Data Structure - Spring 2022

16. Graph: Basics

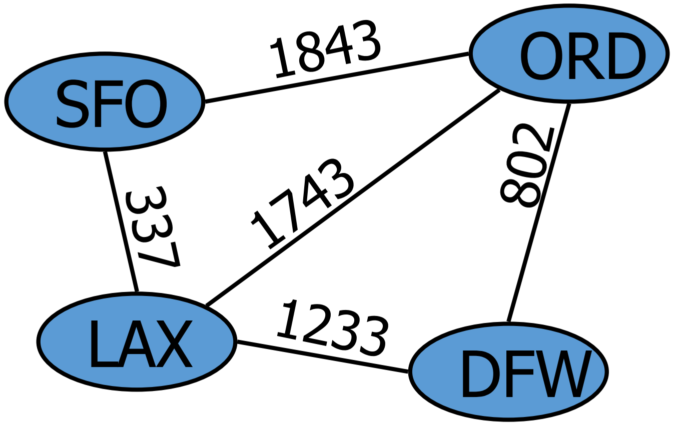
Walid Abdullah Al
Computer and Electronic Systems Engineering
Hankuk University of Foreign Studies

TA: **Seong Joo Kim**

Based on:
Goodrich, Chapter 9,11
Karumanchi, Chapter 6-7
Slides by Prof. Yung Yi, KAIST
Slides by Prof. Chansu Shin, HUFS

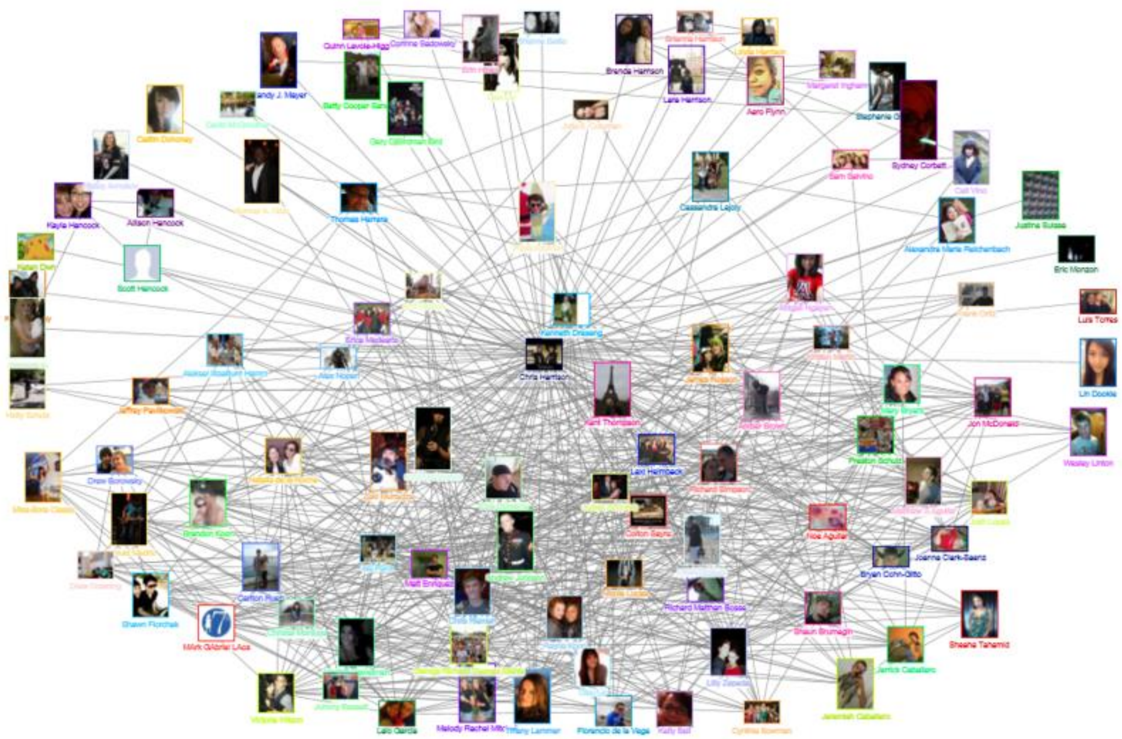
Computer Vision Lab
Hankuk University of Foreign Studies

Graphs: Basics



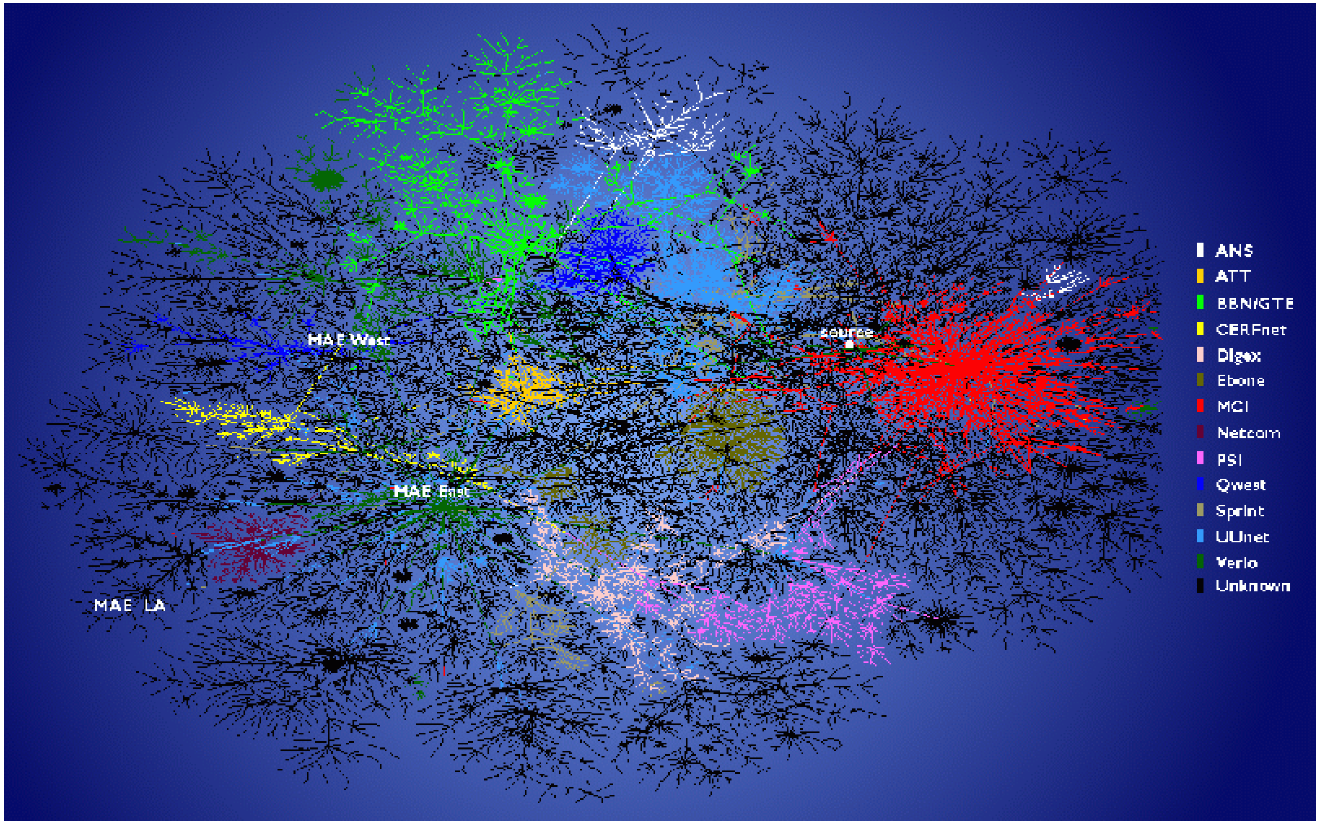
Real Life Examples

- **On-line/Off-line Social Network**



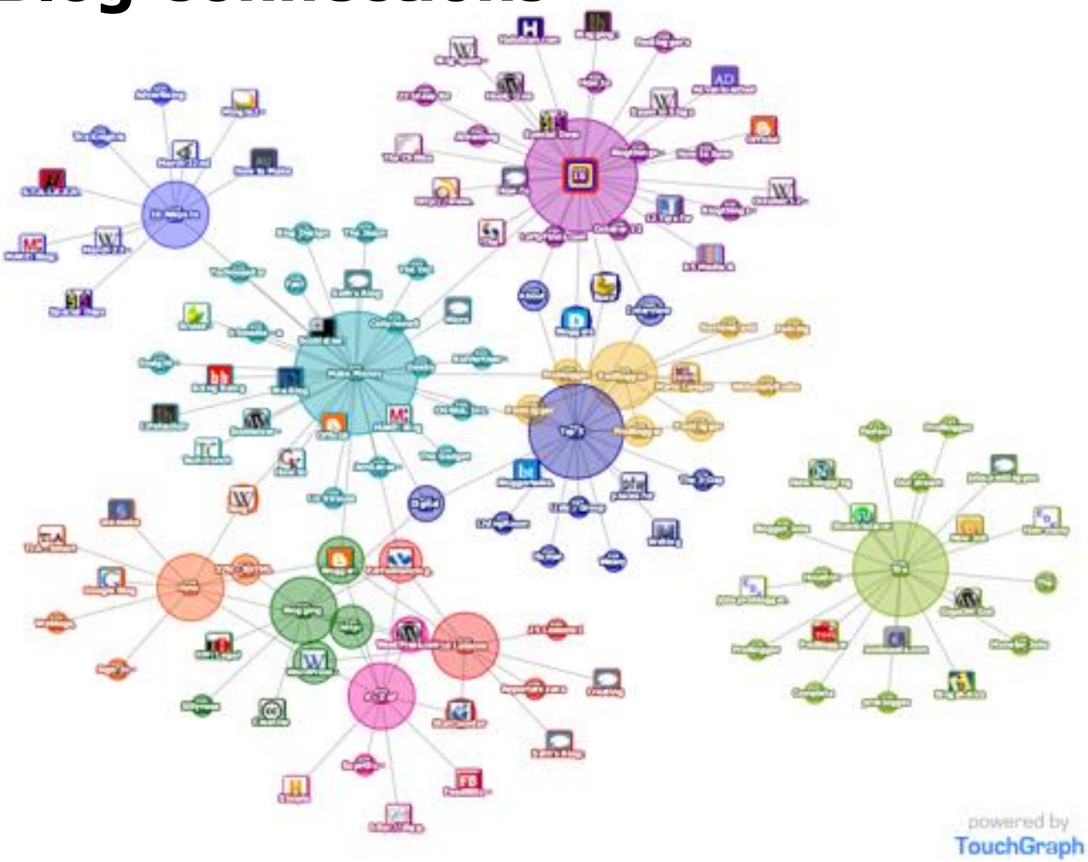
Real Life Examples

- **Internet Connectivity**



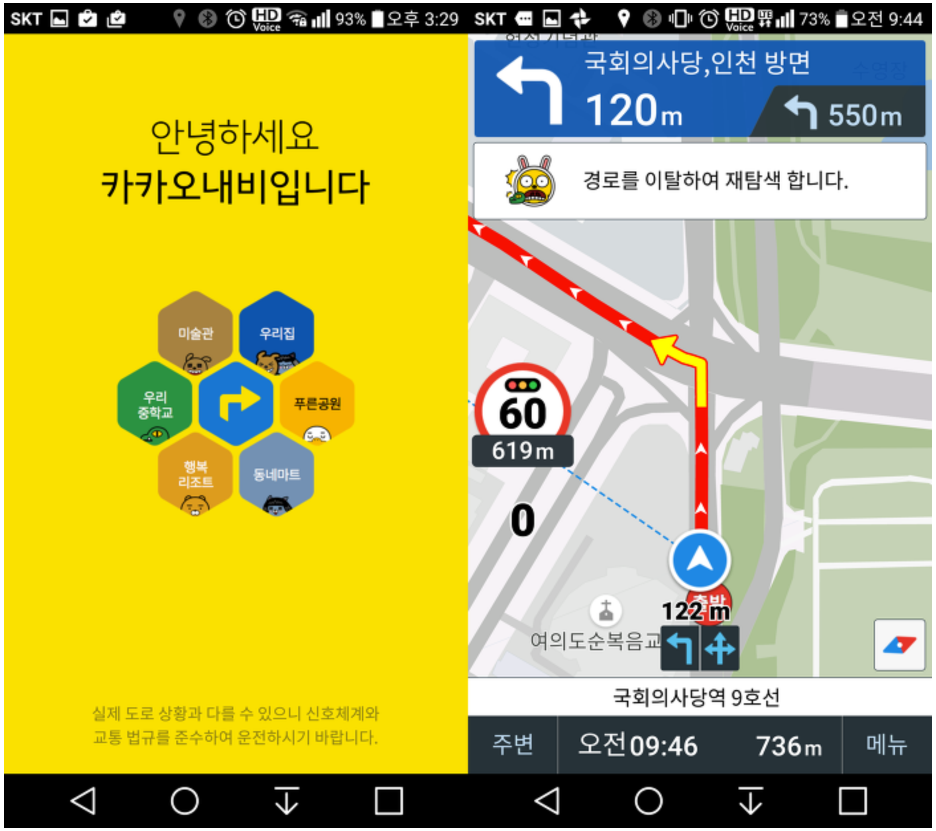
Real Life Examples

- **WebBlog Connections**



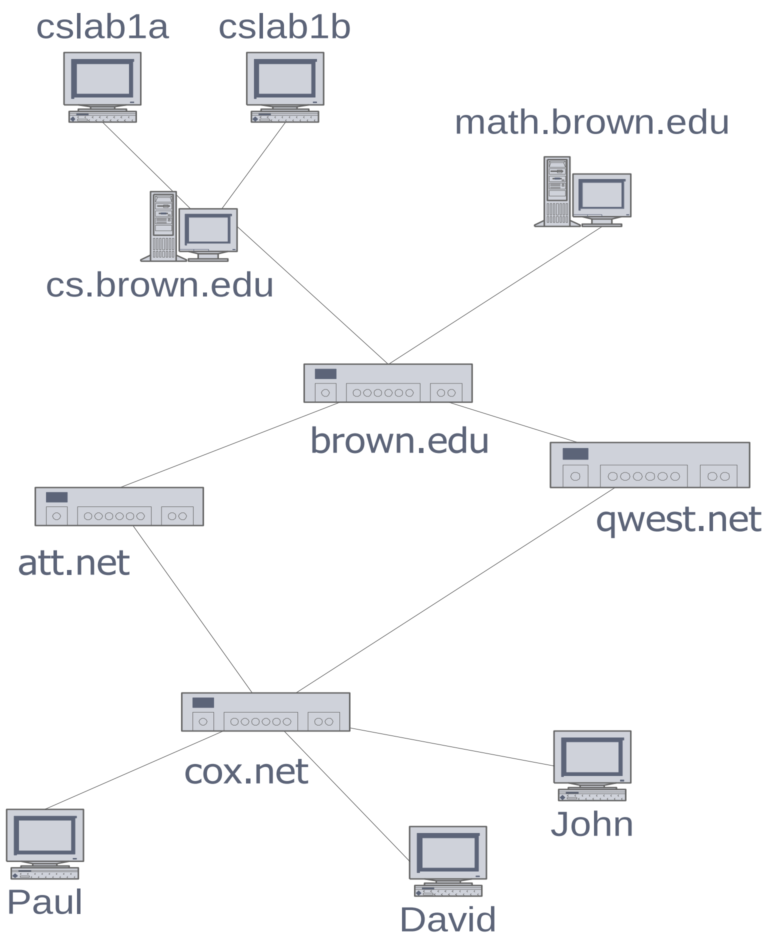
Real Life Examples

- **Navigator**



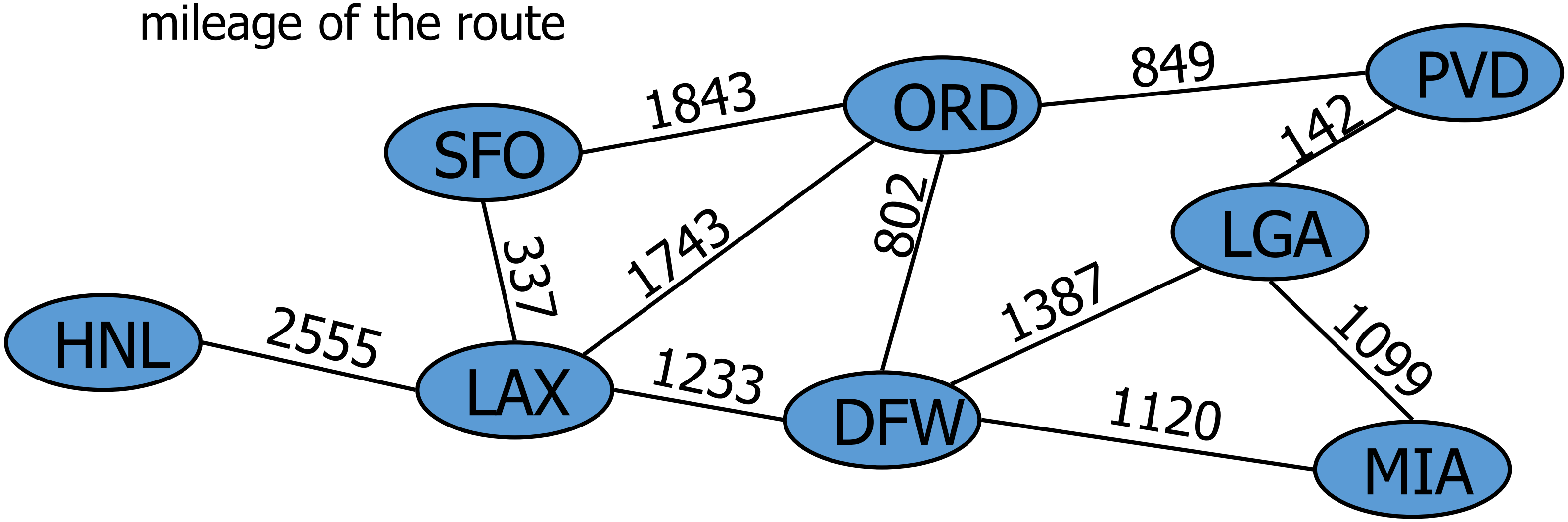
Other Applications

- **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

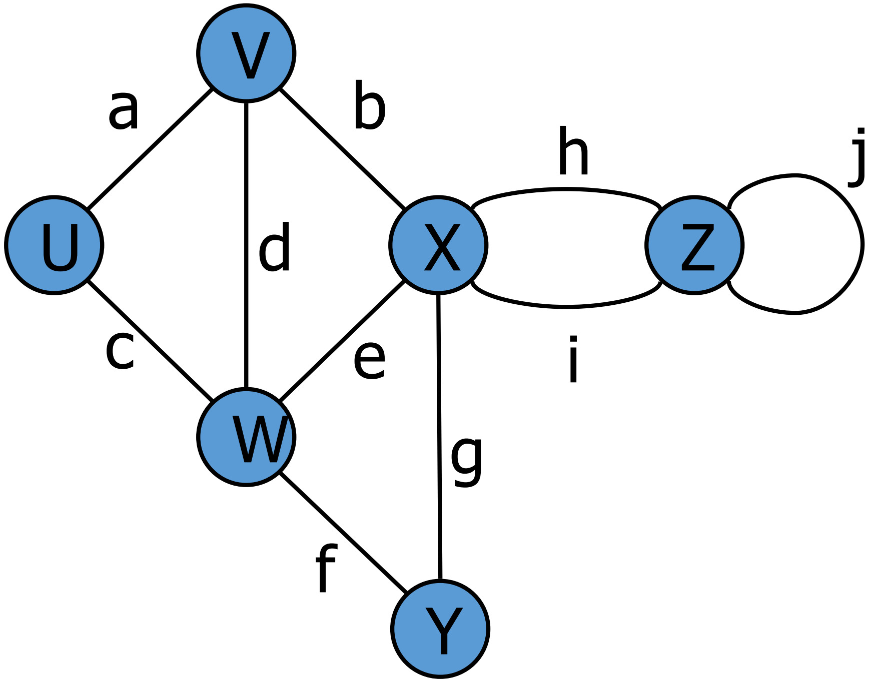


Edge Types

- **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

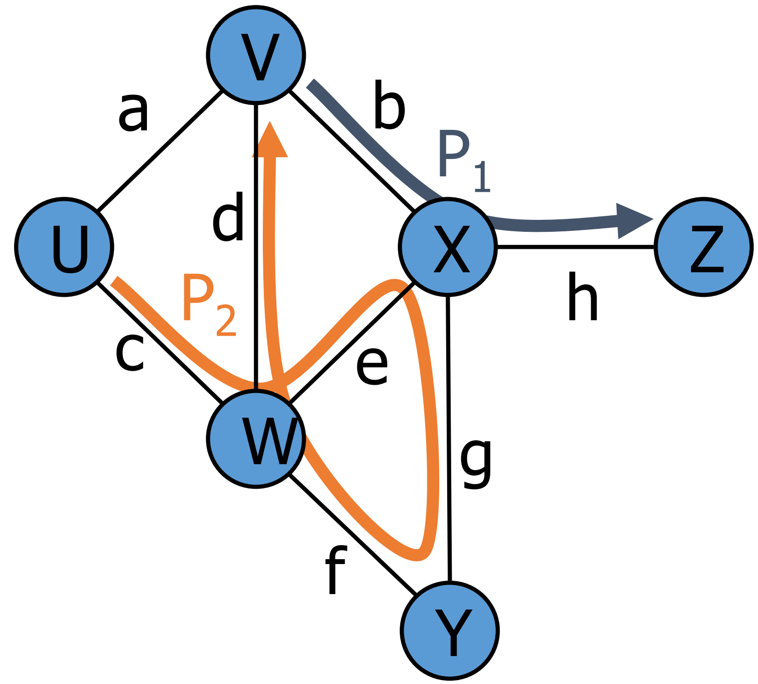
Terminology

- **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.)

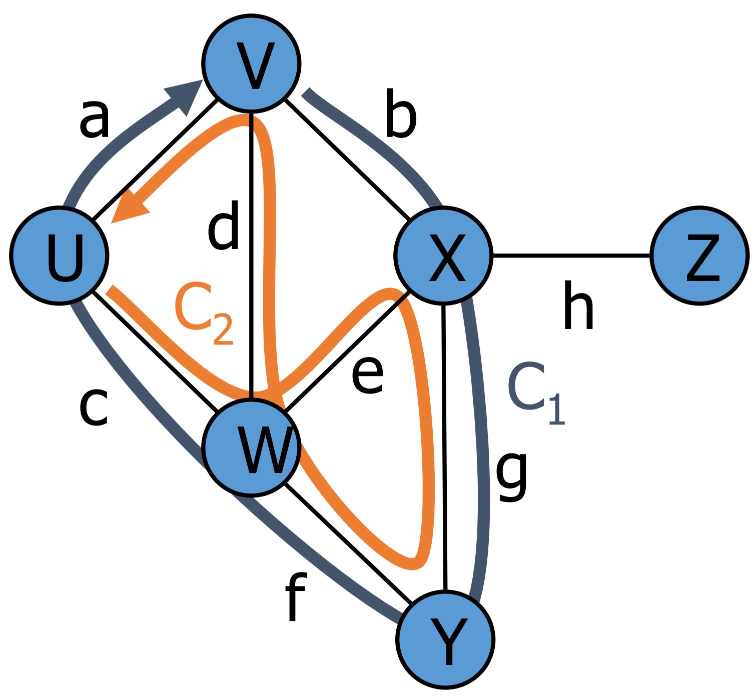
- **Path**
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- **Simple path**
  - path such that all its vertices and edges are distinct
- **Examples**
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V)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



Terminology (cont.)

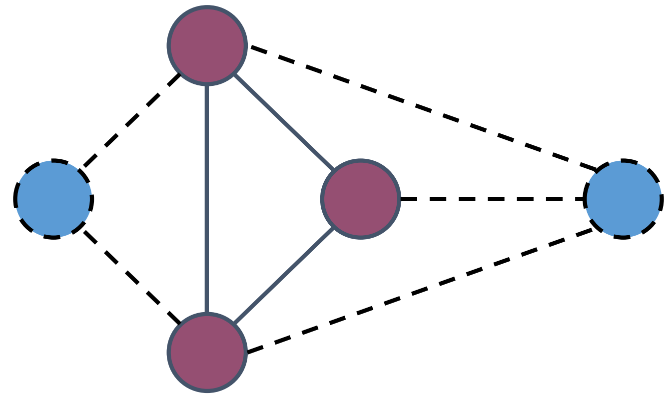
- **Cycle**
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- **Simple cycle**
  - cycle such that all its vertices and edges are distinct
- **Examples**
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵)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

- **Note) Tree is a graph without cycles**

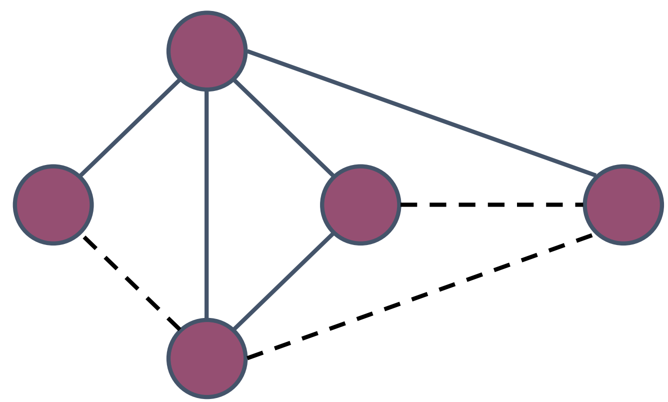


Subgraphs

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



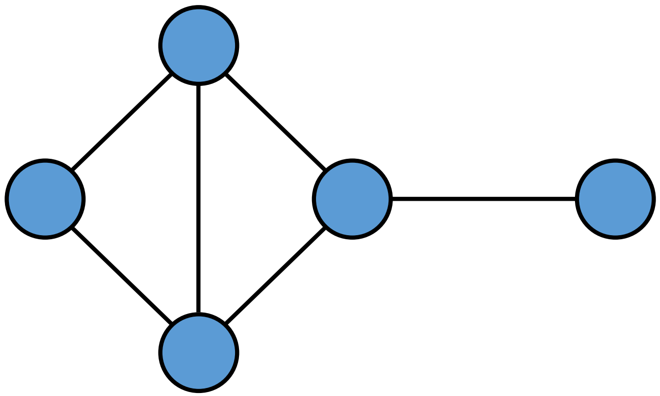
Subgraph



Spanning subgraph

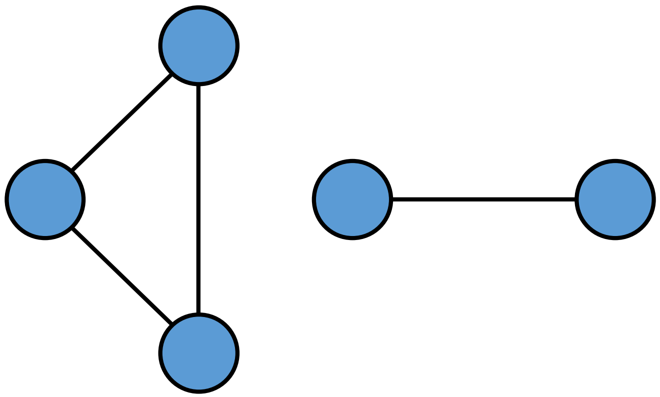
Connectivity

- **A graph is connected if there is a path between every pair of vertices**



Connected graph

- **A connected component of a graph G is a maximal connected subgraph of G**



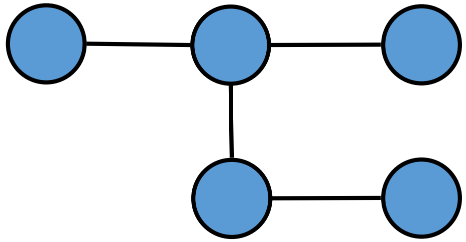
Non connected graph with two connected components

- **“Maximal”?**

Trees and Forests

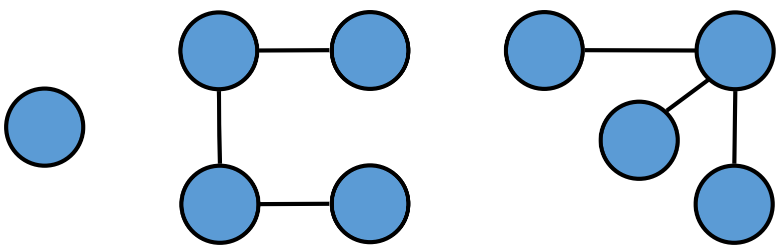
- **A (free) tree is an undirected graph T such that**
  - T is connected
  - T has no cycles

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



Tree

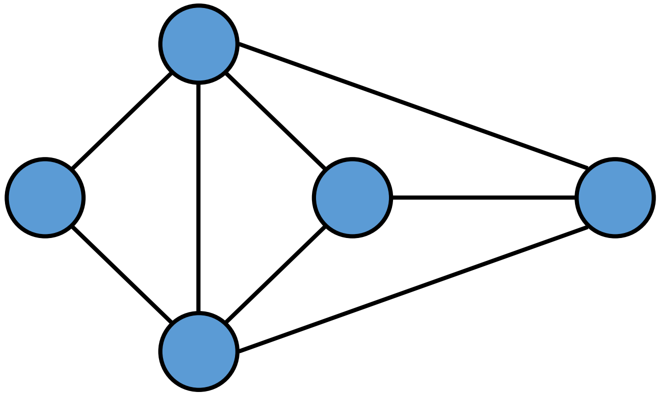
- **A forest is an undirected graph without cycles**
- **The connected components of a forest are trees**



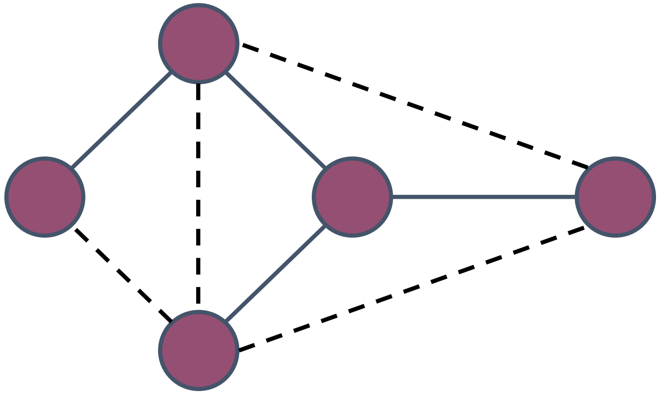
Forest

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

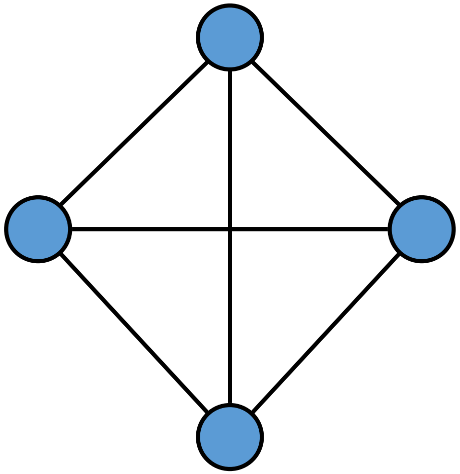
$m \leq n\,(n-1)/2$

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

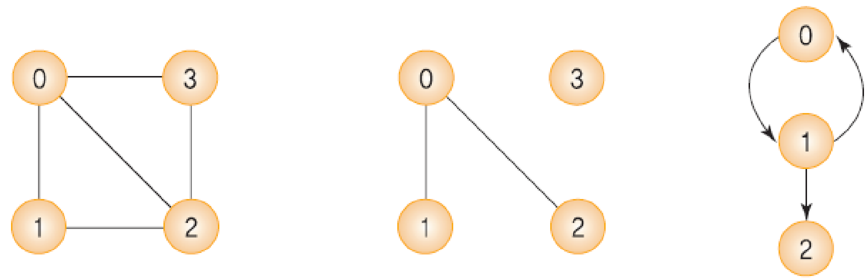
- **Vertices and edges**
  - are positions
  - store elements
- **Accessor methods**
  - e.endVertices(): a list of the two endvertices of e
  - e.opposite(v): the vertex opposite of v on e
  - u.isAdjacentTo(v): true if u and v are adjacent
  - *v: reference to element associated with vertex v
  - *e: 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 the data structure to represent a graph?

We will discuss three ways

Representation: 2D Array

- **Adjacency matrix)**
  - If edge (i, j) exists, M[i][j] = 1.
    Otherwise, M[i][j] = 0
  - The adj. matrix for an undirected graph has zero-diagonal and is symmetric.



	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	1
3	1	0	1	0

(a)

	0	1	2	3
0	0	1	1	0
1	1	0	0	0
2	1	0	0	0
3	0	0	0	0

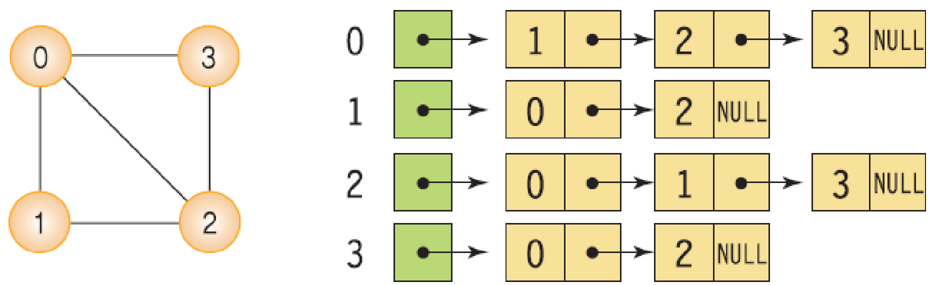
(b)

	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

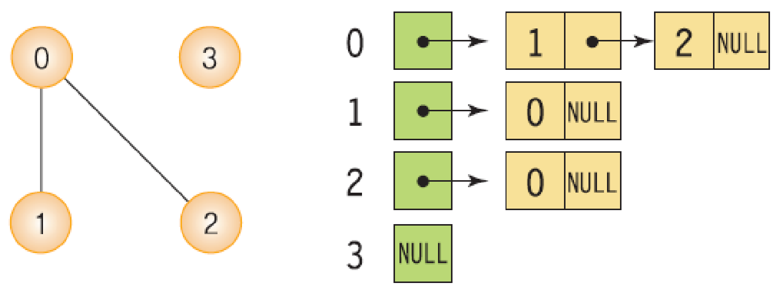
(c)

Representation : Adjacency List

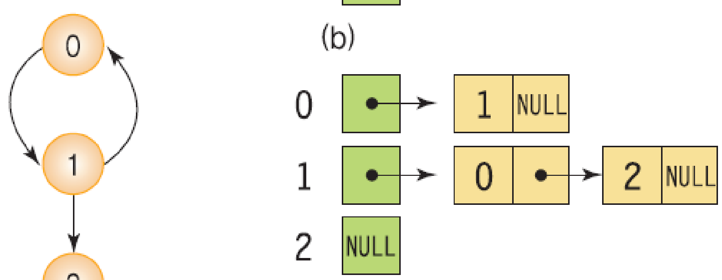
- **List of adjacent vertices**



(a)



(b)



(c)

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

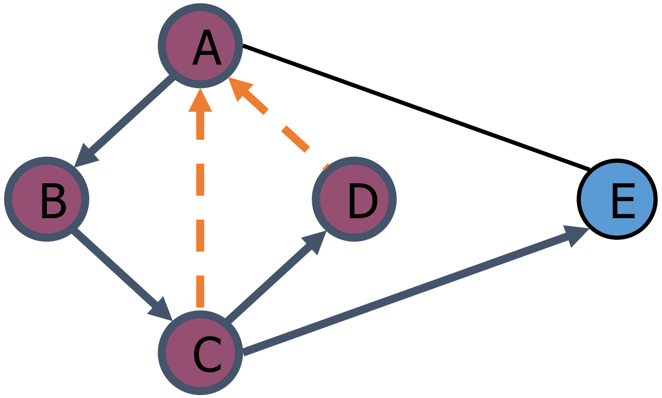
- v.incidentEdges(): matrix row check
- u.isAdjacentTo(v): using v's key

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

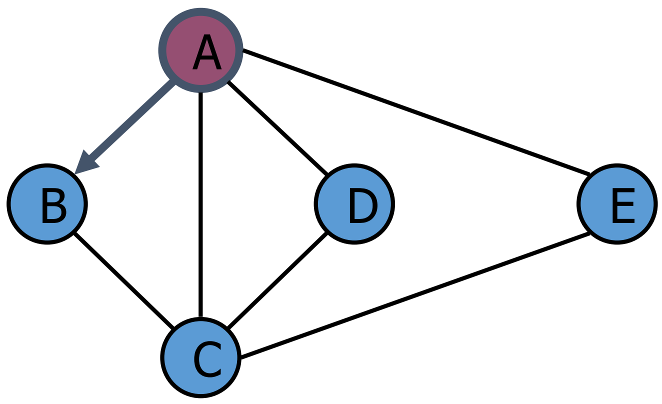


Depth-First Search

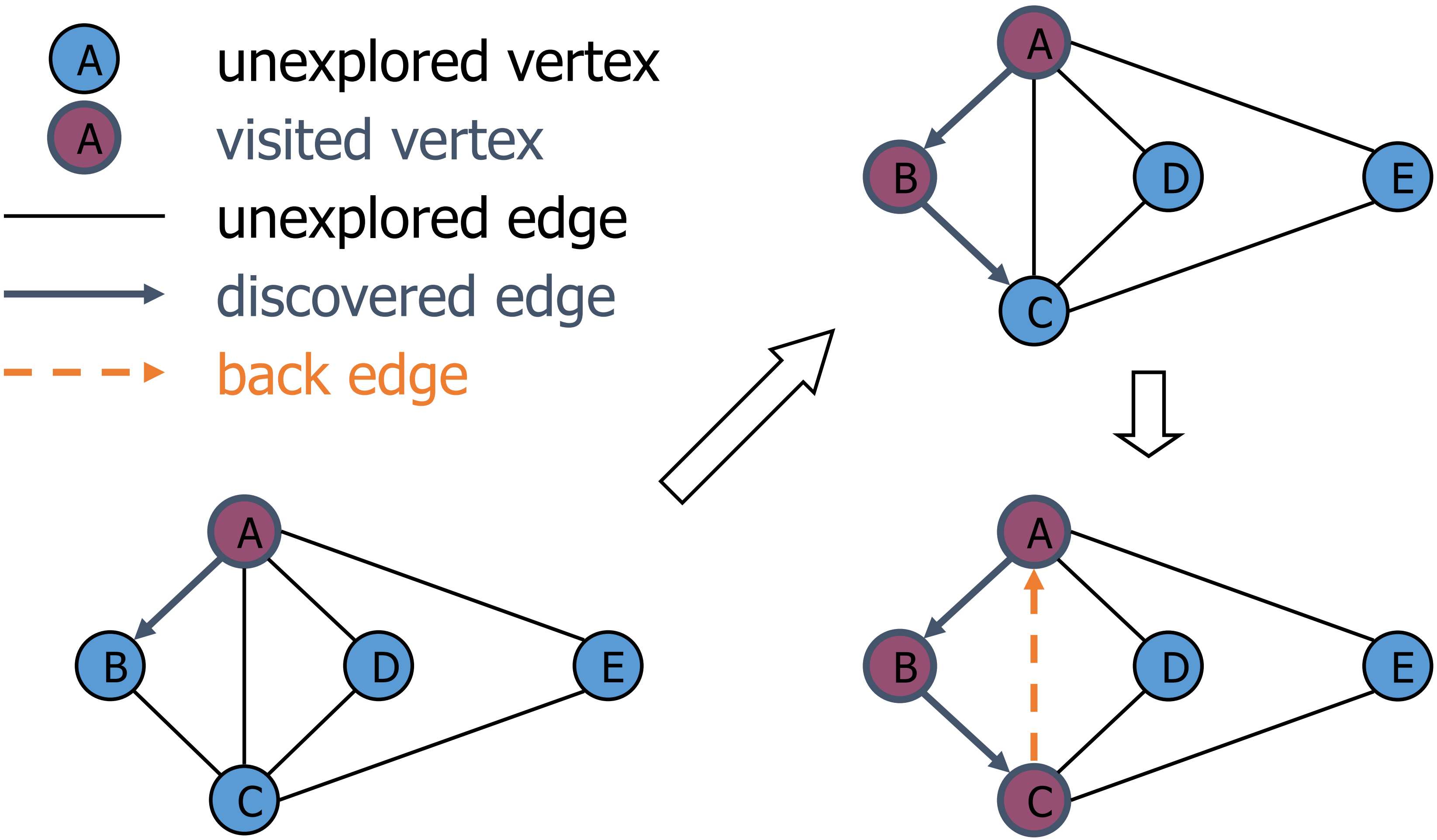
- **Depth-first search (DFS) is a general technique for traversing a graph**

- **Why is this traversal important?**

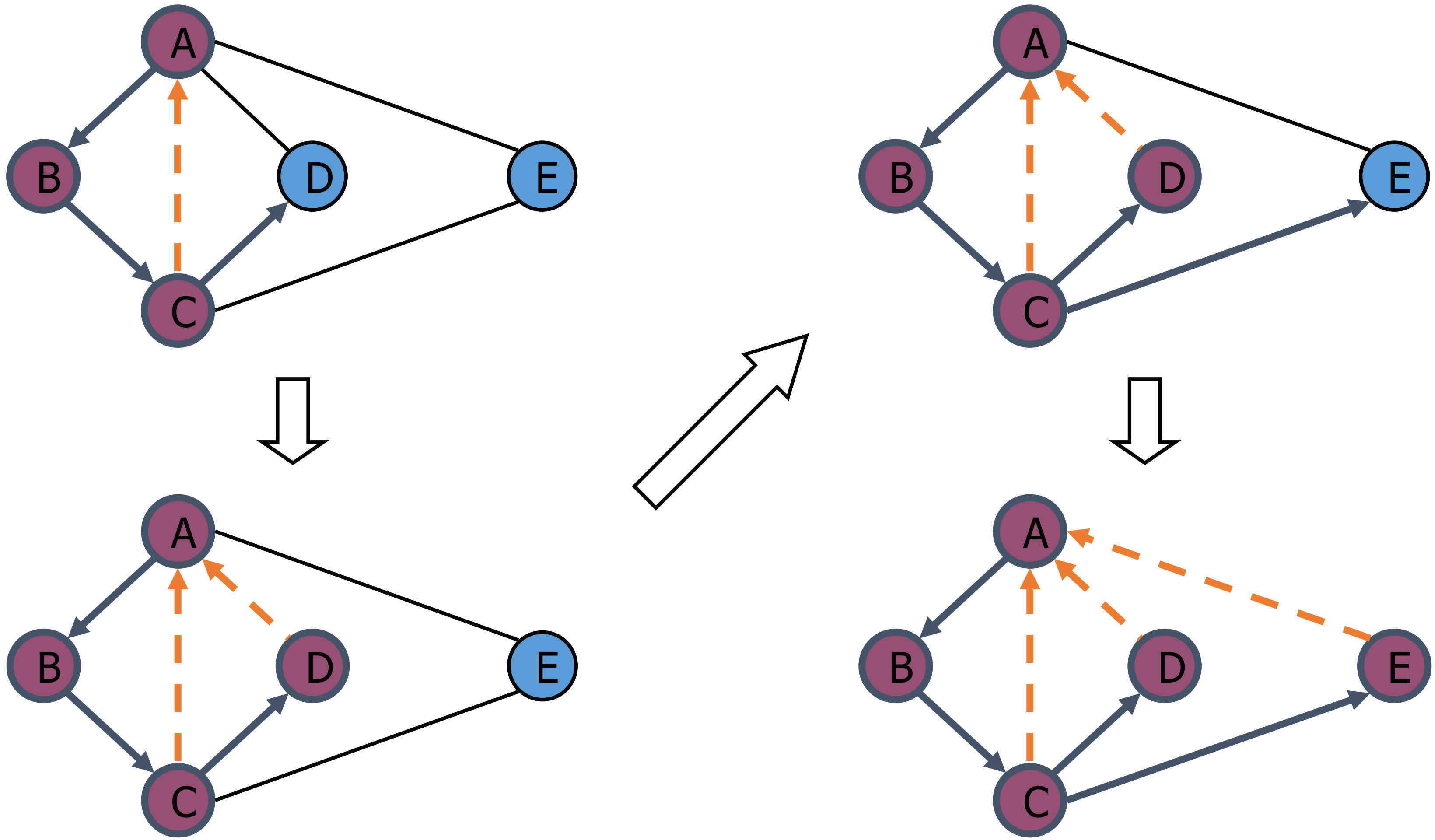
- **Let's first see the example**



Example


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

Example (cont.)



One implication: discovery edges form a spanning tree.

Depth-First Search

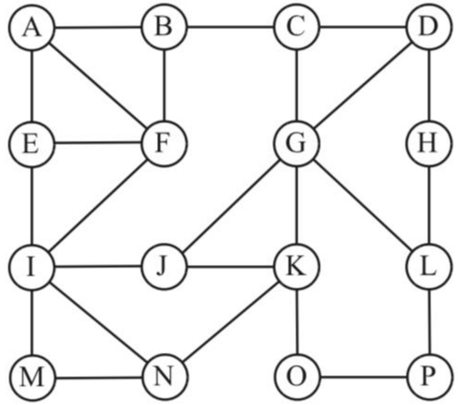
- **A DFS traversal of a graph G**
  - Visits all the vertices and edges of G
  - Determines whether G is connected (how?)
  - Computes the connected components of G (how?)
  - Computes a spanning forest of G
- **DFS on a graph with n vertices and m edges takes $O(n + m)$ time**
- **DFS can be further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

DFS Algorithm

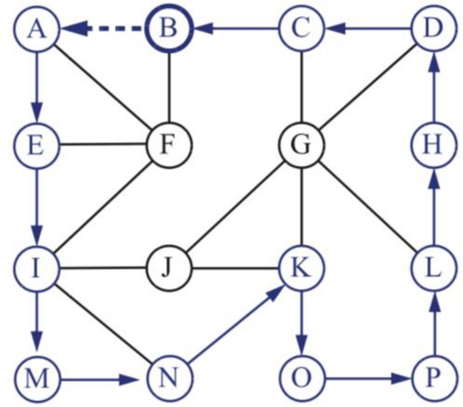
- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

```
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)
```

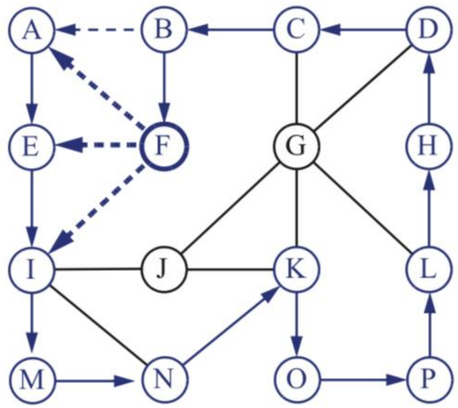
```
Algorithm DFS(G, v)
    Input graph G and a start vertex v of G
    Output labeling of the edges of G
        in the connected component of v
        as discovery edges and back edges
    v.setLabel(VISITED)
    for all e ∈ G.incidentEdges(v)
        if e.getLabel() = UNEXPLORED
            w ← e.opposite(v)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


A B C D
E F G H
I J K L
M N O P

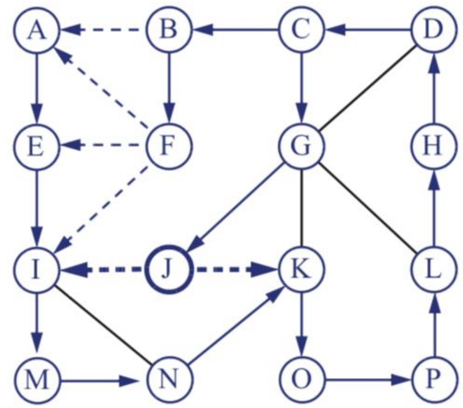
(a)


A B C D
E F G H
I J K L
M N O P

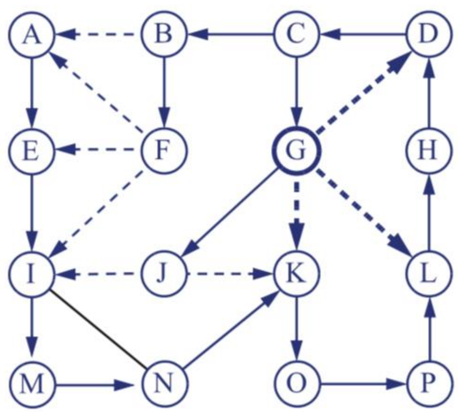
(b)


A B C D
E F G H
I J K L
M N O P

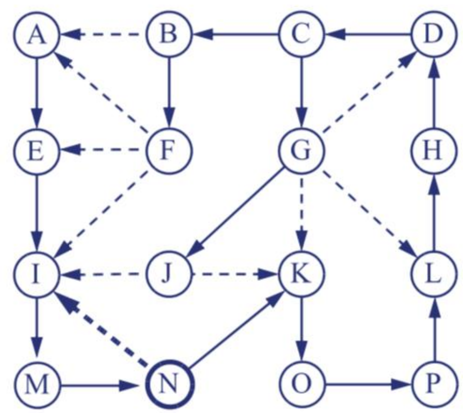
(c)


A B C D
E F G H
I J K L
M N O P

(d)

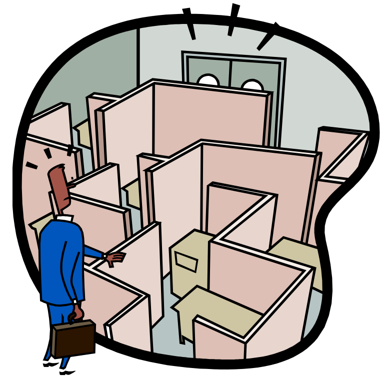

A B C D
E F G H
I J K L
M N O P

(e)

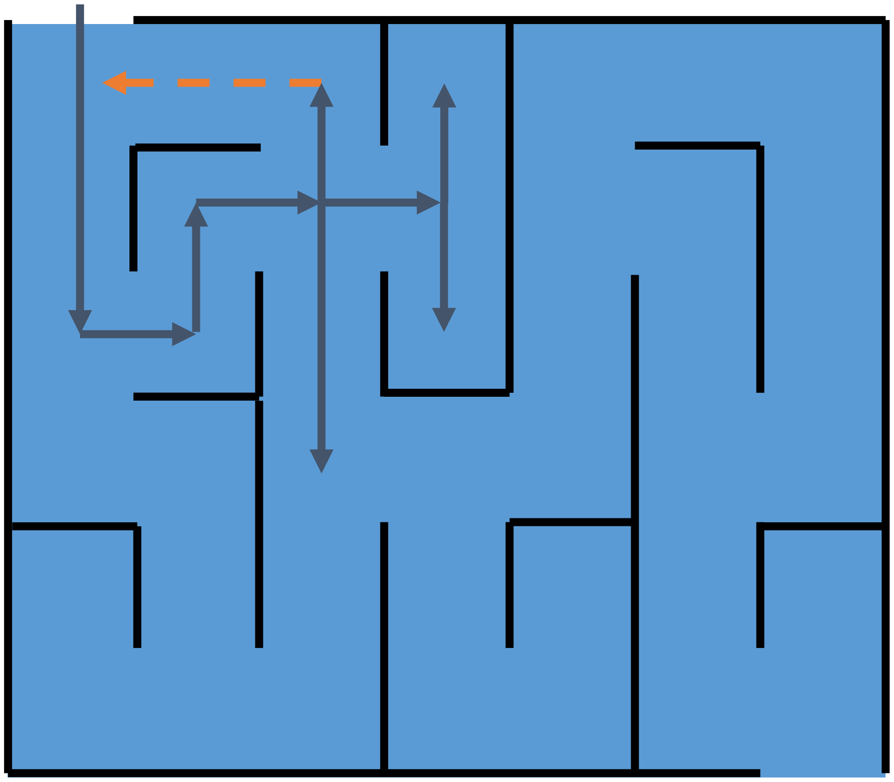

A B C D
E F G H
I J K L
M N O P

(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)



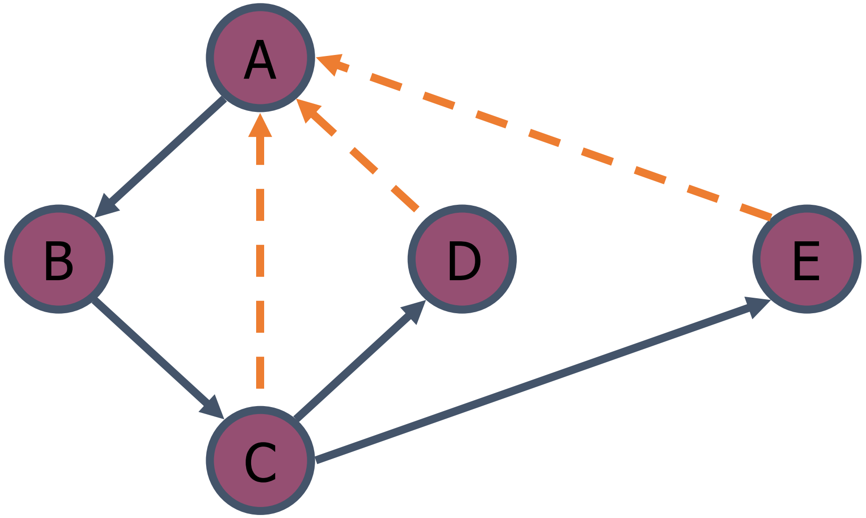
Properties of DFS

Property 1

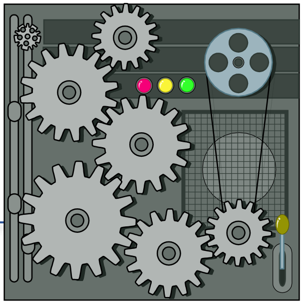
DFS*(*G*, *v*) visits all the vertices and edges in the connected component of *v

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)
```