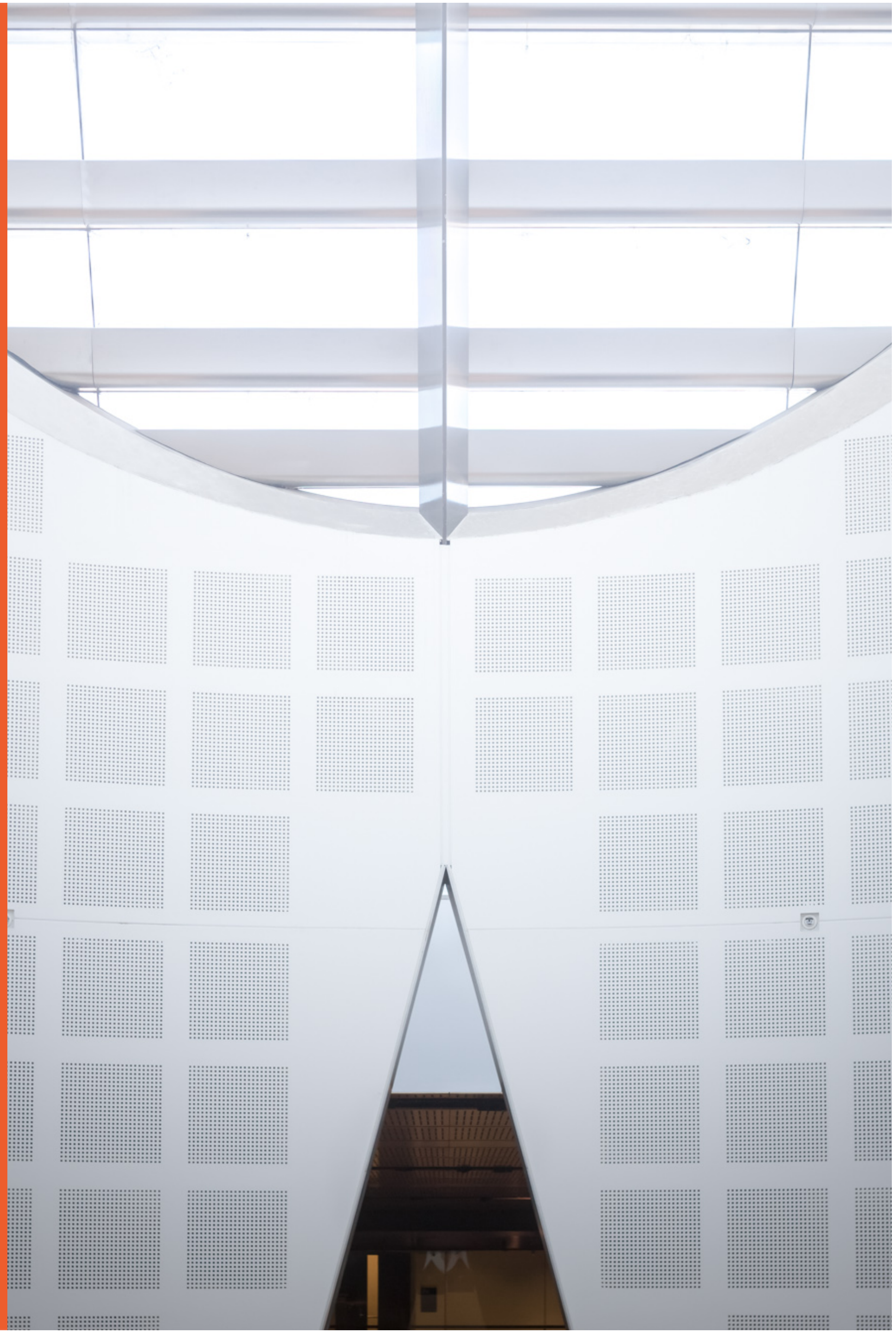
INFO1105/1905
Data Structures
Week 7: Graphs (start)
see textbook section 14.1, 14.2, 14.3

Professor Alan Fekete
Dr John Stavrakakis
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 (5^{th} & 6^{th} edition)
- With modifications and additions from the University of Sydney

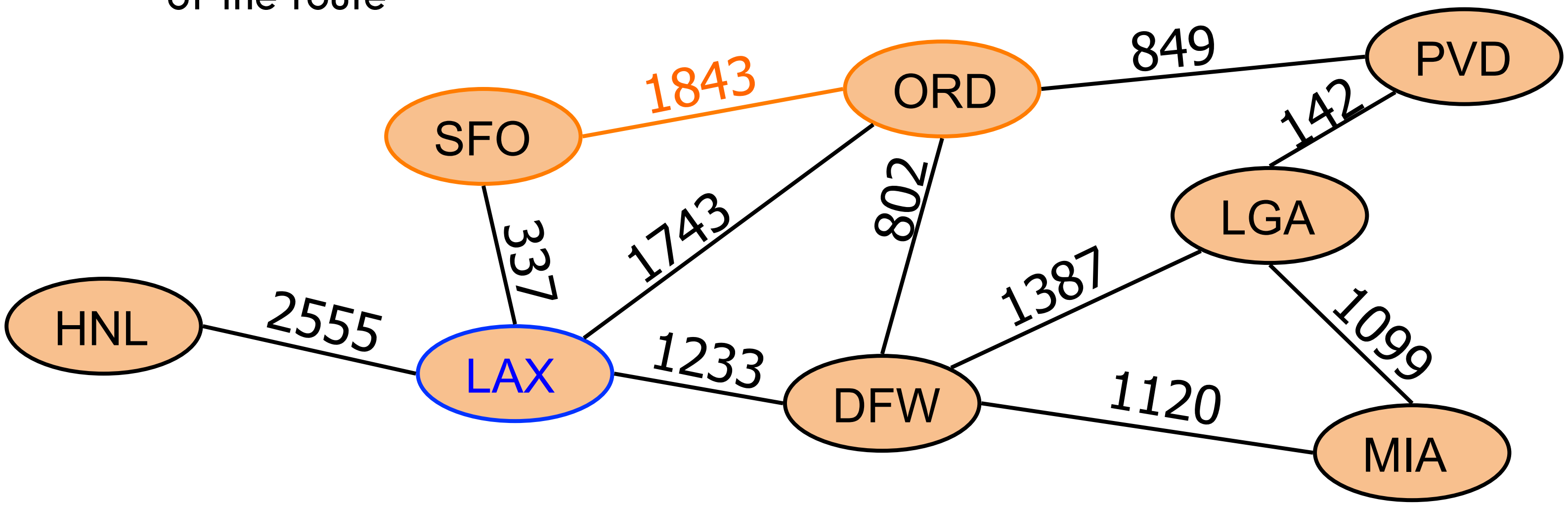
- The slides are a guide or overview of some big ideas
  - Students are responsible for knowing what is in the referenced sections of the textbook, not just what is in the slides

Outline

- Graphs: definitions and ADT
- Data structures for graphs
- Graphs traversals

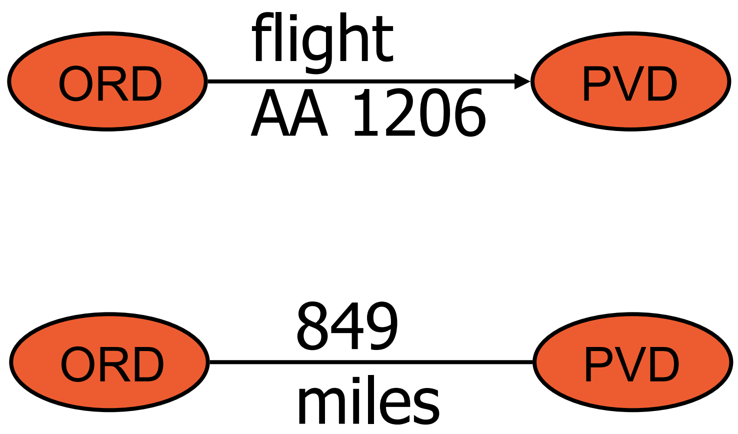
Graphs

- 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



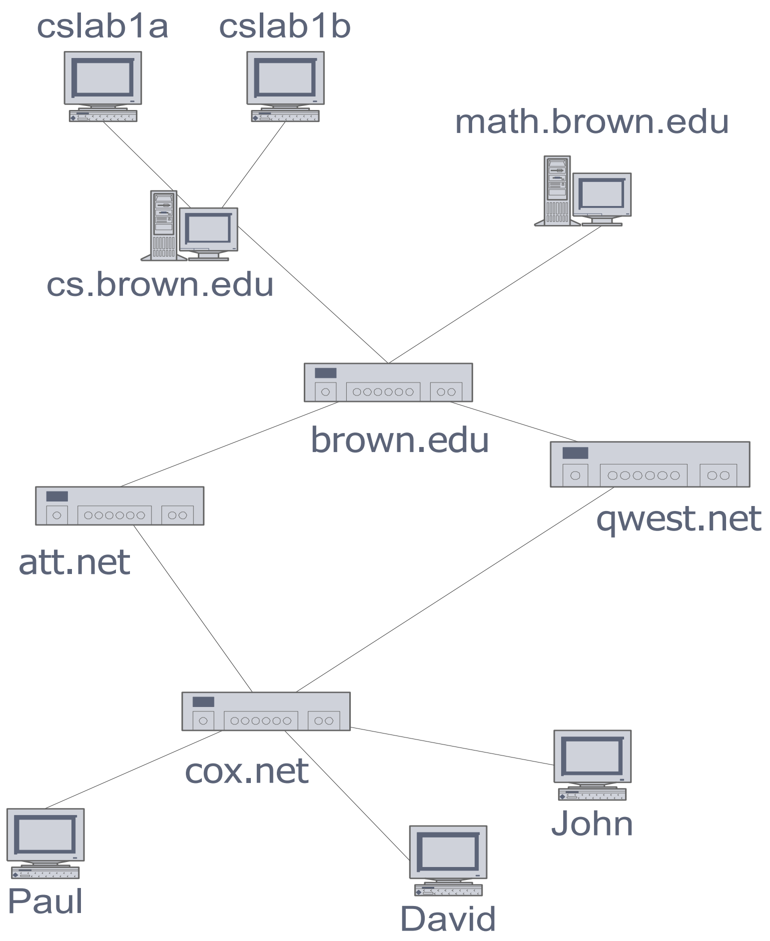
Edge Types

- Directed edge
  - 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 coauthorship relationship
- Directed graph
  - all the edges are directed
  - e.g., route network
- Undirected graph
  - all the edges are undirected
  - e.g., collaboration network



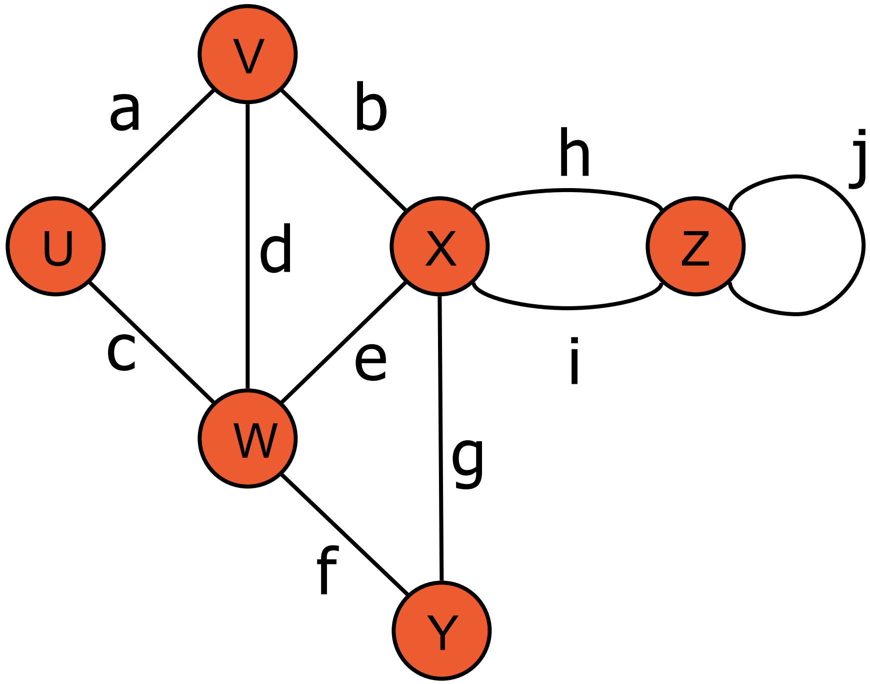
Applications

- 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



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
- Simple graph
  - no parallel edges or self-loops

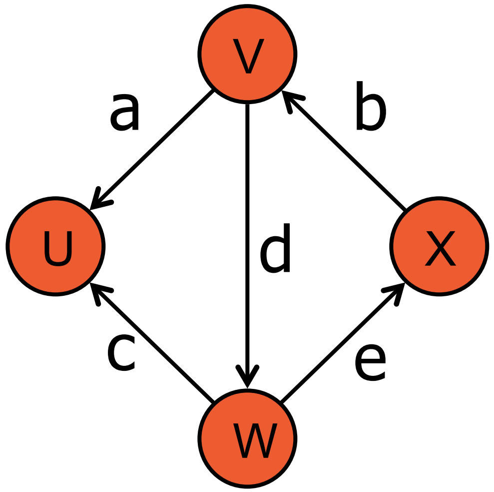


Terminology (cont.)

If edge is directed

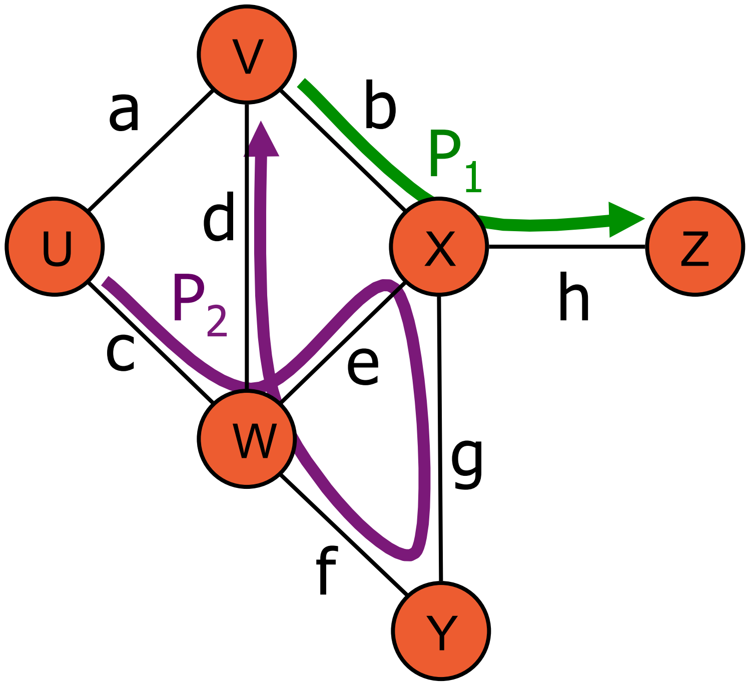
- Origin, destination vertices
- Outgoing edges of V are a,d
- Incoming edge of V is b

- Degree of a vertex
  - deg(V) is 3
  - indeg(V) is 1
  - outdeg(V) is 2



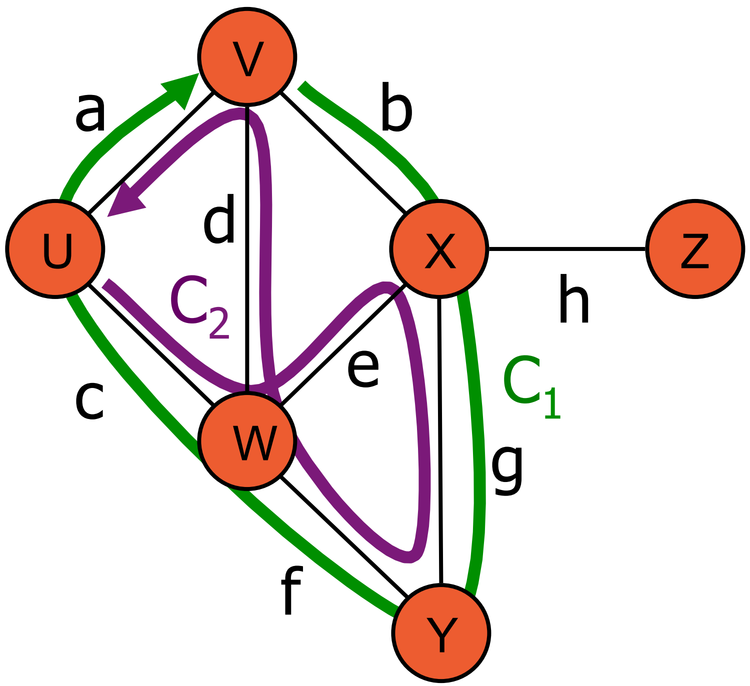
Terminology (cont.)

- 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
- In directed graph, directed paths



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
- 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

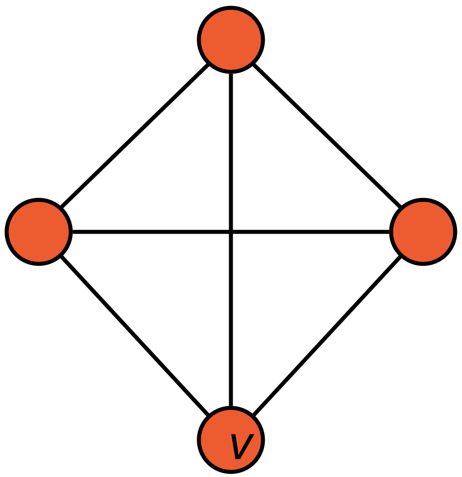
$m \leq n\,(n - 1)/2$

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

What is the bound for a directed simple graph?

Notation

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



Example

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

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.

Graph ADT

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.

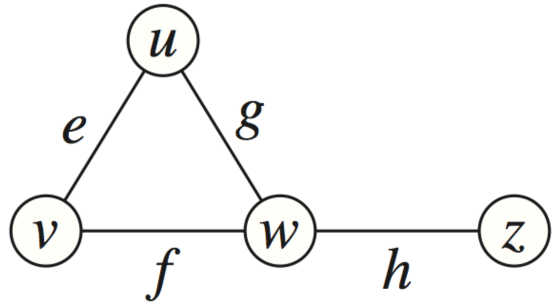
removeEdge(e): Removes edge e from the graph.

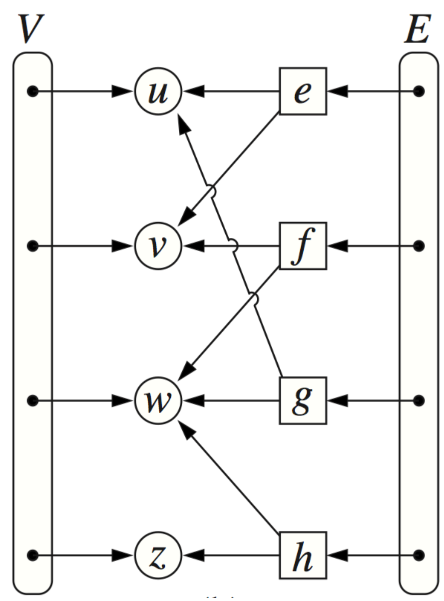
Outline

- Graphs: definitions and ADT
- Data structures for graphs
  - Edge list structure
  - Adjacency list structure
  - Adjacency map structure
  - Adjacency matrix
- Graphs traversals

Edge List Structure

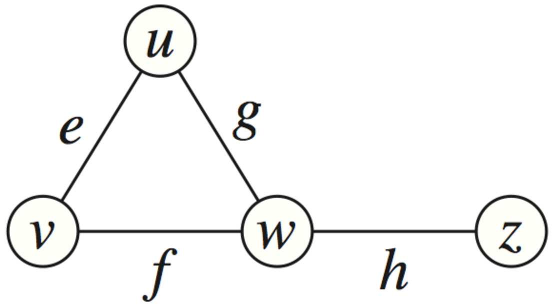
- Unordered list of all edges
- Vertex object
  - element
  - reference to position in vertex sequence
- Edge object
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- Vertex sequence V
  - sequence of vertex objects
- Edge sequence E
  - sequence of edge objects

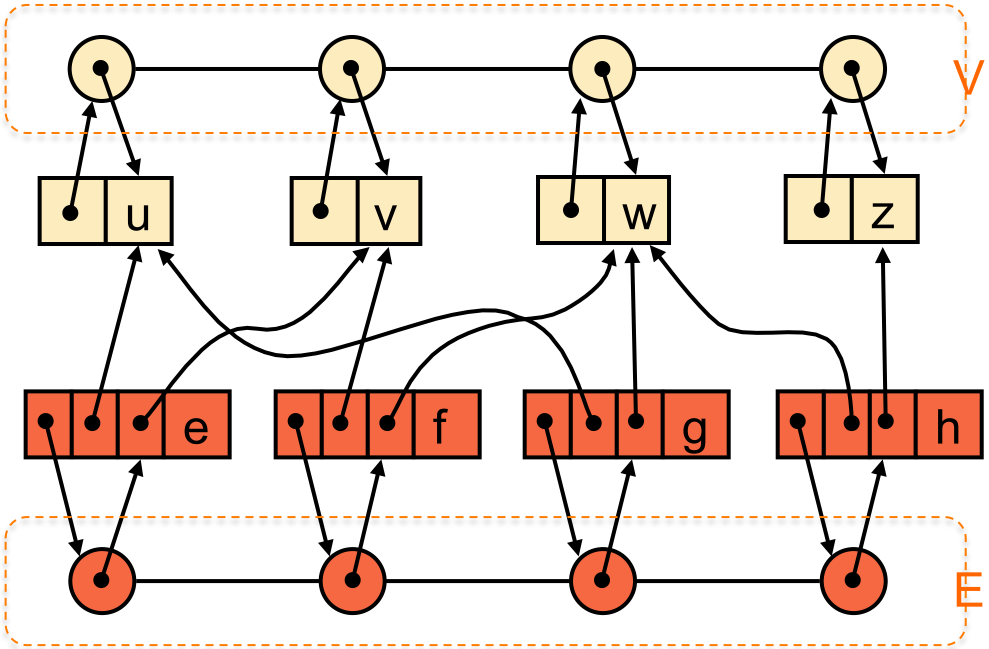




Edge List Structure

- 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

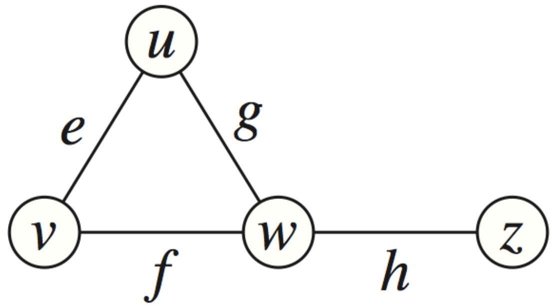
▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List
Space	
incidentEdges(v)	
areAdjacent (v, w)	
insertVertex(o)	
insertEdge(v, w, o)	
removeVertex(v)	
removeEdge(e)	

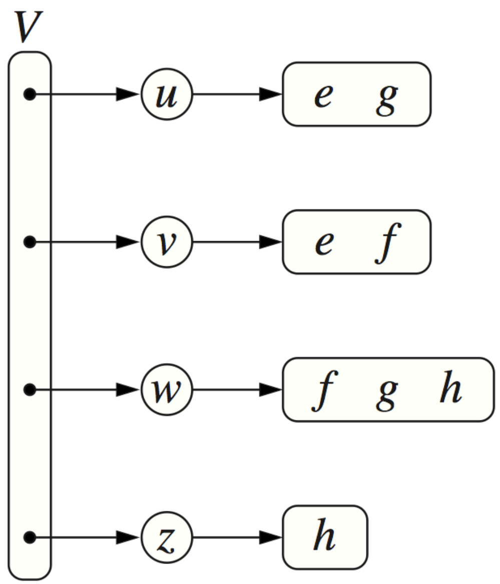
Edge list: performance

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

Adjacency List Structure

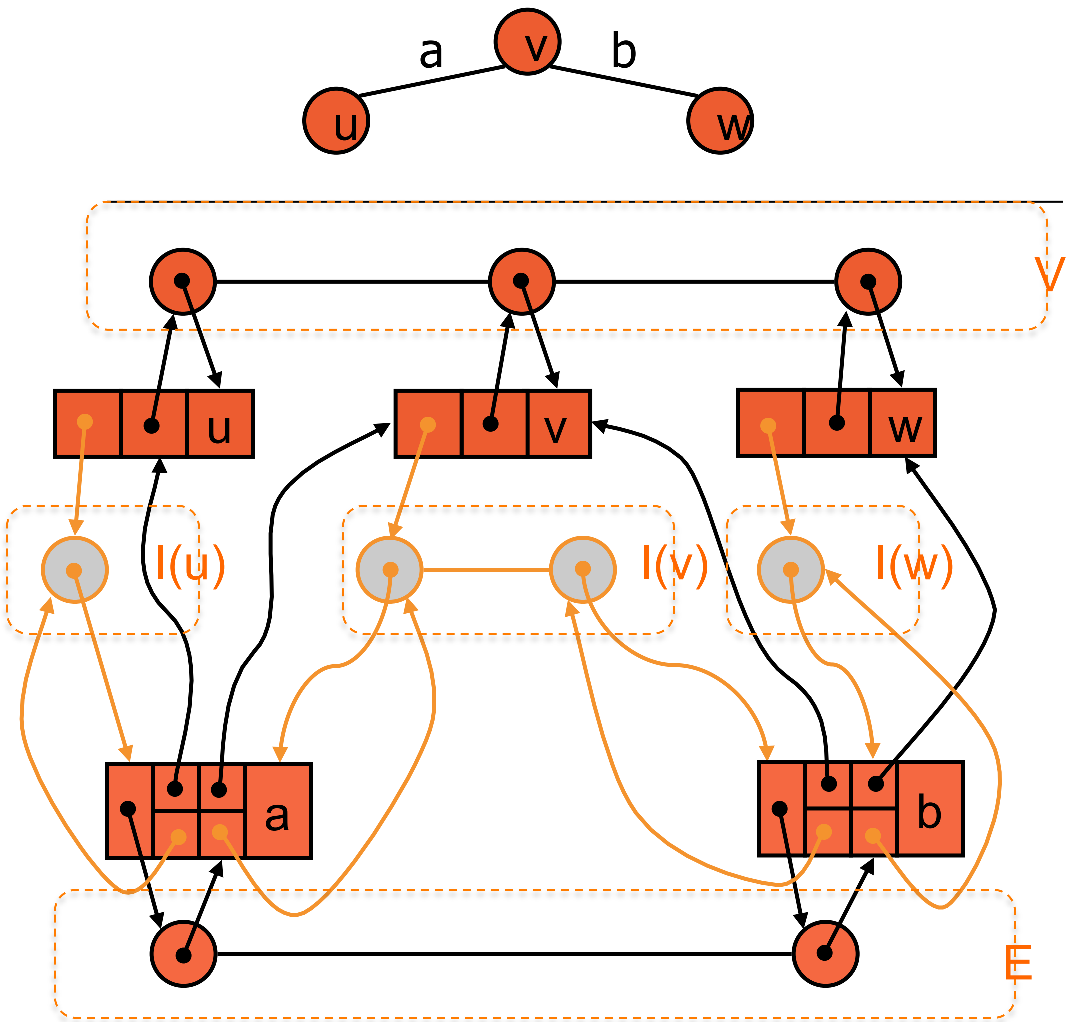
- Unordered list with additional list structure
- Incidence sequence for each vertex v
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices





Adjacency List Structure

- Unordered list with additional list structure
- Incidence sequence I(v) 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



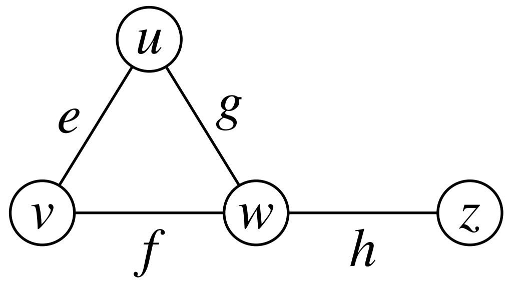
Adjacency list: performance

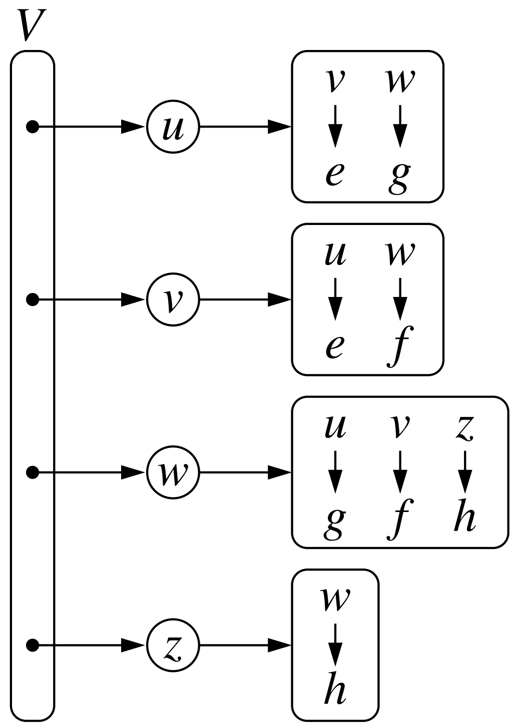
– All methods in O(1) with Edge list structure still O(1)

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

Adjacency Map Structure

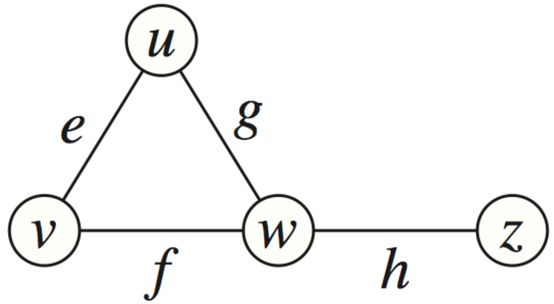
- Same as adjacency list, but uses a hash-based map for storing incident edges (see week 9)
- Incidence map for each vertex v
  - Key = opposite endpoint Value =edge
- getEdge(u,v) now is in *expected* O(1)
  - Although still worst case O(min(deg(u),deg(v)))





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



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

Adjacency Matrix: performance

n vertices, m edges - no parallel edges - no self-loops	Adjacency Matrix
Space	n^2
incidentEdges(v)	n
areAdjacent (v, w)	1
insertVertex(o)	$\underline{n^2}$
insertEdge(v, w, o)	1
removeVertex(v)	$\underline{n^2}$
removeEdge(e)	1

Performance

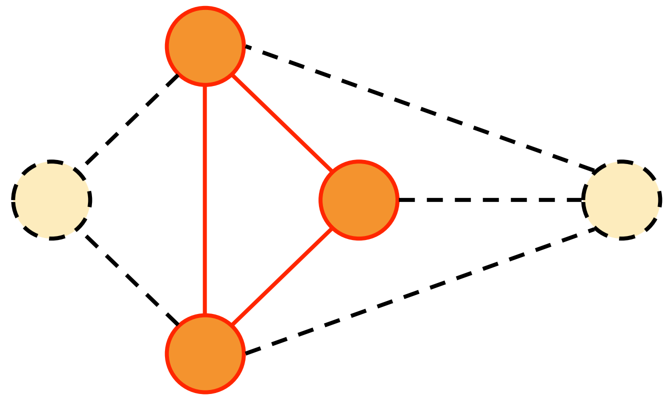
▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Map	Adjacency Matrix
Space	$n + m$	$n + m$	$n + m$	n^2
getEdge(u,v)	m	min(deg(v), deg(w))	1(exp.)	1
outDegree(v), inDegree(v)	m	1	1	n
incidentEdges(v)	m	deg(v)	deg(v)	n
areAdjacent (v, w)	m	min(deg(v), deg(w))	min(deg(v), deg(w))	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

Outline

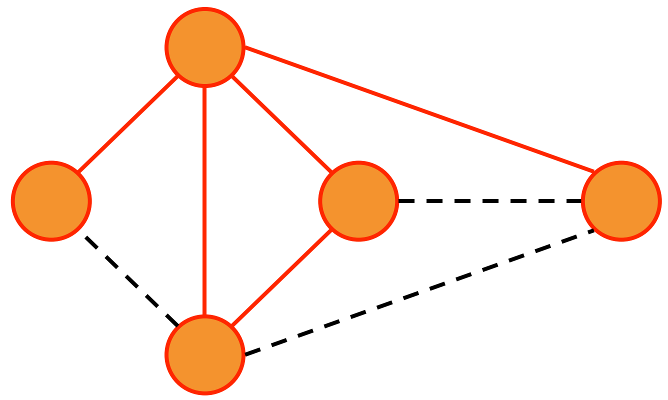
- Graphs: definitions and ADT
- Data structures for graphs
- Graphs traversals
  - More definitions
  - DFS
  - BFS

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



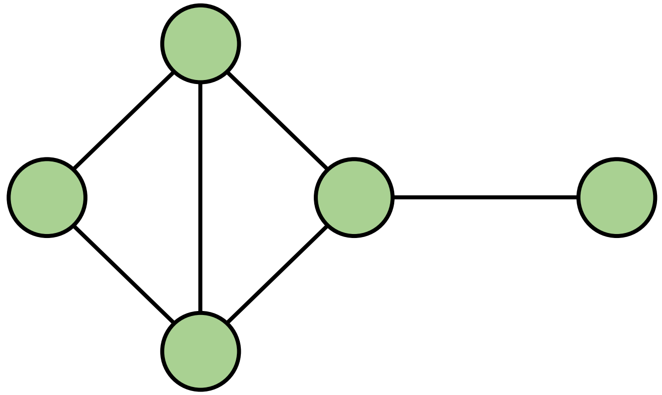
Subgraph



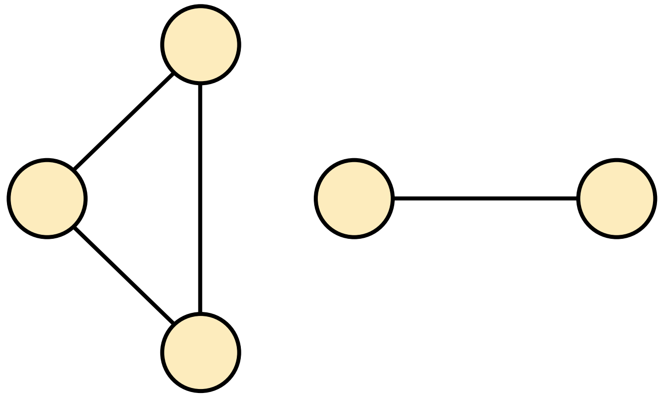
Spanning subgraph

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

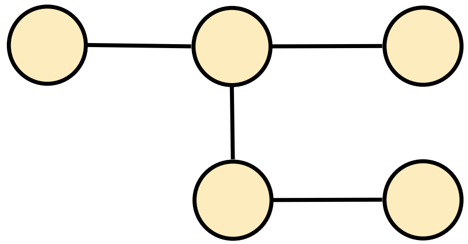


Non connected graph with two connected components

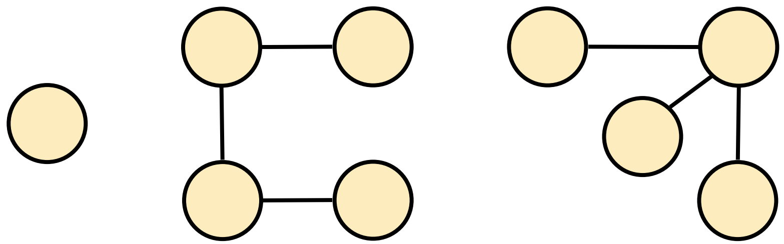
Trees and Forests

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



Tree

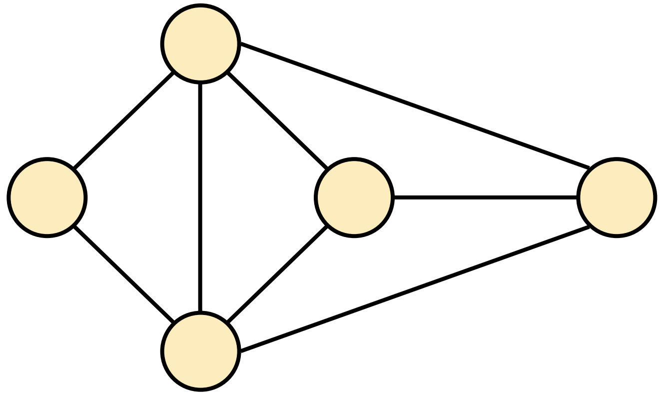


Forest

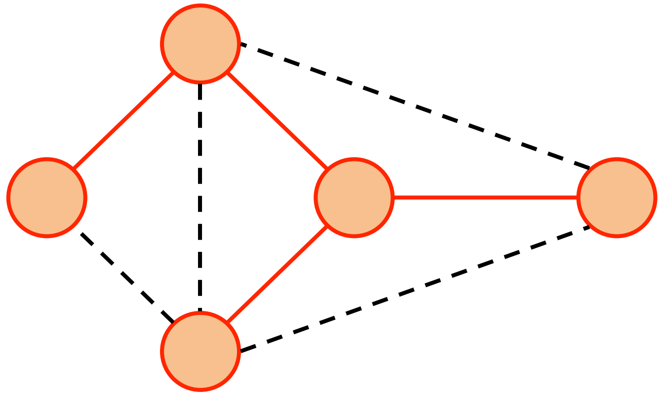
Spanning Trees and Forests

- 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

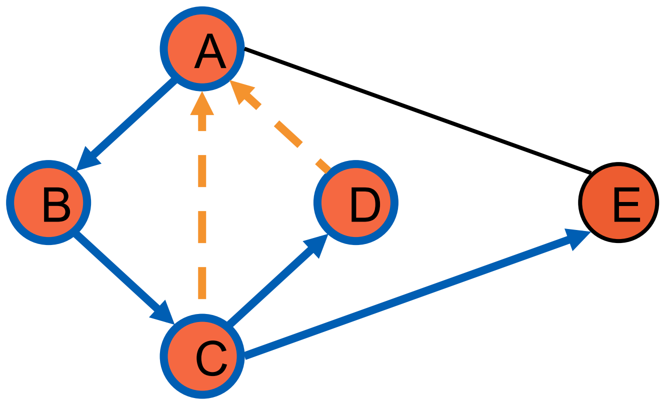
- Properties for G (undirected) graph with n vertices and m edges
  - If G is connected then m>= n-1
  - If G is a (free) tree then m=n-1
  - If G is a forest then m<=n-1



Graph



Spanning tree



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

DFS Algorithm from a Vertex

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**

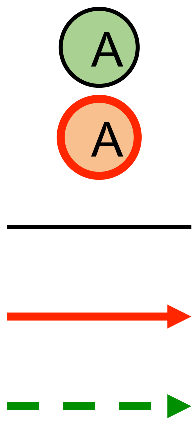
    Record edge e as the discovery edge for vertex v.

    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



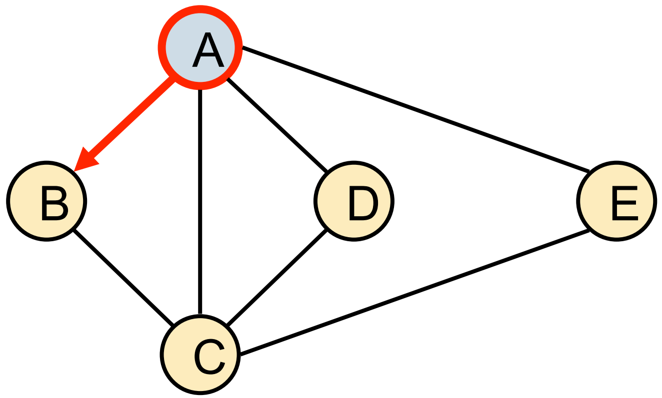
unexplored vertex

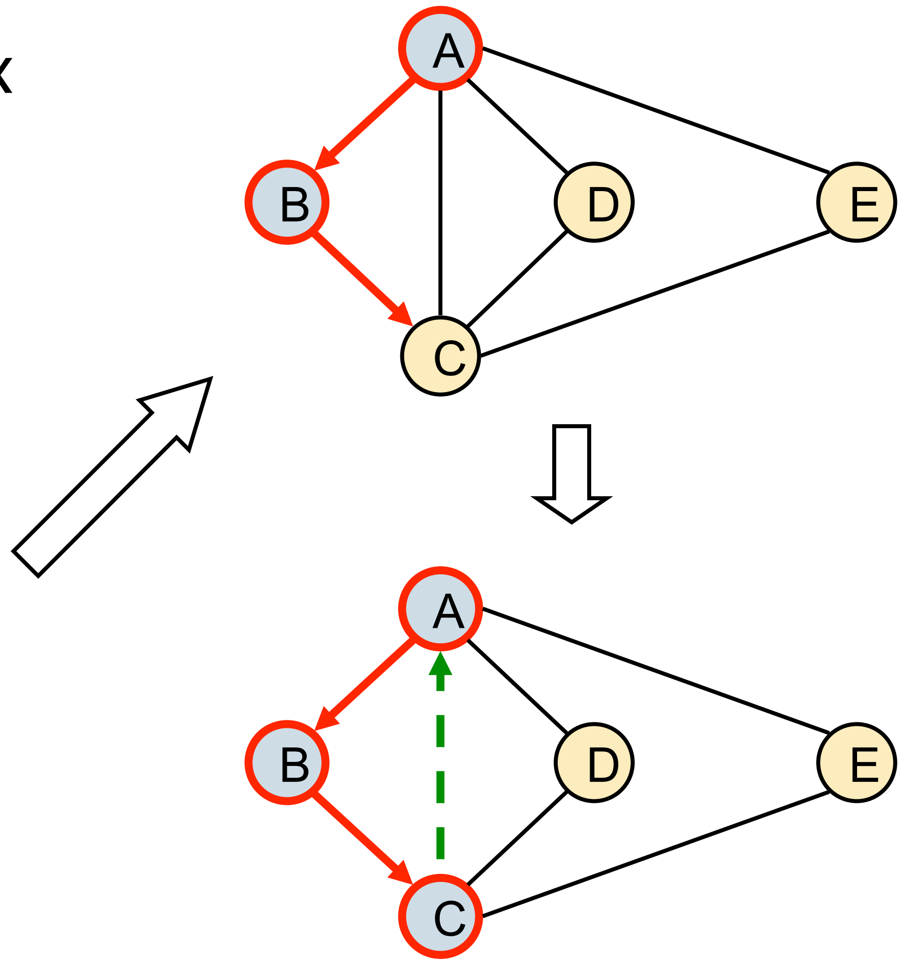
visited vertex

unexplored edge

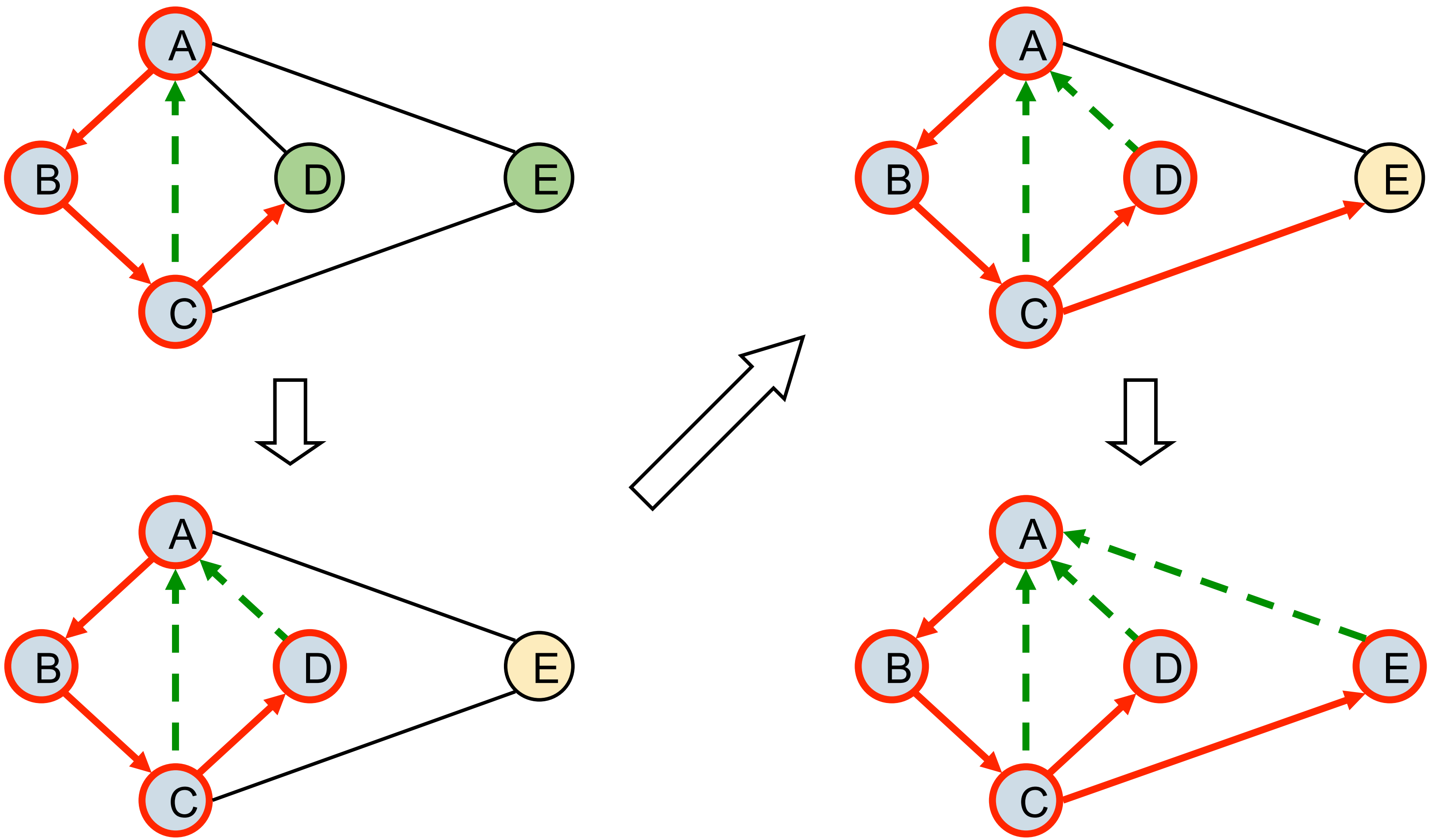
discovery edge

back edge

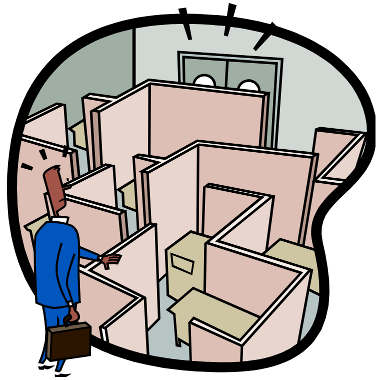




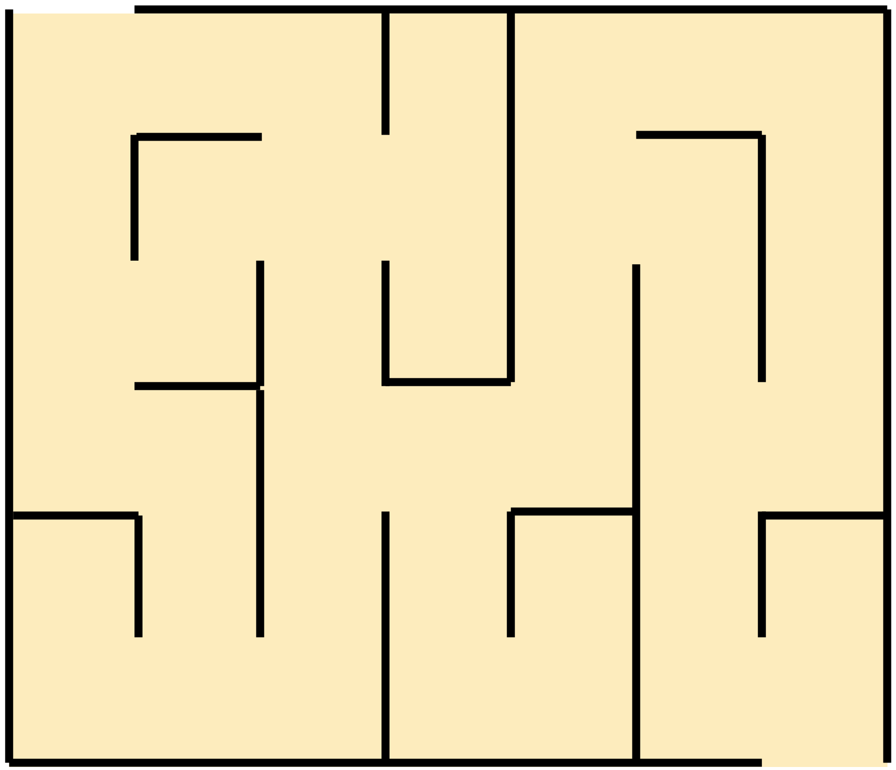
Example (cont.)



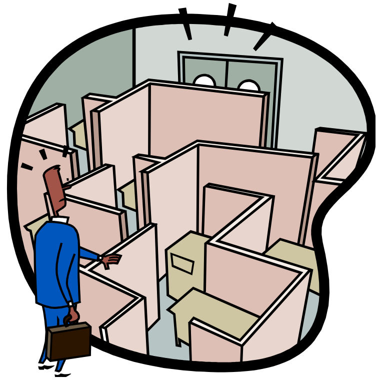
DFS and Maze Traversal



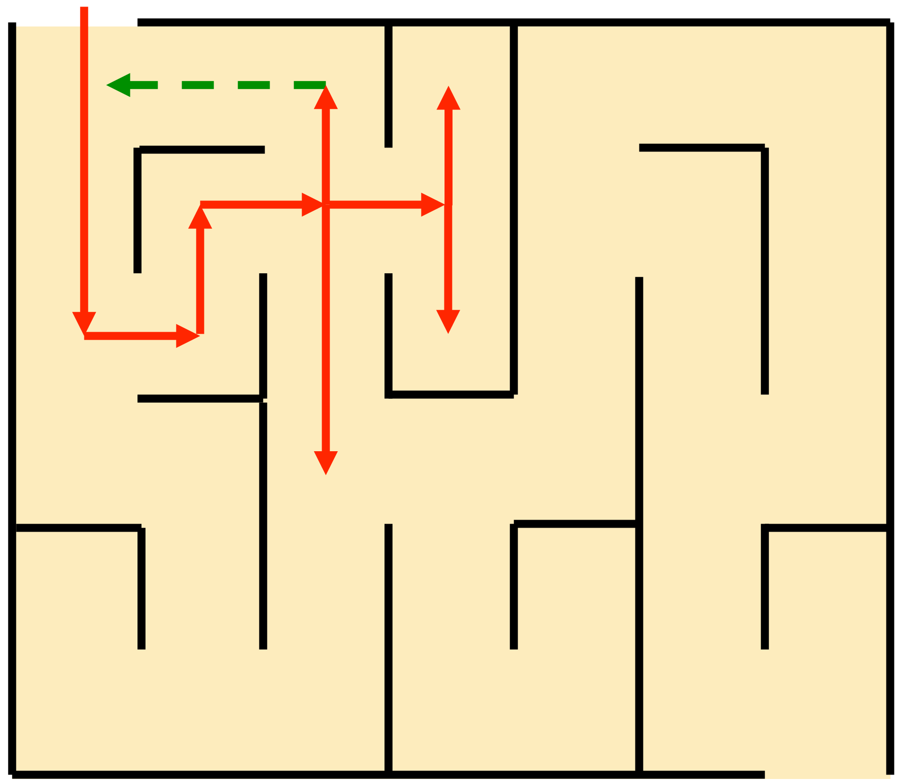
- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



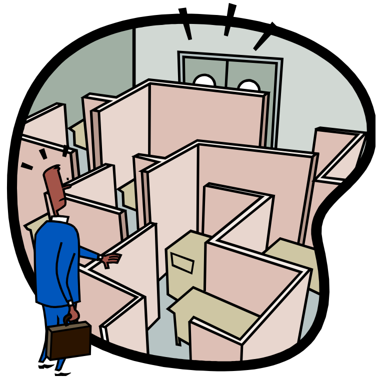
DFS and Maze Traversal



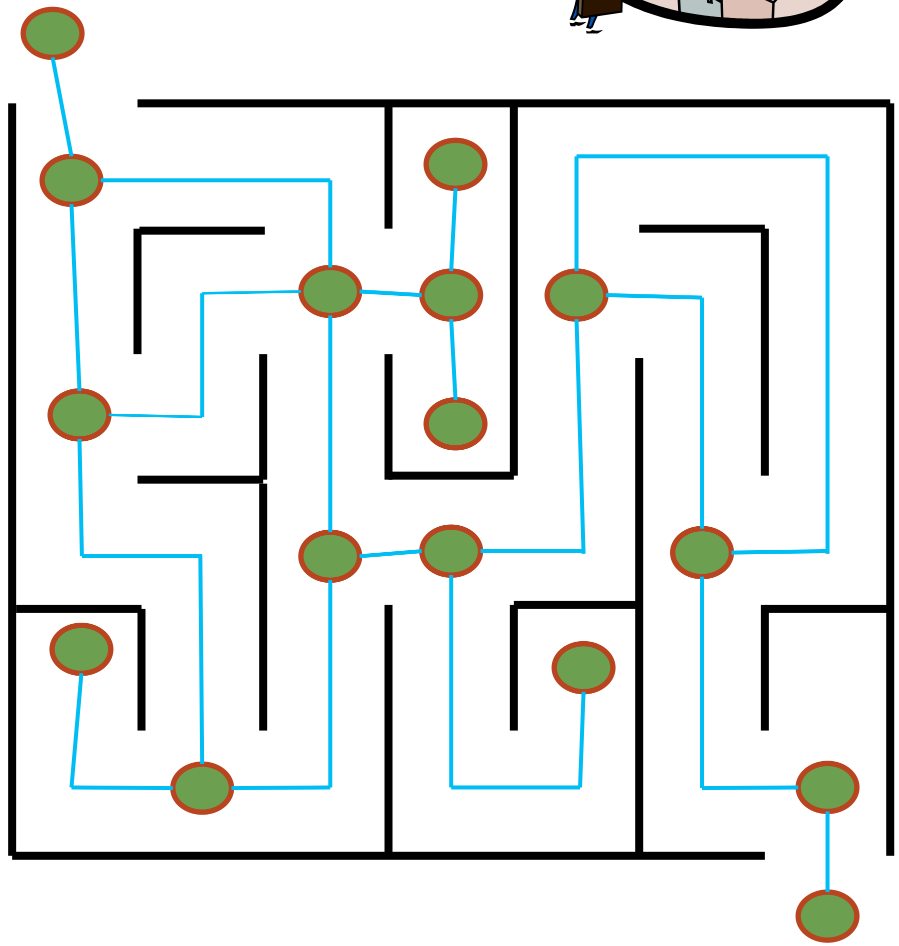
- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



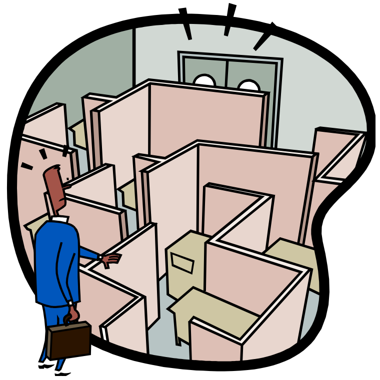
DFS and Maze Traversal



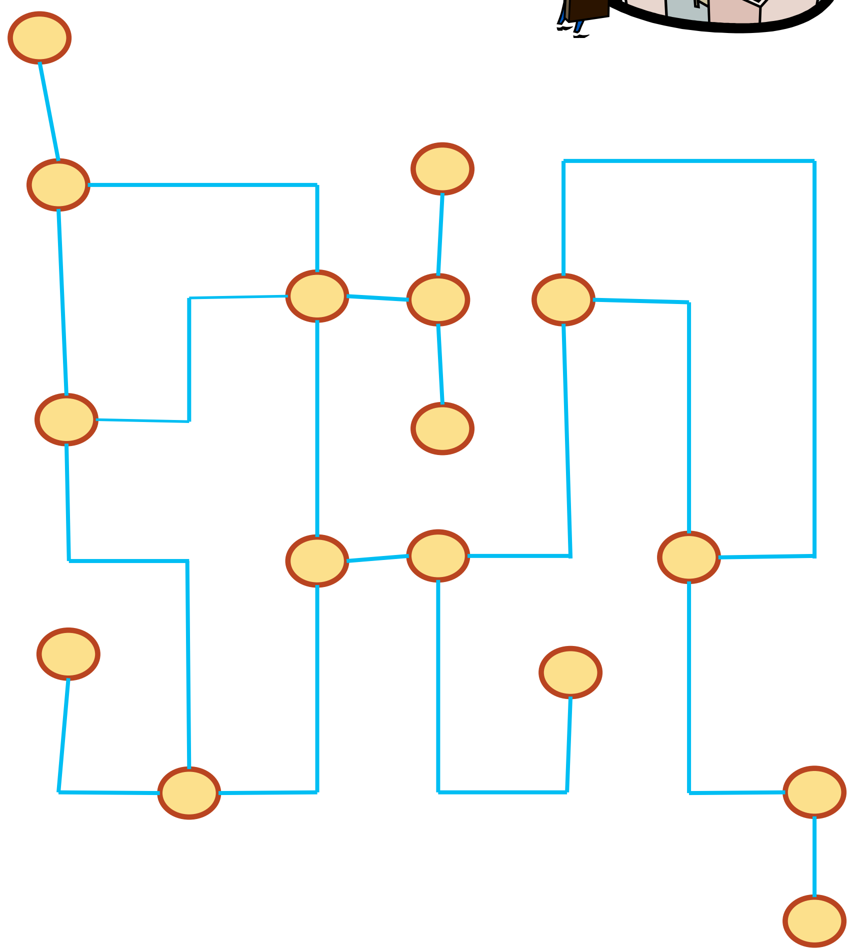
- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



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)



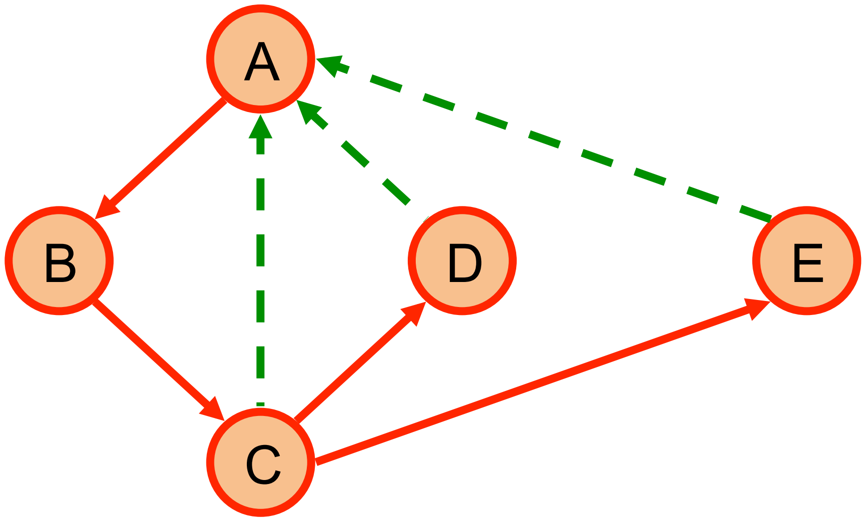
Properties of DFS

Property 1

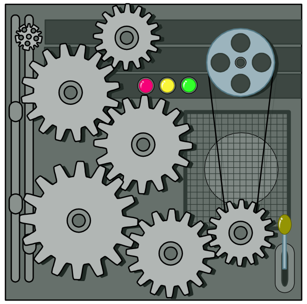
DFS(*G*, *v*) visits all the vertices and edges in the connected component of *v*

Property 2

The discovery edges labeled by *DFS*(*G*, *v*) form a spanning tree 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
- 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 $e \in G.incidentEdges(v)$
    if $getLabel(e) = UNEXPLORED$
      $w \leftarrow opposite(v,e)$
      if $getLabel(w) = UNEXPLORED$
        $setLabel(e, DISCOVERY)$
        $DFS(G, w)$
      else
        $setLabel(e, BACK)$

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;
}
```

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)
  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

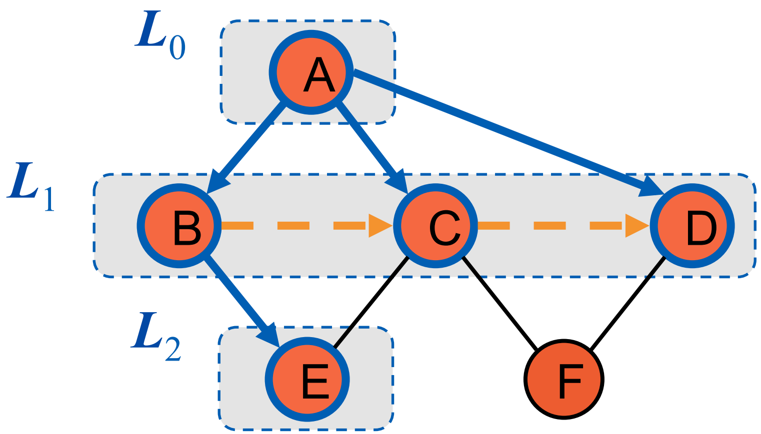
```
Algorithm DFS(G)
   Input graph G
   Output labeling of the edges of G
      as discovery edges and
      back edges
  for all u ∈ G.vertices()
   setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
   setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
   if getLabel(v) = UNEXPLORED
      DFS(G, v)
```

```
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)
```

All Connected Components

- Loop over all vertices, doing a DFS from each unvisted 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
  - 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
  - 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 $u \in G.vertices()$
    setLabel(*u, UNEXPLORED*)
  for all $e \in G.edges()$
    setLabel(*e, UNEXPLORED*)
  for all $v \in G.vertices()$
    if *getLabel*(*v*) = *UNEXPLORED*
      BFS(*G, v*)

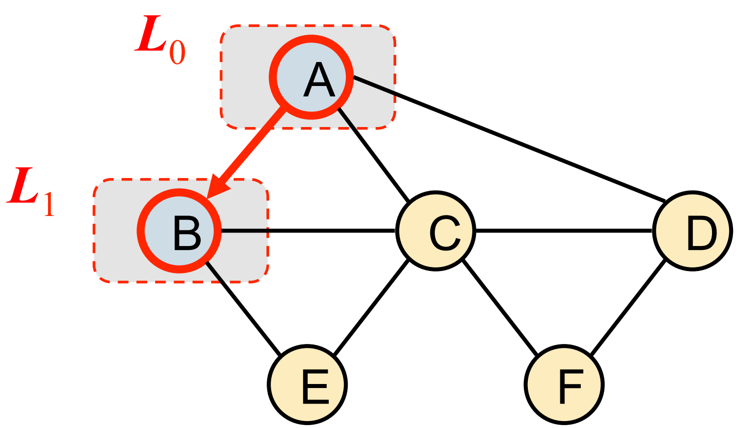
Algorithm *BFS*(*G, s*)
  $L_0 \leftarrow$ new empty sequence
  $L_0.addLast(s)$
  setLabel(*s, VISITED*)
  $i \leftarrow 0$
  while $\neg L_i.isEmpty()$
    $L_{i+1} \leftarrow$ new empty sequence
    for all $v \in L_i.elements()$
      for all $e \in G.incidentEdges(v)$
        if *getLabel*(*e*)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

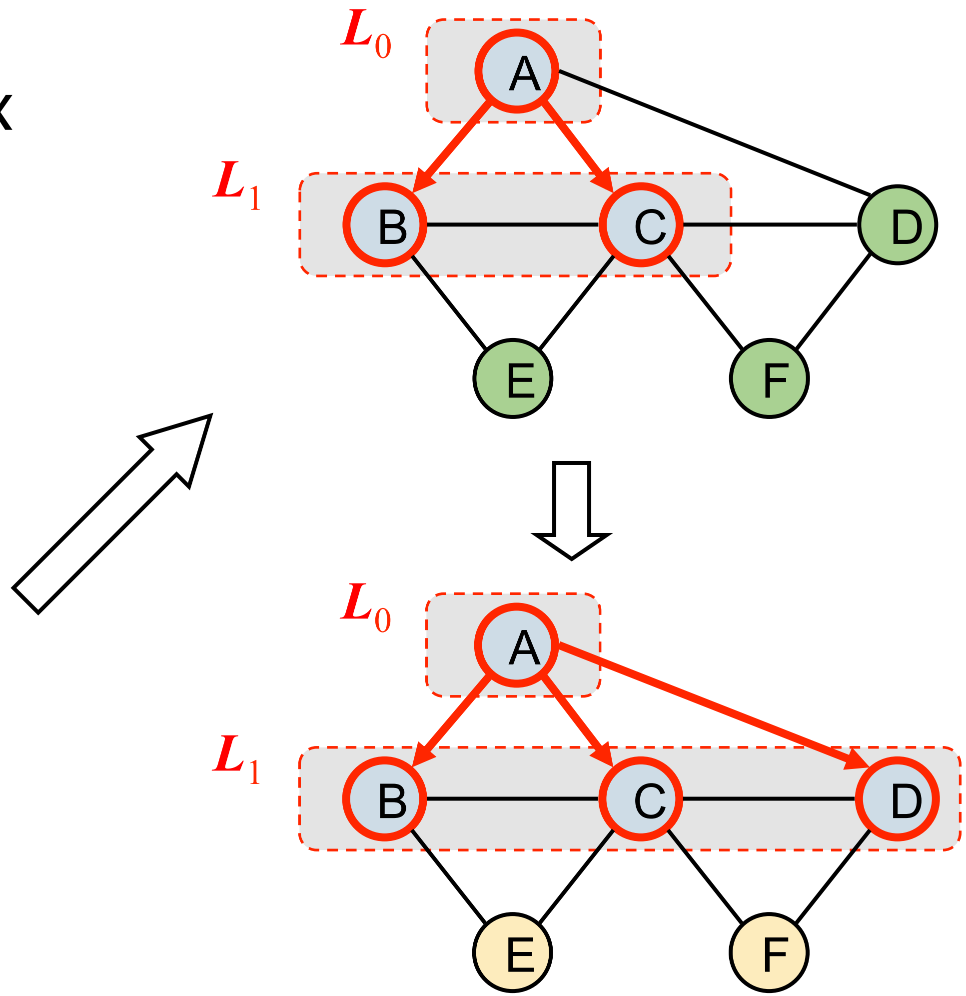
Java Implementation (fragment 14.8)

```
/** 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
  }
}
```

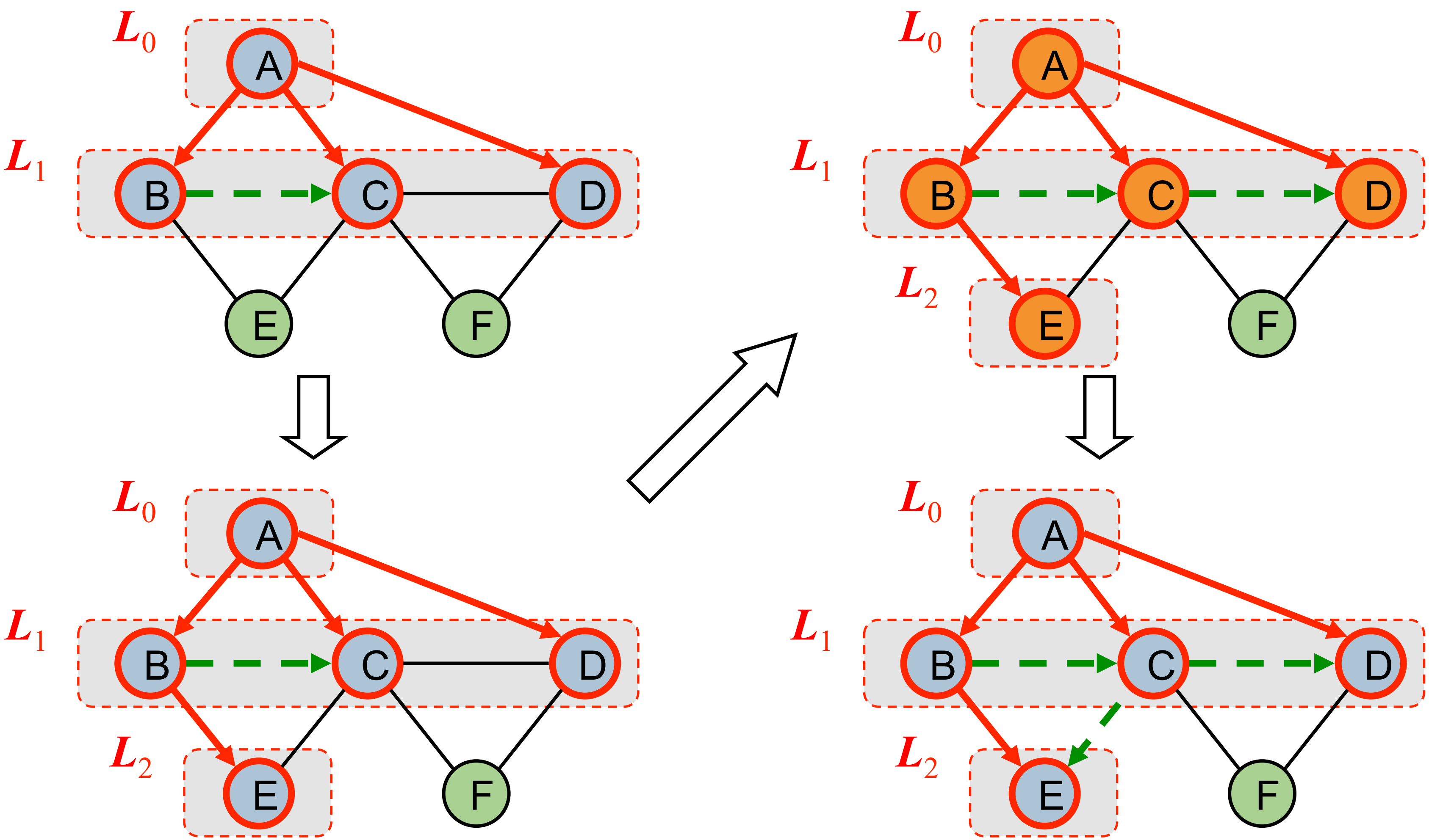
Example

unexplored vertex
visited vertex
unexplored edge
discovery edge
cross edge

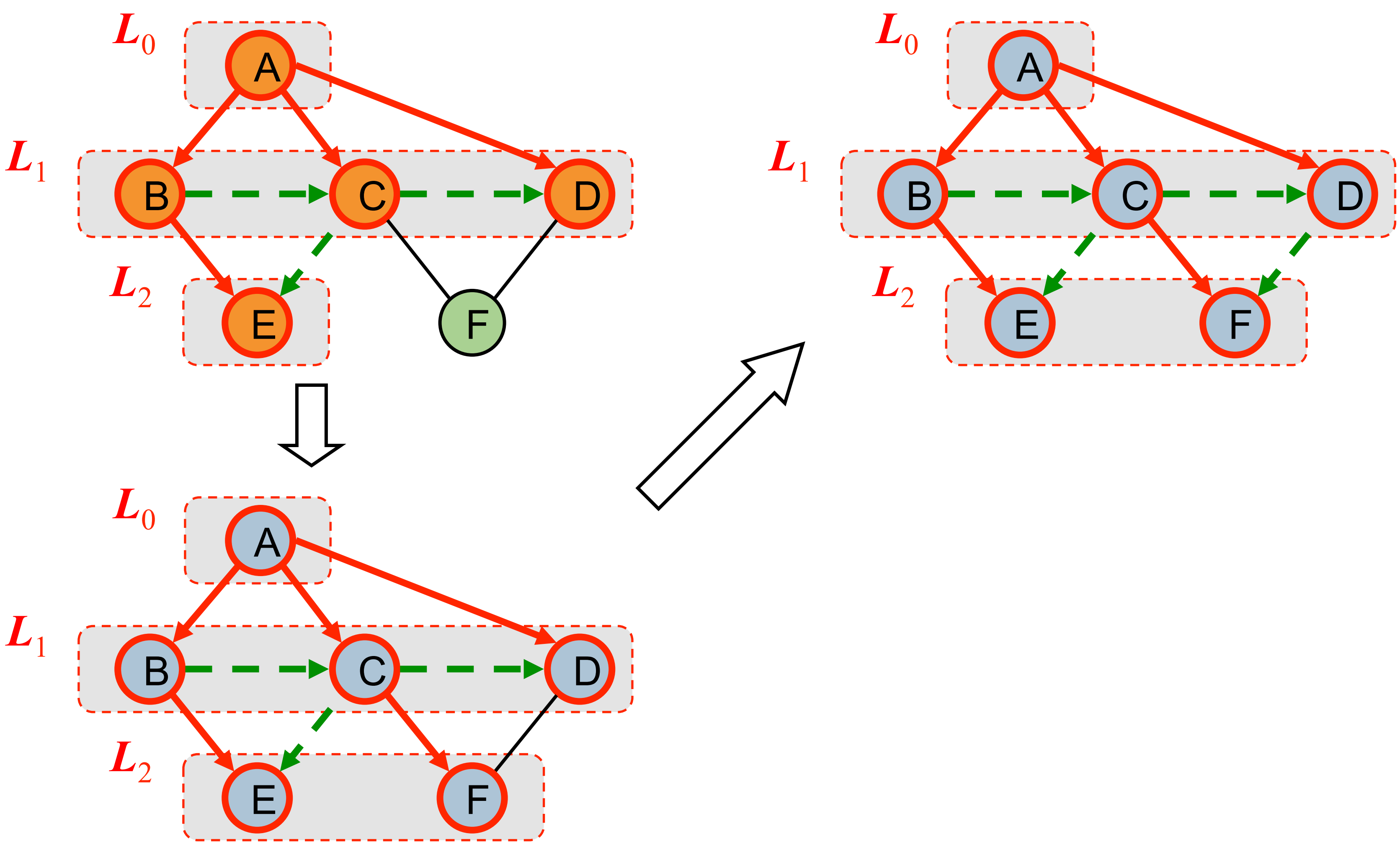




Example (cont.)



Example (cont.)



Properties

Notation

G_s: connected component of s

Property 1

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

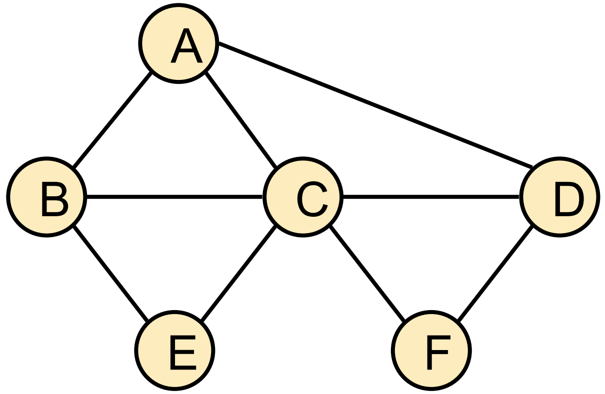
Property 2

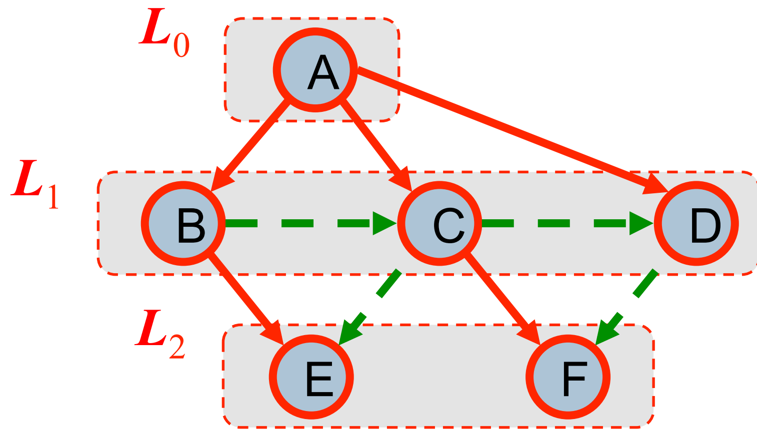
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

Property 3

For each vertex v in L_i

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





Analysis

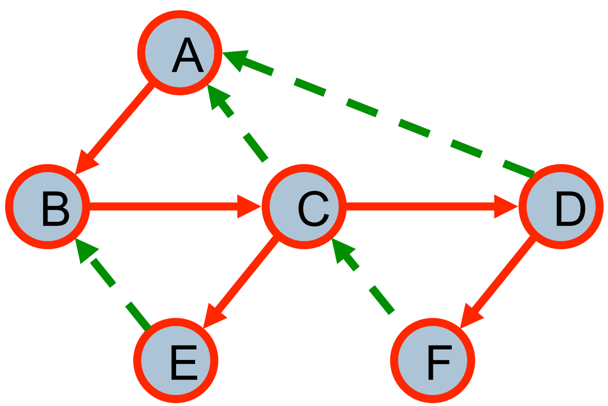
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges is called once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

Applications

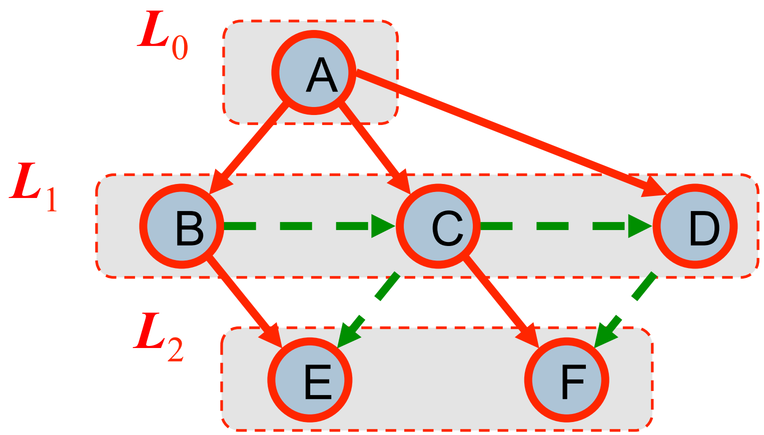
- Using the template method pattern, we can specialise the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, 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	√	



DFS



BFS

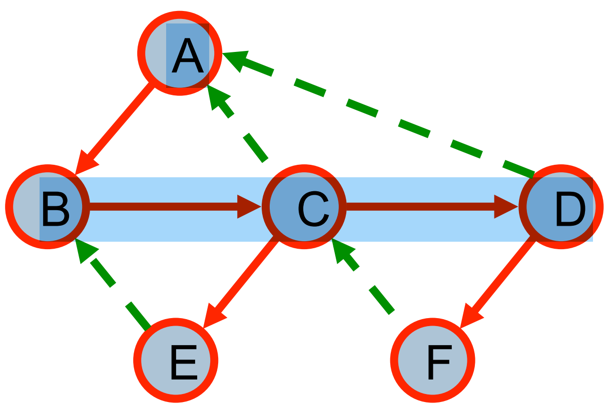
DFS vs. BFS (cont.)

Back edge (v,w)

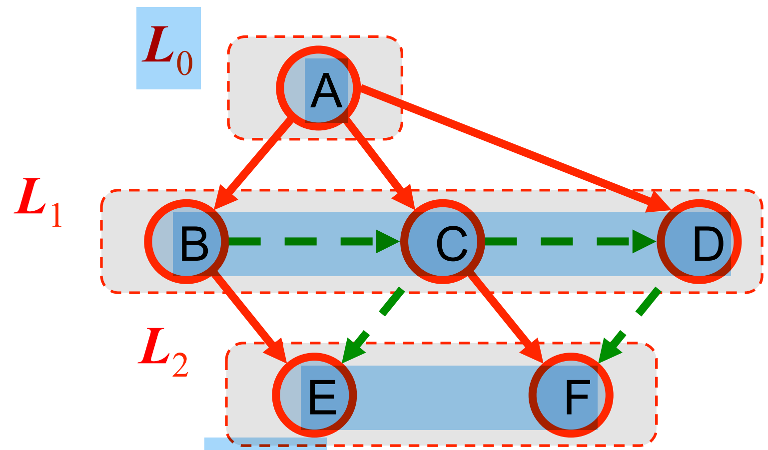
- w is an ancestor of v in the tree of discovery edges

Cross edge (v,w)

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



DFS



BFS

Outline

- Graphs: definitions and ADT (section 14.1)

- Data structures for graphs (section 14.2)

- Graphs traversals (section 14.3)