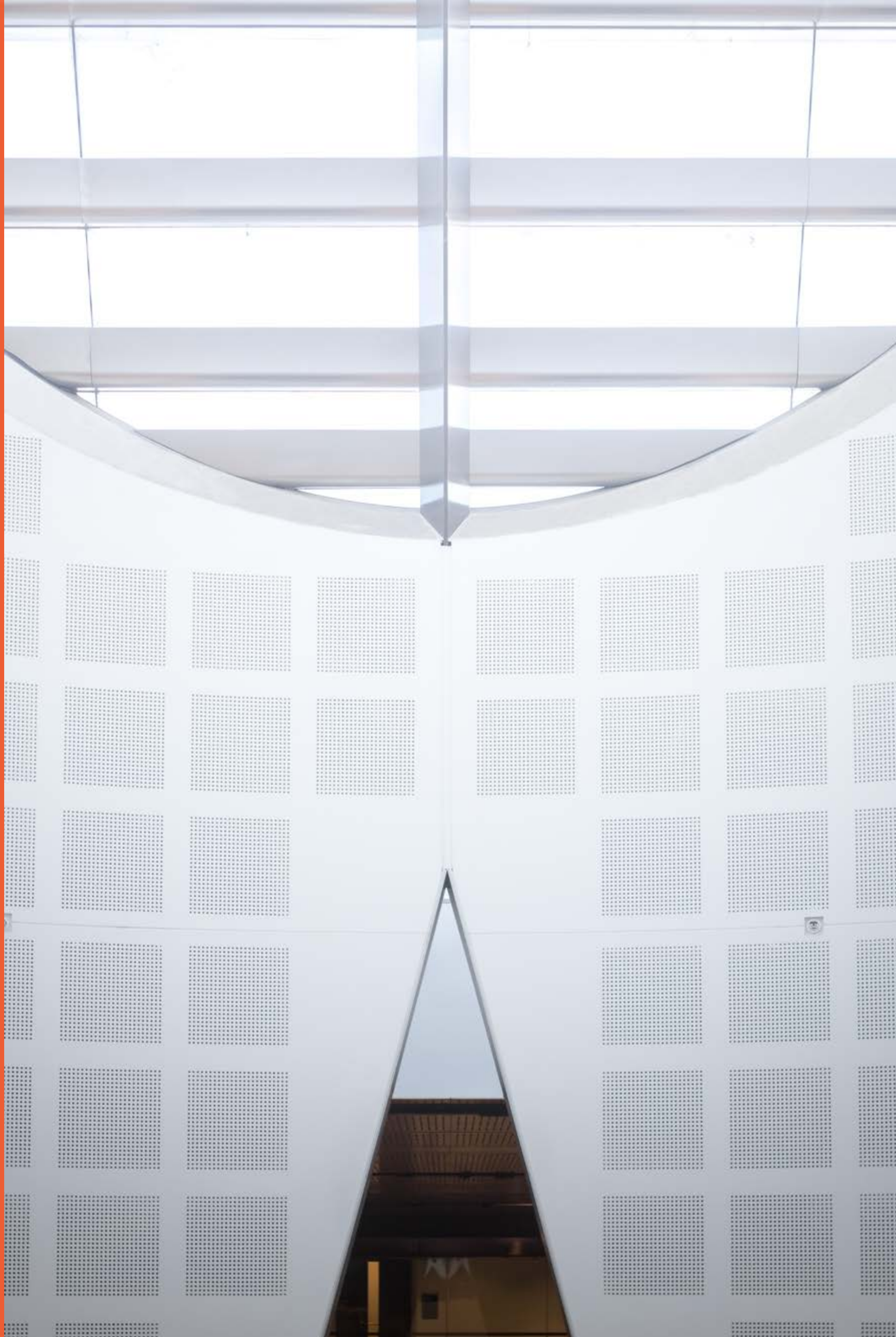
# INFO1105/1905
## Data Structures
## Week 9: Graphs (start)
see textbook section 14.1, 14.2, 14.3


Professor Alan Fekete
Professor Seokhee Hong
School of Information Technologies


using material from the textbook
and A/Prof Kalina Yacef


THE UNIVERSITY OF SYDNEY

# Copyright warning

- These slides contain material from the textbook (Goodrich, Tamassia & Goldwasser)
  - Data structures and algorithms in Java (5th & 6th edition)
- With modifications and additions from the University of Sydney

- The slides are a guide or overview of some big ideas
  - Students are responsible for knowing what is in the referenced sections of the textbook, not just what is in the slides

# Reminder! Quiz 4

- Quiz 4 will take place during lab in week 10
- Done online, over a 20 minutes duration,
  - during the last 30 minutes of the lab period, or as indicated by your tutor
- A few multiple choice questions,
  - covering material from weeks 7, 8, also part of week 9
    - hash tables and collision handling (including chaining and open addressing)
    - recurrence equations
    - graph definition
    - graph representations
      - However, graph traversal (BFS, DFS) will be in quiz 5!

# Outline

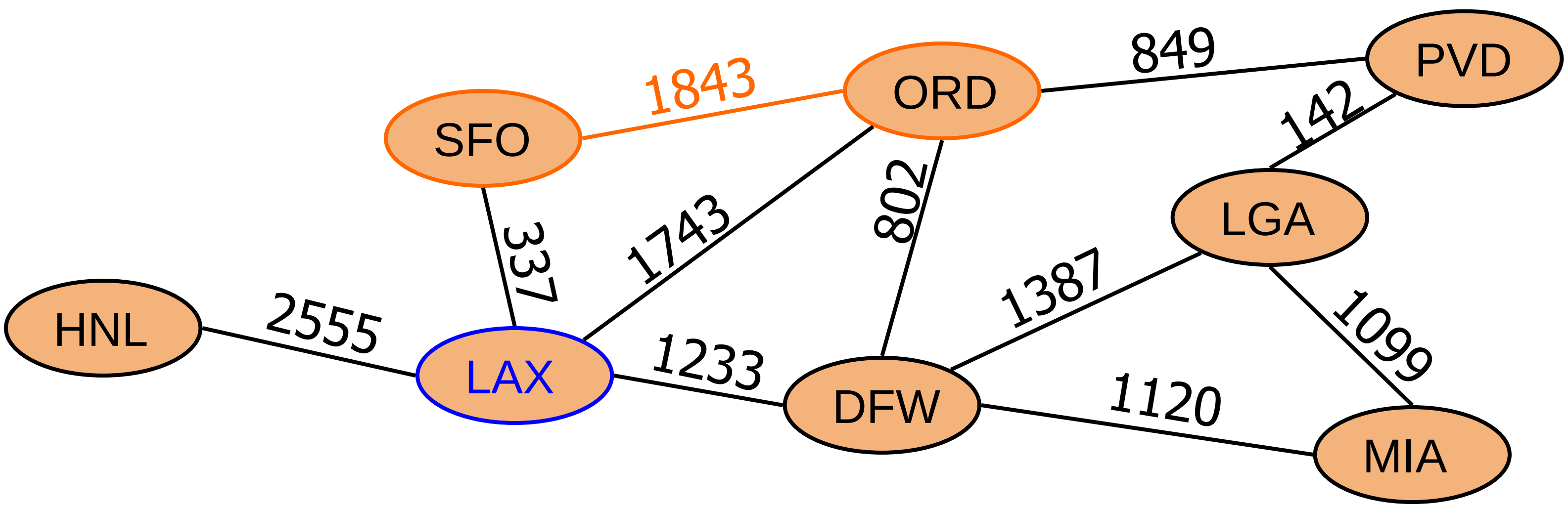
- **Graphs:**
  1. **Definitions**
  2. **Graph ADT**

- Data structures for graphs

- Graphs traversals

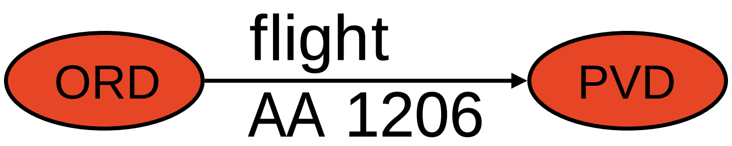
# Graphs

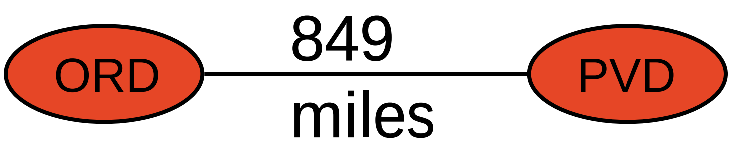
- Graph: represent relationships between objects (vertices)
- A graph is a pair $(V, E)$, where
  - $V$ is a set of nodes, called **vertices** (singular : vertex)
  - $E$ is a collection of pairs of vertices, called **edges**
  - Vertices and edges are positions and store elements
- Example:
  - A vertex 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

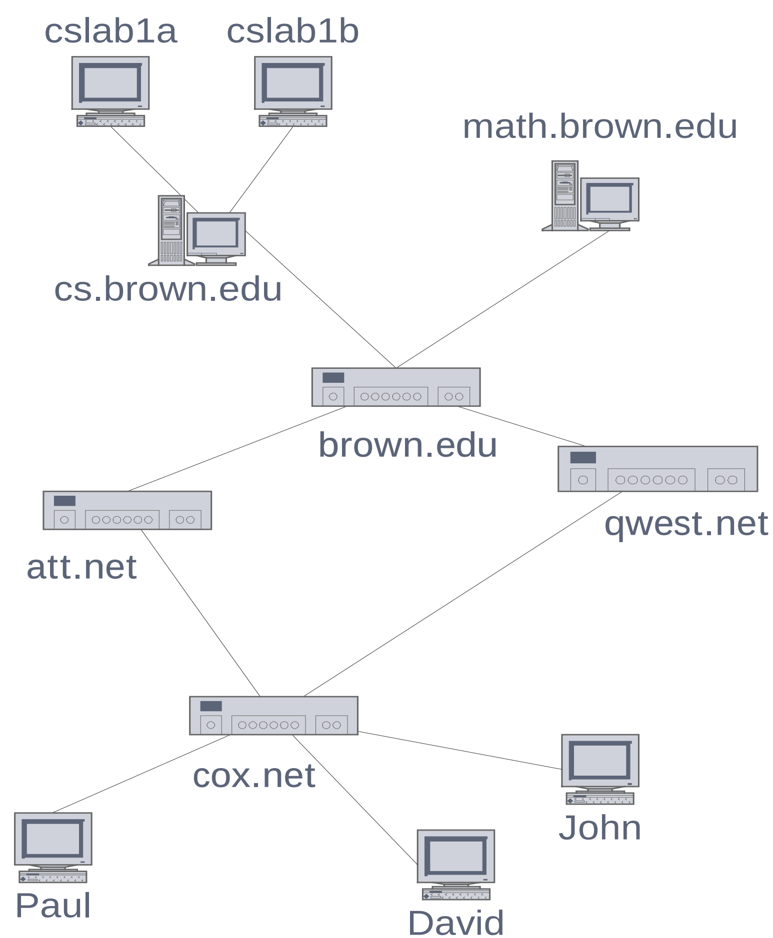
- **Directed** edge
  - **<u>ordered pair</u>** of vertices $(u,v)$
  - first vertex $u$ is the **origin**
  - second vertex $v$ is the **destination**
  - e.g., a flight
- **Undirected** edge
  - unordered pair of vertices $(u,v)$
  - e.g., a coauthorship relationship
- **Directed** graph
  - all the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



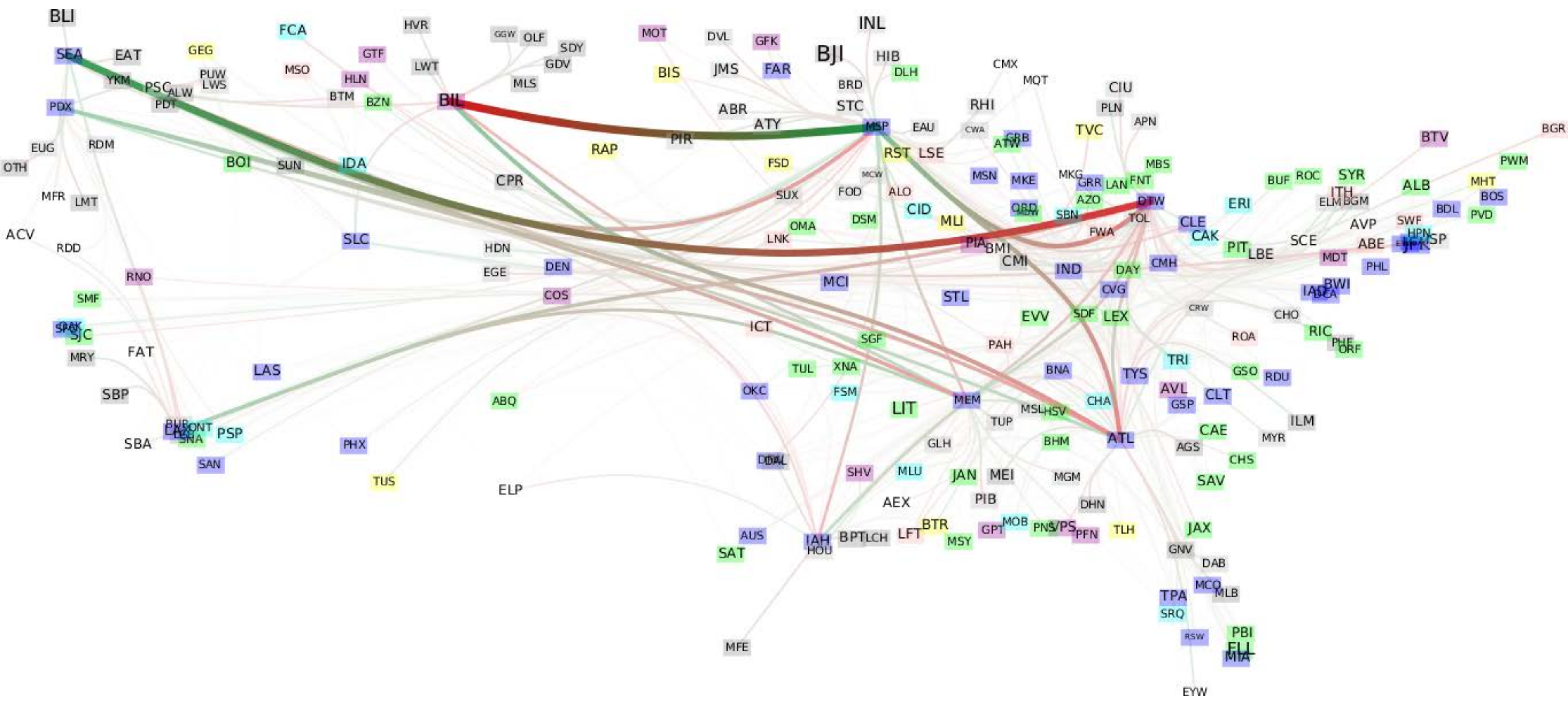



# Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
  - City maps
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram
- OO programming
  - Class inheritance

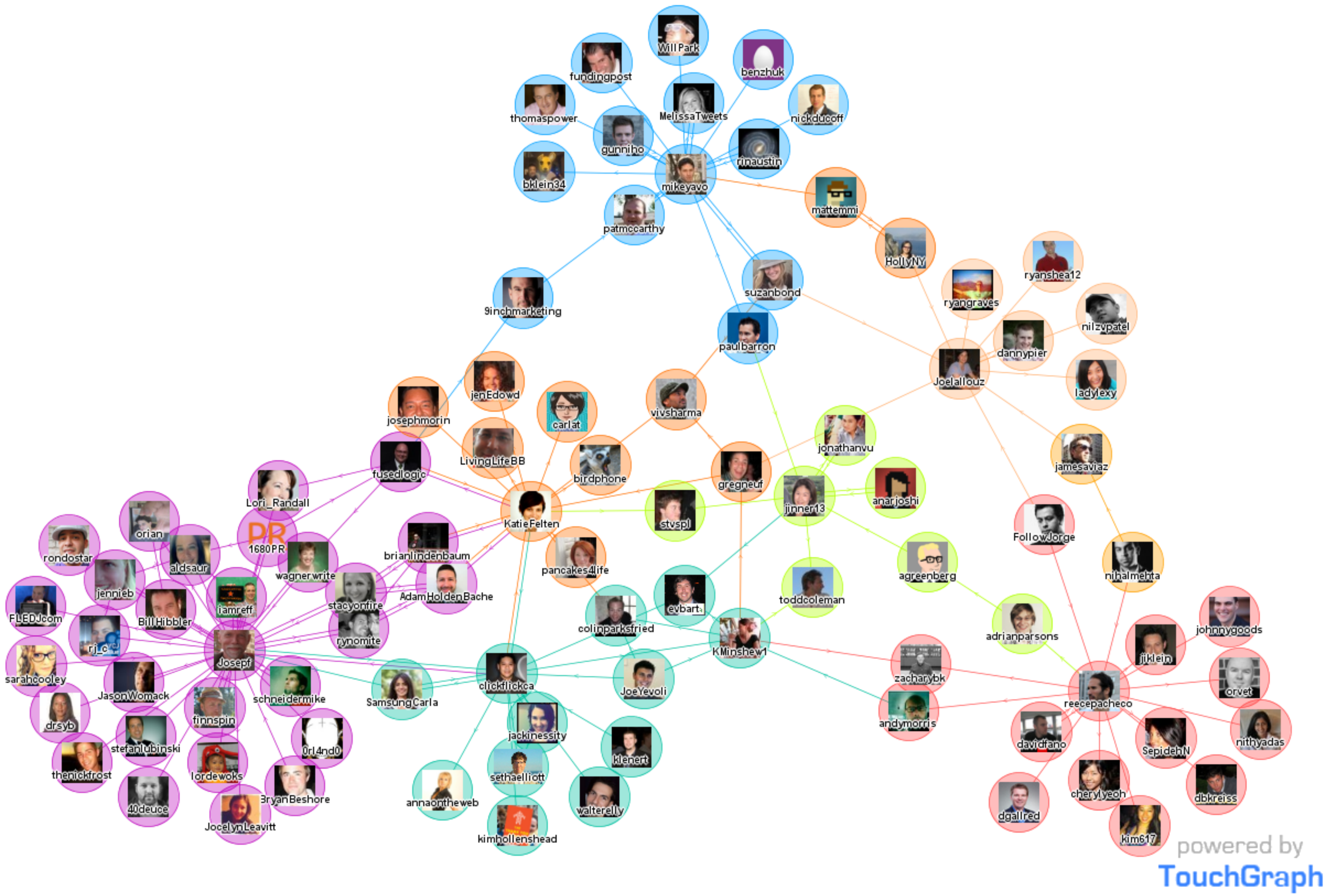


# US Airline Network

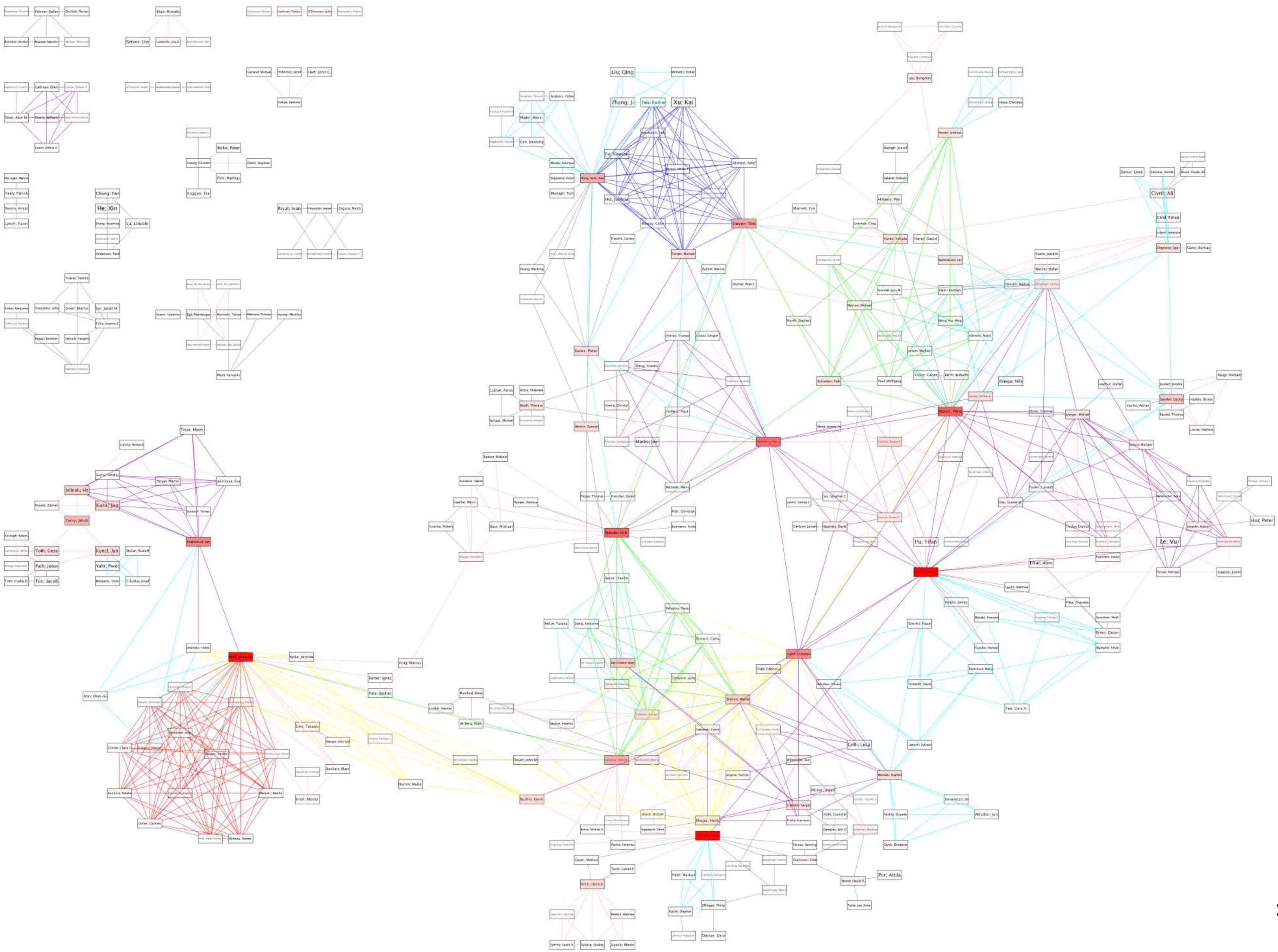


# London Metro Map

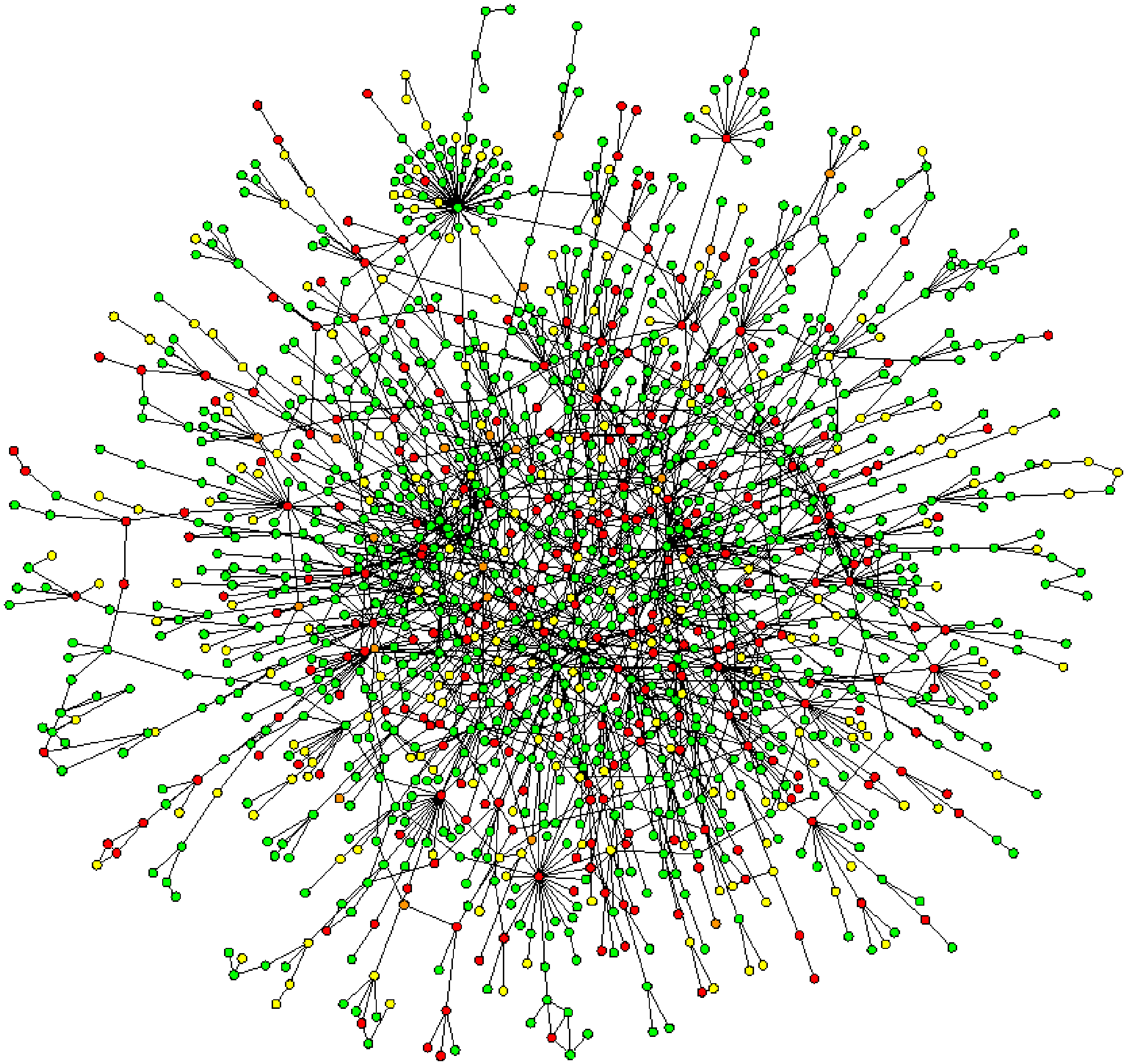
# Social Network: Facebook Network



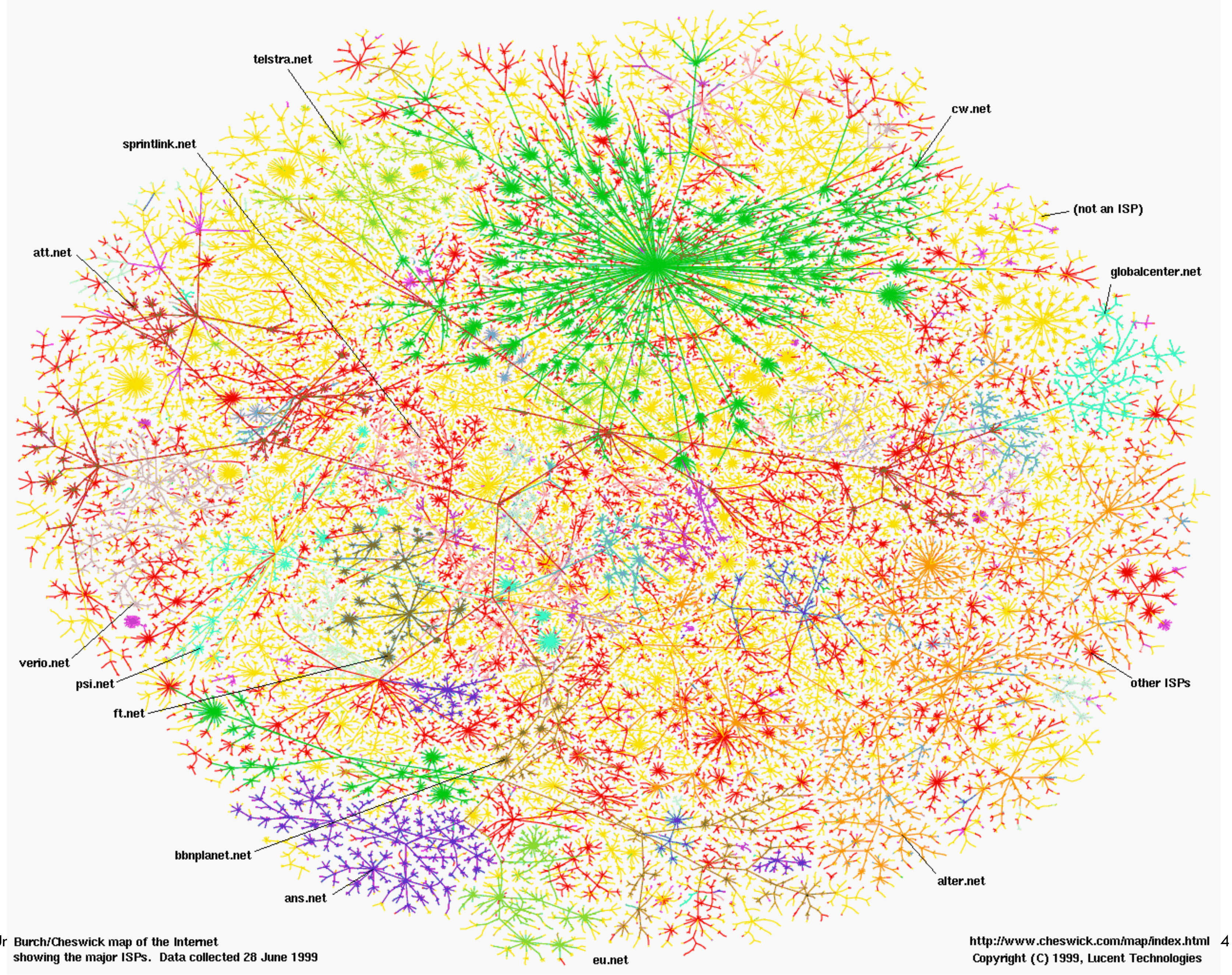
# Social Network: Research Collaboration Network



# Biological Network: Protein-Protein Interaction Network

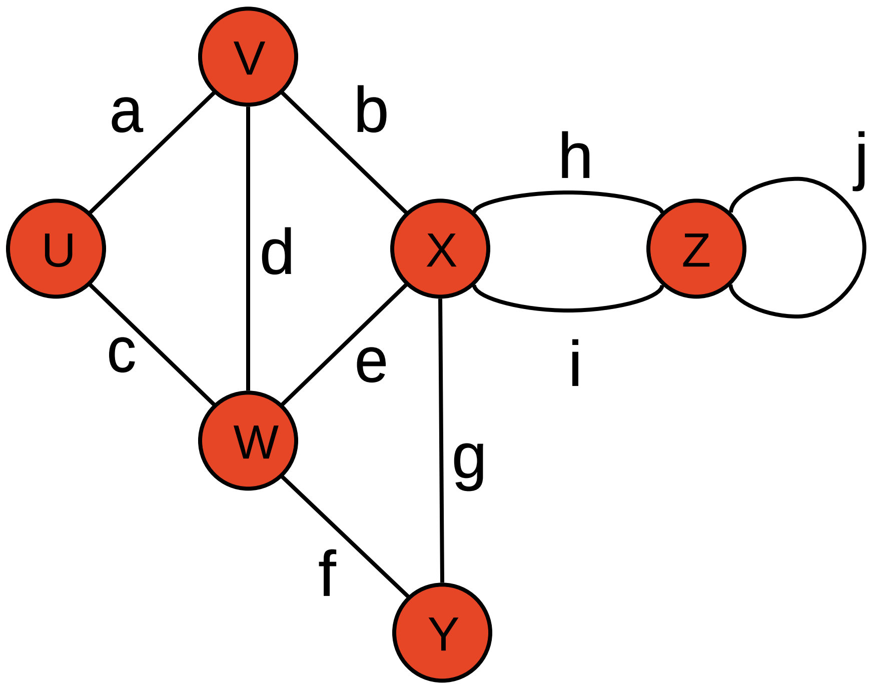


# Internet Mapping Project



# Terminology

– **End vertices** (or endpoints) of an edge
  – U and V are the endpoints of a
– Edges **incident** to 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
– **Simple** graph
  – no parallel edges or self-loops

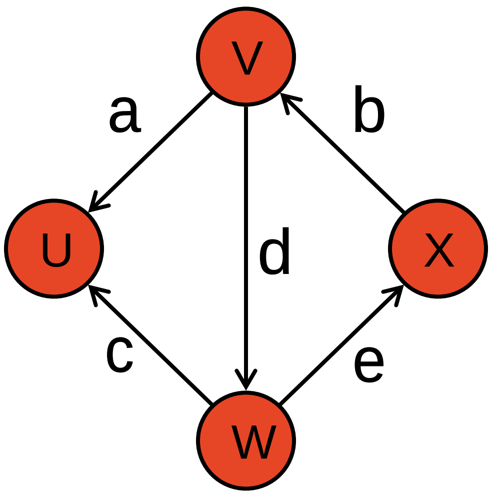


# Terminology (cont.)

If edge is **directed**

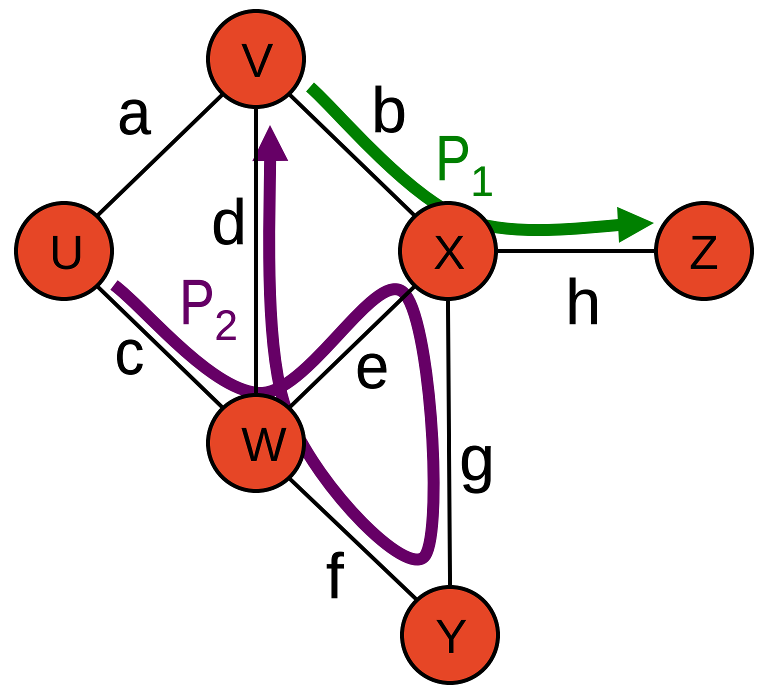
- **Origin, destination** vertices
- **Outgoing** edges of V are a,d
- **Incoming** edge of V is b

- Degree of a vertex
  - deg(V) is 3
  - **indeg**(V) is 1: in-degree
  - **outdeg**(V) is 2: out-degree



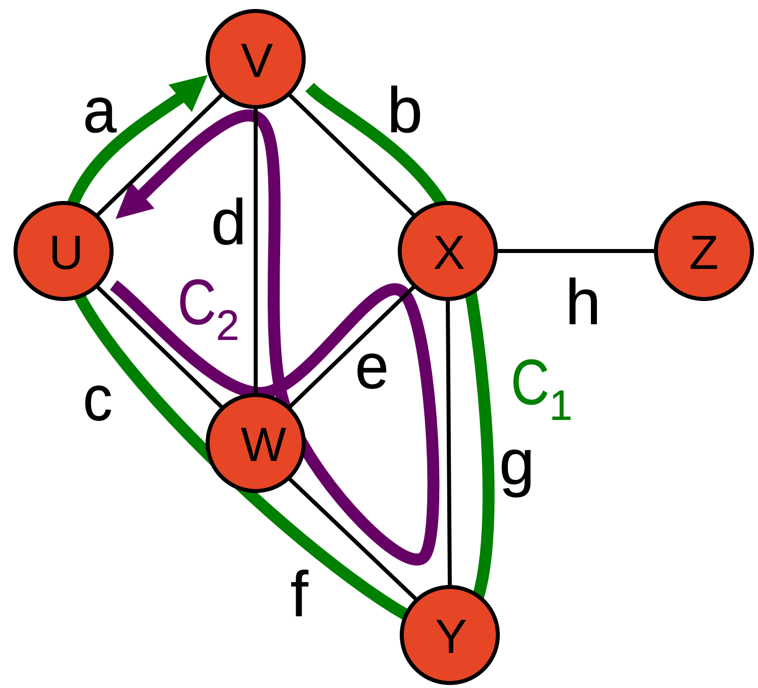
# Terminology (cont.)

- **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 <u>distinct</u>
- 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
- In directed graph, **directed** paths



# Terminology (cont.)

- **Cycle**
  - circular sequence of alternating vertices and edges
  - Path that starts and ends at the same vertex
- **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
- **Acyclic graph** has no cycle



# Properties

## Property 1

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

Proof: each edge is counted twice

## Property 2

In an undirected simple graph

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

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

directed simple graph:

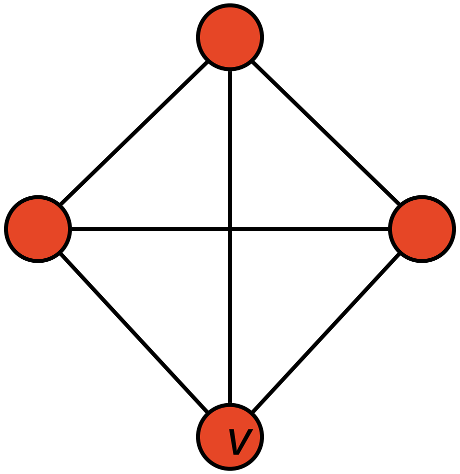
$m \leq n\,(n-1)$

* Complete graph $Kn$ (clique):

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

Notation

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




Example: K4

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

# Graph ADT: 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, getElement(), to retrieve the stored element.
- An **Edge** stores an associated object (e.g., a flight number, travel distance, cost), retrieved with the getEelement( ) method.

# Graph ADT Methods

- Old version

areAdjacent($v$, $w$): getEdge(u,v)

incidentEdges($v$): outgoingEdges(v) incomingEdges(v)

| | |
|---|---|
| numVertices( ): | Returns the number of vertices of the graph. |
| vertices( ): | Returns an iteration of all the vertices of the graph. |
| numEdges( ): | Returns the number of edges of the graph. |
| edges( ): | Returns an iteration of all the edges of the graph. |
| getEdge($u$, $v$): | Returns the edge from vertex $u$ to vertex $v$, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge($u$, $v$) and getEdge($v$, $u$). |
| endVertices($e$): | Returns an array containing the two endpoint vertices of edge $e$. If the graph is directed, the first vertex is the origin and the second is the destination. |
| opposite($v$, $e$): | For edge $e$ incident to vertex $v$, returns the other vertex of the edge; an error occurs if $e$ is not incident to $v$. |
| outDegree($v$): | Returns the number of outgoing edges from vertex $v$. |
| inDegree($v$): | Returns the number of incoming edges to vertex $v$. For an undirected graph, this returns the same value as does outDegree($v$). |
| outgoingEdges($v$): | Returns an iteration of all outgoing edges from vertex $v$. |
| incomingEdges($v$): | Returns an iteration of all incoming edges to vertex $v$. For an undirected graph, this returns the same collection as does outgoingEdges($v$). |
| insertVertex($x$): | Creates and returns a new Vertex storing element $x$. |
| insertEdge($u$, $v$, $x$): | Creates and returns a new Edge from vertex $u$ to vertex $v$, storing element $x$; an error occurs if there already exists an edge from $u$ to $v$. |
| removeVertex($v$): | Removes vertex $v$ and all its incident edges from the graph. |
| removeEdge($e$): | Removes edge $e$ from the graph. |

# Outline

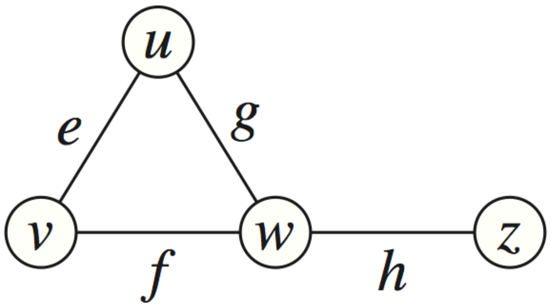
– Graphs: definitions and ADT

– **Data structures for graphs**

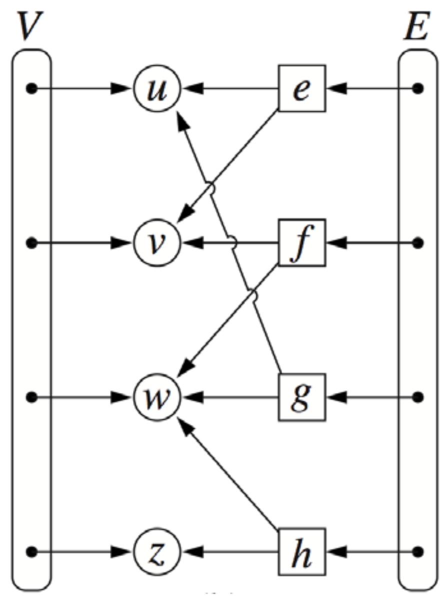
1. **Edge list structure**
2. **Adjacency list structure**
3. **Adjacency map structure**
4. **Adjacency matrix**

– Graphs traversals

# 1. Edge List Structure

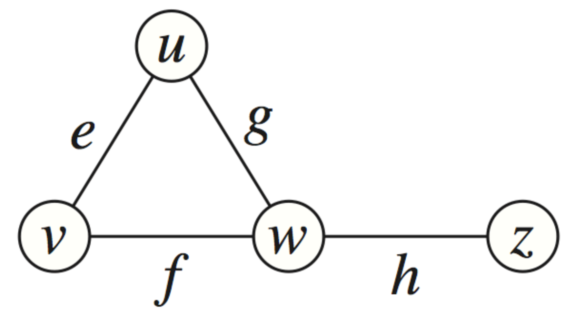


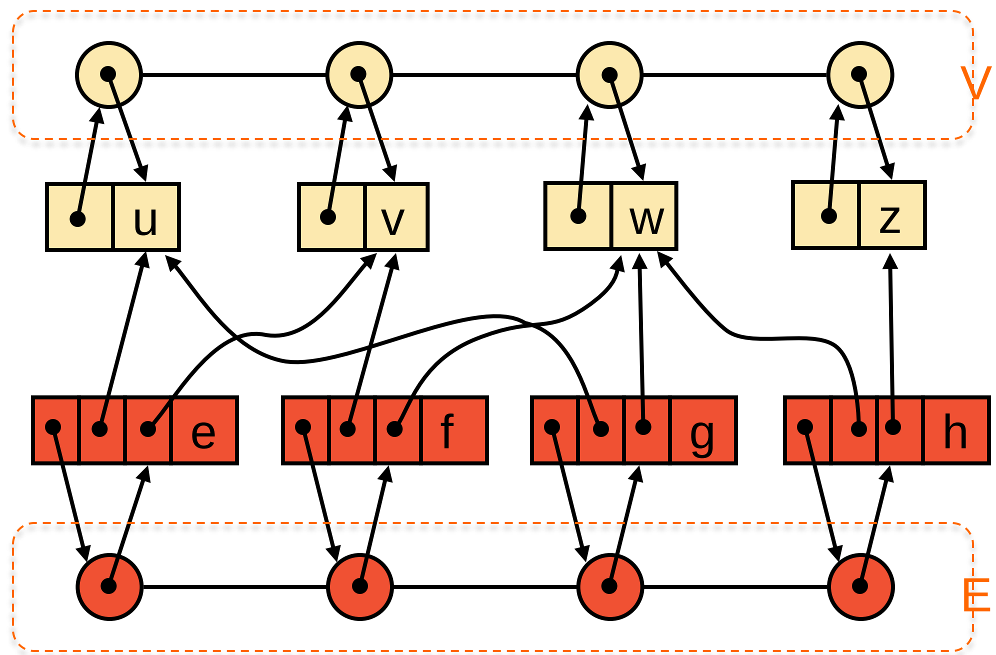

- Simple, but inefficient
- **Unordered list V, E**
- **Vertex object**
  - element
  - reference to position in vertex sequence
- **Edge object**
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- Vertex sequence **V**
  - sequence of vertex objects
- Edge sequence **E**
  - sequence of edge objects



# Edge List Structure

- Unordered list  of all edges
- 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 V
  - sequence of vertex objects
- Edge sequence E
  - sequence of edge objects



# Edge list: performance

| **Method** | Edge List |
|---|---|
| Space | $n + m$ |
| incidentEdges($v$) | $m$ |
| areAdjacent ($v, w$) | $m$ |
| insertVertex($o$) | 1 |
| insertEdge($v, w, o$) | 1 |
| removeVertex($v$) | $m$ |
| removeEdge($e$) | 1 |

- $n$ vertices, $m$ edges
- no parallel edges
- no self-loops

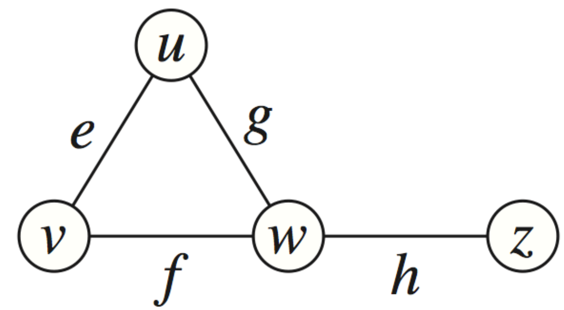
* Method

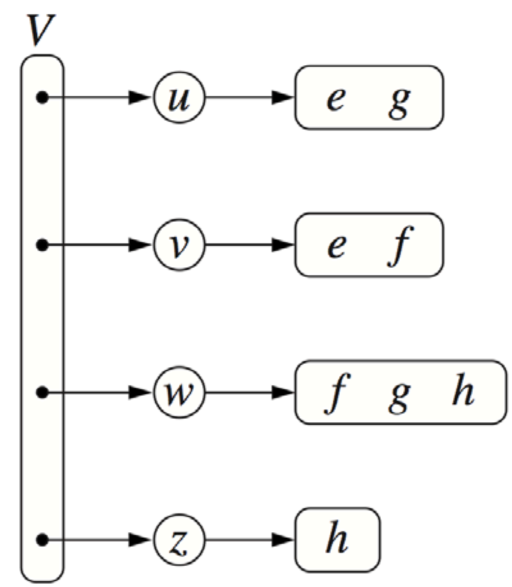
areAdjacent ($v, w$):
getEdge(u,v)

incidentEdges($v$):
outgoingEdges(v)
incomingEdges(v)

# 2. Adjacency List Structure

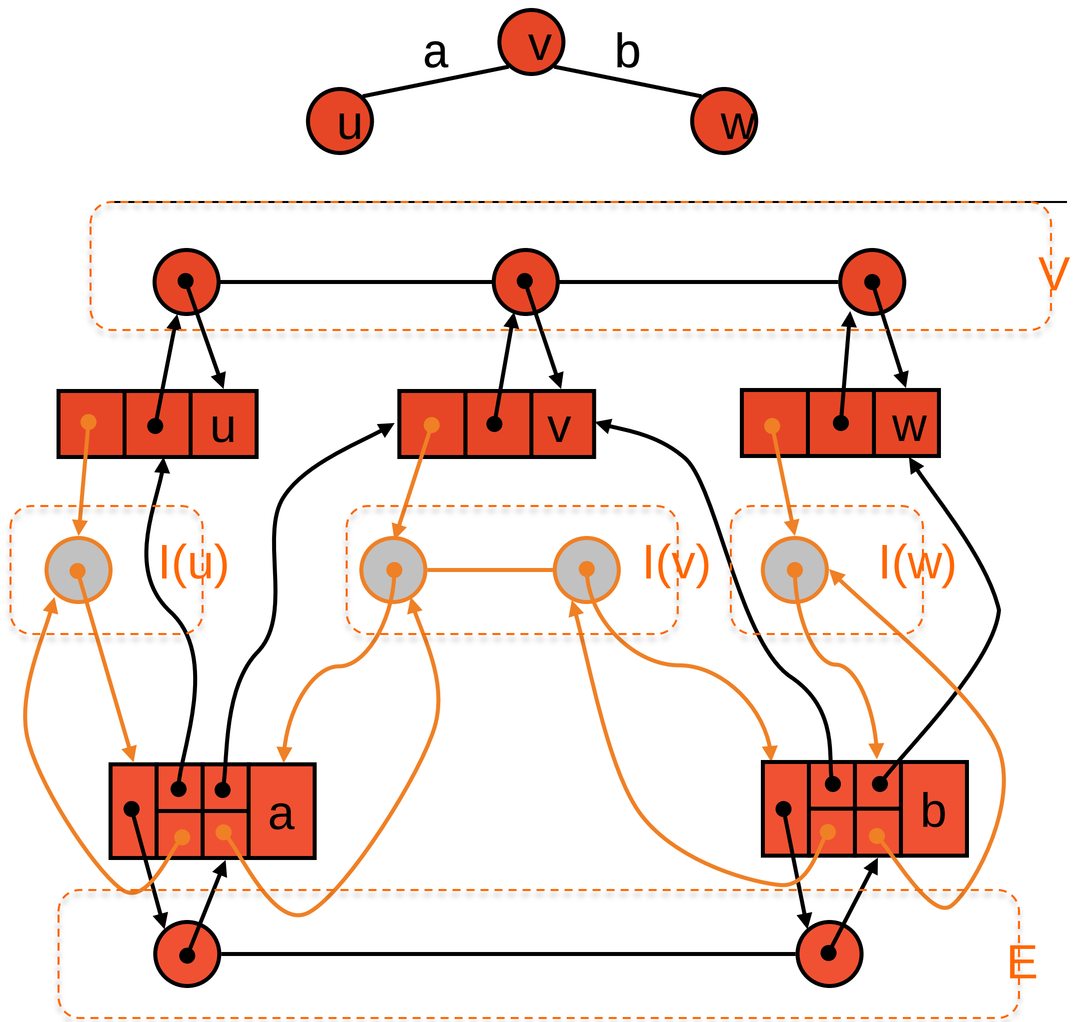
- Unordered list with additional list structure
- Incidence sequence for each vertex v
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices
- Sparse graphs: O(n) edges



# Adjacency List Structure

- Unordered list with additional list structure
- <u>Incidence sequence I(v)</u> for each vertex v
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices



# Adjacency list: performance

– All methods in O(1) with Edge list structure still O(1)

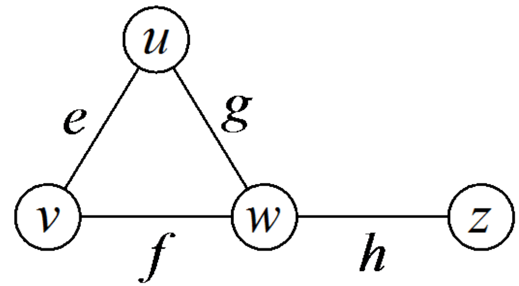
| **Method** | Adjacency List |
|---|---|
| Space | $n + m$ |
| incidentEdges($v$) | deg($v$) |
| areAdjacent ($v$, $w$) | min(deg($v$), deg($w$)) |
| insertVertex($o$) | 1 |
| insertEdge($v$, $w$, $o$) | 1 |
| removeVertex($v$) | deg($v$) |
| removeEdge($e$) | 1 |

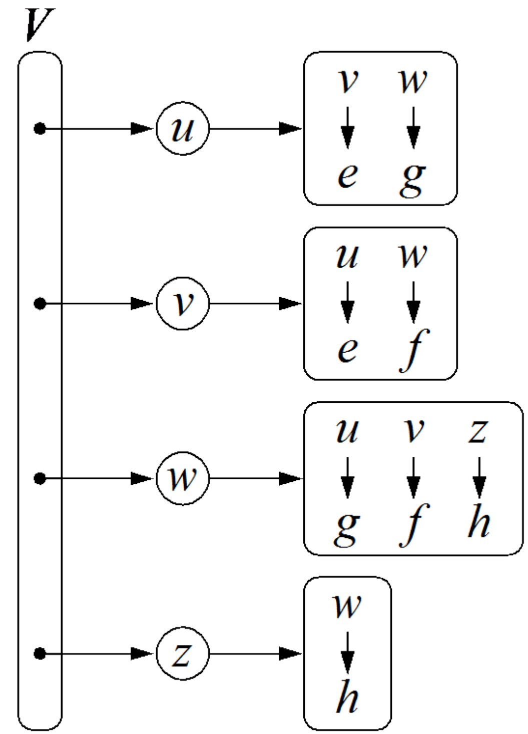
# 3. Adjacency Map Structure

- Same as adjacency list, but uses a <u>hash-based map</u> for storing incident edges (see week 8)

- Incidence map for each vertex v
  - Key = opposite endpoint Value =edge

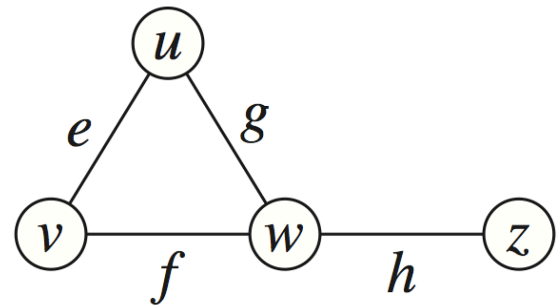
- <u>getEdge(u,v)</u> now is in <u>*expected* O(1)</u>
  - Although still worst case O(min(deg(u),deg(v)))



# 4. Adjacency Matrix Structure

- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array A
  - 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
- Dense graphs: O(n^2) edges




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

# Adjacency Matrix: performance

| Method | Adjacency Matrix |
|---|---|
| Space | $n^2$ |
| incidentEdges($v$) | $n$ |
| areAdjacent ($v$, $w$) | 1 |
| insertVertex($o$) | $n^2$ |
| insertEdge($v$, $w$, $o$) | 1 |
| removeVertex($v$) | $n^2$ |
| removeEdge($e$) | 1 |

# Performance: summary

- $n$ vertices, $m$ edges, no parallel edges/ self-loops
- incidentEdges($v$): outgoingEdges(v), incomingEdges(v)

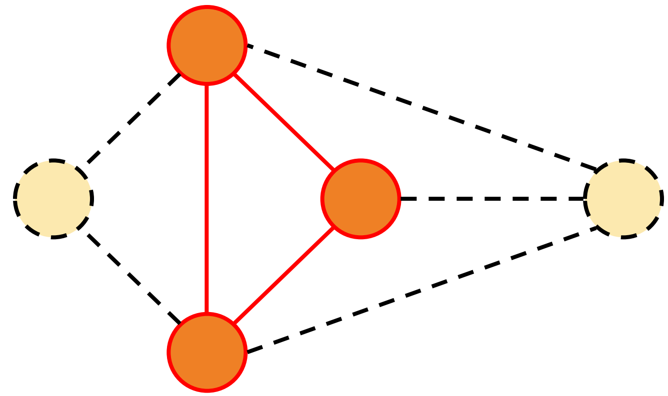
| Method | Edge List | Adjacency List | Adjacency Map | Adjacency Matrix |
|---|---|---|---|---|
| Space | $n + m$ | $n + m$ | $n + m$ | $n^2$ |
| getEdge($u,v$) | $m$ | min(deg($v$), deg($w$)) | 1(exp.) | 1 |
| outDegree(v), inDegree(v) | $m$ | 1 | 1 | $n$ |
| incidentEdges($v$) | $m$ | deg($v$) | deg($v$) | $n$ |
| areAdjacent ($v, w$) | $m$ | min(deg($v$), deg($w$)) | 1 (exp.) | 1 |
| insertVertex($o$) | 1 | 1 | 1 | $n^2$ |
| insertEdge($v, w, o$) | 1 | 1 | 1 (exp.) | 1 |
| removeVertex($v$) | $m$ | deg($v$) | deg($v$) | $n^2$ |
| removeEdge($e$) | 1 | 1 | 1 | 1 |

# Outline

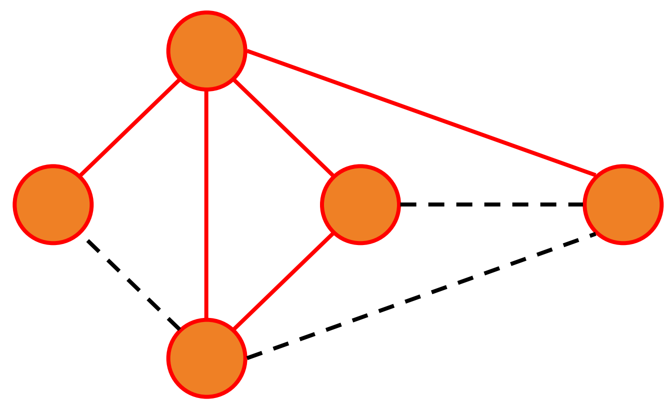
- Graphs: definitions and ADT
- Data structures for graphs
- **Graphs traversals**
  1. **More definitions**
  2. **DFS**
  3. **BFS**

# 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

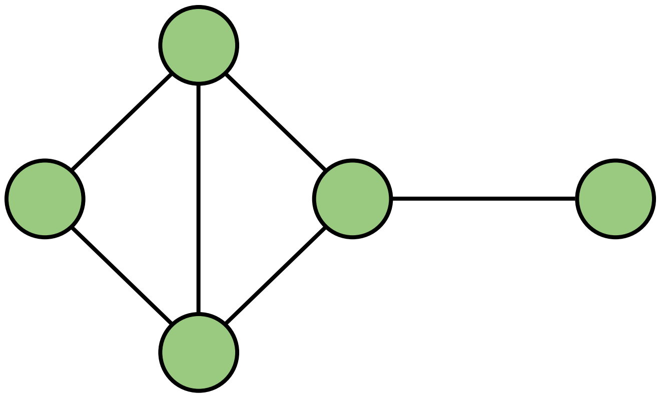


Spanning subgraph

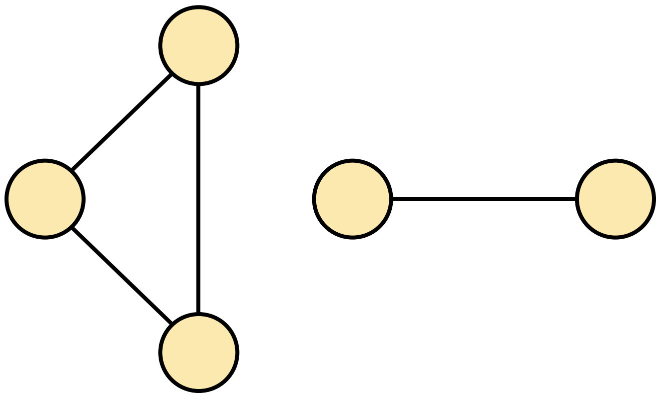
# Connectivity

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

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



Connected graph

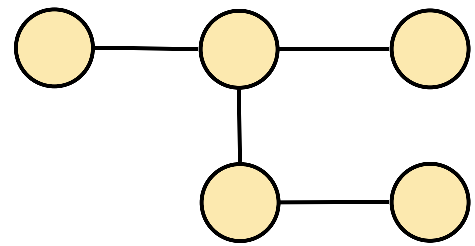


Non connected graph with two connected components

# Trees and Forests

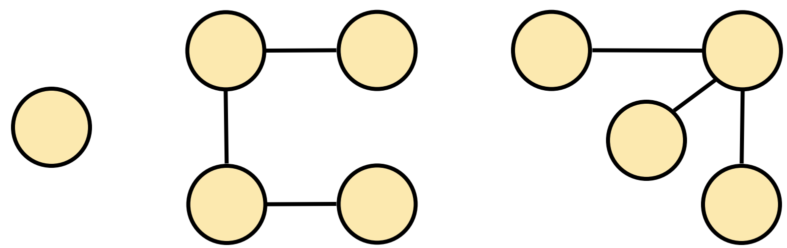
- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles

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



Tree

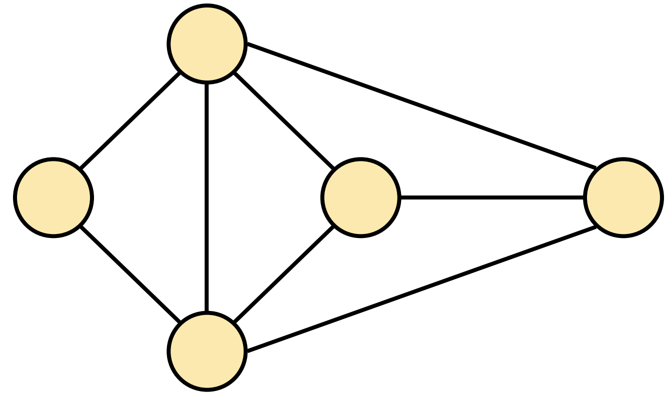
- A forest is an undirected graph without cycles
- The connected components of a forest are trees



Forest

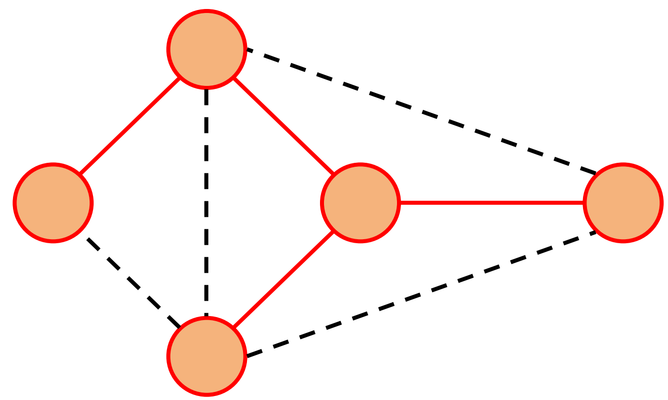
# Spanning Trees and Forests

- 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

- Properties for G (undirected) graph with n vertices and m edges
  - If G is connected then m>= n-1
  - If G is a (free) tree then m=n-1
  - If G is a forest then m<=n-1

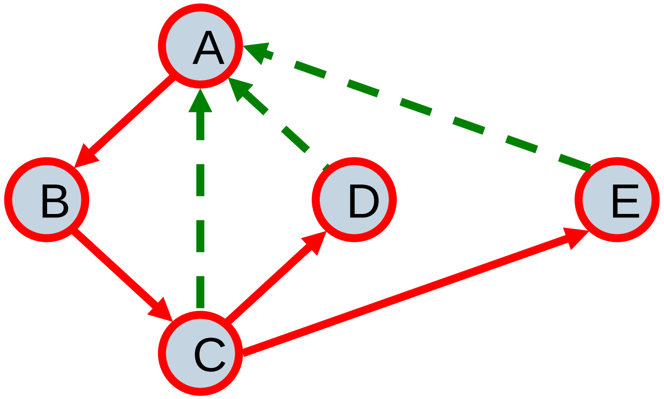


Spanning tree

# Graph Traversal: DFS, BFS

- Graph traversal algorithms are key to answering "Reachability" problems:
  - How to travel from one vertex to another following paths of a graph

- For example: given graph G (undirected/directed)
  - Find and report a path between two given vertices
  - Compute a shortest path (min # of edges) between two vertices
  - Test whether G is connected/strongly-connected
  - Computes a spanning tree/forest of G
  - Computes the (strongly) connected components of G
  - Find a cycle in the graph




# 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

- 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
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G

# DFS Algorithm from a starting vertex u

**Algorithm** DFS($G$, $u$):

*Input:* A graph $G$ and a vertex $u$ of $G$

*Output:* A collection of vertices reachable from $u$, with their discovery edges

Mark vertex $u$ as visited.

**for** each of $u$'s outgoing edges, $e = (u, v)$ **do**

  **if** vertex $v$ has not been visited **then**

    Record edge $e$ as the discovery edge for vertex $v$.

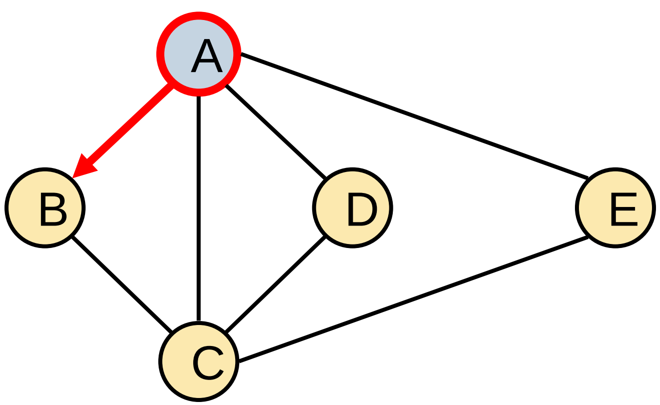
    Recursively call DFS($G$, $v$).

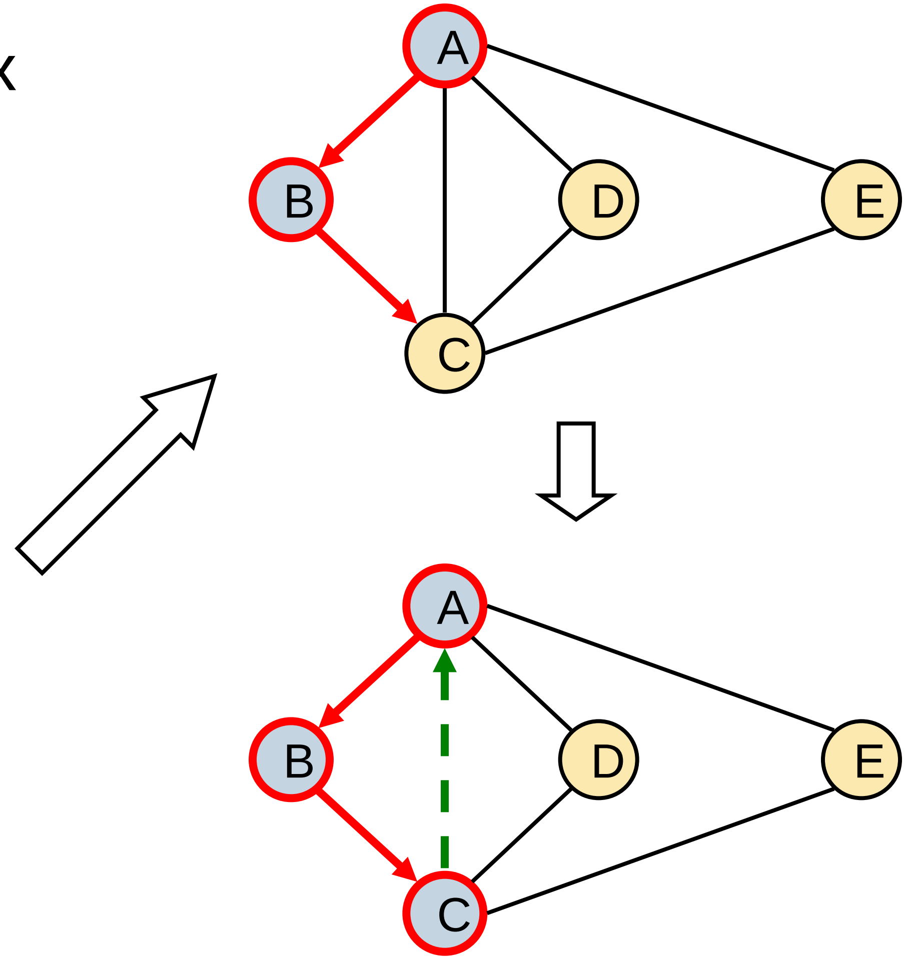
# Java implementation (fragment 14.5)

```
/** Performs depth-first search of Graph g starting at Vertex u. */
public static <V,E> void DFS(Graph<V,E> g, Vertex<V> u,
                      Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
  known.add(u);                              // u has been discovered
  for (Edge<E> e : g.outgoingEdges(u)) {     // for every outgoing edge from u
    Vertex<V> v = g.opposite(u, e);
    if (!known.contains(v)) {
      forest.put(v, e);                      // e is the tree edge that discovered v
      DFS(g, v, known, forest);              // recursively explore from v
    }
  }
}
```

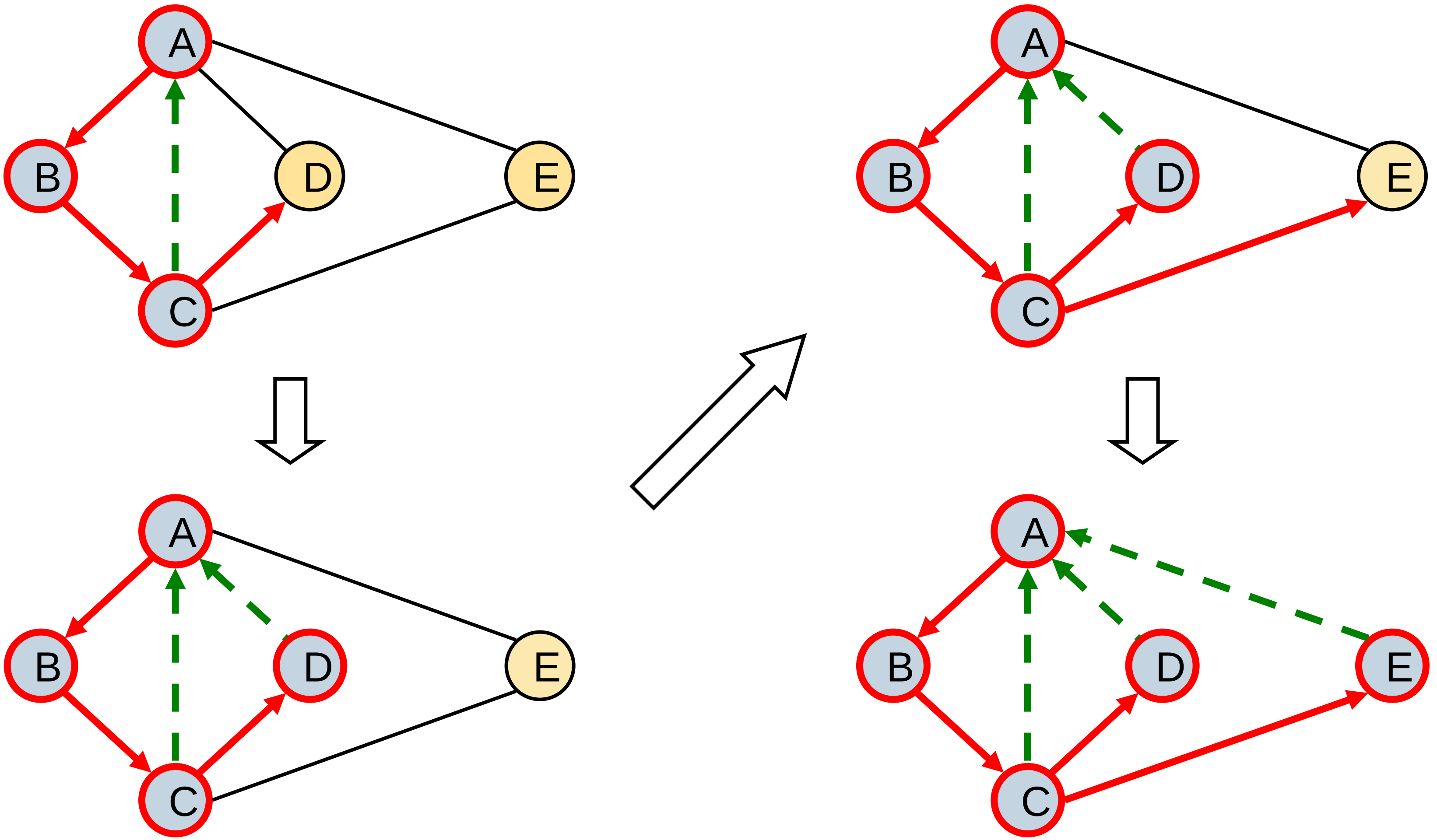
# Example

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



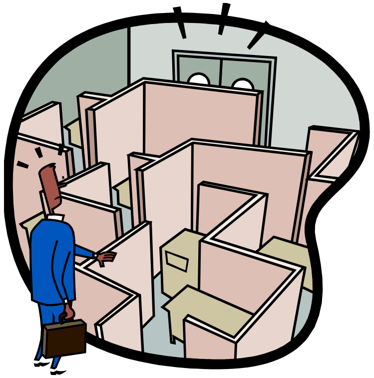



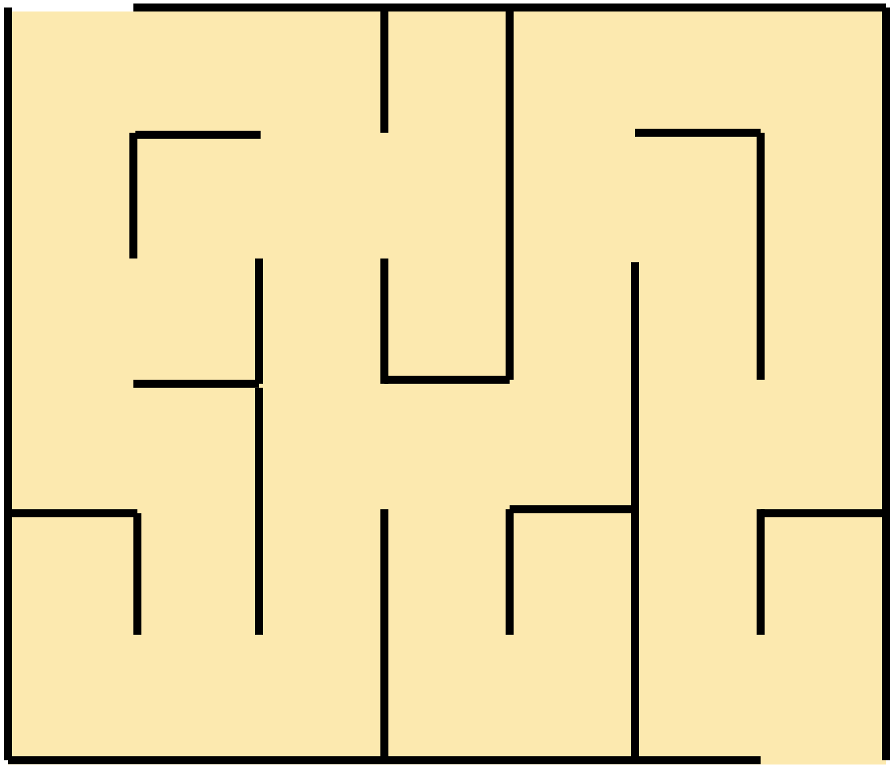
# Example (cont.)



# DFS and Maze Traversal

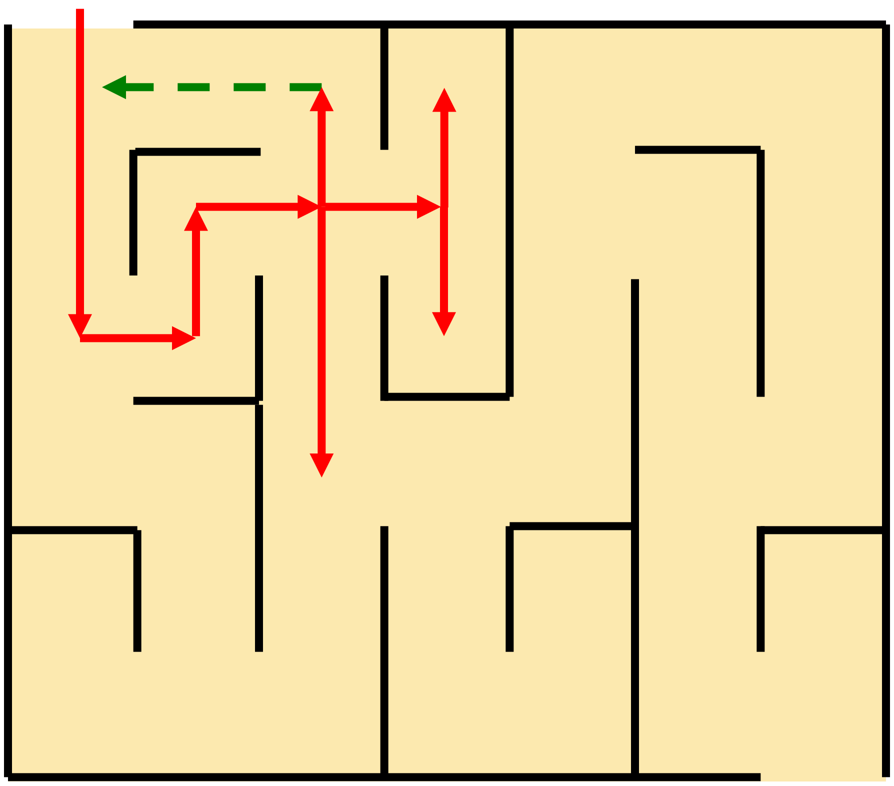
- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge ) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)





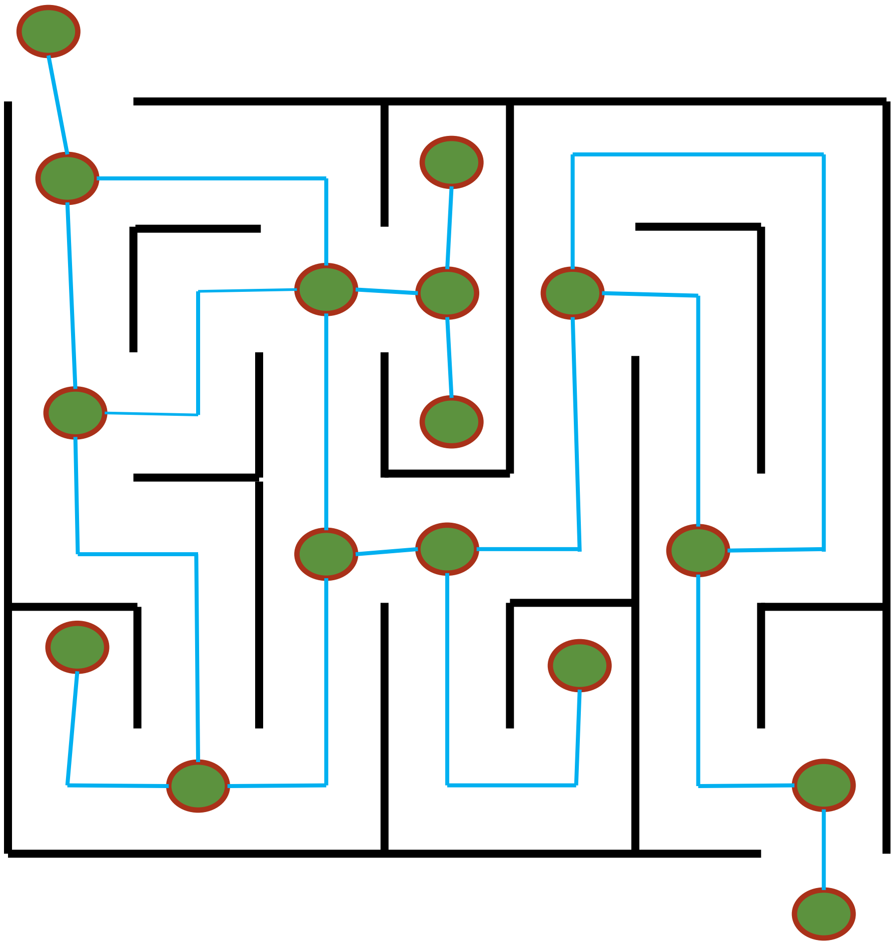
# DFS and Maze Traversal

- 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)



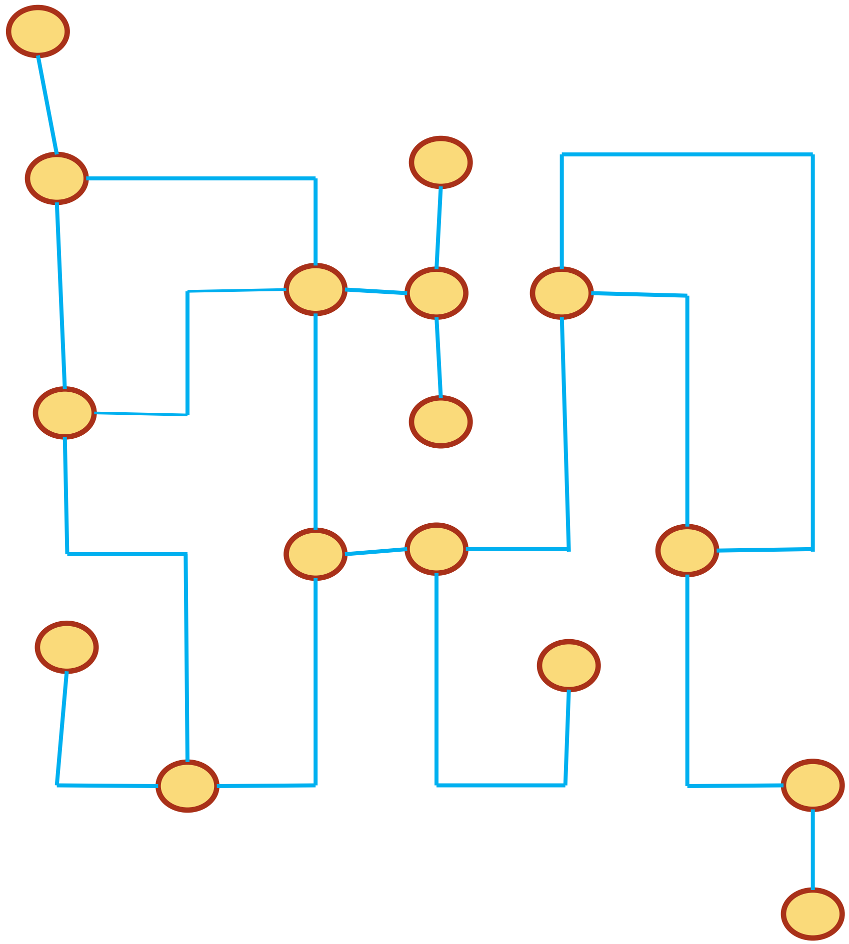
# DFS and Maze Traversal

- 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 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

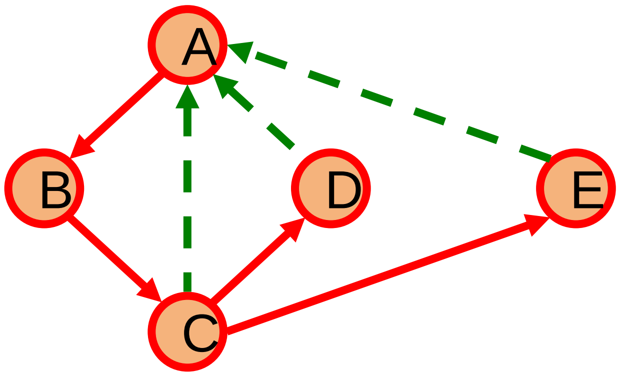
$DFS(G, v)$ visits all the vertices and edges in the connected component of $v$

## Property 2

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

* Directed graphs: DFS tree T

1. Tree edges: discovery edges
2. Non-tree edges
   - Back edges: to ancestor in T
   - Forward edges: to descendant in T
   - Cross edges



# Analysis of DFS

- Setting/getting a vertex/edge label takes $O(1)$ time
- DFS is called at most once for each vertex
- Each edge is examined at most twice (once for directed graph)
- 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 (outgoingEdges(v)) takes O(deg(v)) time
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

```
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)
```

# DFS Application

- DFS can be further extended to solve other graph problems in $O(n + m)$ time
  - Find and report a path between two given vertices
  - Find a cycle in the graph
  - Test whether G is connected
  - Computes the connected components of G
  - Computes a spanning tree of G (if G is connected)

- DFS Implementation (see DFS method in Section 14.3.2)
  - Keep track of DFS tree: two auxiliary data structures
    - a set (known): vertices that have already been visited
    - a map (forest) : associates a vertex v and edge e (used to discover v)
  - Hash-based implementation: O(1) expected time (mark vertex v "explored")

# 1. 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)
    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)
```

# Path Finding in Java

```
/** Returns an ordered list of edges comprising the directed path from u to v. */
public static <V,E> PositionalList<Edge<E>>
constructPath(Graph<V,E> g, Vertex<V> u, Vertex<V> v,
                  Map<Vertex<V>,Edge<E>> forest) {
  PositionalList<Edge<E>> path = new LinkedPositionalList<>( );
  if (forest.get(v) != null) {                 // v was discovered during the search
    Vertex<V> walk = v;                        // we construct the path from back to front
    while (walk != u) {
      Edge<E> edge = forest.get(walk);
      path.addFirst(edge);                     // add edge to *front* of path
      walk = g.opposite(walk, edge);           // repeat with opposite endpoint
    }
  }
  return path;
}
```

# 2. 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

- A cycle exists iff a back edge exists

- 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 for an Entire Graph

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

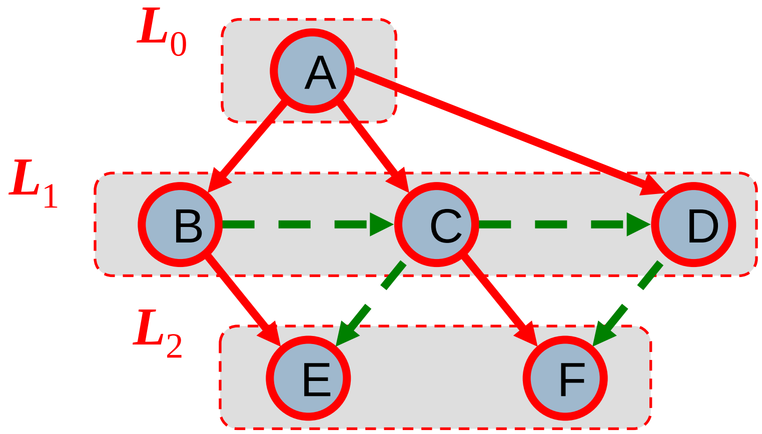
**Algorithm** ***DFS*(*G*)**
    **Input** graph ***G***
    **Output** labeling of the edges of ***G***
        as discovery edges and
        back edges
  **for all** ***u*** $\in$ ***G.vertices*()**
   ***setLabel*(*u, UNEXPLORED*)**
  **for all** ***e*** $\in$ ***G.edges*()**
   ***setLabel*(*e, UNEXPLORED*)**
  **for all** ***v*** $\in$ ***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*** $\in$ ***G.incidentEdges*(*v*)**
    **if** ***getLabel*(*e*)** = ***UNEXPLORED***
      ***w*** $\leftarrow$ ***opposite*(*v,e*)**
      **if** ***getLabel*(*w*)** = ***UNEXPLORED***
        ***setLabel*(*e, DISCOVERY*)**
        ***DFS*(*G, w*)**
      **else**
        ***setLabel*(*e, BACK*)**

# 3. All Connected Components

– Loop over all vertices, doing a DFS from each unvisited one.

```
/** Performs DFS for the entire graph and returns the DFS forest as a map. */
public static <V,E> Map<Vertex<V>,Edge<E>> DFSComplete(Graph<V,E> g) {
  Set<Vertex<V>> known = new HashSet<>( );
  Map<Vertex<V>,Edge<E>> forest = new ProbeHashMap<>( );
  for (Vertex<V> u : g.vertices( ))
    if (!known.contains(u))
      DFS(g, u, known, forest);                // (re)start the DFS process at u
  return forest;
}
```




# 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
  - Subdivides vertices into levels *Li*
- BFS on a graph with $n$ vertices and $m$ edges takes <u>$O(n + m)$ time</u>

- BFS can be further extended to solve other graph problems
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
  - Find and report a shortest path (<u>minimum # of edges)</u> between two vertices
  - 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()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
        BFS(G, v)
```

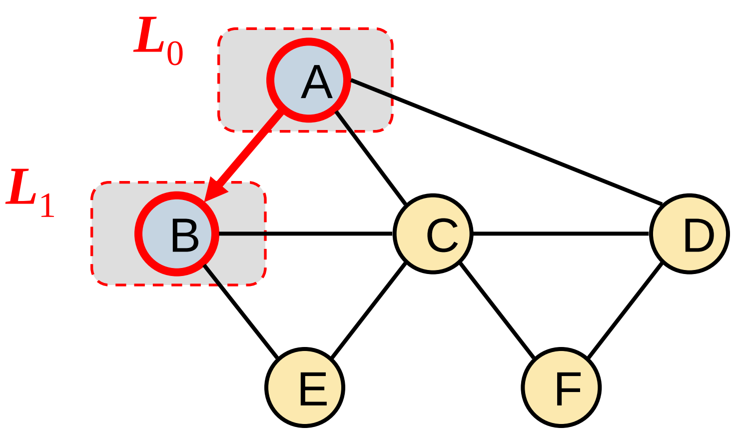
```
Algorithm BFS(G, s)
  L0 ← new empty sequence
  L0.addLast(s)
  setLabel(s, VISITED)
  i ← 0
  while ¬Li.isEmpty()
    Li+1 ← new empty sequence
    for all v ∈ Li.elements()
      for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w)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

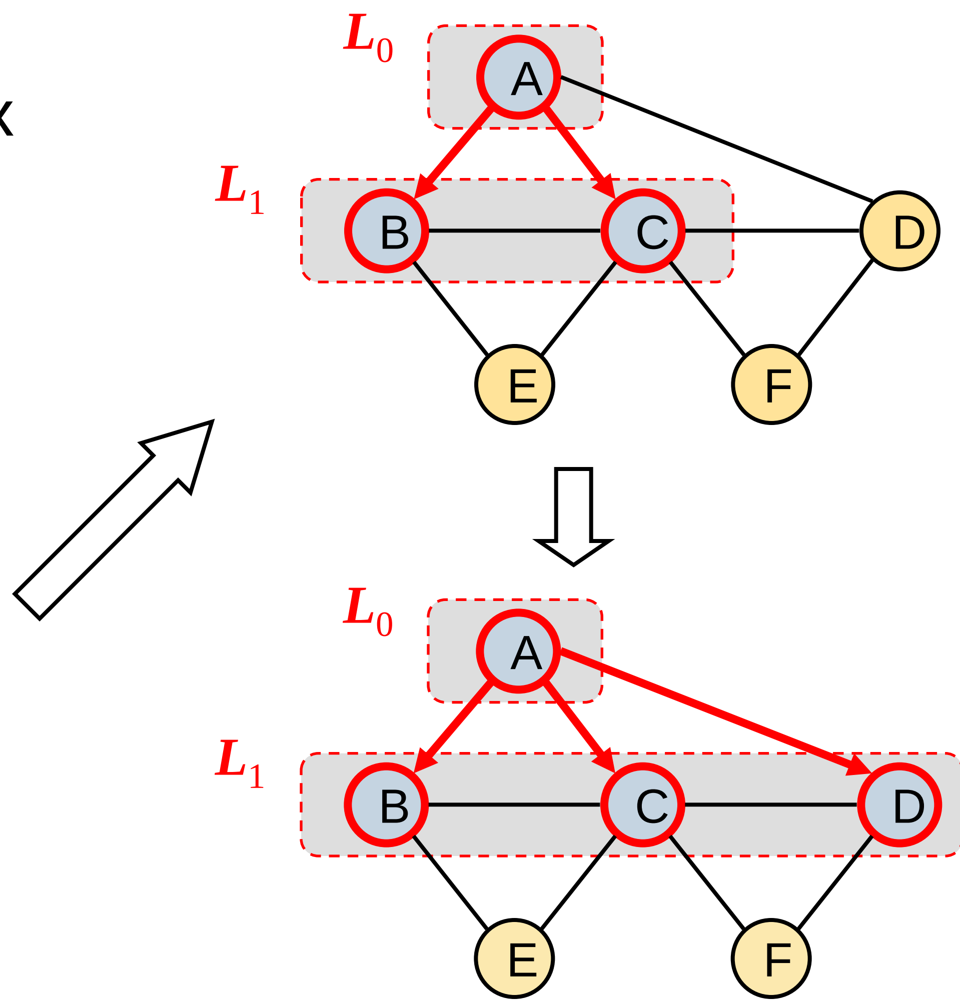
# Java Implementation (fragment 14.8)

```
/** Performs breadth-first search of Graph g starting at Vertex u. */
public static <V,E> void BFS(Graph<V,E> g, Vertex<V> s,
                    Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
  PositionalList<Vertex<V>> level = new LinkedPositionalList<>( );
  known.add(s);
  level.addLast(s);                          // first level includes only s
  while (!level.isEmpty( )) {
    PositionalList<Vertex<V>> nextLevel = new LinkedPositionalList<>( );
    for (Vertex<V> u : level)
      for (Edge<E> e : g.outgoingEdges(u)) {
        Vertex<V> v = g.opposite(u, e);
        if (!known.contains(v)) {
          known.add(v);
          forest.put(v, e);                  // e is the tree edge that discovered v
          nextLevel.addLast(v);              // v will be further considered in next pass
        }
      }
    level = nextLevel;                       // relabel 'next' level to become the current
  }
}
```

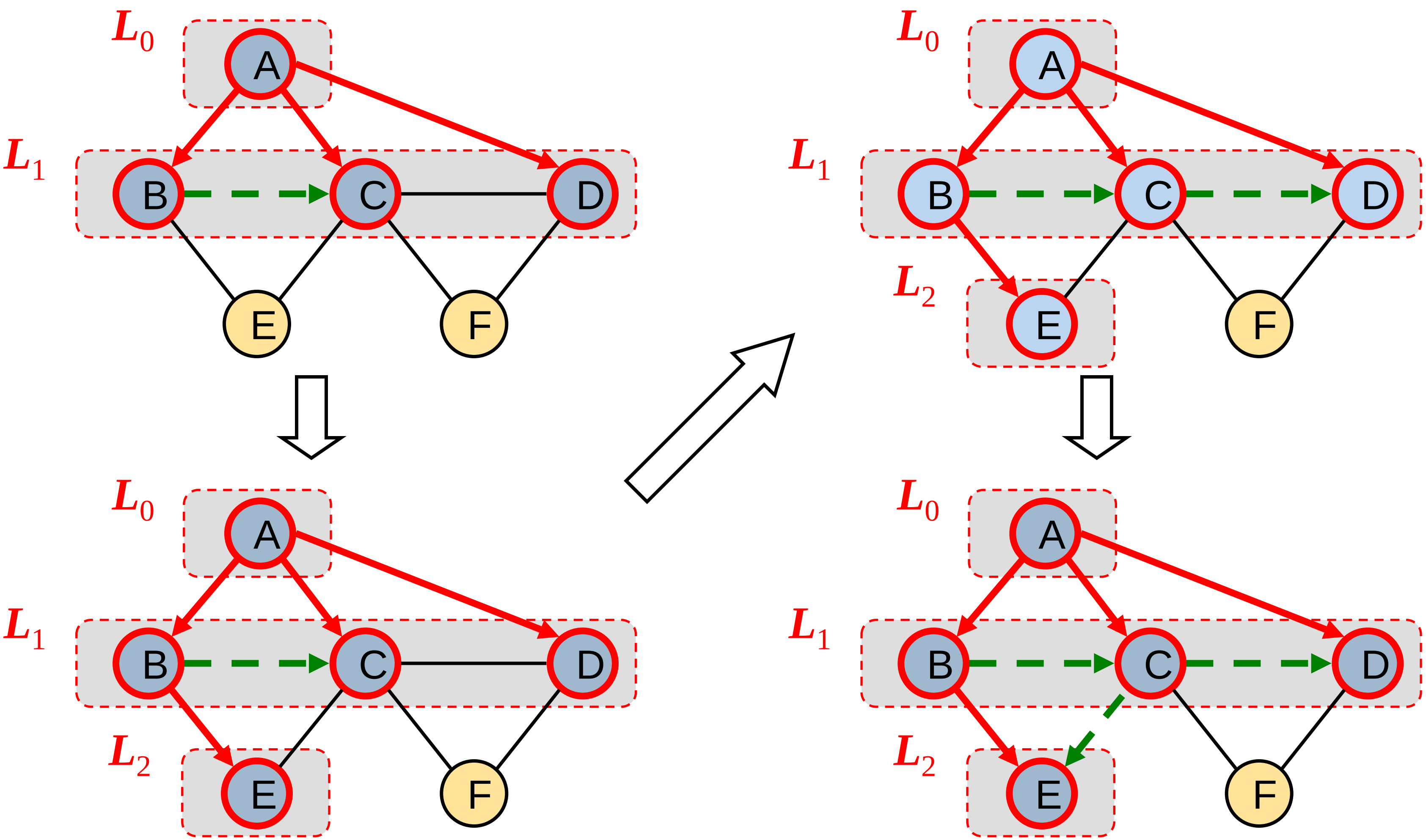
# Example

- A unexplored vertex
- A visited vertex
- unexplored edge
- **discovery edge**
- **cross edge**

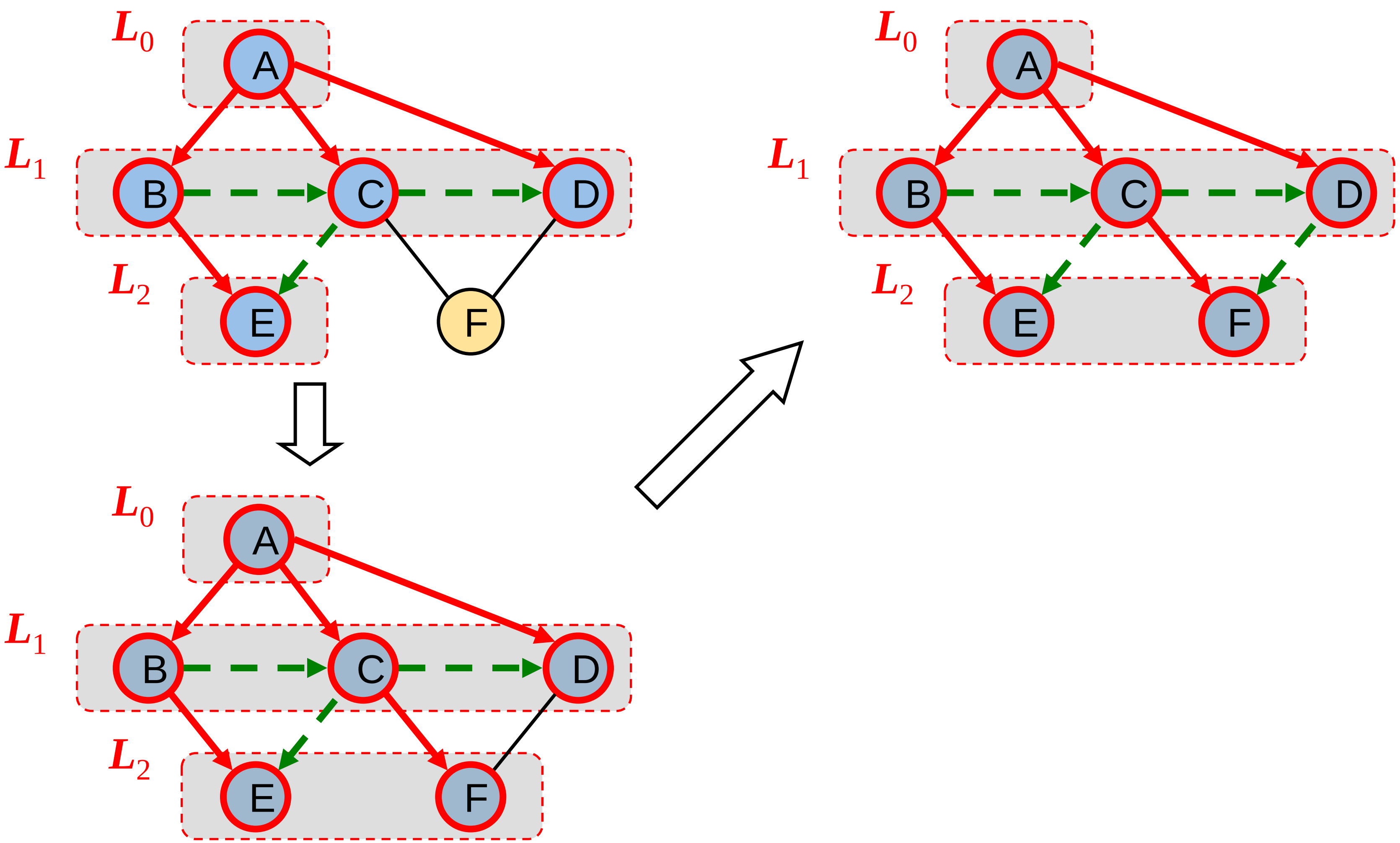


# Example (cont.)



# Example (cont.)



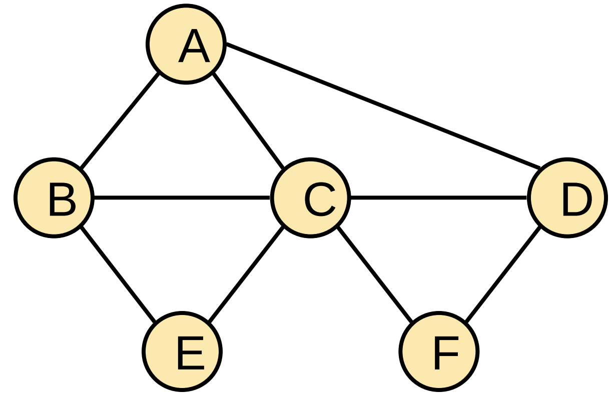
# Properties

## Notation

$G_s$: connected component of $s$

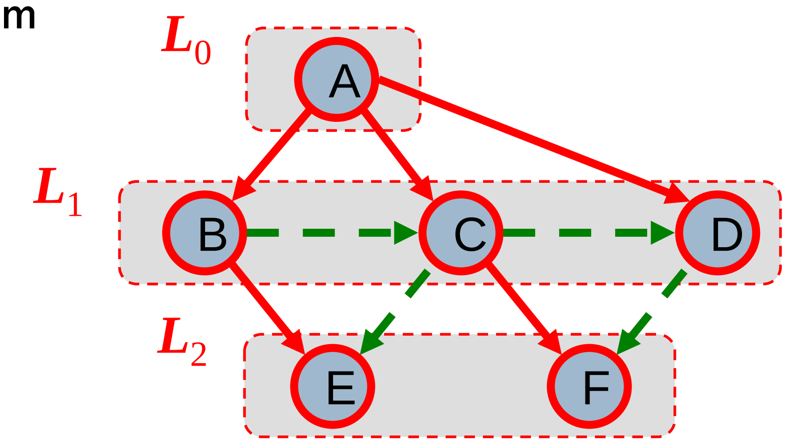
## Property 1

$BFS(G, s)$ 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 (shortest path)
- Every path from $s$ to $v$ in $G_s$ has at least $i$ edges

# Analysis

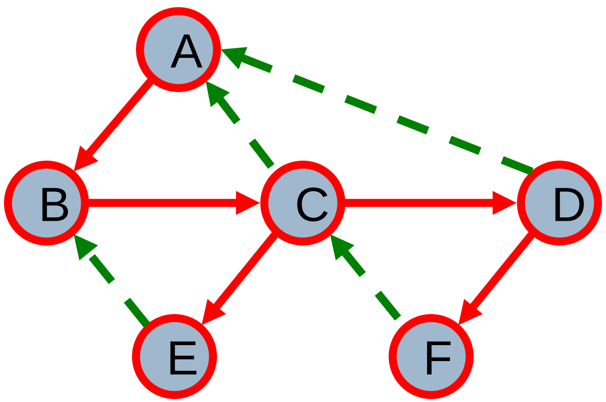
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence $L_i$
- Method incidentEdges is called once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

# Applications

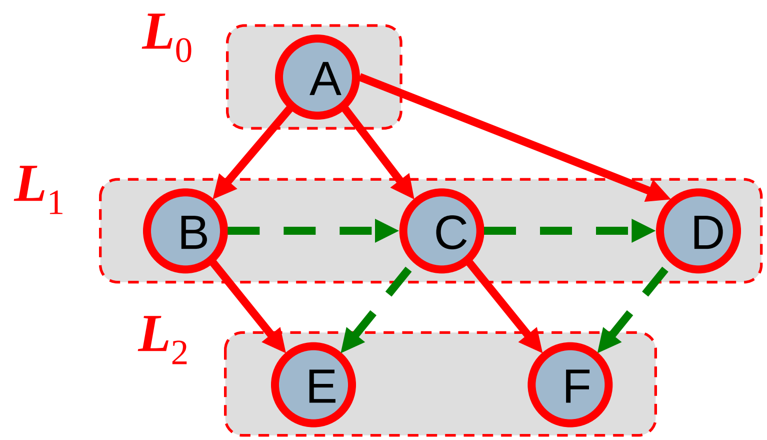
- we can specialise the BFS traversal of a graph $G$ to solve the following problems in $O(n + m)$ time
- using the template method pattern (set, map): similar to DFS
  - Compute the connected components of $G$
  - Compute a spanning forest of $G$
  - Find a simple cycle in $G$, or report that $G$ is a forest
  - Find a shortest path (minimum number of edges) between two vertices, or report that no such path exists

# DFS vs. BFS

| Applications | DFS | BFS |
|---|---|---|
| Spanning forest, connected components, paths, cycles (undirected graph) | √ | √ |
| Shortest paths | | √ |
| Strongly-connected components<br>Directed cycle (directed graph) | √ | |




DFS




BFS

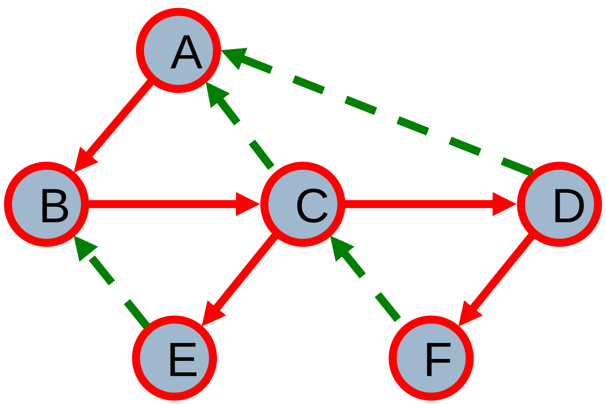
# DFS vs. BFS (cont.)

**Back edge** $(v,w)$

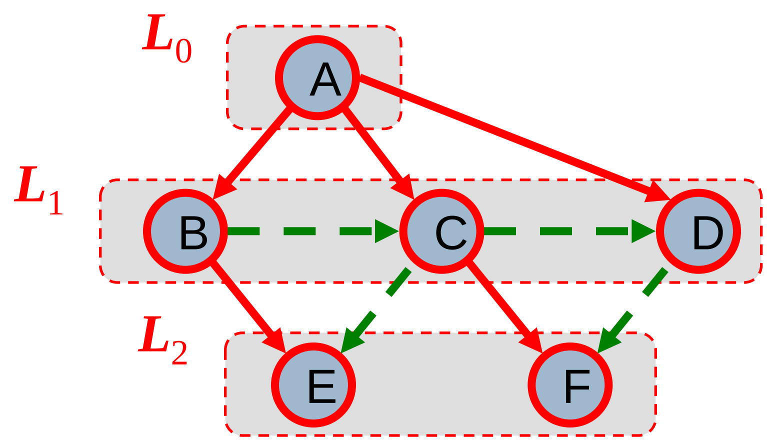
– $w$ is an ancestor of $v$ in the tree of discovery edges

**Cross edge** $(v,w)$

– $w$ is in the same level as $v$ or in the next level




DFS




BFS

# Outline

- **Graphs (section 14.1)**
  1. Definitions
  2. Graph ADT

- **Data structures for graphs (section 14.2)**
  1. Edge list structure
  2. Adjacency list structure
  3. Adjacency map structure
  4. Adjacency matrix

- **Graphs traversals (section 14.3)**
  1. DFS
  2. BFS
  3. Applications