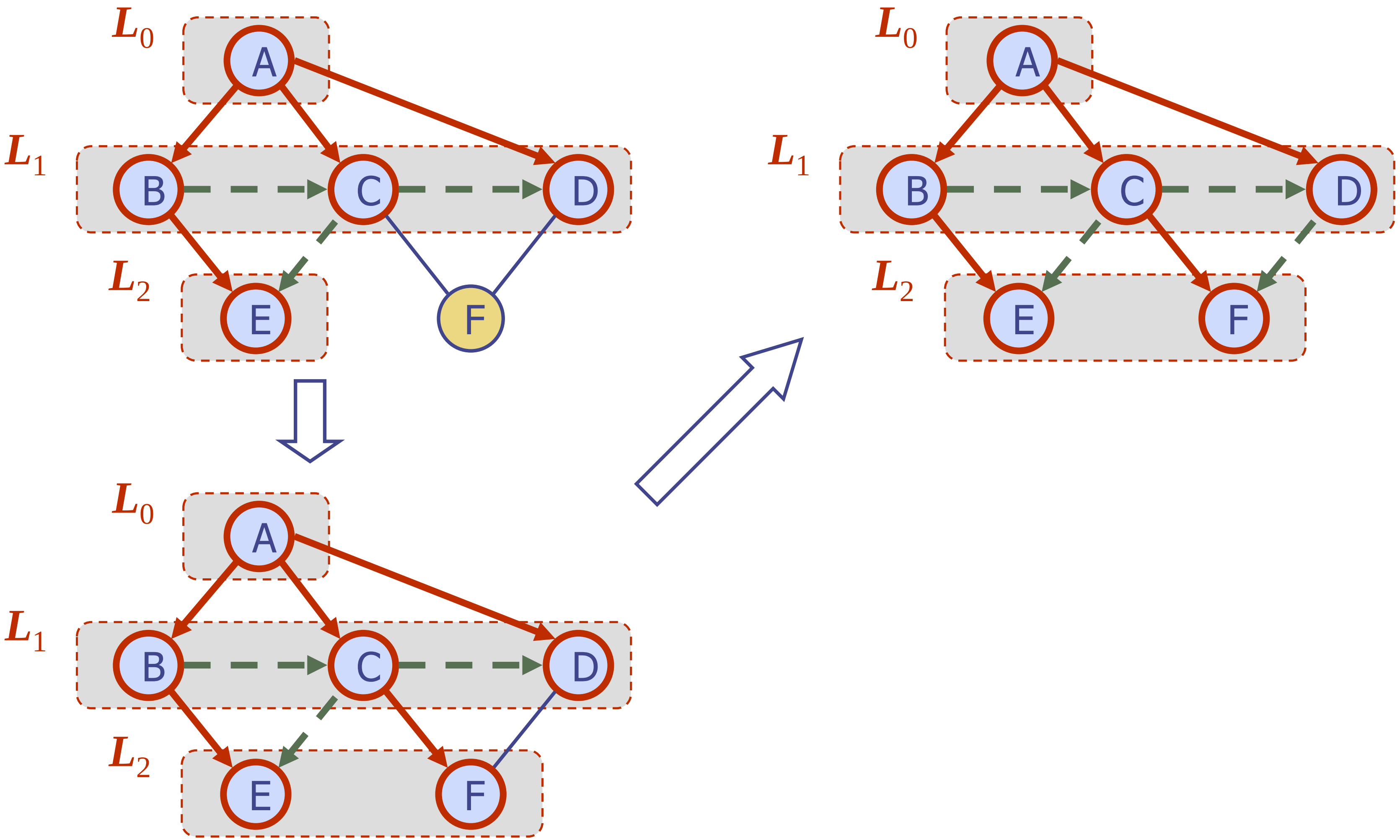




Example (cont.)



Example (cont.)



Breadth-First Search

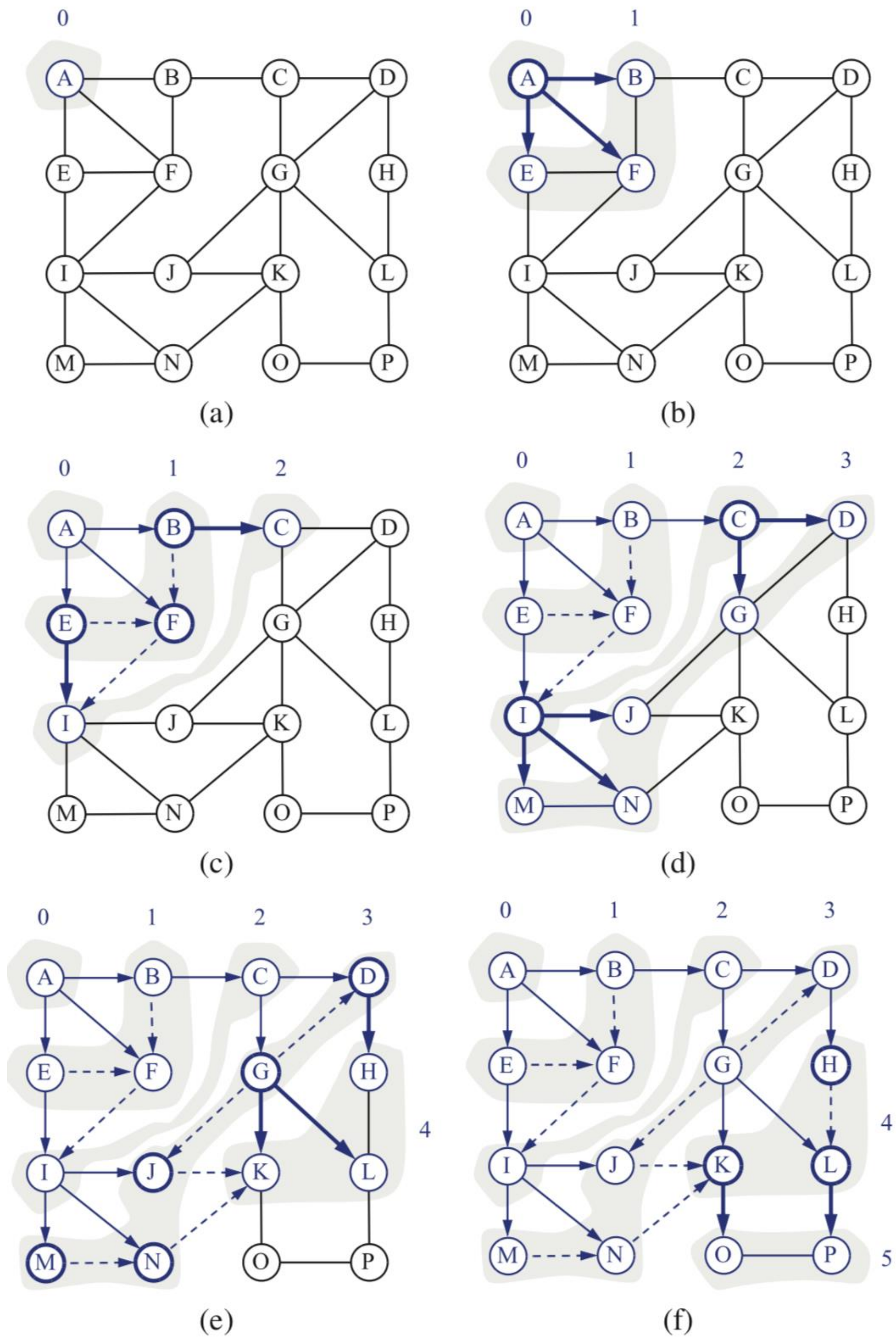
- A BFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- BFS on a graph with n vertices and m edges takes $O(n + m)$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Can label each vertex by the length of a shortest path (in terms of # of edges) from the start vertex s
  - Find a simple cycle, if there is one

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()
    u.setLabel(UNEXPLORED)
  for all  e ∈ G.edges()
    e.setLabel(UNEXPLORED)
  for all  v ∈ G.vertices()
    if  v.getLabel() = UNEXPLORED
        BFS(G, v)
```

```
Algorithm BFS(G, s)
  L0 ← new empty sequence
  L0.insertBack(s)
  s.setLabel(VISITED)
  i ← 0
  while ¬Li.empty()
    Li+1 ← new empty sequence
    for all  v ∈ Li.elements()
      for all  e ∈ v.incidentEdges()
        if  e.getLabel() = UNEXPLORED
          w ← e.opposite(v)
          if  w.getLabel() = UNEXPLORED
            e.setLabel(DISCOVERY)
            w.setLabel(VISITED)
            Li+1.insertBack(w)
          else
            e.setLabel(CROSS)
    i ← i +1
```


(a)
(b)
(c)
(d)
(e)
(f)

Properties

Notation

G_s: connected component of s

Property 1

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

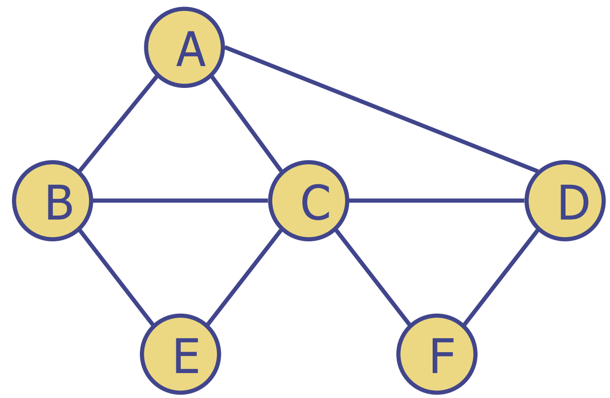
Property 2

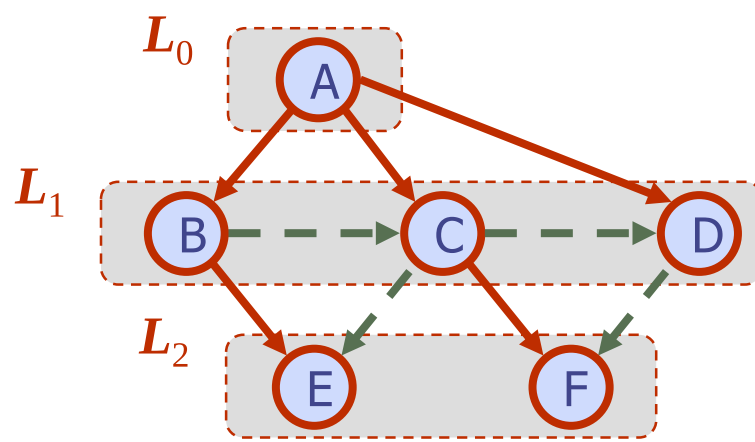
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

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 (i.e., find a shortest path)





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 $\sum_v \deg(v) = 2m$

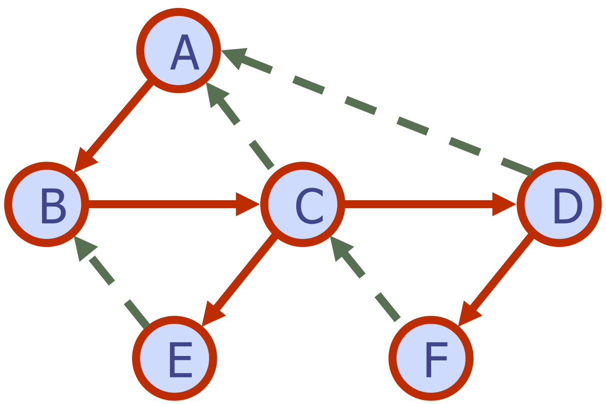
Applications

- Using the template method pattern, 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
  - Find a simple cycle in G, or report that G is a forest
  - 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

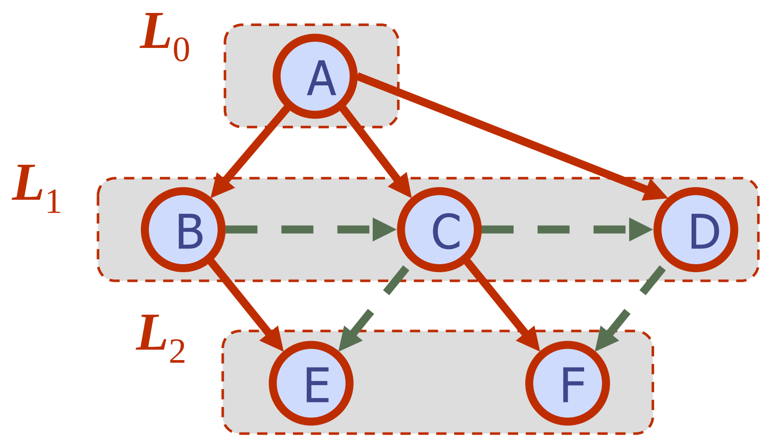
DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components (how?)	√	

Biconnected components:
- Connected
- Even after removing any vertex the graph remains connected



DFS



BFS

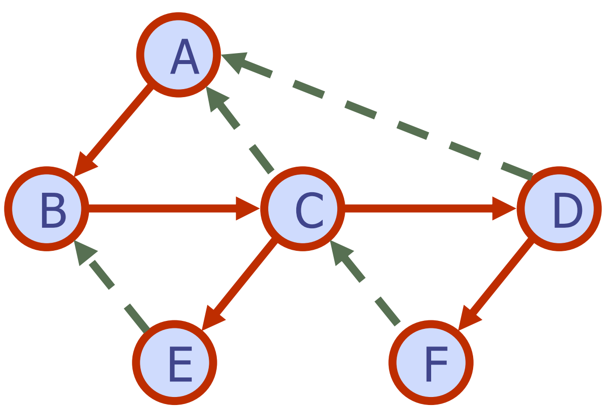
DFS vs. BFS (cont.)

Back edge (v,w)

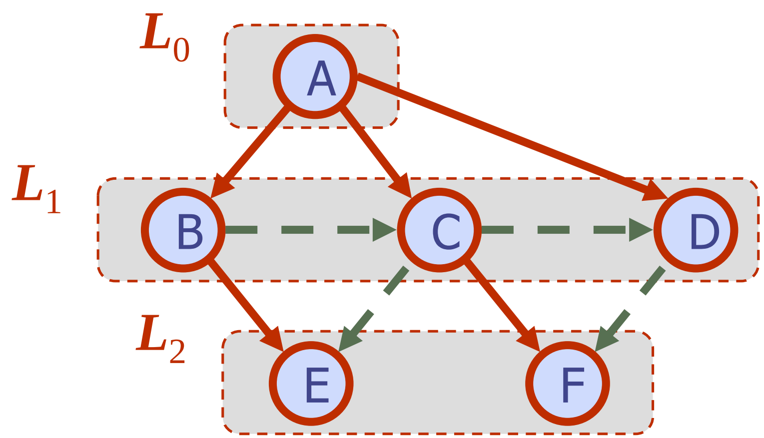
- w is an ancestor of v in the tree of discovery edges

Cross edge (v,w)

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



DFS



BFS

Questions?