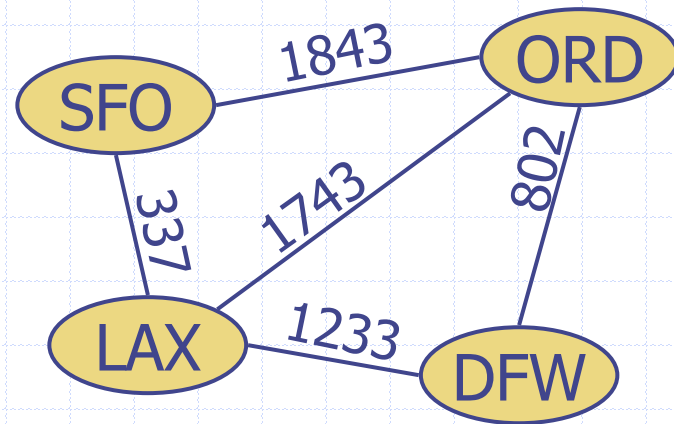


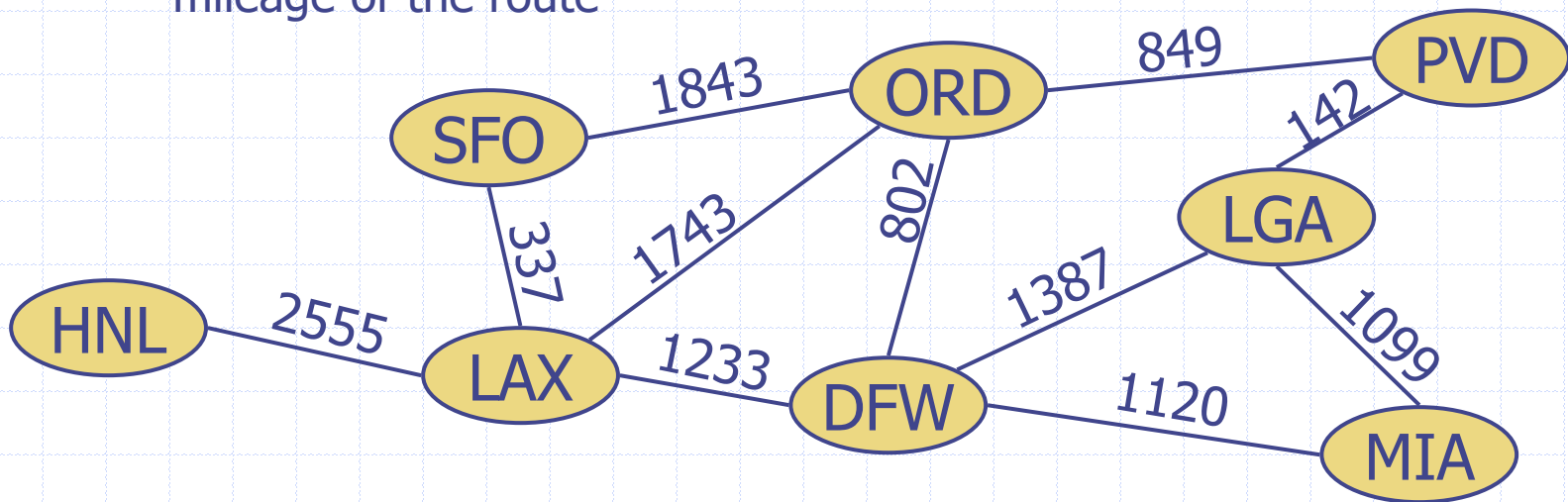
Presentation for use with the textbook **Data Structures and Algorithms in Java, 6th edition**, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Graphs



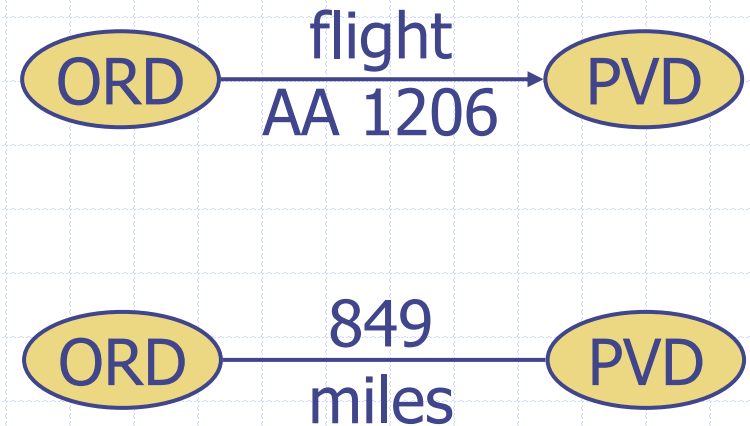
Graphs

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



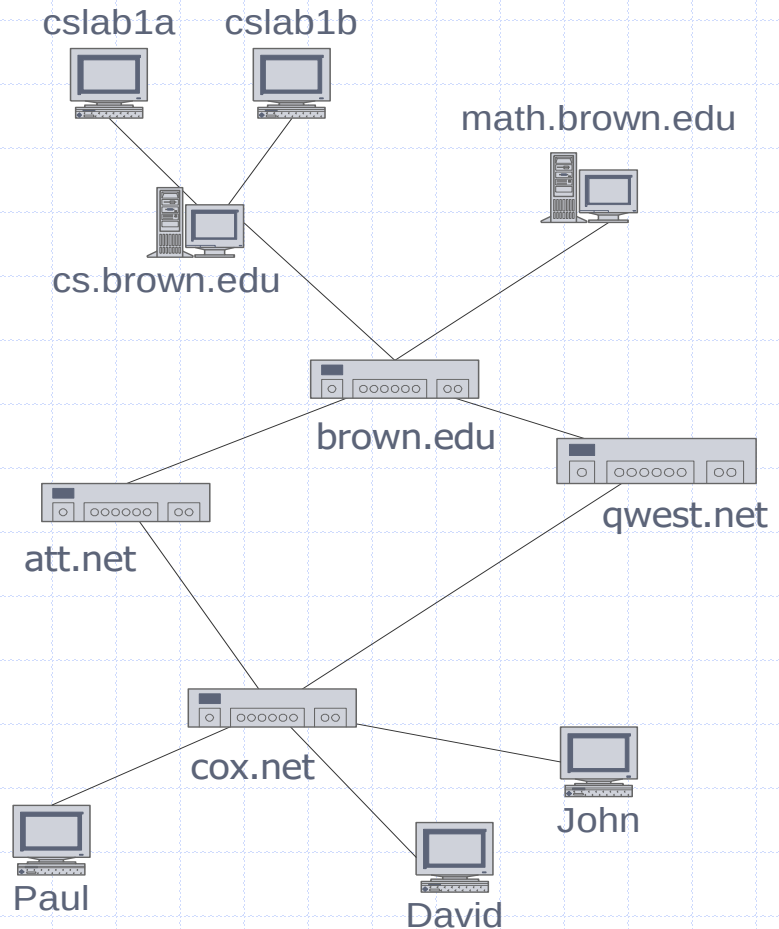
Edge Types

- Directed edge
 - ordered pair of vertices (u,v)
 - first vertex u is the origin
 - second vertex v is the destination
 - e.g., a flight
- 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



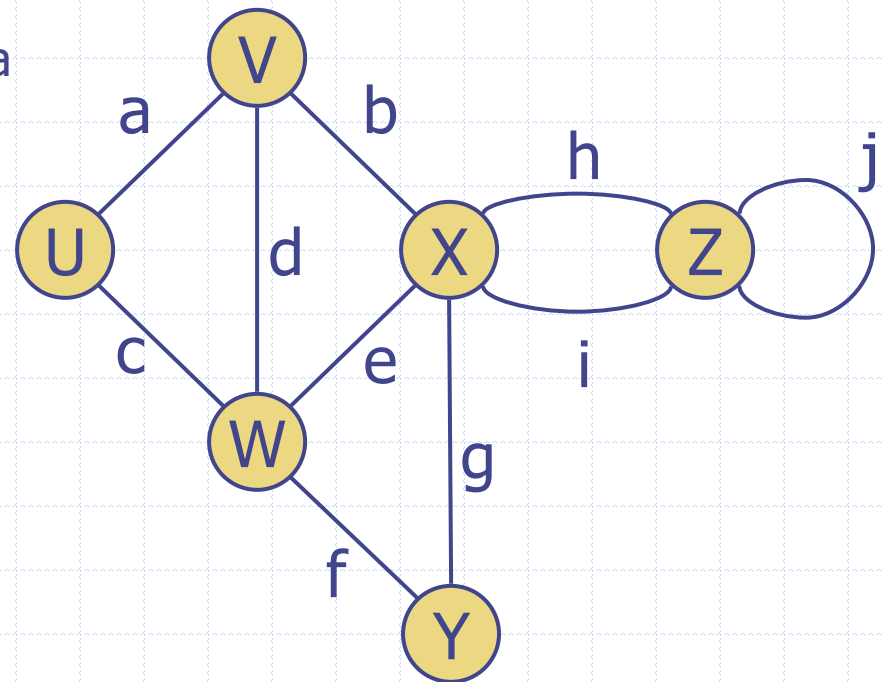
Applications

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



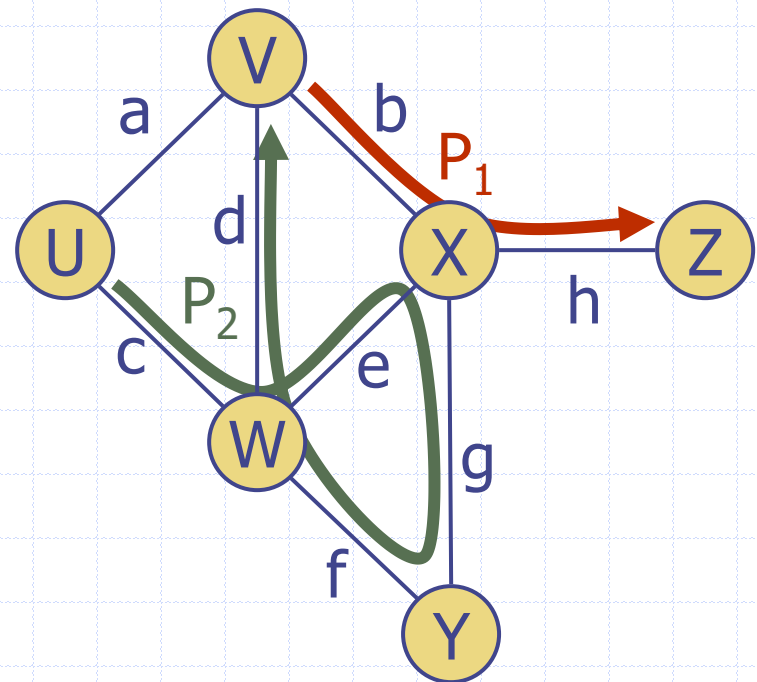
Terminology

- End vertices (or endpoints) of an edge
 - U and V are the endpoints of a
- Edges incident on a vertex
 - a, d, and b are incident on V
- 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



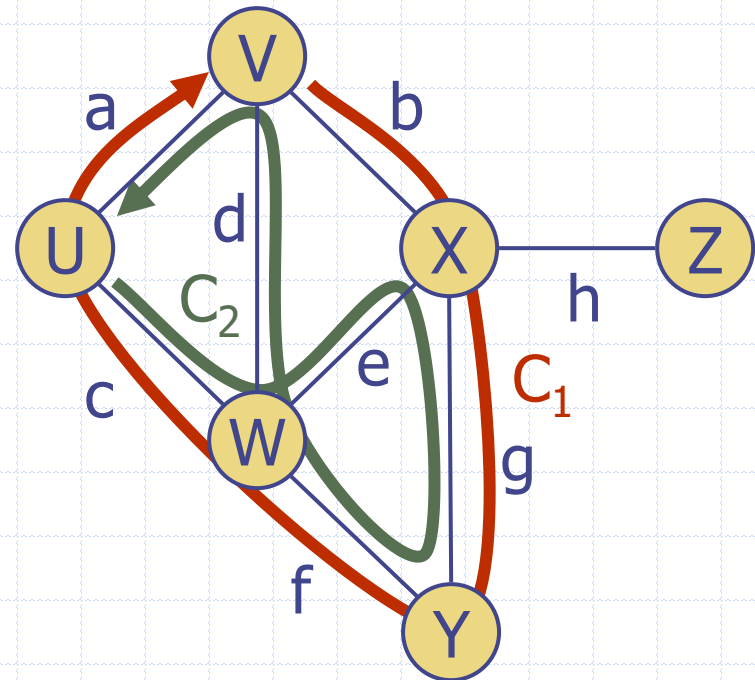
Terminology (cont.)

- 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
 - $C_1 = (V, b, X, g, Y, f, W, c, U, a, \downarrow)$ is a simple cycle
 - $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, \downarrow)$ is a cycle that is not simple



Properties

Property 1

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

Proof: each edge is counted twice

Property 2

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

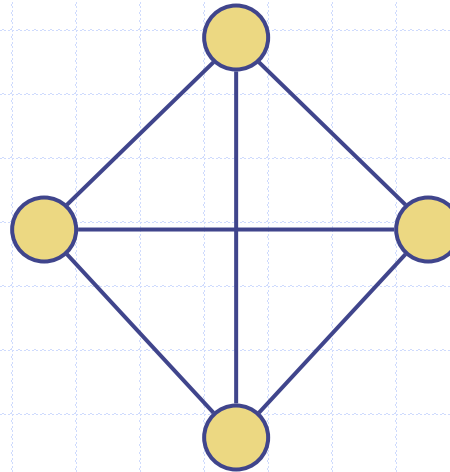
$$m \leq n(n-1)/2$$

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

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

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.

Data Structures for Graphs

- we introduce four data structures for representing a graph.
- In each representation, we maintain a collection to store the vertices of a graph.
- However, the four representations differ greatly in the way they organize the edges.
- These four data structures are:
 - *edge list*
 - *adjacency list*
 - *adjacency map*
 - *adjacency matrix*

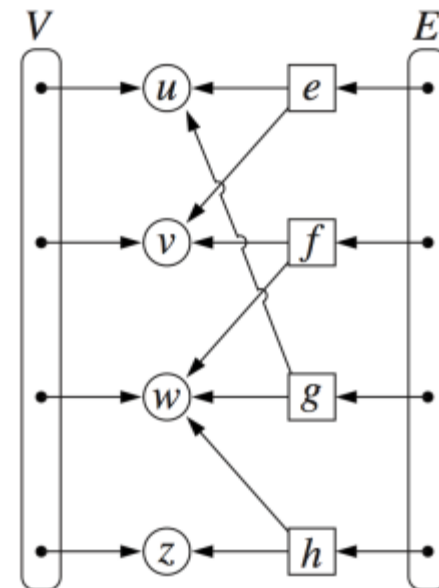
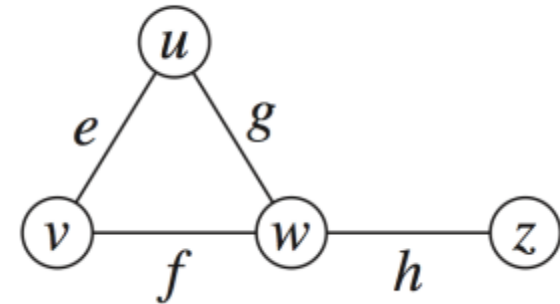
Data Structures for Graphs- Performance

Method	Edge List	Adj. List	Adj. Map	Adj. Matrix
numVertices()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
numEdges()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices()	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges()	$O(m)$	$O(m)$	$O(m)$	$O(m)$
getEdge(u, v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
outDegree(v) inDegree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
outgoingEdges(v) incomingEdges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
insertVertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
removeVertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insertEdge(u, v, x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
removeEdge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

Table 14.1: A summary of the running times for the methods of the graph ADT, using the graph representations discussed in this section. We let n denote the number of vertices, m the number of edges, and d_v the degree of vertex v . Note that the adjacency matrix uses $O(n^2)$ space, while all other structures use $O(n + m)$ space.

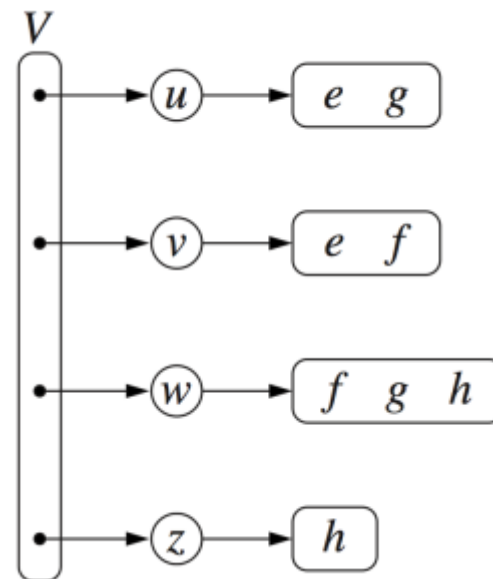
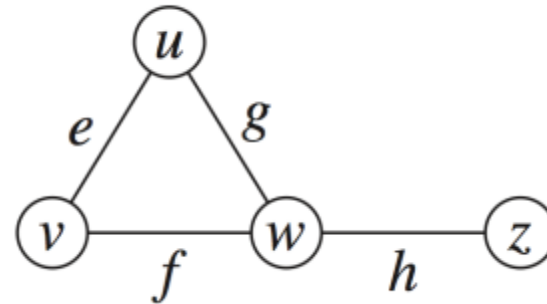
Edge List Structure

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



Adjacency List Structure

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



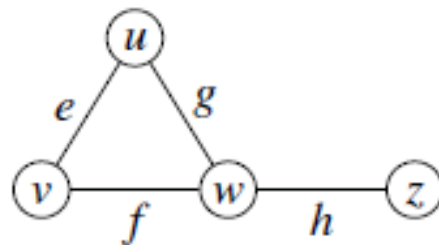
Adjacency Map Structure

- ❑ We can improve the performance by using a hash-based map to implement $I(v)$ for each vertex v .
- ❑ Let the opposite endpoint of each incident edge serve as a key in the map, with the edge structure serving as the value.
- ❑ We call such a graph representation an adjacency map.
- ❑ The space usage for an adjacency map remains $O(n+m)$, because $I(v)$ uses $O(\deg(v))$ space for each vertex v , as with the adjacency list.

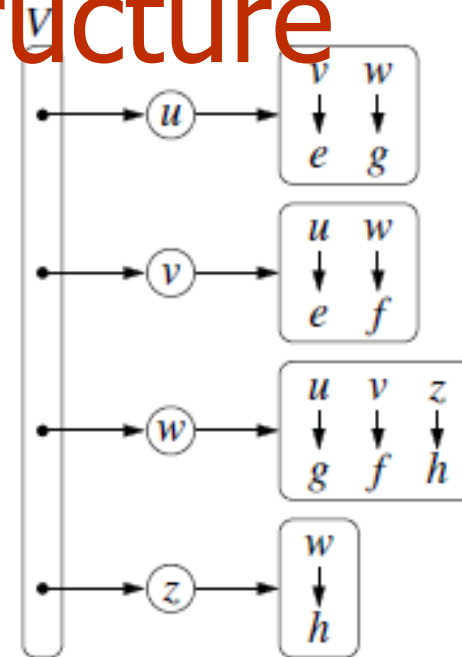
Adjacency Map Structure

- ❑ The advantage of the adjacency map, relative to an adjacency list, is that the `getEdge(u, v)` method can be implemented in expected $O(1)$ time by searching for vertex u as a key in $I(v)$, or vice versa.
- ❑ This provides an improvement over the adjacency list, while retaining the worst-case bound of $O(\min(\deg(u), \deg(v)))$.
- ❑ The performance of adjacency map essentially achieves optimal running times for all methods, making it an excellent all-purpose choice as a graph representation.

Adjacency Map Structure



(a)

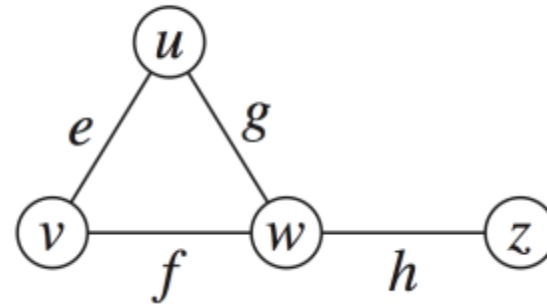


(b)

Figure 14.6: (a) An undirected graph G ; (b) a schematic representation of the adjacency map structure for G . Each vertex maintains a secondary map in which neighboring vertices serve as keys, with the connecting edges as associated values. As with the adjacency list, we presume that there is also an overall list E of all Edge instances.

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 adjacent vertices
- The “old fashioned” version just has 0 for no edge and 1 for edge



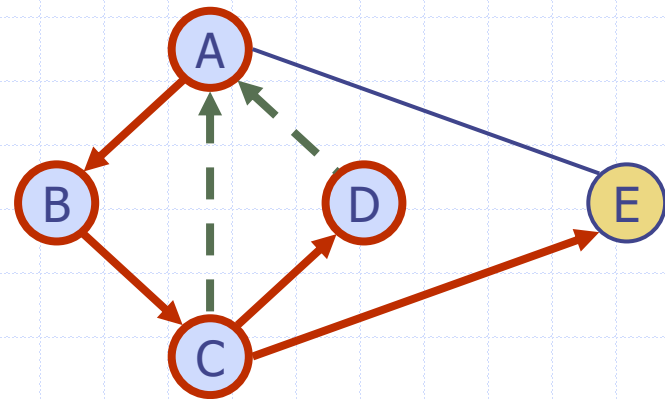
		0	1	2	3		
u	$\longrightarrow$	0		e	g		
v	$\longrightarrow$	1		e		f	
w	$\longrightarrow$	2		g	f		h
z	$\longrightarrow$	3				h	

Performance

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

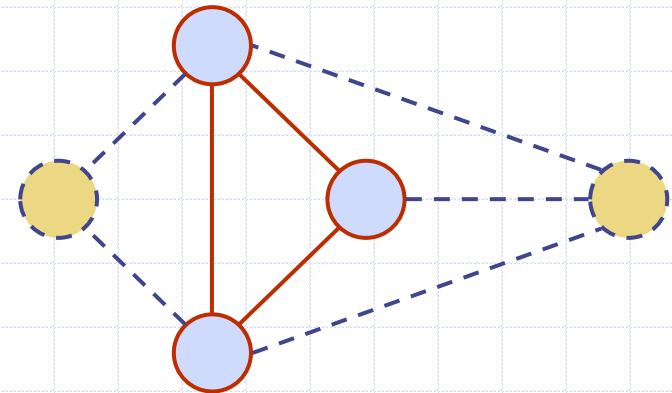
Presentation for use with the textbook **Data Structures and Algorithms in Java, 6th edition**, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Depth-First Search

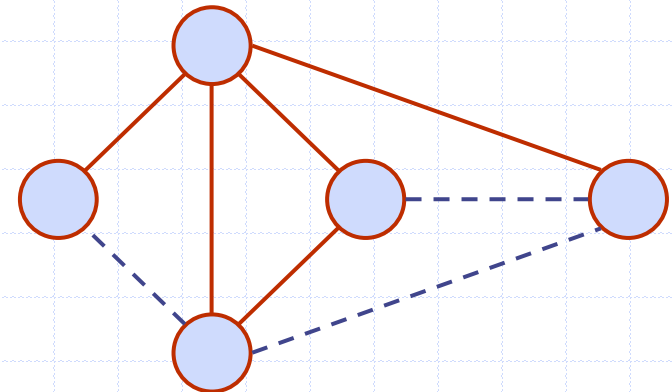


Subgraphs

- 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



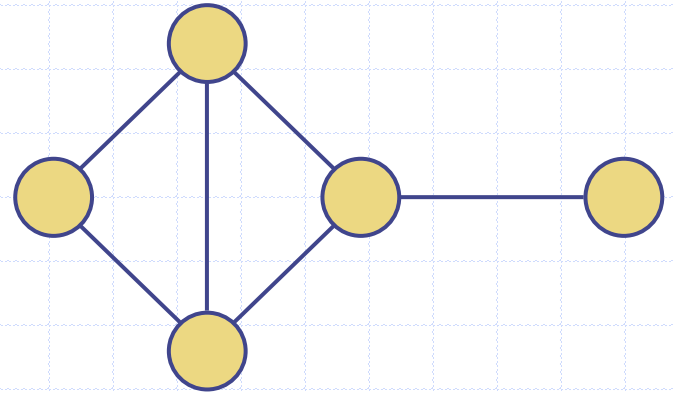
Subgraph



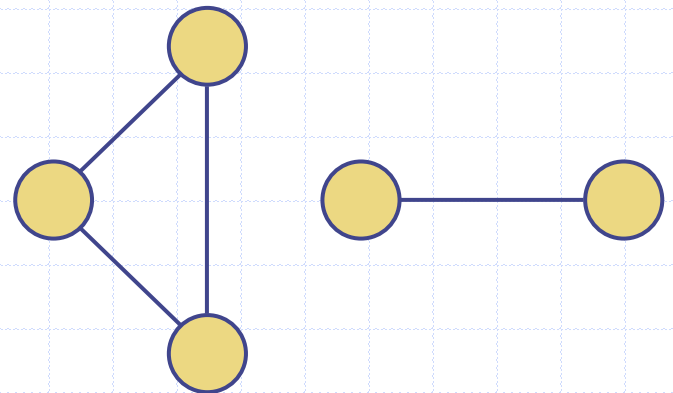
Spanning subgraph

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

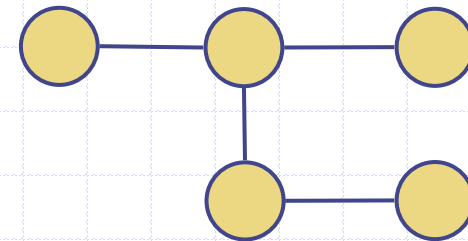
Trees and Forests

- A (free) tree is an undirected graph T such that

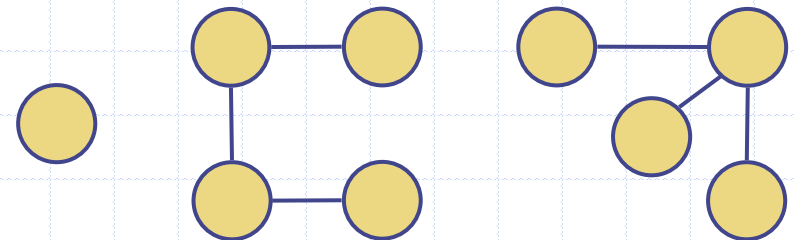
- T is connected
- T has no cycles

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



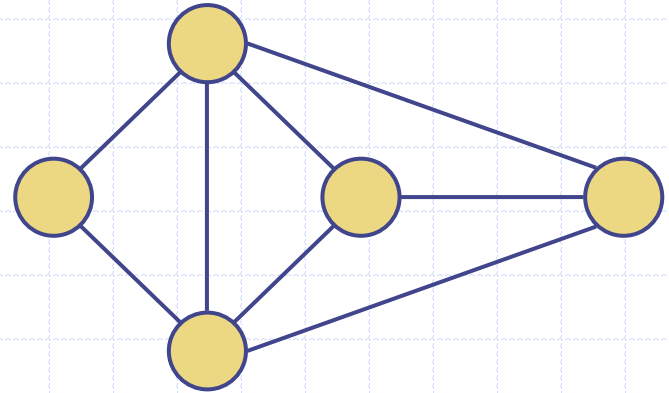
Tree



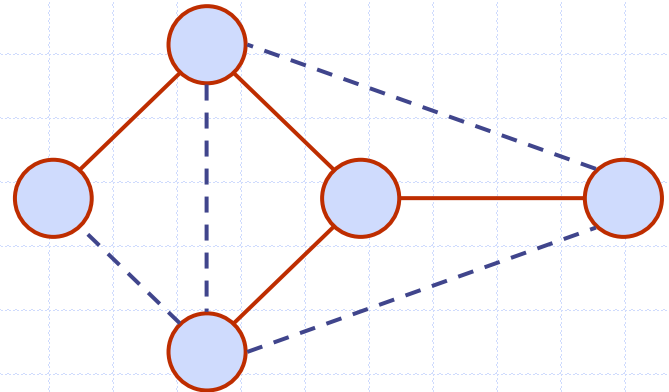
Forest

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



Graph



Spanning tree

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).

Example 1

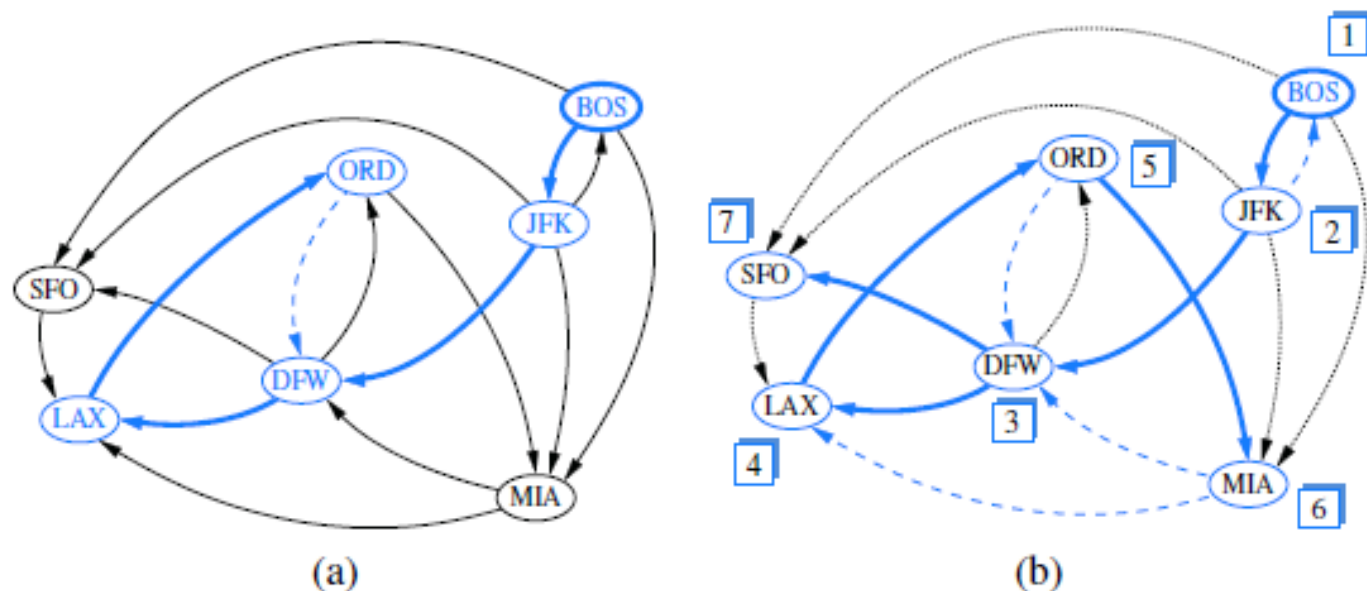


Figure 14.8: An example of a DFS in a directed graph, starting at vertex (BOS): (a) intermediate step, where, for the first time, a considered edge leads to an already visited vertex (DFW); (b) the completed DFS. The tree edges are shown with thick blue lines, the back edges are shown with dashed blue lines, and the forward and cross edges are shown with dotted black lines. The order in which the vertices are visited is indicated by a label next to each vertex. The edge (ORD,DFW) is a back edge, but (DFW,ORD) is a forward edge. Edge (BOS,SFO) is a forward edge, and (SFO,LAX) is a cross edge.

Example 2

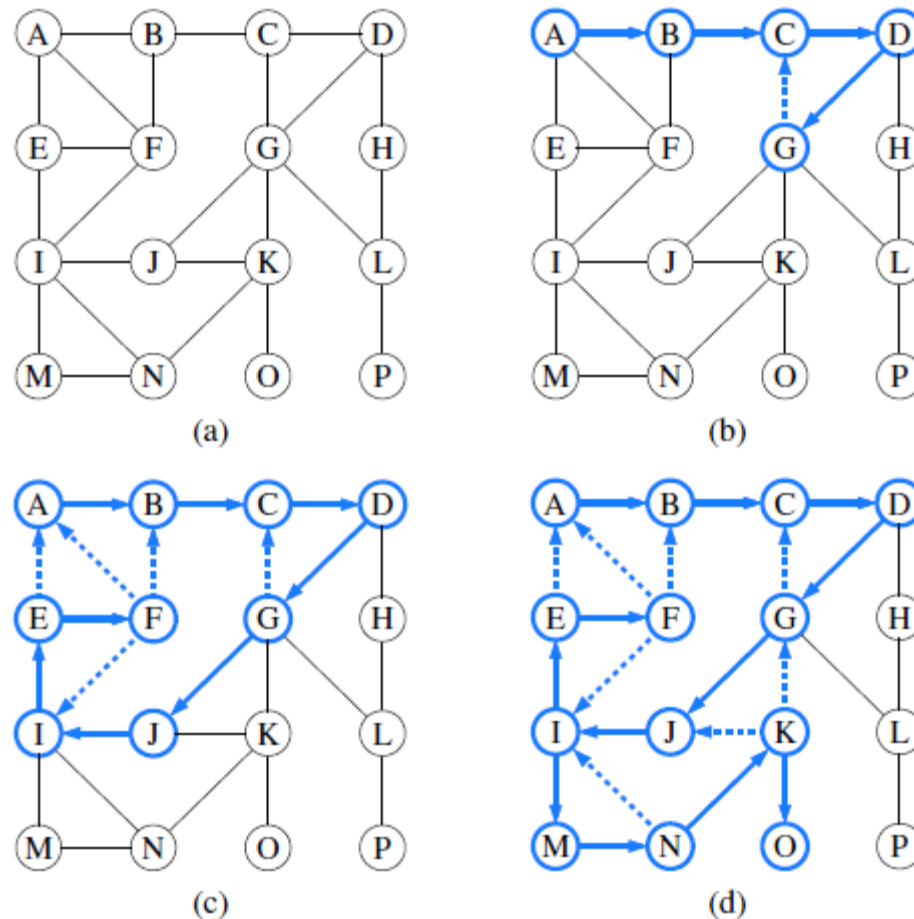


Figure 14.9: Example of depth-first search traversal on an undirected graph starting at vertex A. We assume that a vertex's adjacencies are considered in alphabetical order. Visited vertices and explored edges are highlighted, with discovery edges drawn as solid lines and nontree (back) edges as dashed lines: (a) input graph;

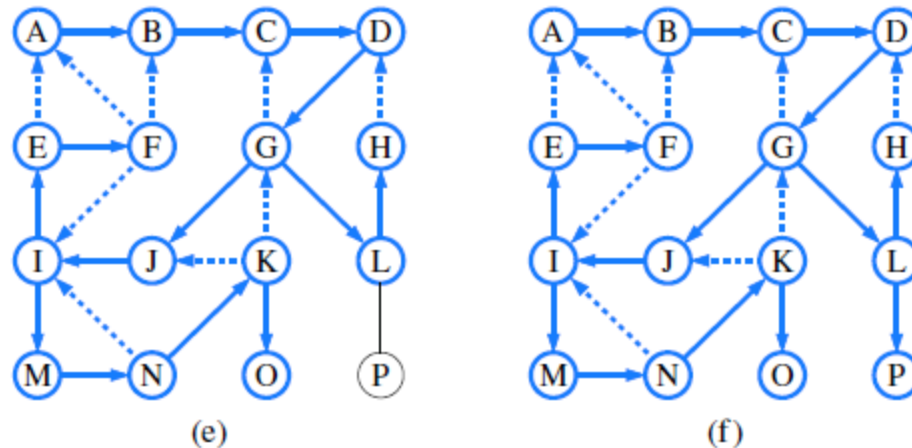


Figure 14.9: Example of depth-first search traversal on an undirected graph starting at vertex *A*. We assume that a vertex's adjacencies are considered in alphabetical order. Visited vertices and explored edges are highlighted, with discovery edges drawn as solid lines and nontree (back) edges as dashed lines: (a) input graph;

Java Implementation

```
1  /** Performs depth-first search of Graph g starting at Vertex u. */
2  public static <V,E> void DFS(Graph<V,E> g, Vertex<V> u,
3                               Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
4      known.add(u); // u has been discovered
5      for (Edge<E> e : g.outgoingEdges(u)) { // for every outgoing edge from u
6          Vertex<V> v = g.opposite(u, e);
7          if (!known.contains(v)) {
8              forest.put(v, e); // e is the tree edge that discovered v
9              DFS(g, v, known, forest); // recursively explore from v
10         }
11     }
12 }
```

Example

A

unexplored vertex

A

visited vertex

—

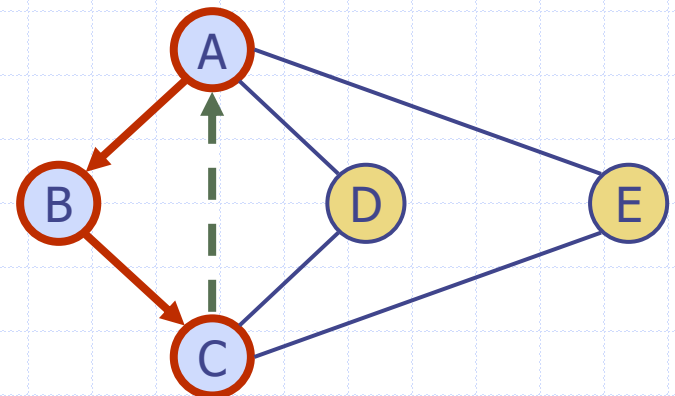
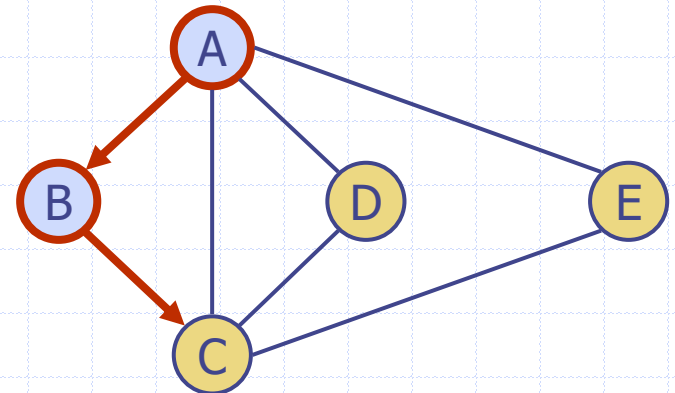
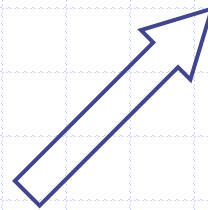
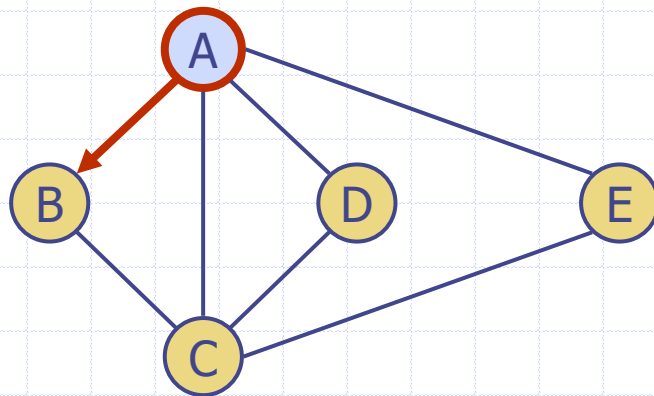
unexplored edge

→

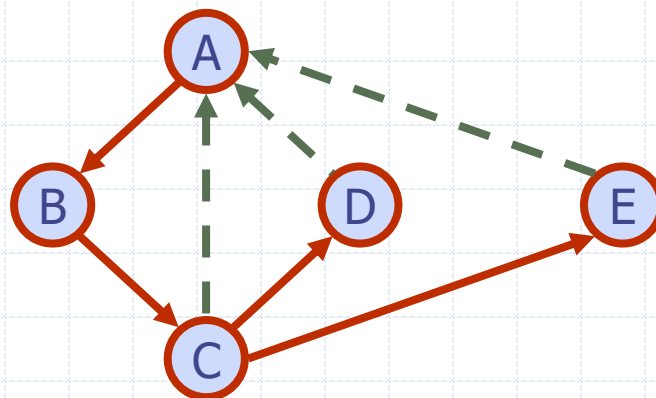
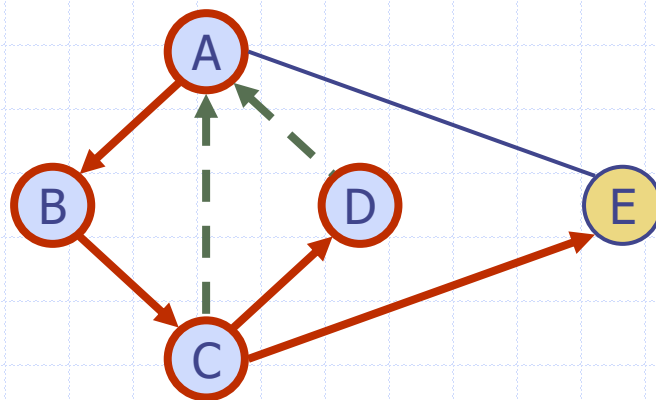
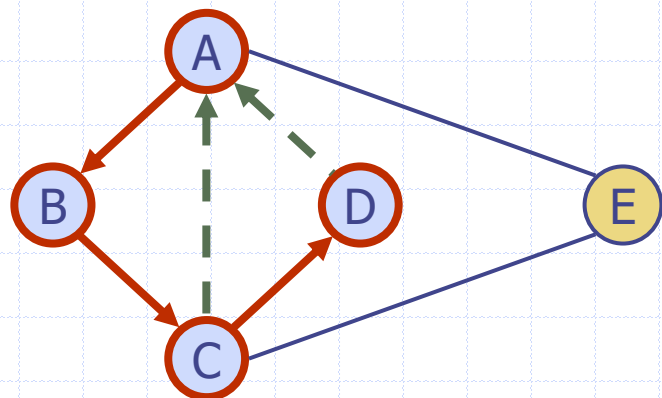
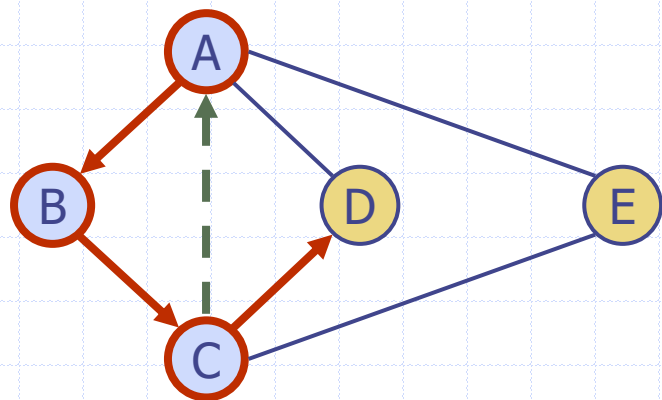
discovery edge

- - -

back edge



Example (cont.)



- [illegible]

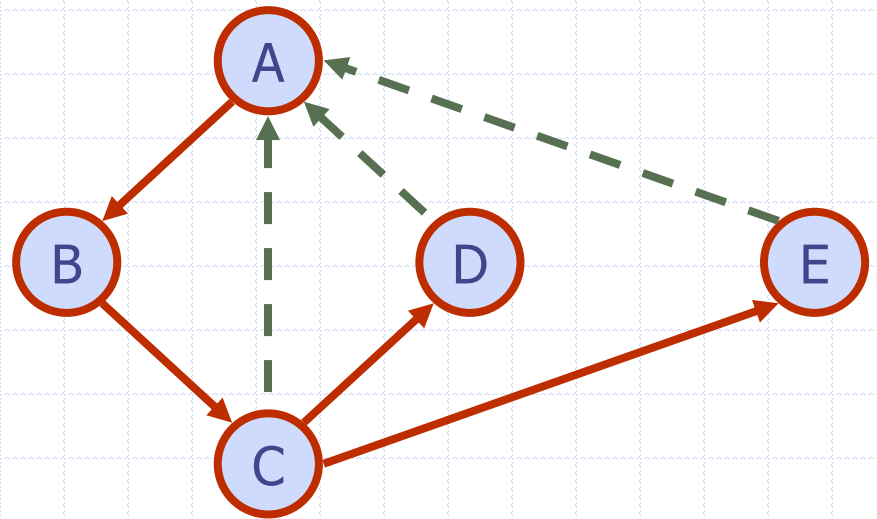
Properties of DFS

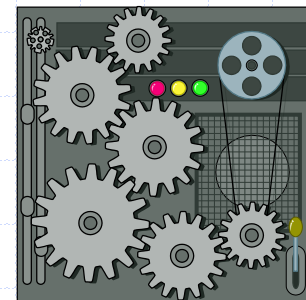
Property 1

$DFS(G, v)$ 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 $\sum_v \deg(v) = 2m$

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( $G, v, z$ )  
  setLabel( $v, VISITED$ )  
  S.push( $v$ )  
  if  $v = z$   
    return S.elements()  
  for all  $e \in G.incidentEdges(v)$   
    if getLabel( $e$ ) = UNEXPLORED  
       $w \leftarrow \text{opposite}(v, e)$   
      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



- We can use the basic DFS method as a tool to identify the (directed) path leading from vertex u to v , if v is reachable from u . This path can easily be reconstructed from the information that was recorded in the forest of discovery edges during the traversal.
- Code Fragment 14.6 provides an implementation of a secondary method that produces an ordered list of vertices on the path from u to v , if given the map of discovery edges that was computed by the original DFS method.

Path Finding



- ❑ To reconstruct the path, we begin at the *end* of the path, examining the forest of discovery edges to determine what edge was used to reach vertex v . We then determine the opposite vertex of that edge and repeat the process to determine what edge was used to discover it.
- ❑ By continuing this process until reaching u , we can construct the entire path.
- ❑ Assuming constant-time lookup in the forest map, the path reconstruction takes time proportional to the length of the path, and therefore, it runs in $O(n)$ time (in addition to the time originally spent calling DFS).

Path Finding in Java

```
1  /** Returns an ordered list of edges comprising the directed path from u to v. */
2  public static <V,E> PositionalList<Edge<E>>
3  constructPath(Graph<V,E> g, Vertex<V> u, Vertex<V> v,
4               Map<Vertex<V>,Edge<E>>> forest) {
5      PositionalList<Edge<E>> path = new LinkedPositionalList<>();
6      if (forest.get(v) != null) {           // v was discovered during the search
7          Vertex<V> walk = v;               // we construct the path from back to front
8          while (walk != u) {
9              Edge<E> edge = forest.get(walk);
10             path.addFirst(edge);           // add edge to *front* of path
11             walk = g.opposite(walk, edge); // repeat with opposite endpoint
12         }
13     }
14     return path;
15 }
```

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  $e \in G.incidentEdges(v)$   
    if getLabel( $e$ ) = UNEXPLORED  
       $w \leftarrow opposite(v, e)$   
      S.push( $e$ )  
      if getLabel( $w$ ) = UNEXPLORED  
        setLabel( $e, DISCOVERY$ )  
        pathDFS( $G, w, z$ )  
        S.pop( $e$ )  
      else  
         $T \leftarrow$  new empty stack  
        repeat  
           $o \leftarrow S.pop()$   
          T.push( $o$ )  
        until  $o = w$   
        return T.elements()  
  S.pop( $v$ )
```

Cycle Finding



- ❑ For both undirected and directed graphs, a cycle exists if and only if a **back edge** exists relative to the DFS traversal of that graph.
- ❑ It is easy to see that if a back edge exists, a cycle exists by taking the back edge from the descendant to its ancestor and then following the tree edges back to the descendant.
- ❑ Conversely, if a cycle exists in the graph, there must be a back edge relative to a DFS (although we do not prove this fact here).

Cycle Finding



- ❑ Algorithmically, detecting a back edge in the undirected case is easy, because all edges are either tree edges or back edges.
- ❑ In the case of a directed graph, additional modifications to the core DFS implementation are needed to properly categorize a nontree edge as a back edge.
- ❑ When a directed edge is explored leading to a previously visited vertex, we must recognize whether that vertex is an ancestor of the current vertex.
- ❑ This can be accomplished, for example, by maintaining another set, with all vertices upon which a recursive call to DFS is currently active.

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)
```

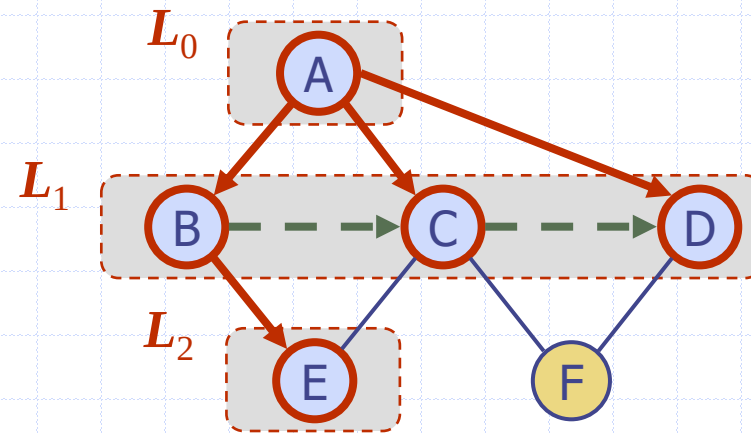
All Connected Components

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

```
1  /** Performs DFS for the entire graph and returns the DFS forest as a map. */
2  public static <V,E> Map<Vertex<V>,Edge<E>> DFSComplete(Graph<V,E> g) {
3      Set<Vertex<V>> known = new HashSet<>();
4      Map<Vertex<V>,Edge<E>> forest = new ProbeHashMap<>();
5      for (Vertex<V> u : g.vertices())
6          if (!known.contains(u))
7              DFS(g, u, known, forest);           // (re)start the DFS process at u
8      return forest;
9  }
```

Code Fragment 14.7: Top-level method that returns a DFS forest for an entire graph.

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
                     $setLabel(e, CROSS)$ 
     $i \leftarrow i + 1$ 
```

Example



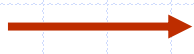
unexplored vertex



visited vertex



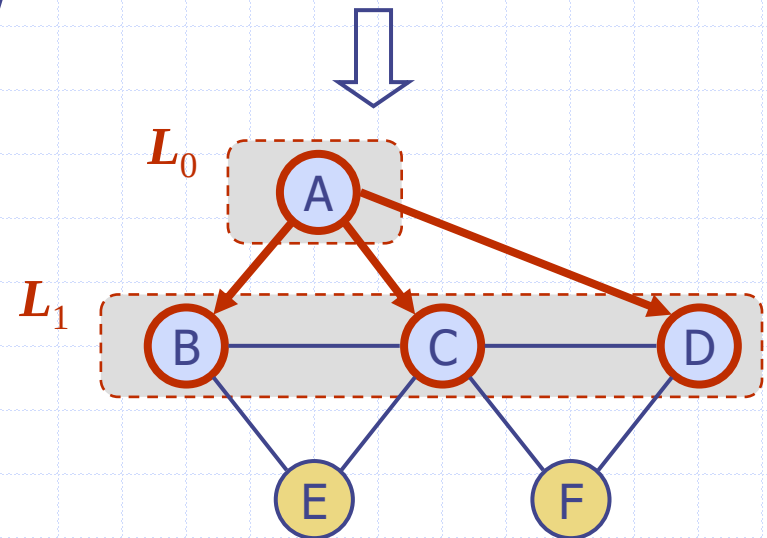
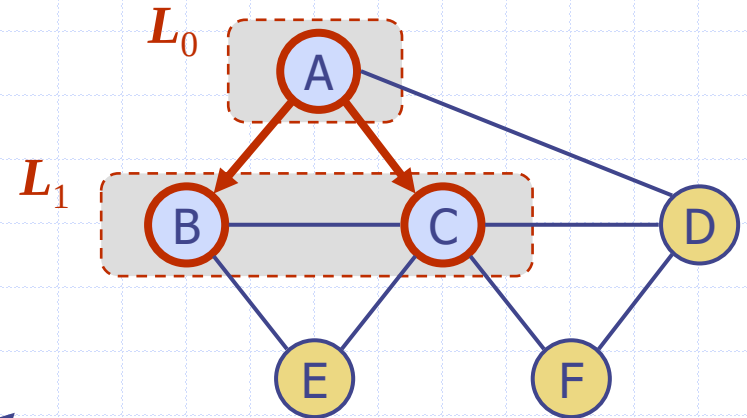
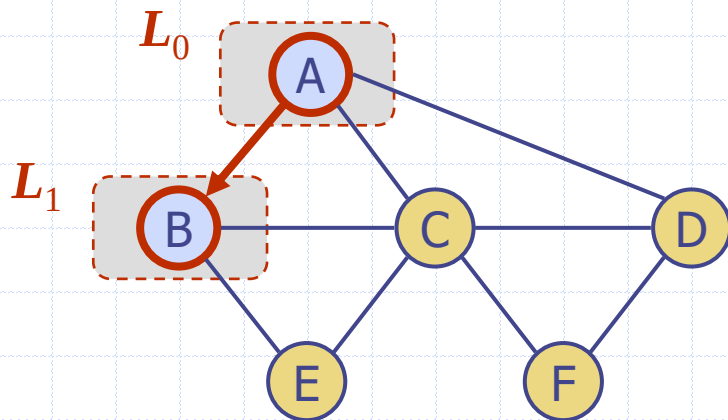
unexplored edge



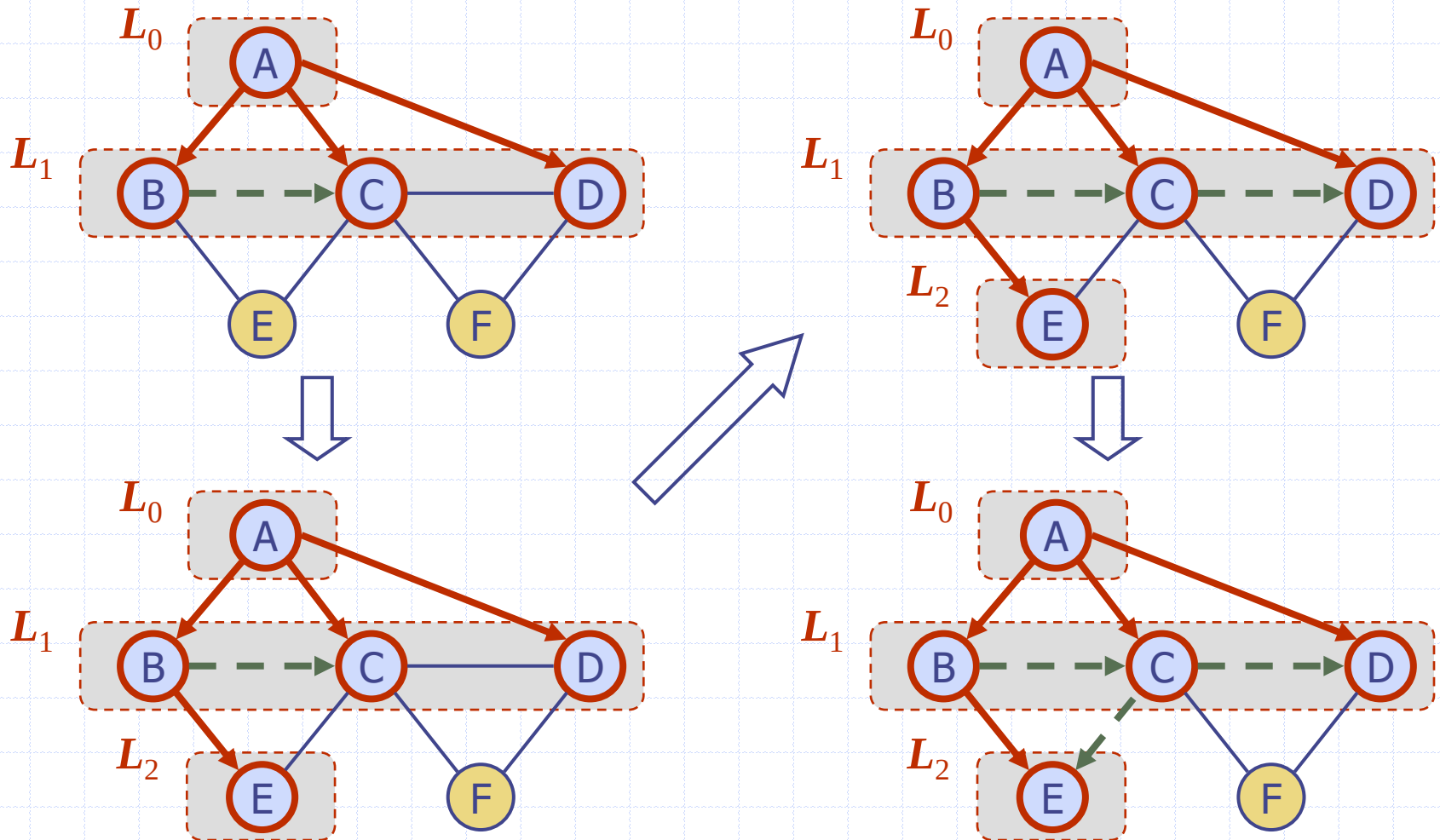
discovery edge



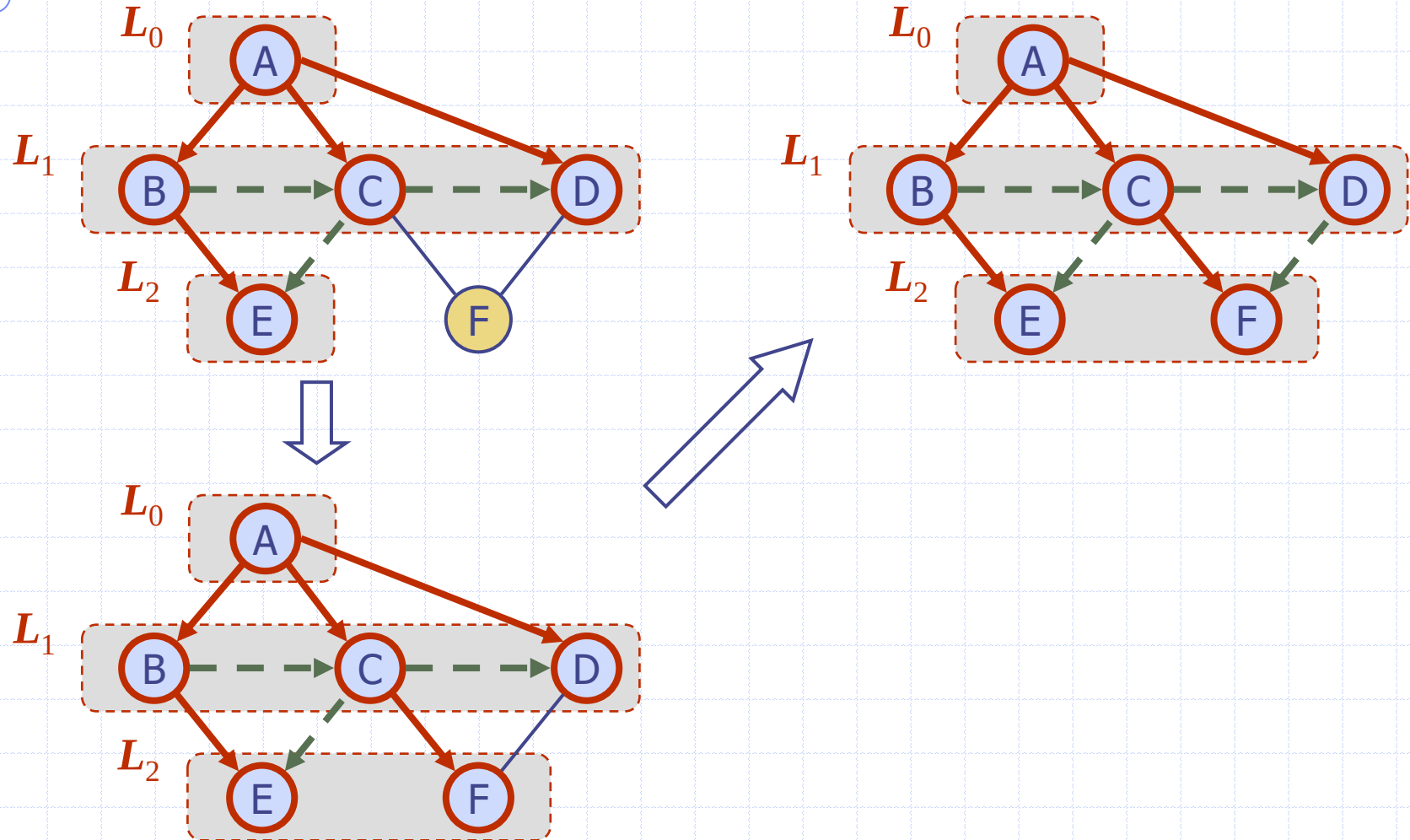
cross edge



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

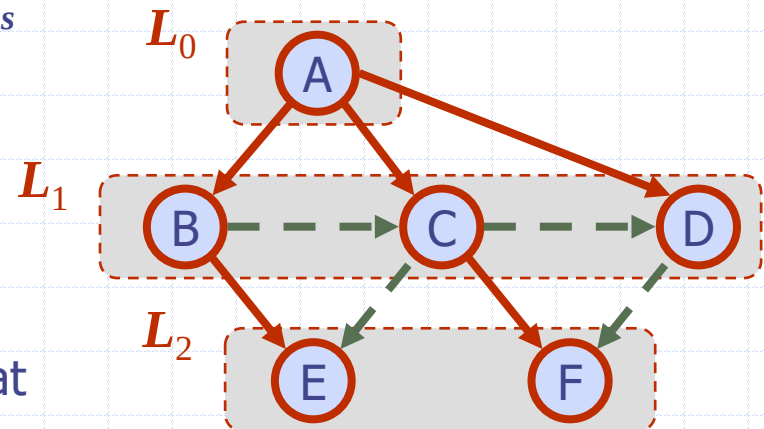
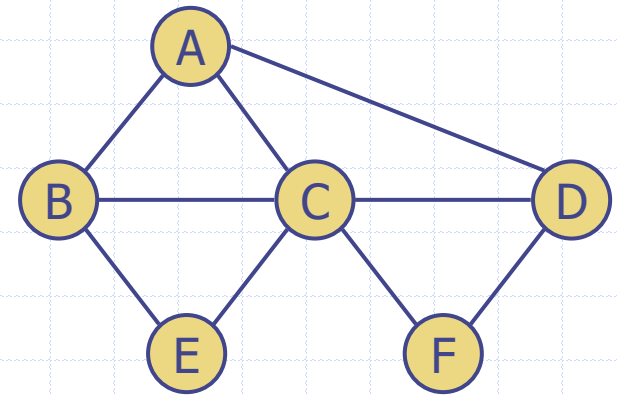
Property 2

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

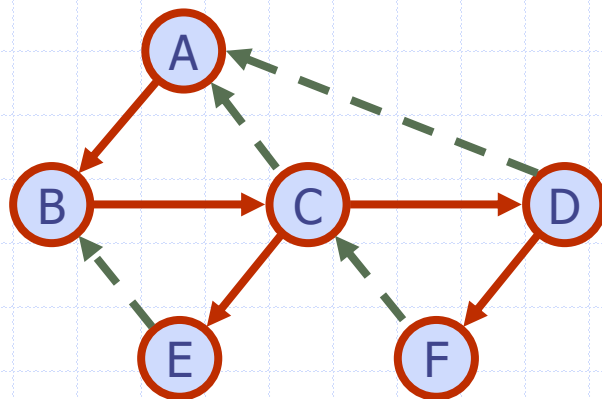
- ❑ Setting/getting a vertex/edge label takes $O(1)$ time
- ❑ Each vertex is labeled twice
 - once as UNEXPLORED
 - once as VISITED
- ❑ Each edge is labeled twice
 - once as UNEXPLORED
 - once as DISCOVERY or CROSS
- ❑ Each vertex is inserted once into a sequence L_i
- ❑ Method incidentEdges is called 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

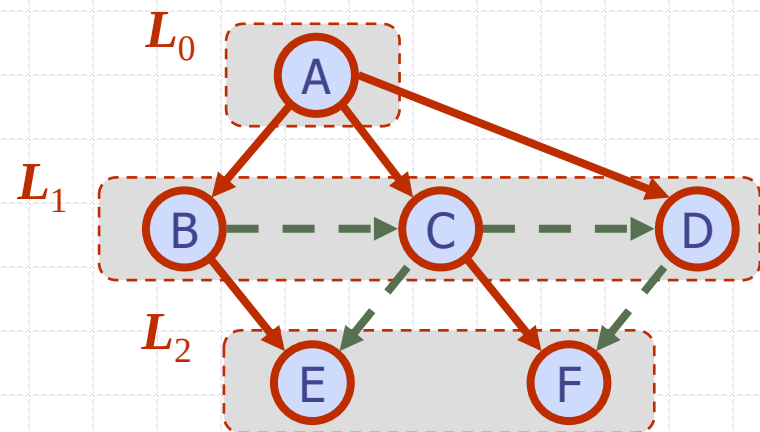
- Using the **template method pattern**, we can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time
 - Compute the connected components of G
 - Compute a spanning forest of G
 - Find a simple cycle in G , or report that G is a forest
 - Given two vertices of G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



DFS

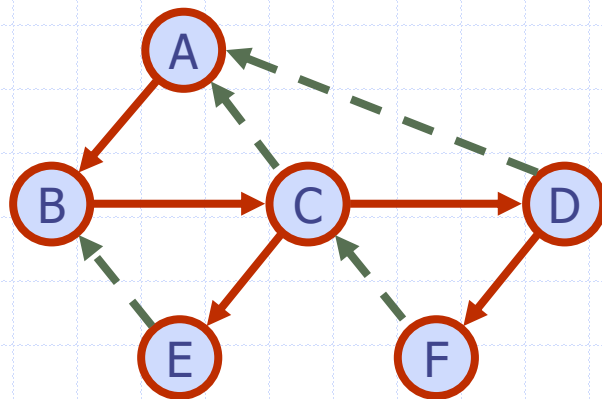


BFS

DFS vs. BFS (cont.)

Back edge (v,w)

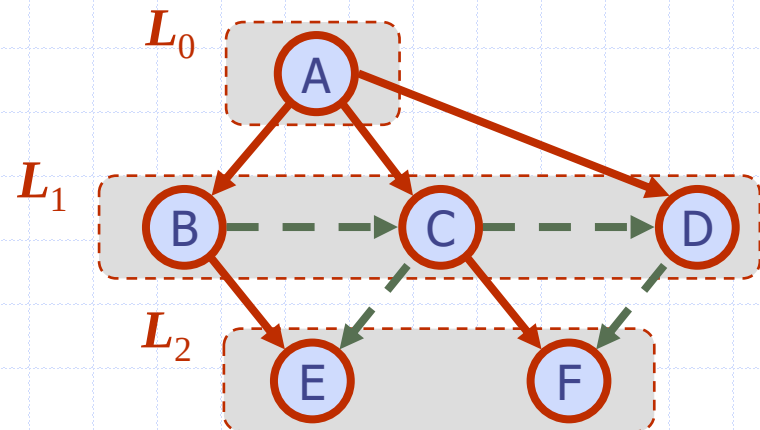
- w is an ancestor of v in the tree of discovery edges



DFS

Cross edge (v,w)

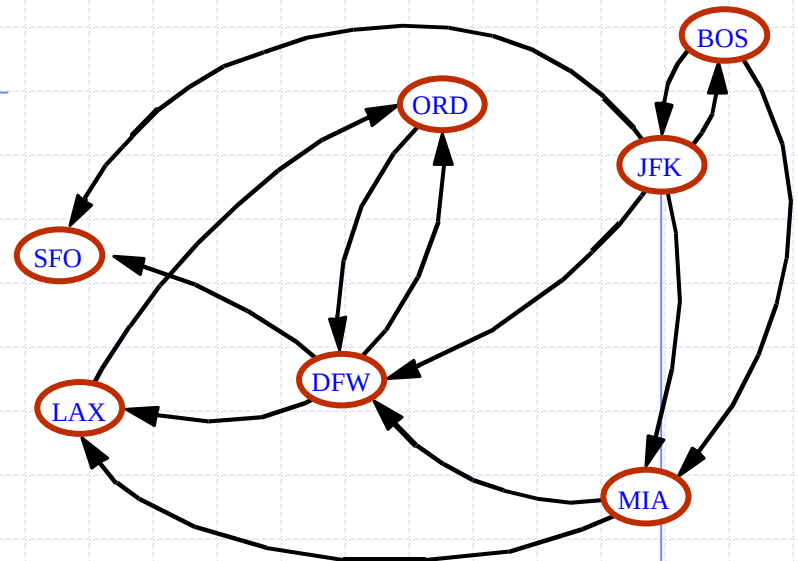
- w is in the same level as v or in the next level



BFS

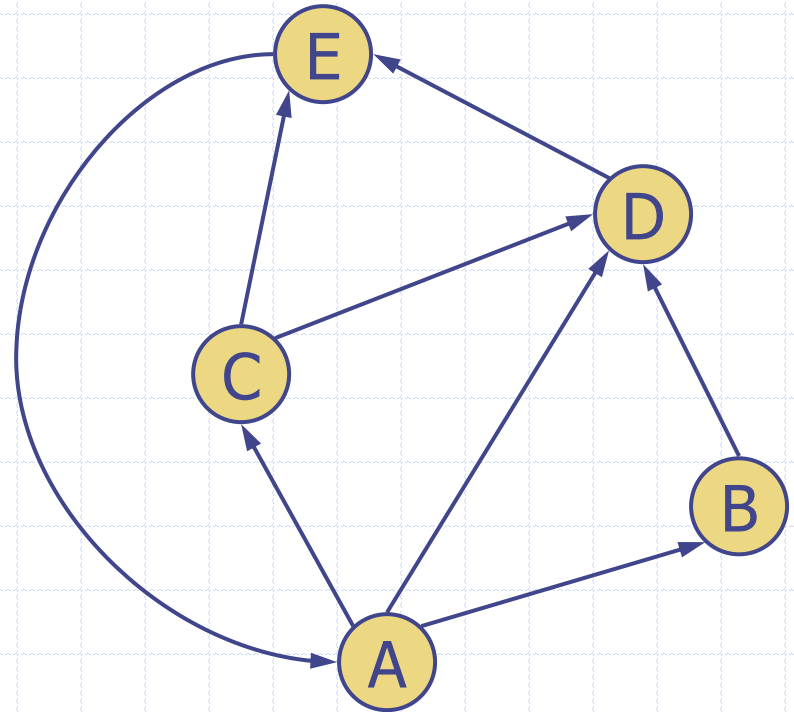
Presentation for use with the textbook **Data Structures and Algorithms in Java, 6th edition**, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Directed Graphs

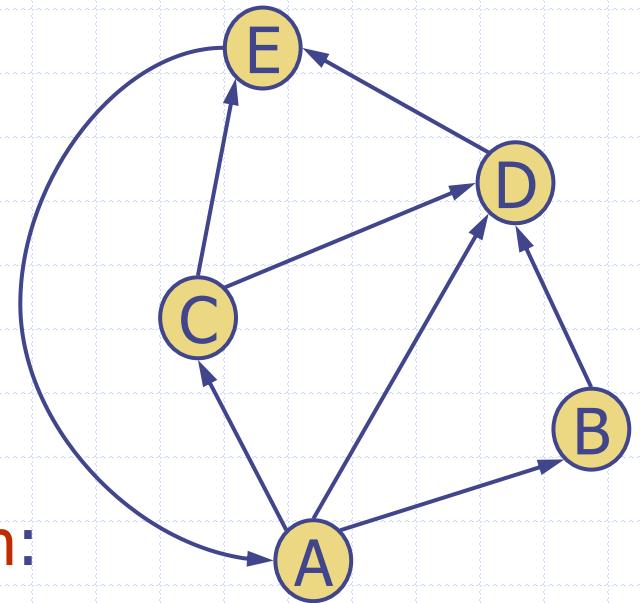


Digraphs

- A **digraph** is a graph whose edges are all directed
 - Short for “directed graph”
- Applications
 - one-way streets
 - flights
 - task scheduling



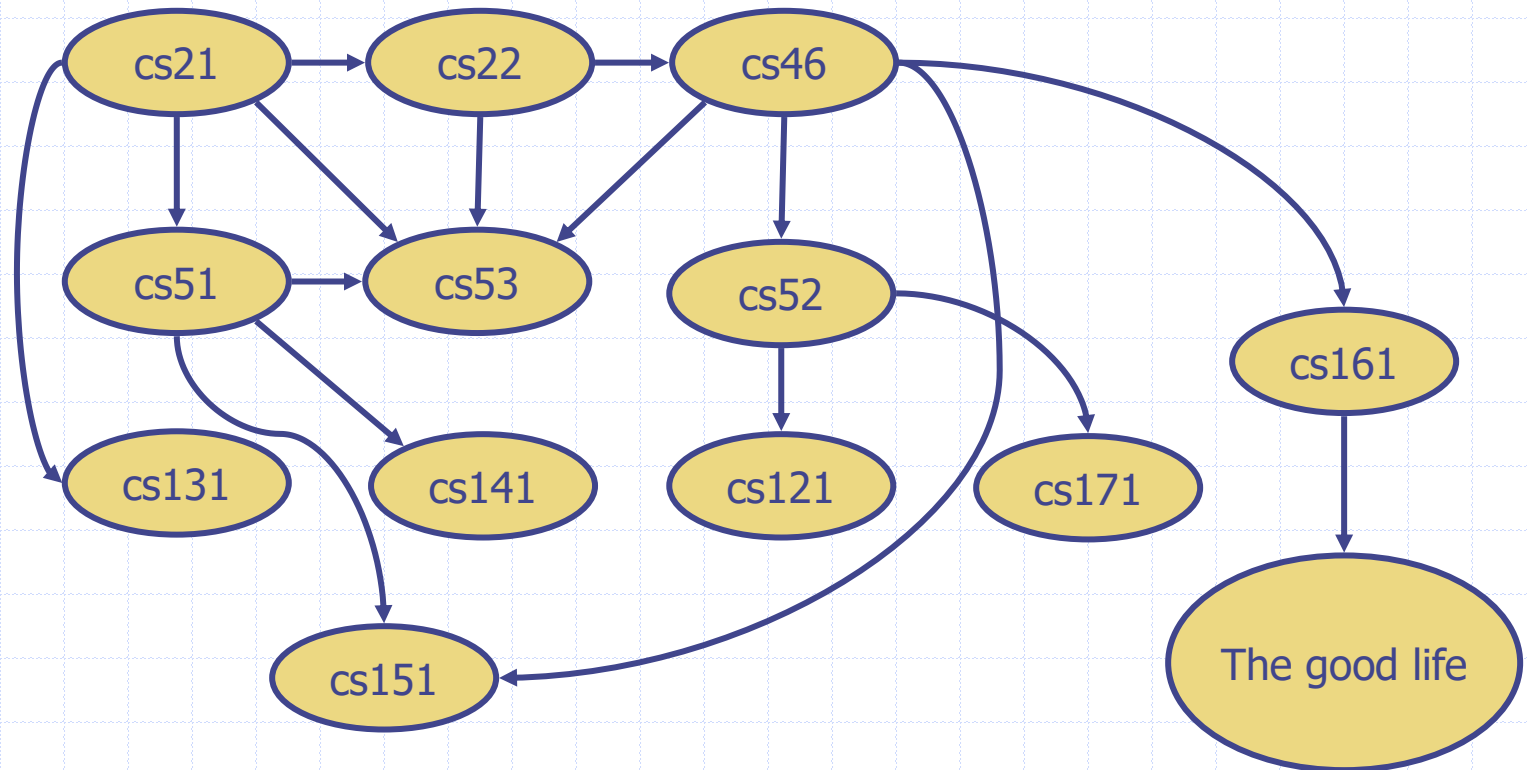
Digraph Properties



- A graph $G=(V,E)$ such that
 - Each edge goes in **one direction**:
 - Edge (a,b) goes from a to b , but not b to a
- If G is simple, **$m \leq n \cdot (n - 1)$**
- If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of incoming edges and outgoing edges in time proportional to their size

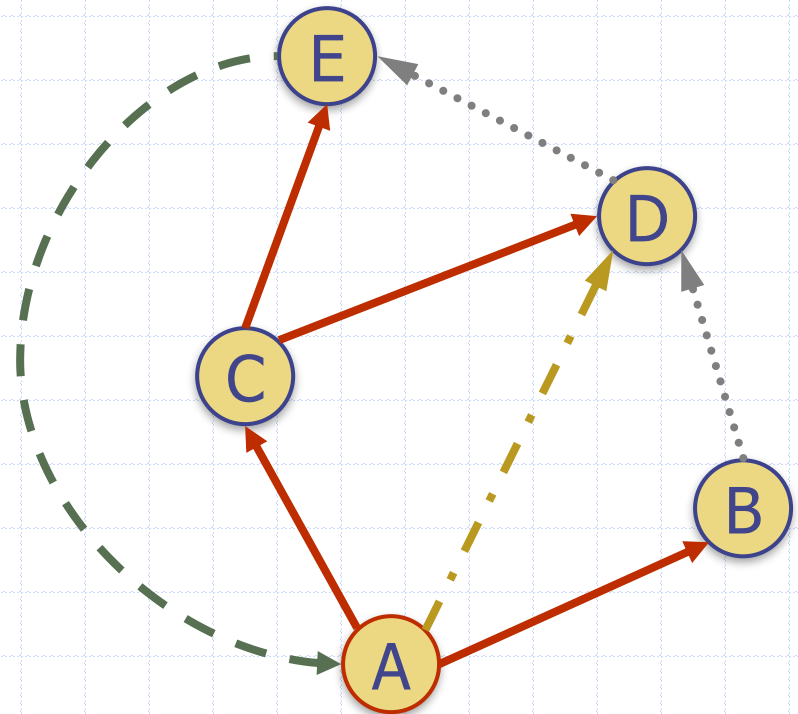
Digraph Application

- **Scheduling:** edge (a,b) means task a must be completed before b can be started



Directed DFS

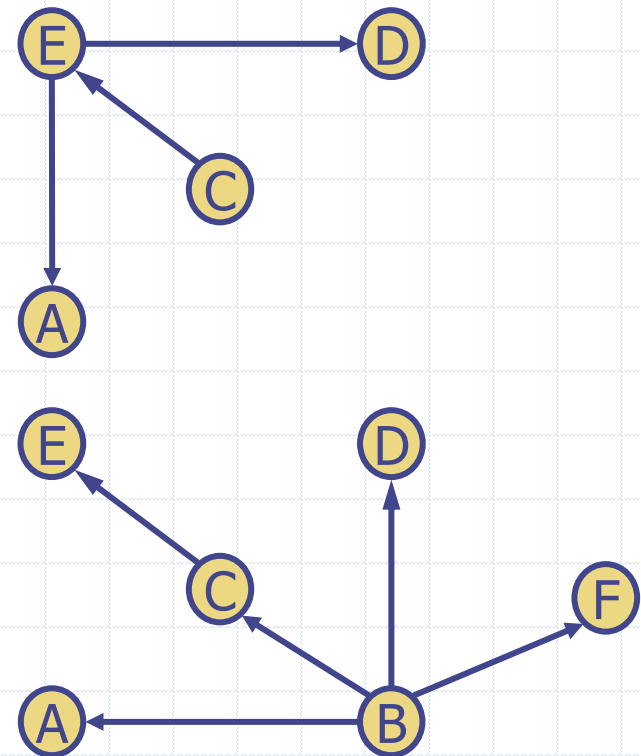
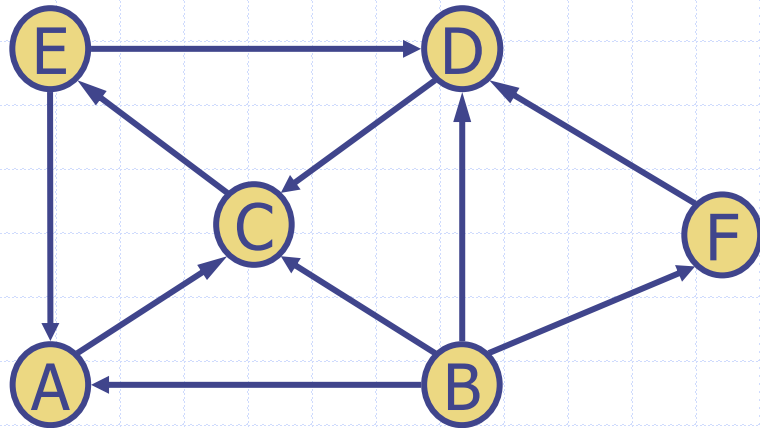
- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction
- In the directed DFS algorithm, we have four types of edges
 - **discovery edges**
 - back edges
 - **forward edges**
 - cross edges
- A directed DFS starting at a vertex s determines the vertices **reachable** from s

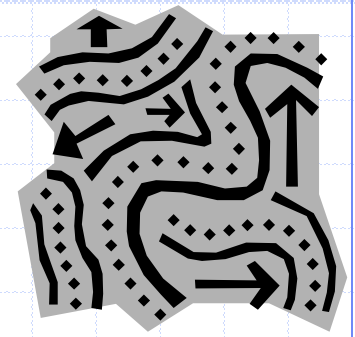




Reachability

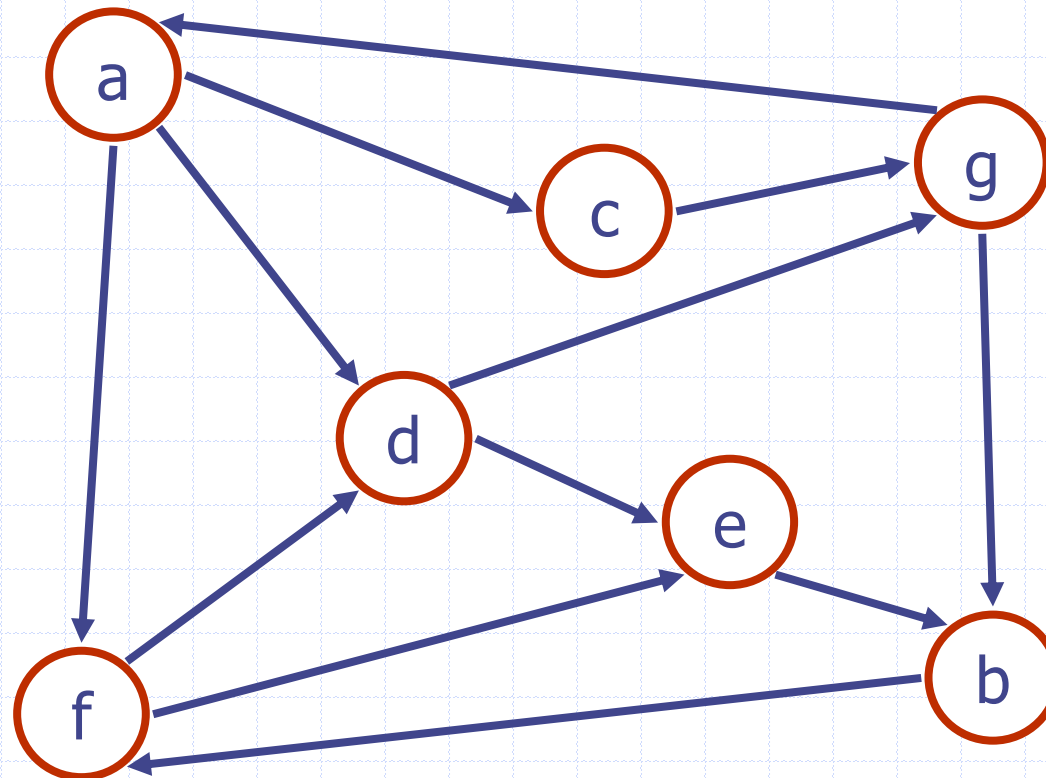
- DFS **tree** rooted at v : vertices reachable from v via directed paths



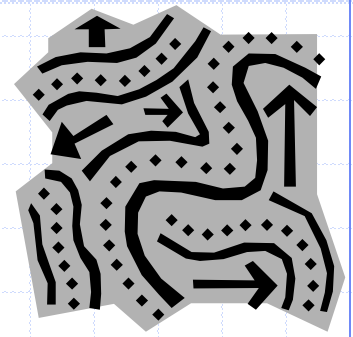


Strong Connectivity

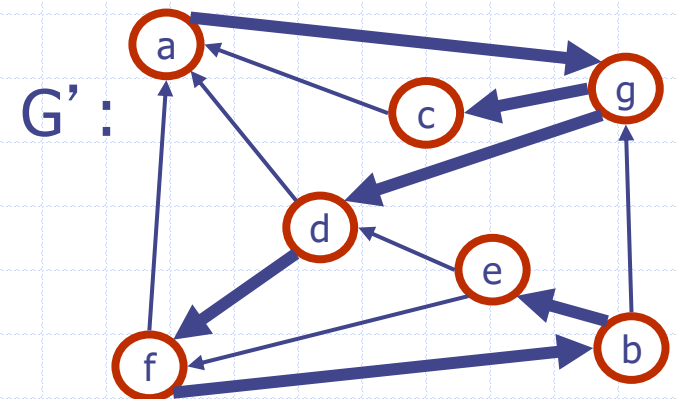
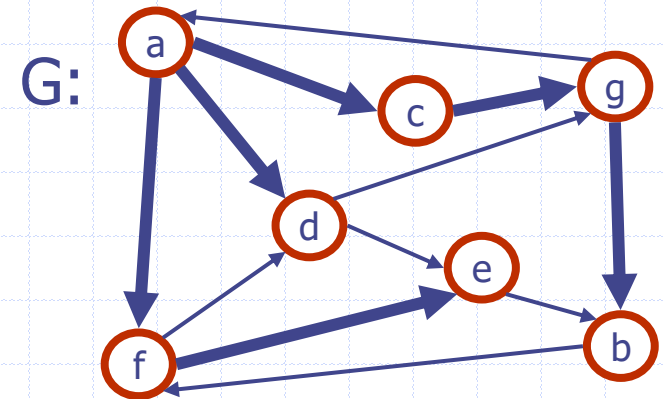
- Each vertex can reach all other vertices



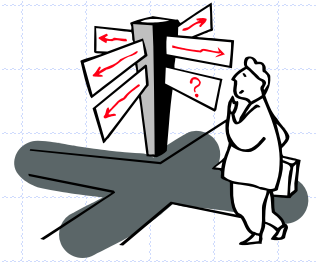
Strong Connectivity Algorithm



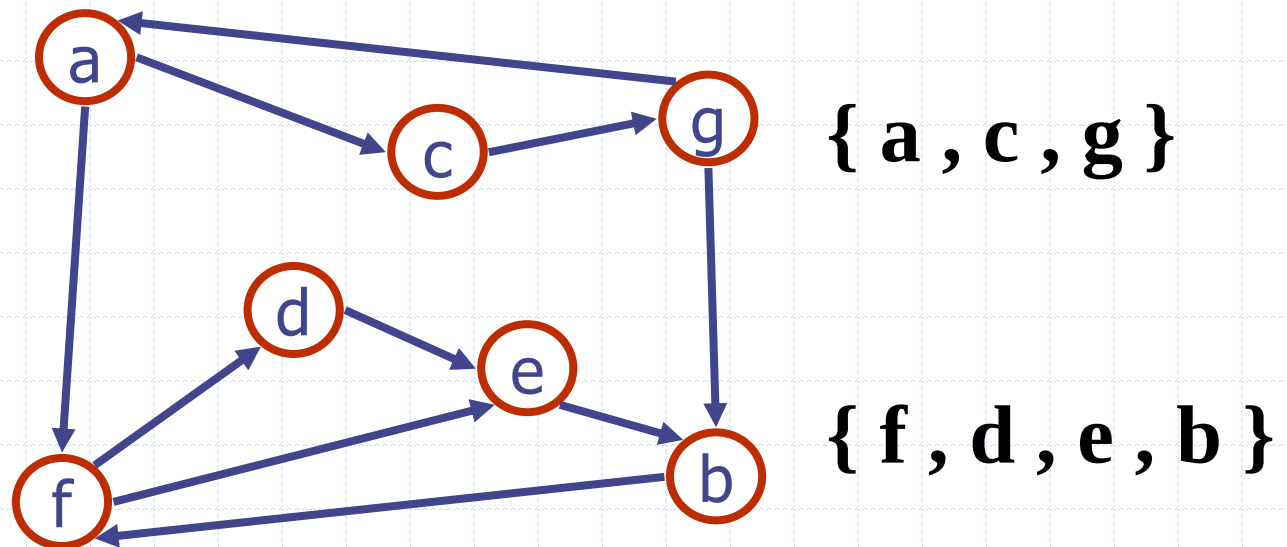
- ❑ Pick a vertex v in G
- ❑ Perform a DFS from v in G
 - If there's a w not visited, print "no"
- ❑ Let G' be G with edges reversed
- ❑ Perform a DFS from v in G'
 - If there's a w not visited, print "no"
 - Else, print "yes"
- ❑ Running time: $O(n+m)$



Strongly Connected Components



- Maximal subgraphs such that each vertex can reach all other vertices in the subgraph
- Can also be done in $O(n+m)$ time using DFS, but is more complicated (similar to biconnectivity).

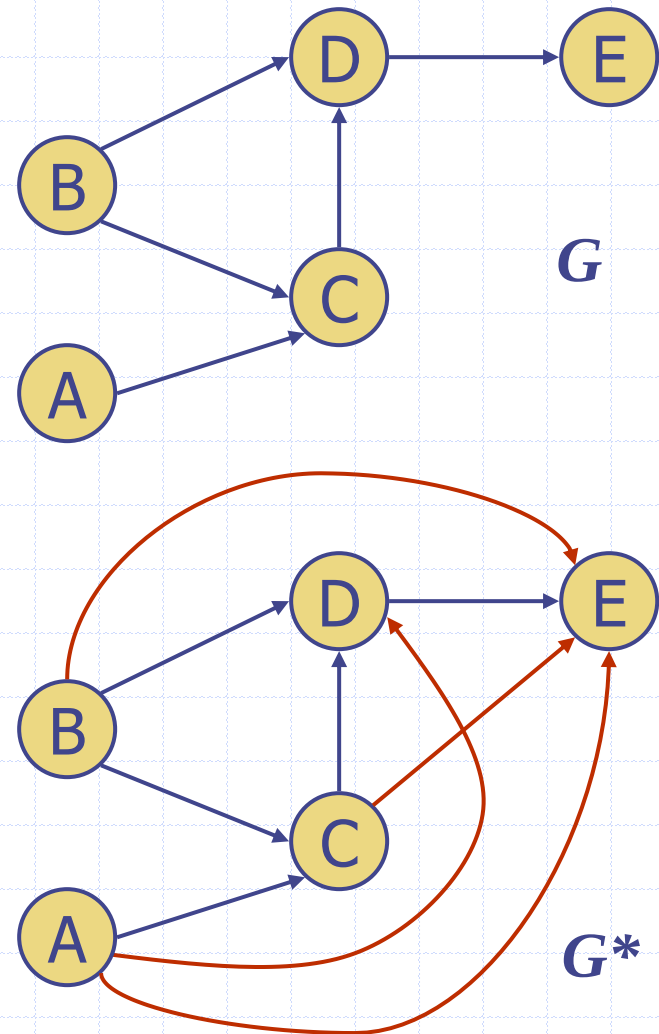


Biconnected graph

- In [graph theory](#), a **biconnected graph** is a connected and "nonseparable" [graph](#), meaning that if any [vertex](#) were to be removed, the graph will remain connected. Therefore a biconnected graph has no [articulation vertices](#).
- The property of being [2-connected](#) is equivalent to biconnectivity, with the caveat that the [complete graph](#) of two vertices is sometimes regarded as biconnected but not 2-connected.
- A **biconnected** [undirected graph](#) is a connected graph that is not broken into disconnected pieces by deleting any single vertex (and its incident edges).
- A **biconnected** [directed graph](#) is one such that for any two vertices v and w there are two directed paths from v to w which have no vertices in common other than v and w .

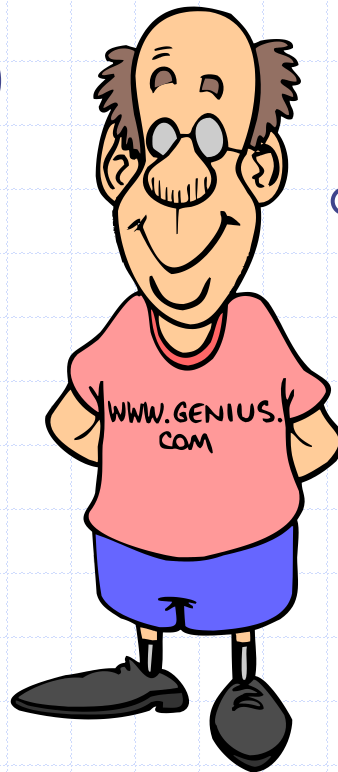
Transitive Closure

- Given a digraph G , the transitive closure of G is the digraph G^* such that
 - G^* has the same vertices as G
 - if G has a directed path from u to v ($u \neq v$), G^* has a directed edge from u to v
- The transitive closure provides reachability information about a digraph



Computing the Transitive Closure

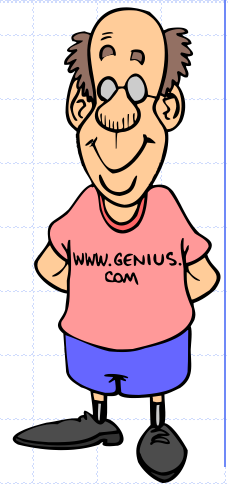
- We can perform DFS starting at each vertex
 - $O(n(n+m))$



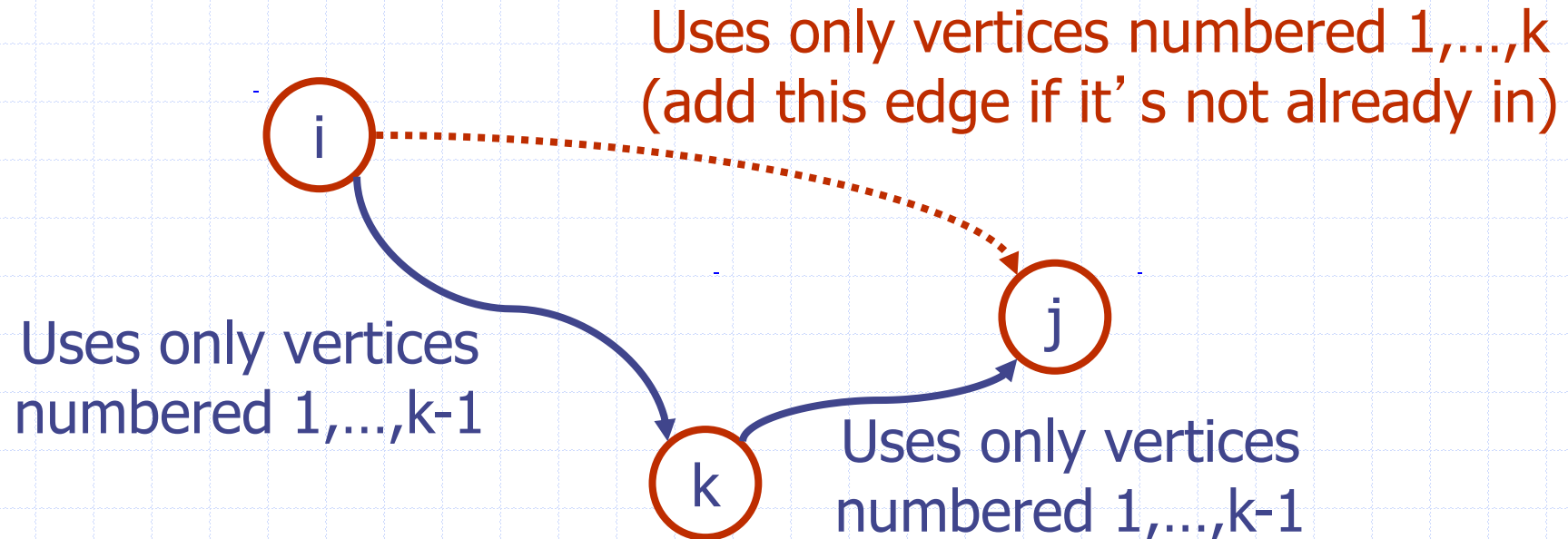
If there's a way to get from **A** to **B** and from **B** to **C**, then there's a way to get from **A** to **C**.

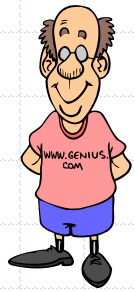
Alternatively ... Use dynamic programming:
The Floyd-Warshall Algorithm

Floyd-Warshall Transitive Closure



- ❑ Idea #1: Number the vertices $1, 2, \dots, n$.
- ❑ Idea #2: Consider paths that use only vertices numbered $1, 2, \dots, k$, as intermediate vertices:





Floyd-Warshall's Algorithm

- Number vertices $v_1, \dots, v_n$
- Compute digraphs $G_0, \dots, G_n$
 - $G_0 = G$
 - G_k has directed edge (v_i, v_j) if G has a directed path from v_i to v_j with intermediate vertices in $\{v_1, \dots, v_k\}$
- We have that $G_n = G^*$
- In phase k , digraph G_k is computed from G_{k-1}
- Running time: $O(n^3)$, assuming `areAdjacent` is $O(1)$ (e.g., adjacency matrix)

Algorithm *FloydWarshall*(G)

Input digraph G

Output transitive closure G^* of G

$i \leftarrow 1$

for all $v \in G.vertices()$

denote v as v_i

$i \leftarrow i + 1$

$G_0 \leftarrow G$

for $k \leftarrow 1$ **to** n **do**

$G_k \leftarrow G_{k-1}$

for $i \leftarrow 1$ **to** n ($i \neq k$) **do**

for $j \leftarrow 1$ **to** n ($j \neq i, k$) **do**

if $G_{k-1}.areAdjacent(v_i, v_k) \wedge$

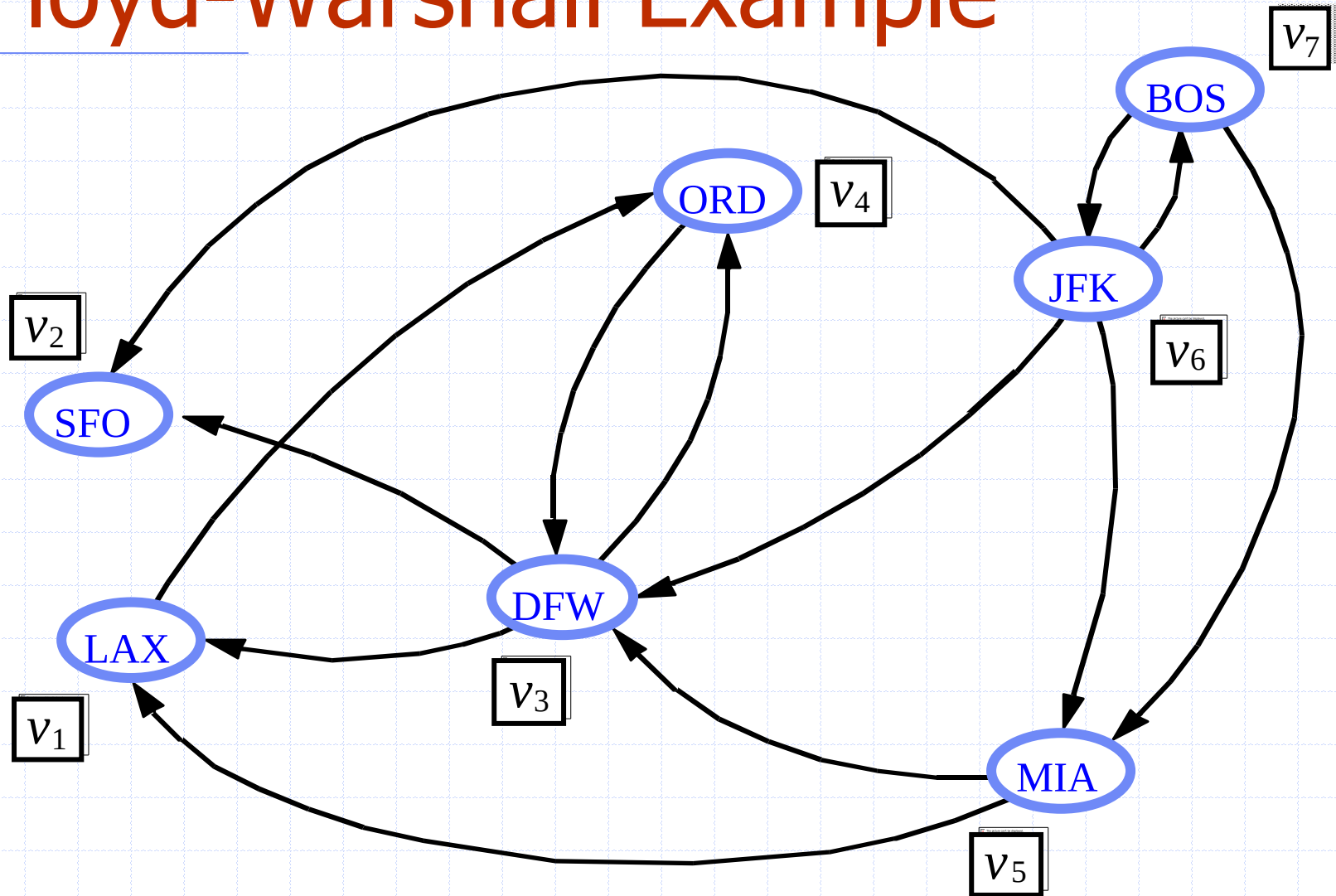
$G_{k-1}.areAdjacent(v_k, v_j)$

if $\neg G_k.areAdjacent(v_i, v_j)$

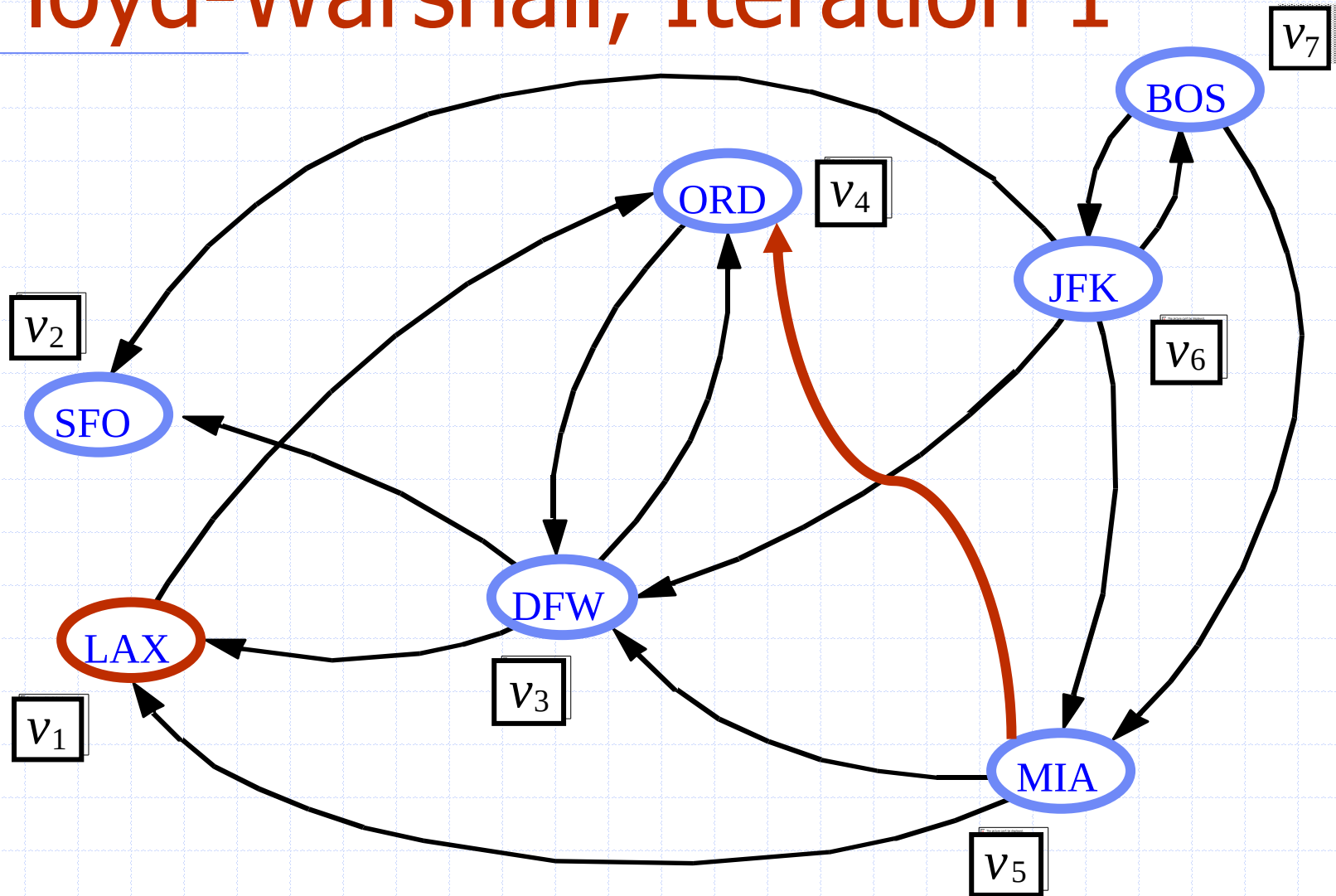
$G_k.insertDirectedEdge(v_i, v_j, k)$

return G_n

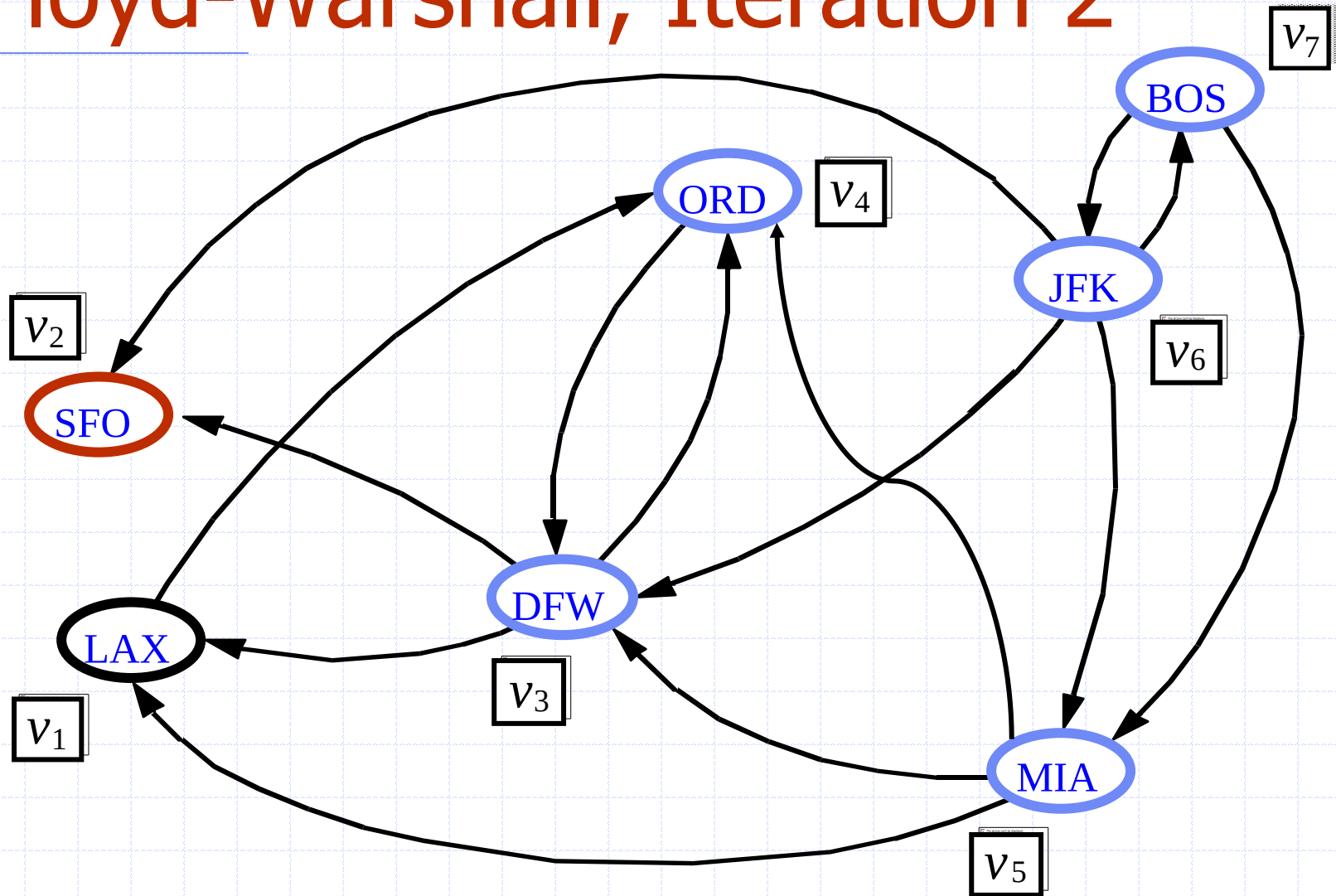
Floyd-Warshall Example



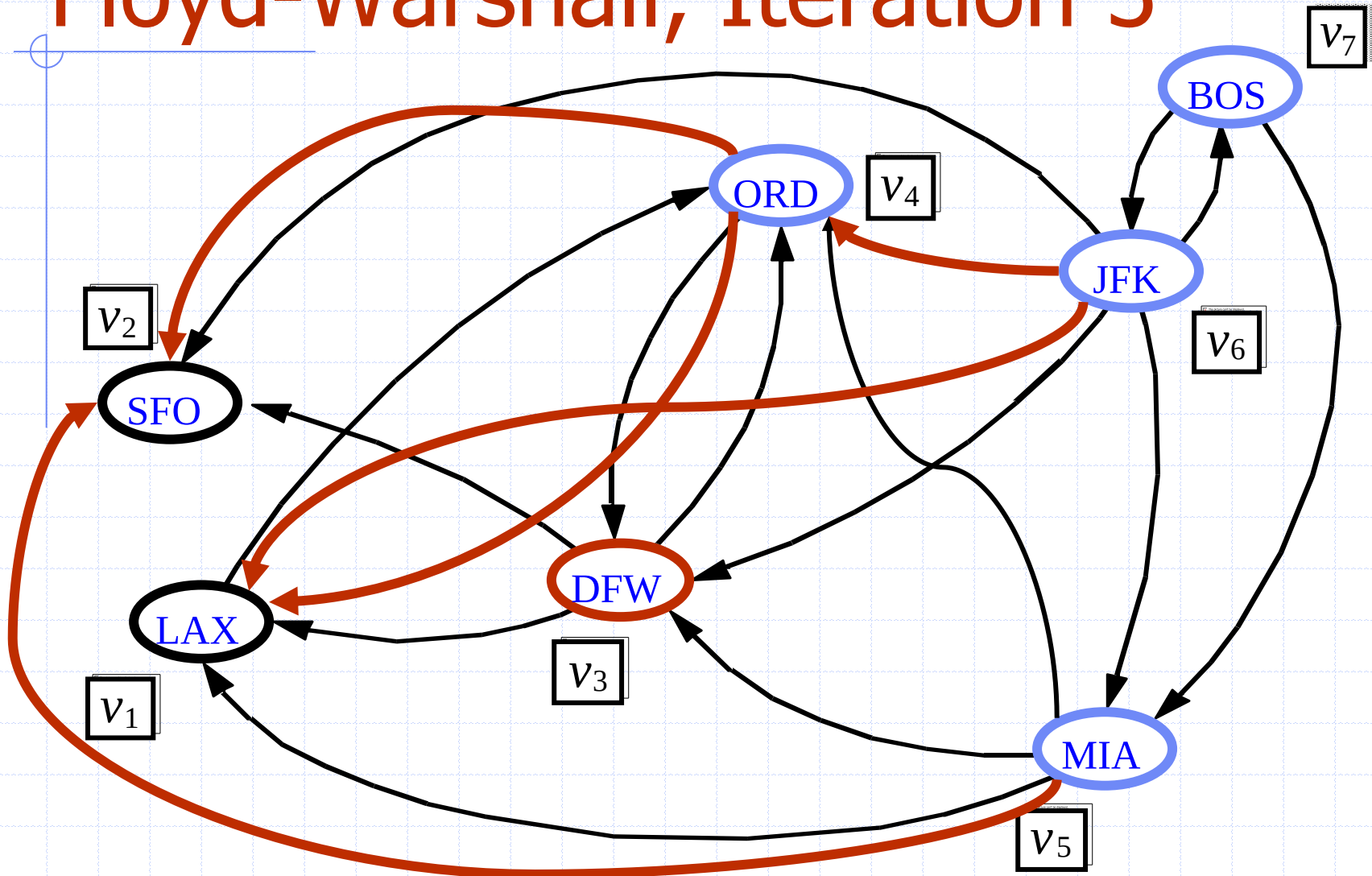
Floyd-Warshall, Iteration 1



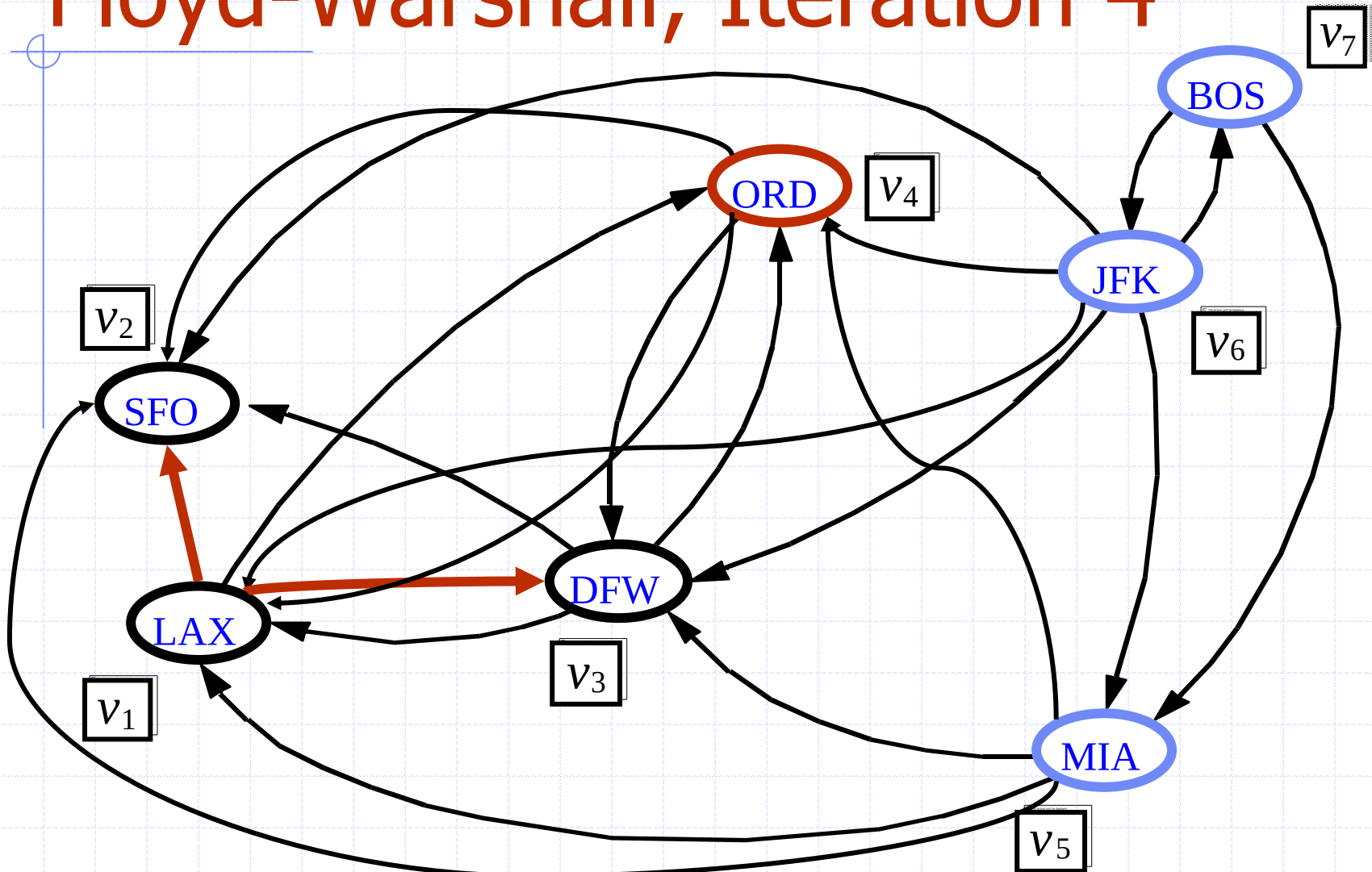
Floyd-Warshall, Iteration 2



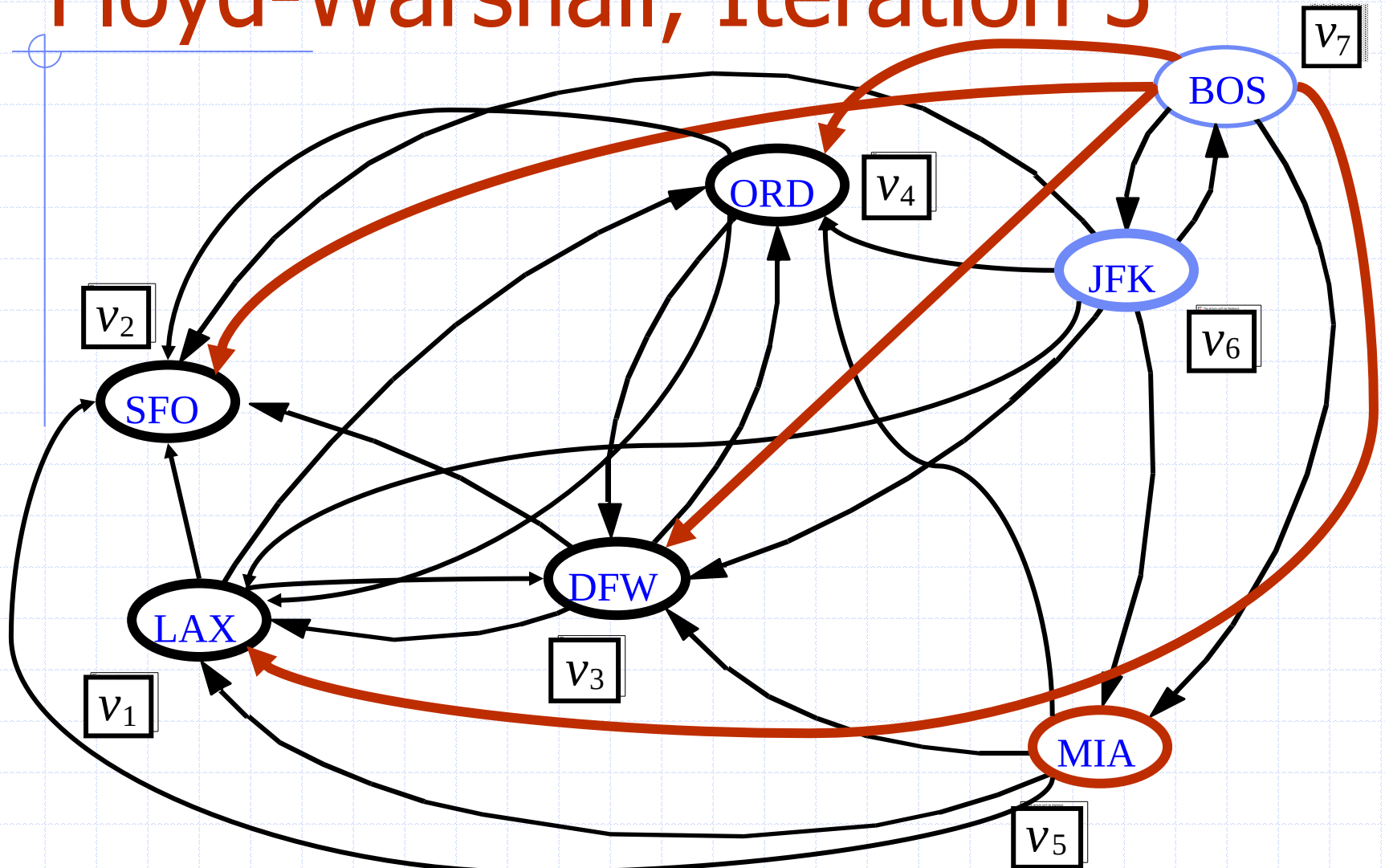
Floyd-Warshall, Iteration 3



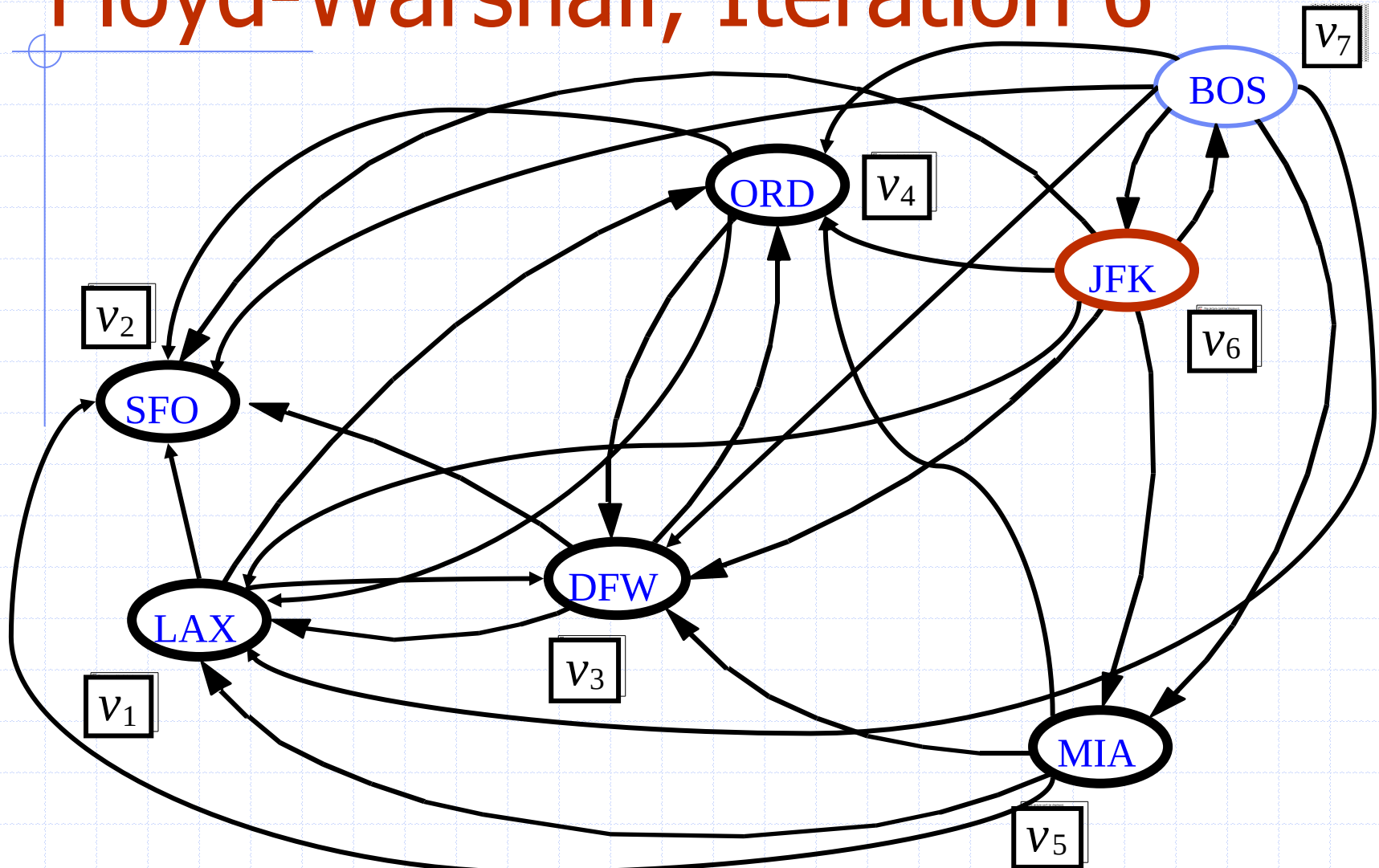
Floyd-Warshall, Iteration 4



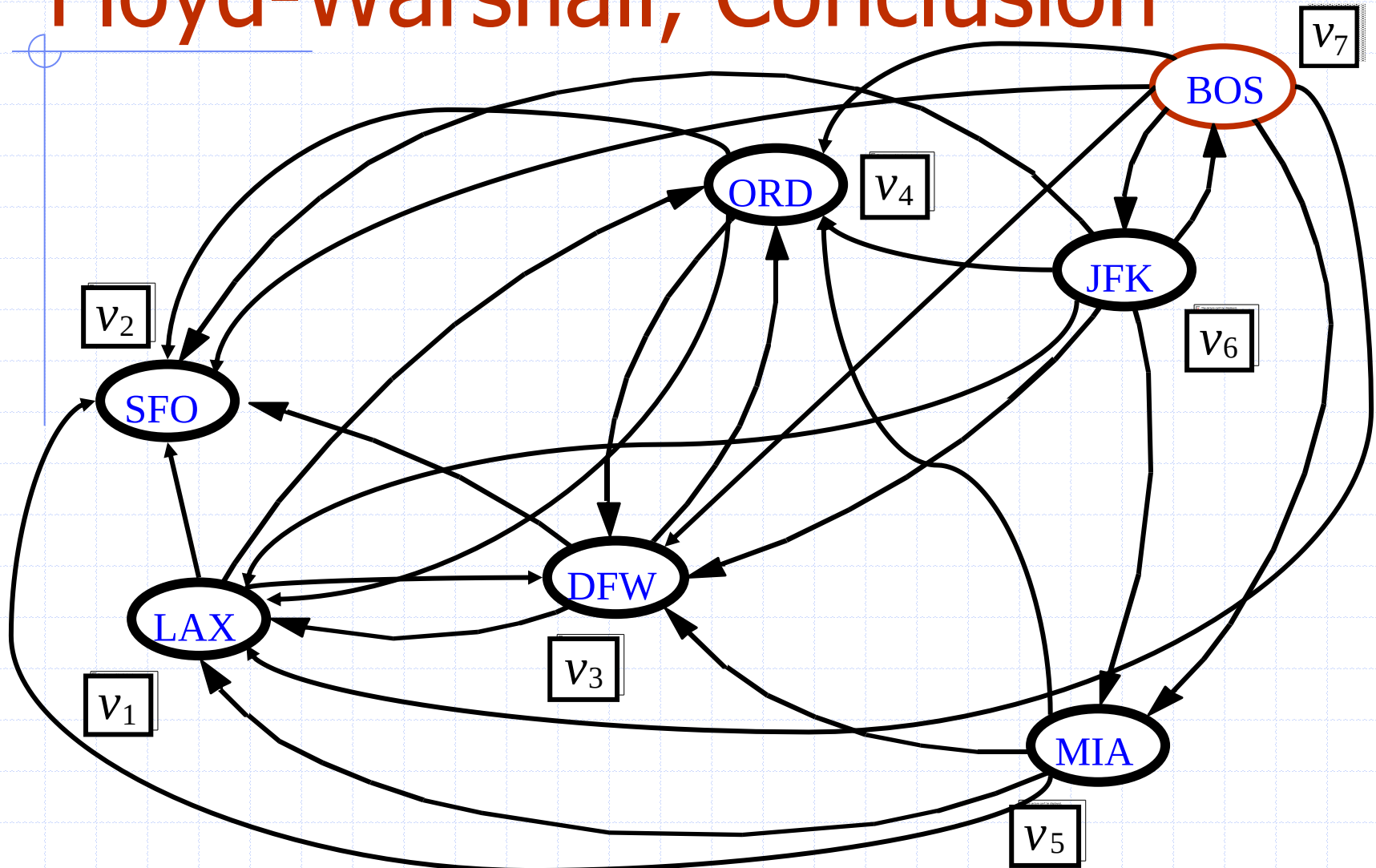
Floyd-Warshall, Iteration 5



Floyd-Warshall, Iteration 6



Floyd-Warshall, Conclusion



DAGs and Topological Ordering

- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

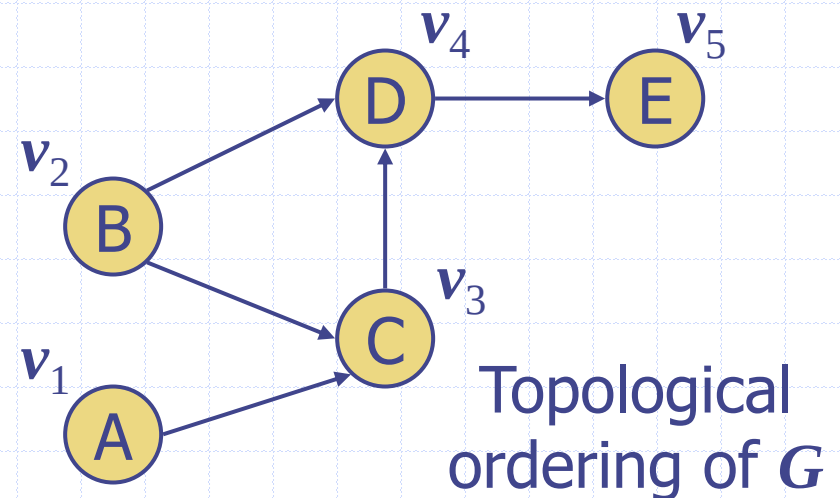
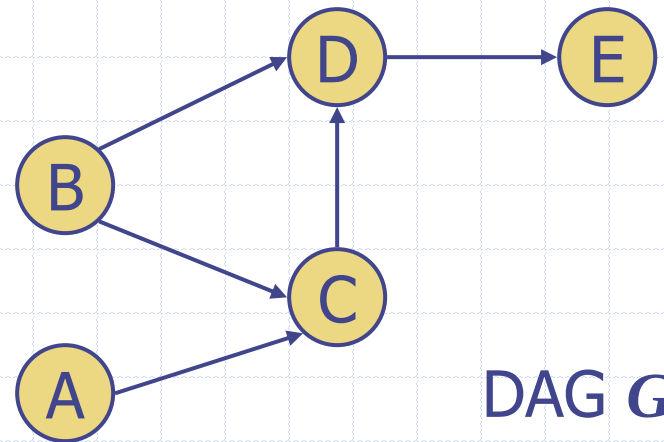
$$v_1, \dots, v_n$$

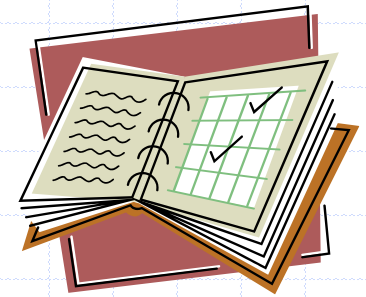
of the vertices such that for every edge (v_i, v_j) , we have $i < j$

- Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints

Theorem

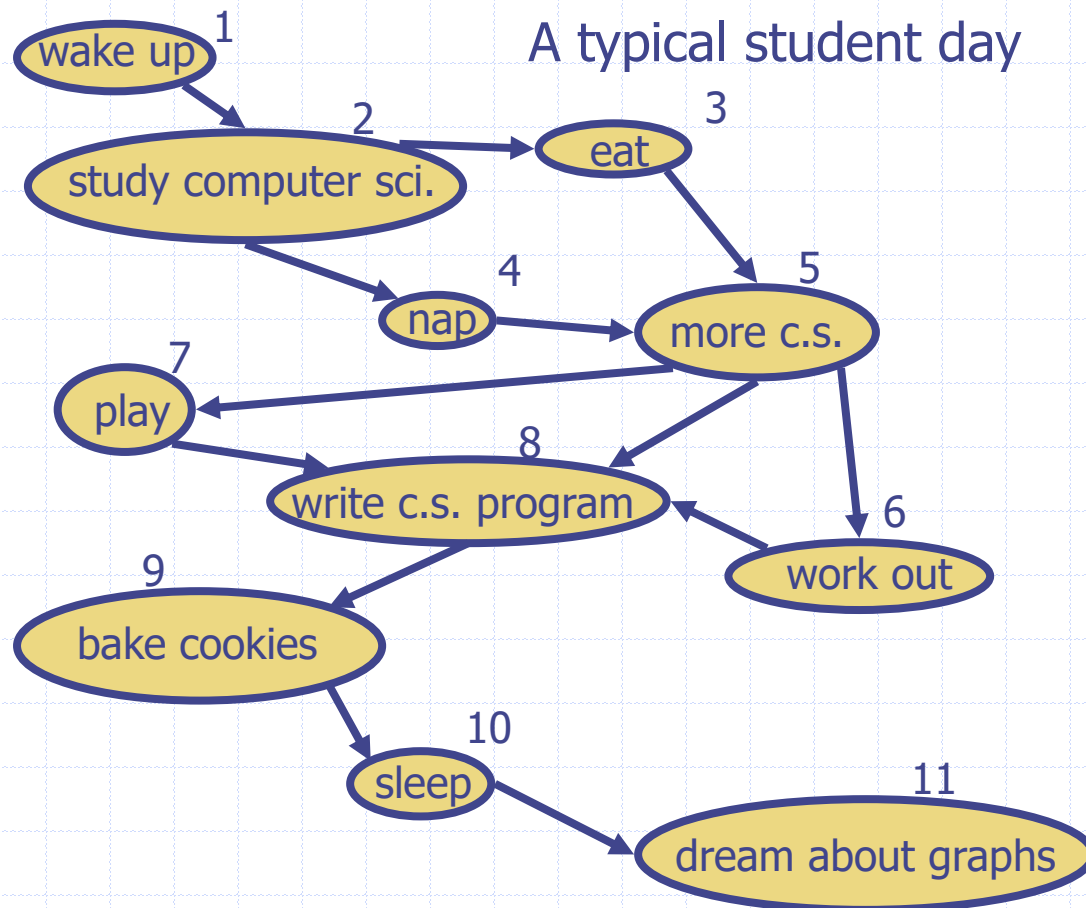
A digraph admits a topological ordering if and only if it is a DAG





Topological Sorting

- Number vertices, so that (u,v) in E implies $u < v$



Algorithm for Topological Sorting

- Note: This algorithm is different than the one in the book

Algorithm TopologicalSort(G)

$H \leftarrow G$ // Temporary copy of G

$n \leftarrow G.\text{numVertices}()$

while H is not empty **do**

Let v be a vertex with no outgoing edges

Label $v \leftarrow n$

$n \leftarrow n - 1$

Remove v from H

- Running time: $O(n + m)$

Implementation with DFS

- Simulate the algorithm by using depth-first search
- $O(n+m)$ time.

Algorithm *topologicalDFS(G)*

Input dag G

Output topological ordering of G
 $n \leftarrow G.\text{numVertices}()$

for all $u \in G.\text{vertices}()$

$\text{setLabel}(u, \text{UNEXPLORED})$

for all $v \in G.\text{vertices}()$

if $\text{getLabel}(v) = \text{UNEXPLORED}$
 $\text{topologicalDFS}(G, v)$

Algorithm *topologicalDFS(G, v)*

Input graph G and a start vertex v of G

Output labeling of the vertices of G
in the connected component of v

$\text{setLabel}(v, \text{VISITED})$

for all $e \in G.\text{outEdges}(v)$

 { outgoing edges }

$w \leftarrow \text{opposite}(v, e)$

if $\text{getLabel}(w) = \text{UNEXPLORED}$

 { e is a discovery edge }

$\text{topologicalDFS}(G, w)$

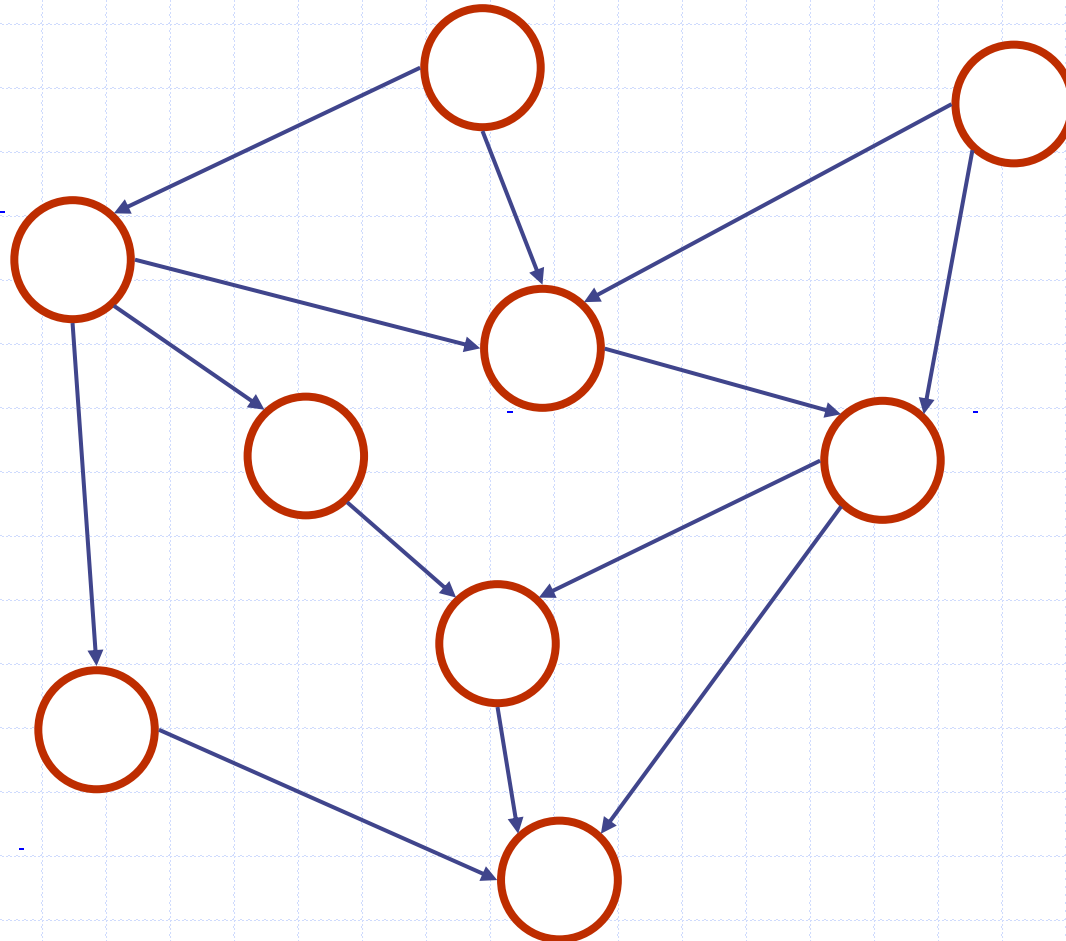
else

 { e is a forward or cross edge }

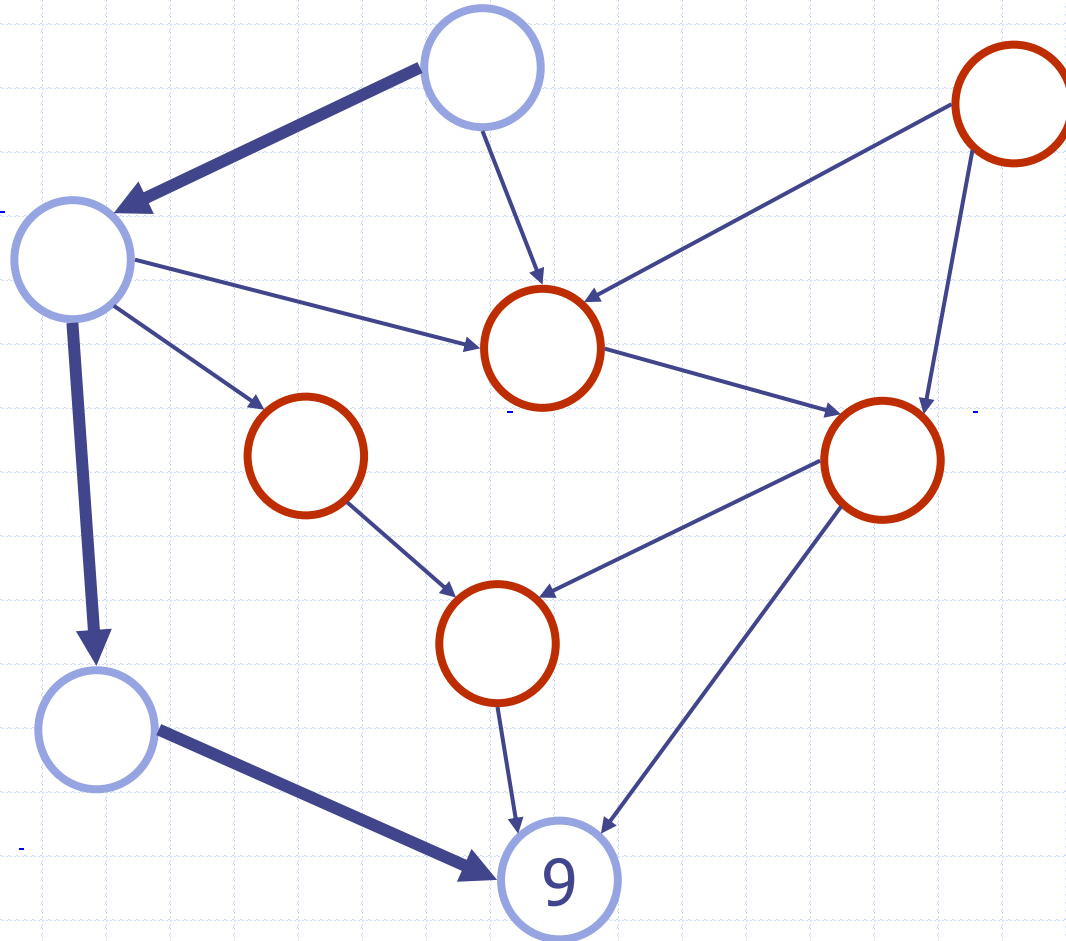
Label v with topological number n

$n \leftarrow n - 1$

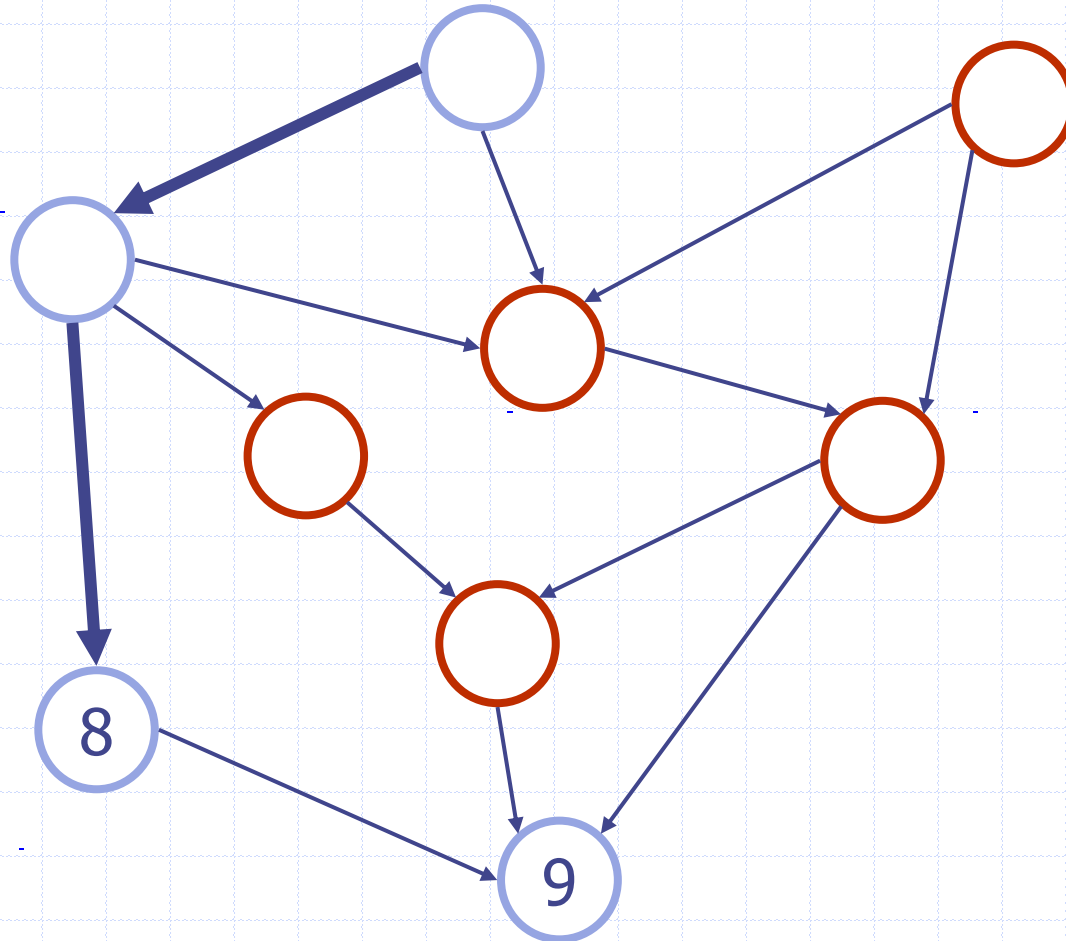
Topological Sorting Example

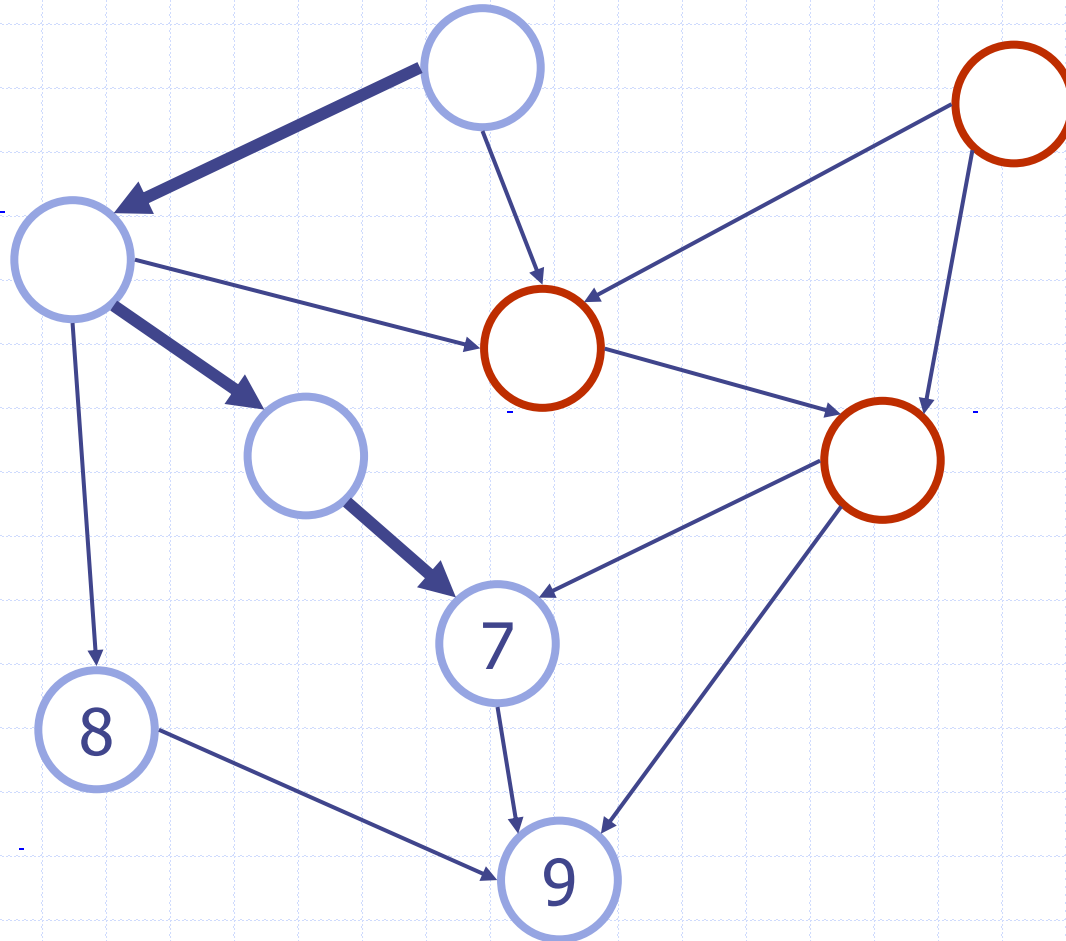


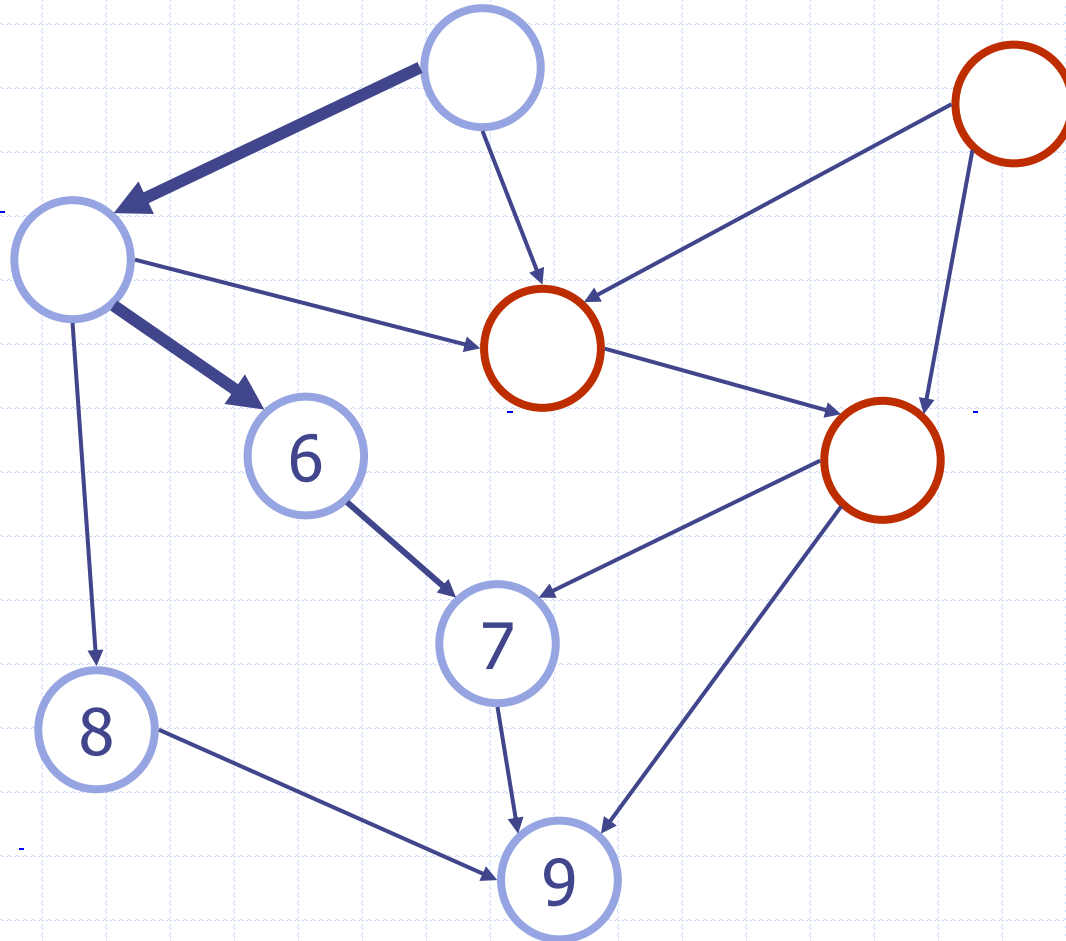
Topological Sorting Example



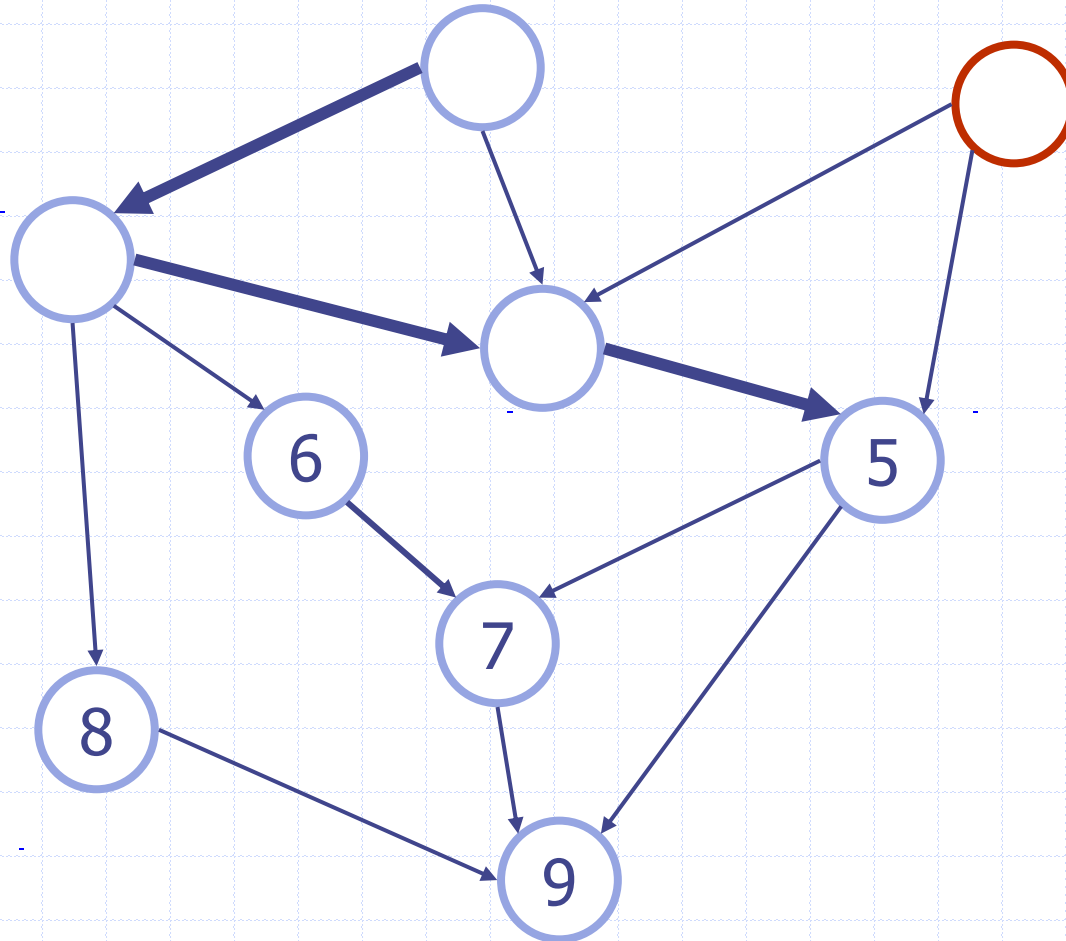
Topological Sorting Example



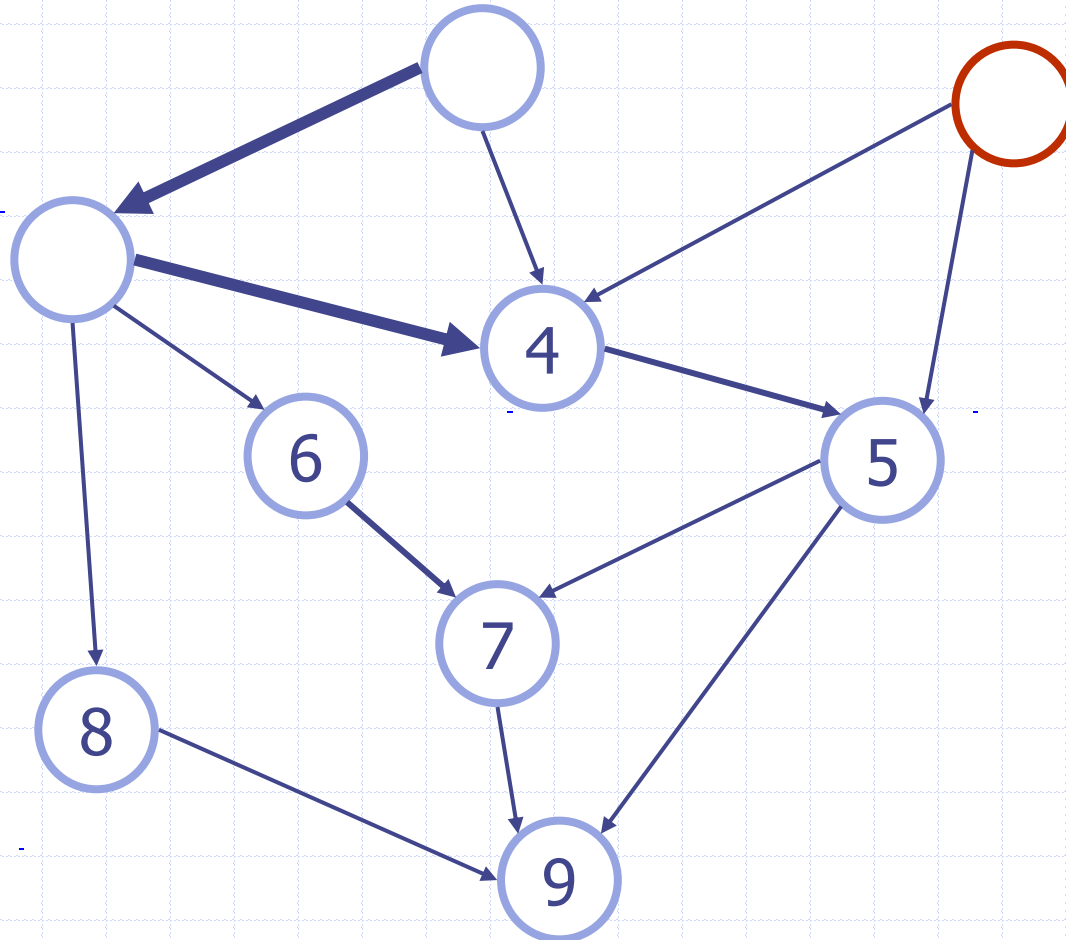




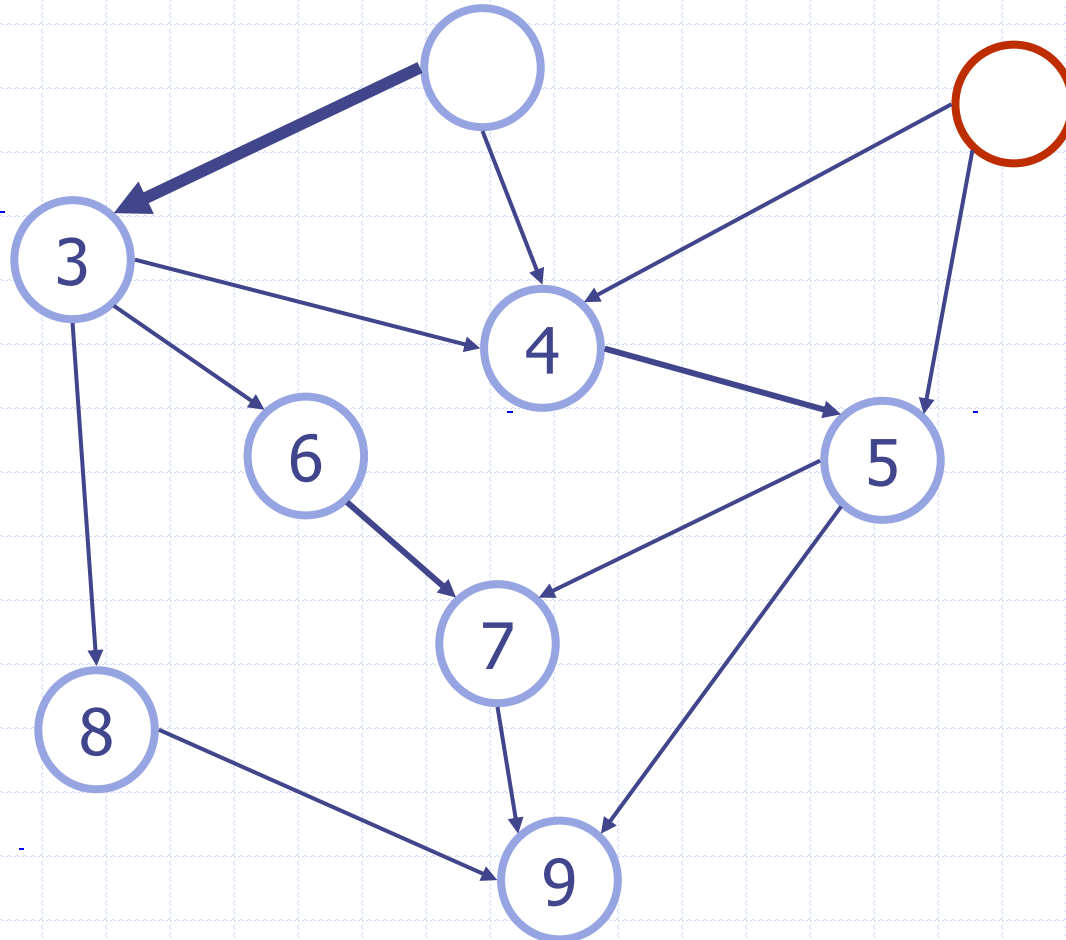
Topological Sorting Example



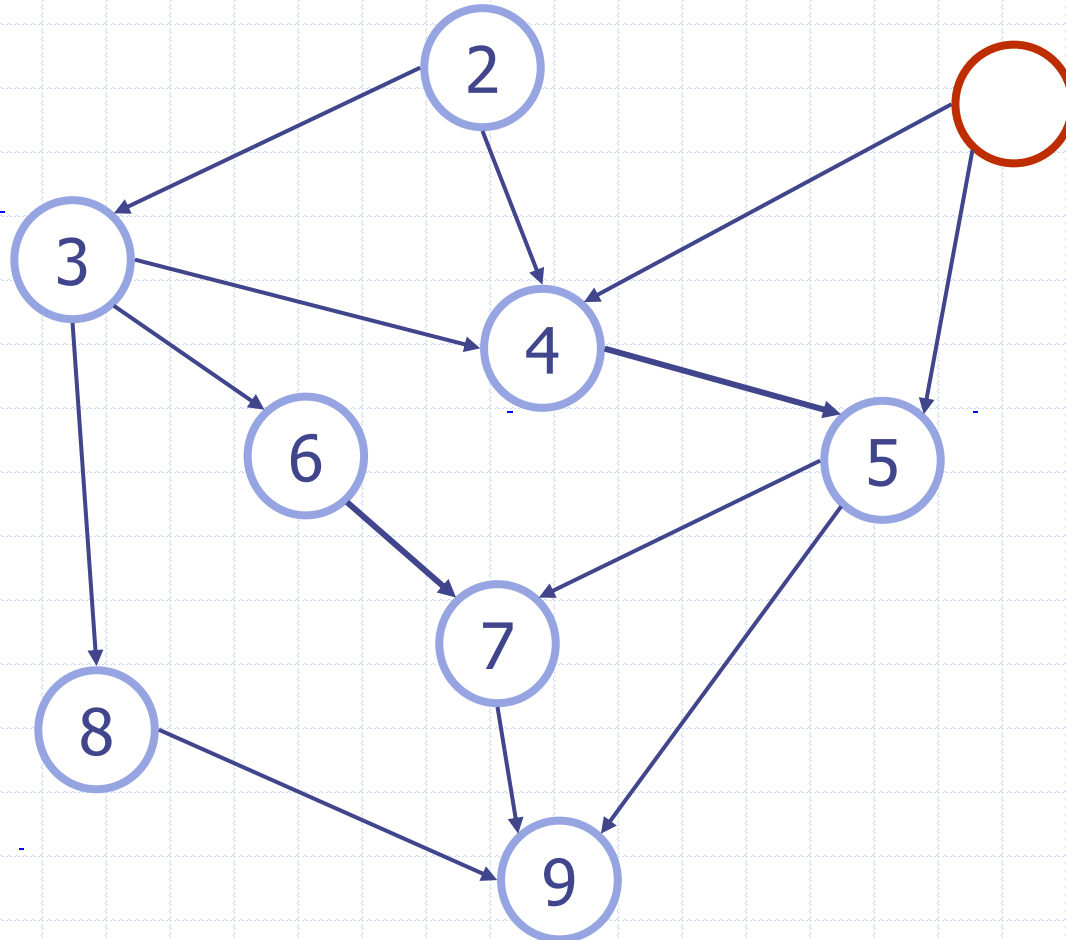
Topological Sorting Example



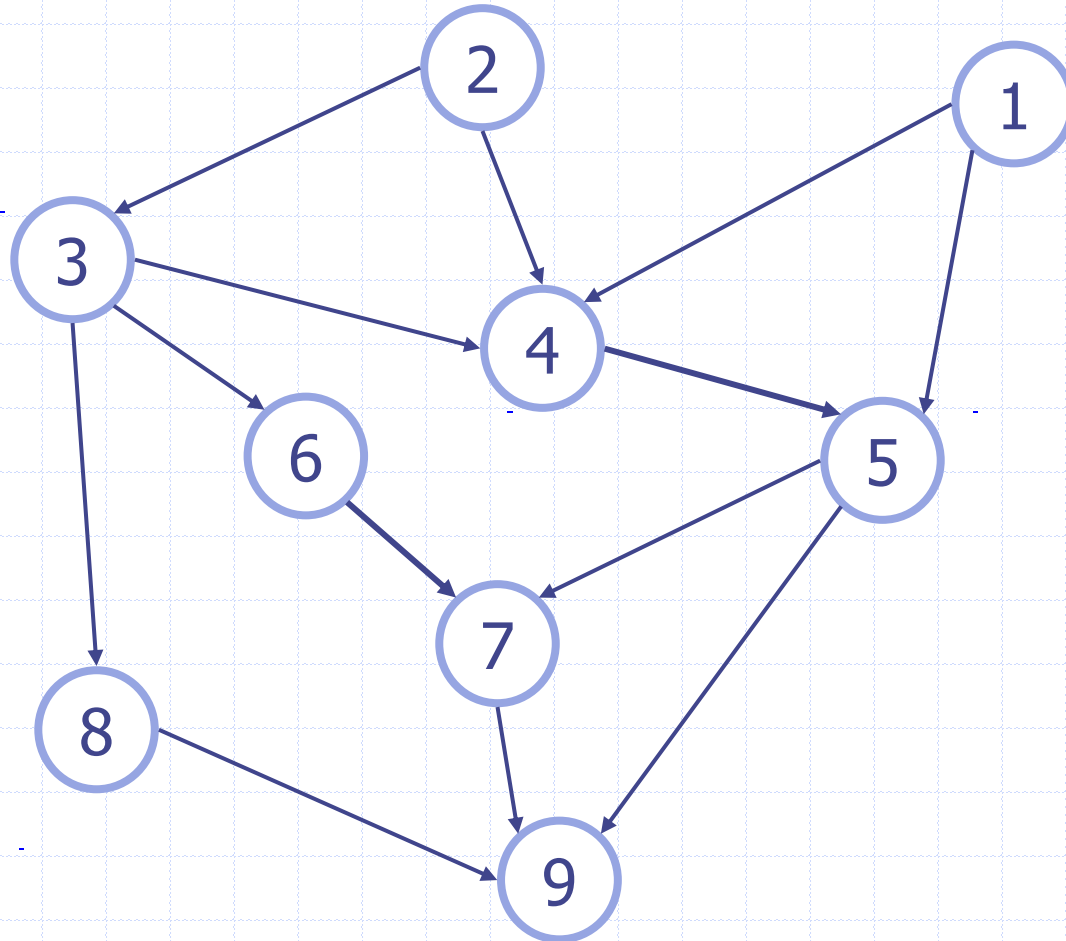
Topological Sorting Example



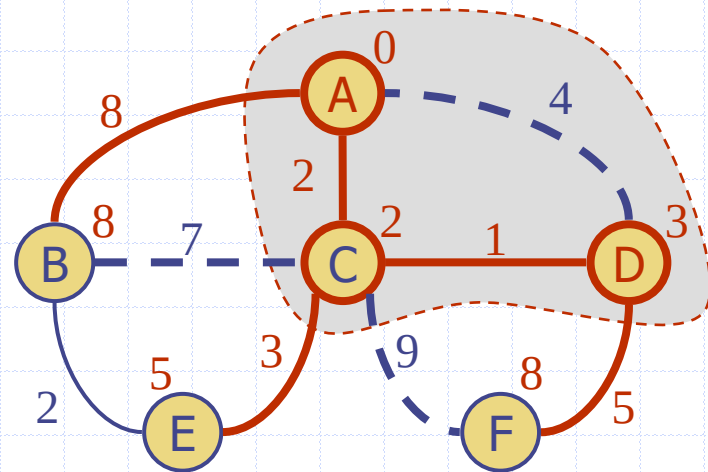
Topological Sorting Example



Topological Sorting Example

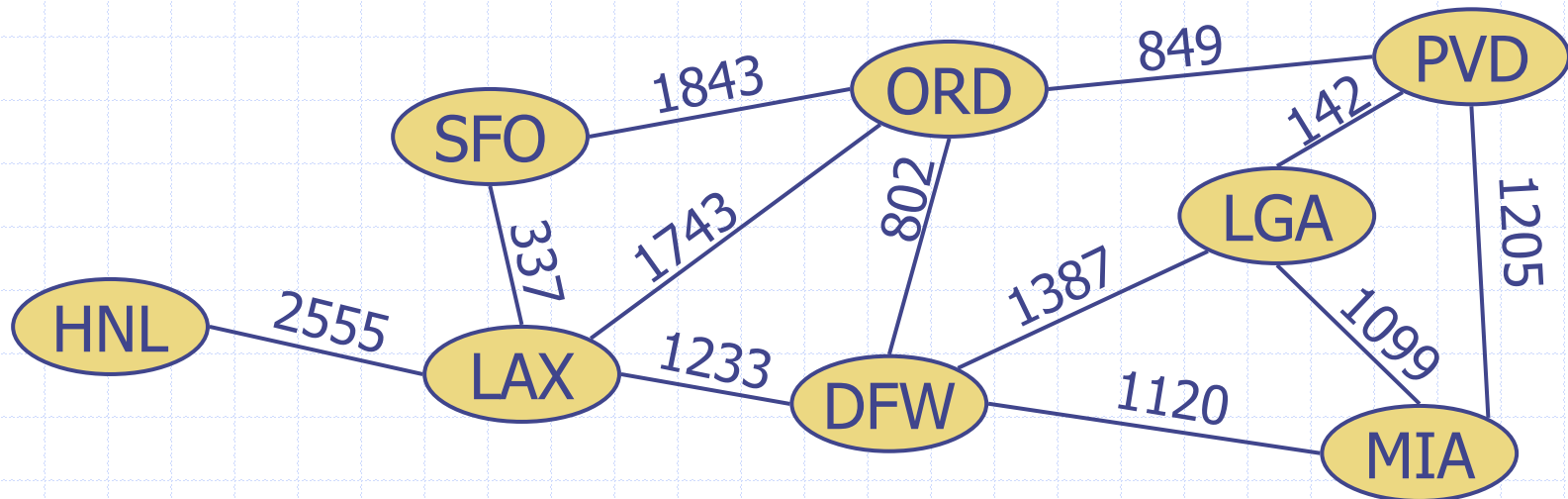


Shortest Paths



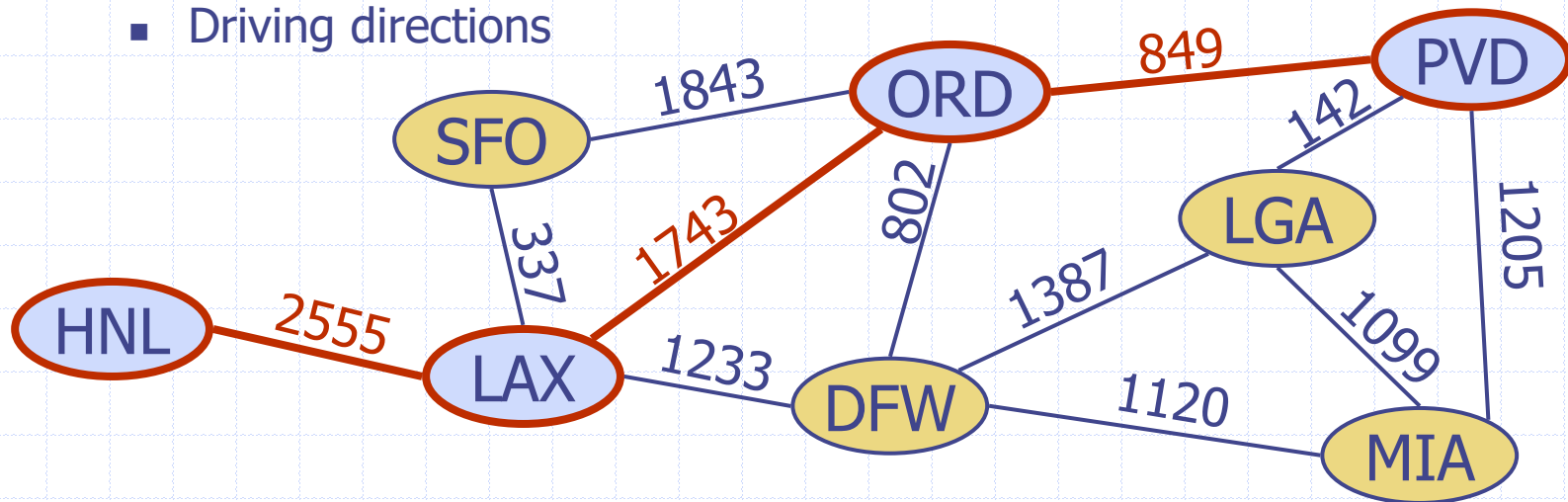
Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
 - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Shortest Paths

- Given a weighted graph and two vertices u and v , we want to find a path of minimum total weight between u and v .
 - Length of a path is the sum of the weights of its edges.
- Example:
 - Shortest path between Providence and Honolulu
- Applications
 - Internet packet routing
 - Flight reservations
 - Driving directions



Shortest Path Properties

Property 1:

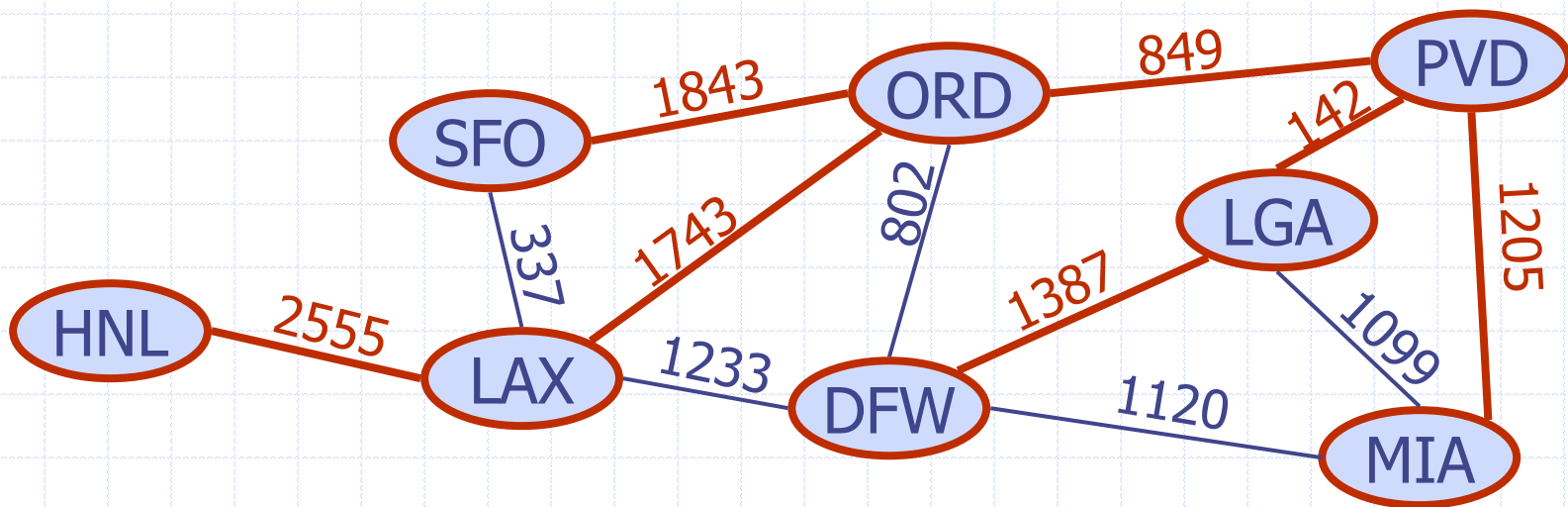
A subpath of a shortest path is itself a shortest path

Property 2:

There is a tree of shortest paths from a start vertex to all the other vertices

Example:

Tree of shortest paths from Providence

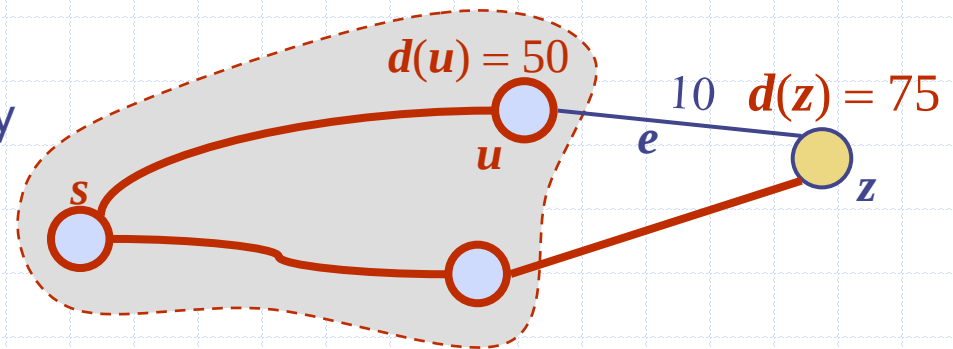


Dijkstra's Algorithm

- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s
- Assumptions:
 - the graph is connected
 - the edges are undirected
 - the edge weights are **nonnegative**
- We grow a “**cloud**” of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a **label** $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- At each step
 - We add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$
 - We update the labels of the vertices adjacent to u

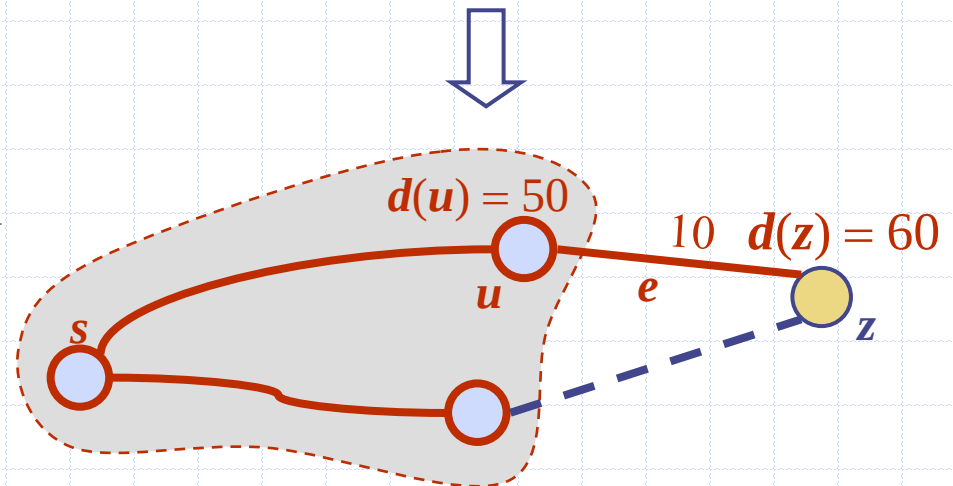
Edge Relaxation

- Consider an edge $e = (u, z)$ such that
 - u is the vertex most recently added to the cloud
 - z is not in the cloud

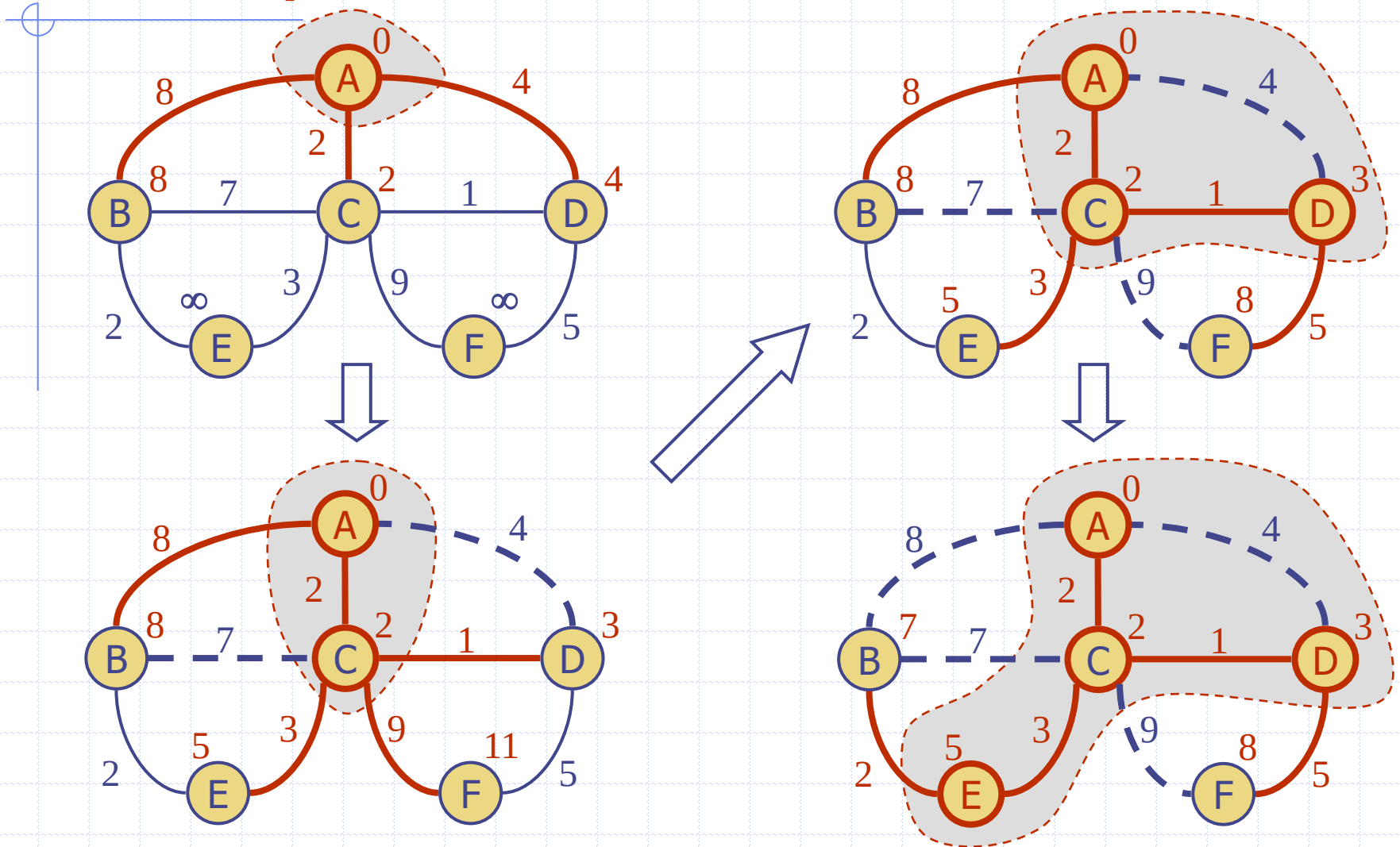


- The relaxation of edge e updates distance $d(z)$ as follows:

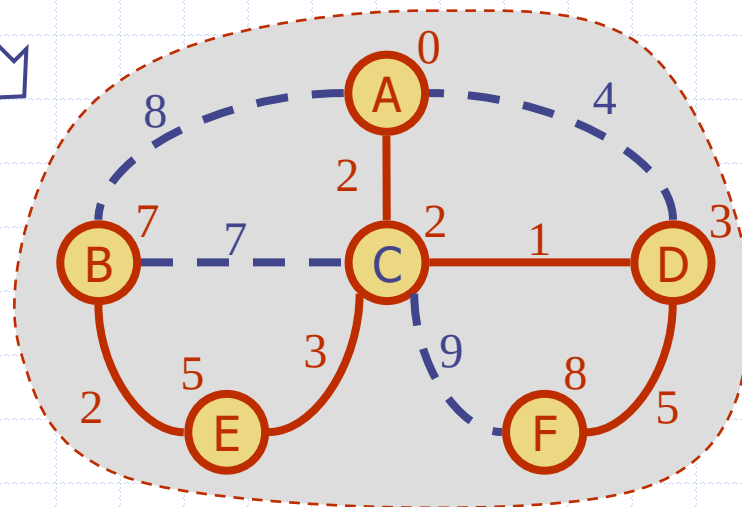
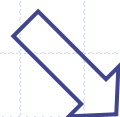
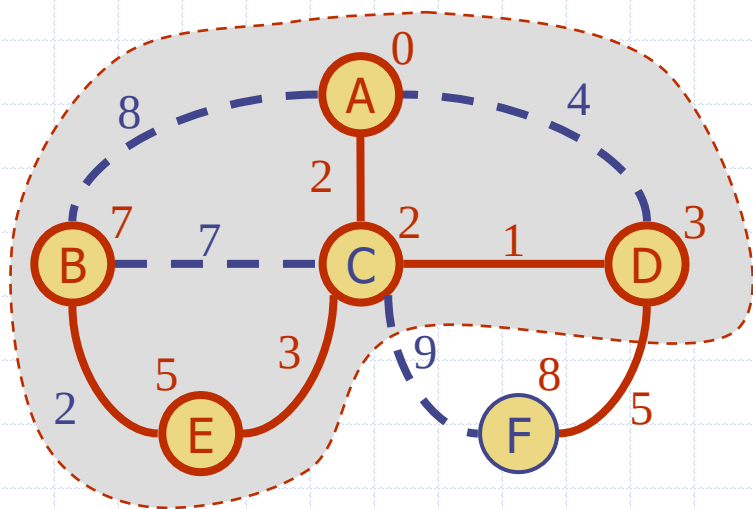
$$d(z) \leftarrow \min\{d(z), d(u) + \text{weight}(e)\}$$



Example



Example (cont.)



Dijkstra's Algorithm

Algorithm ShortestPath(G, s):

Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G .

Output: The length of a shortest path from s to v for each vertex v of G .

Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.

Let a priority queue Q contain all the vertices of G using the D labels as keys.

while Q is not empty **do**

 {pull a new vertex u into the cloud}

$u =$ value returned by $Q.remove_min()$

for each vertex v adjacent to u such that v is in Q **do**

 {perform the *relaxation* procedure on edge (u, v) }

if $D[u] + w(u, v) < D[v]$ **then**

$D[v] = D[u] + w(u, v)$

 Change to $D[v]$ the key of vertex v in Q .

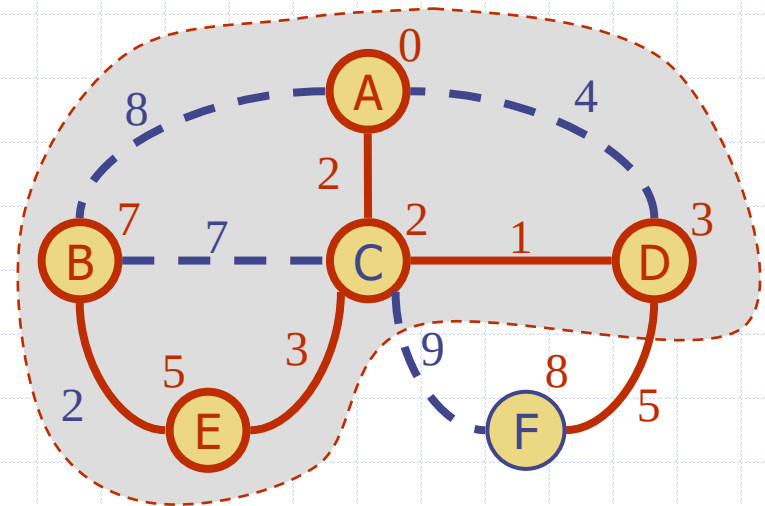
return the label $D[v]$ of each vertex v

Analysis of Dijkstra's Algorithm

- Graph operations
 - We find all the incident edges once for each vertex
- Label operations
 - We set/get the distance and locator labels of vertex z $O(\deg(z))$ times
 - Setting/getting a label takes $O(1)$ time
- Priority queue operations
 - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
 - The key of a vertex in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Dijkstra's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list/map structure
 - Recall that $\sum_v \deg(v) = 2m$
- The running time can also be expressed as $O(m \log n)$ since the graph is connected

Why Dijkstra's Algorithm Works

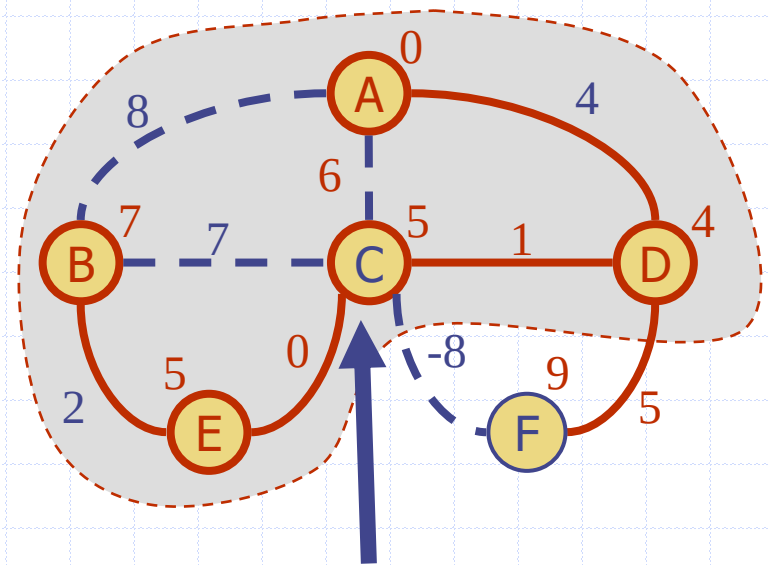
- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
 - Suppose it didn't find all shortest distances. Let F be the first wrong vertex the algorithm processed.
 - When the previous node, D, on the true shortest path was considered, its distance was correct
 - But the edge (D,F) was **relaxed** at that time!
 - Thus, so long as $d(F) \geq d(D)$, F's distance cannot be wrong. That is, there is no wrong vertex



Why It Doesn't Work for Negative-Weight Edges

◆ Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.

- If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C's true distance is 1, but it is already in the cloud with $d(C)=5$!

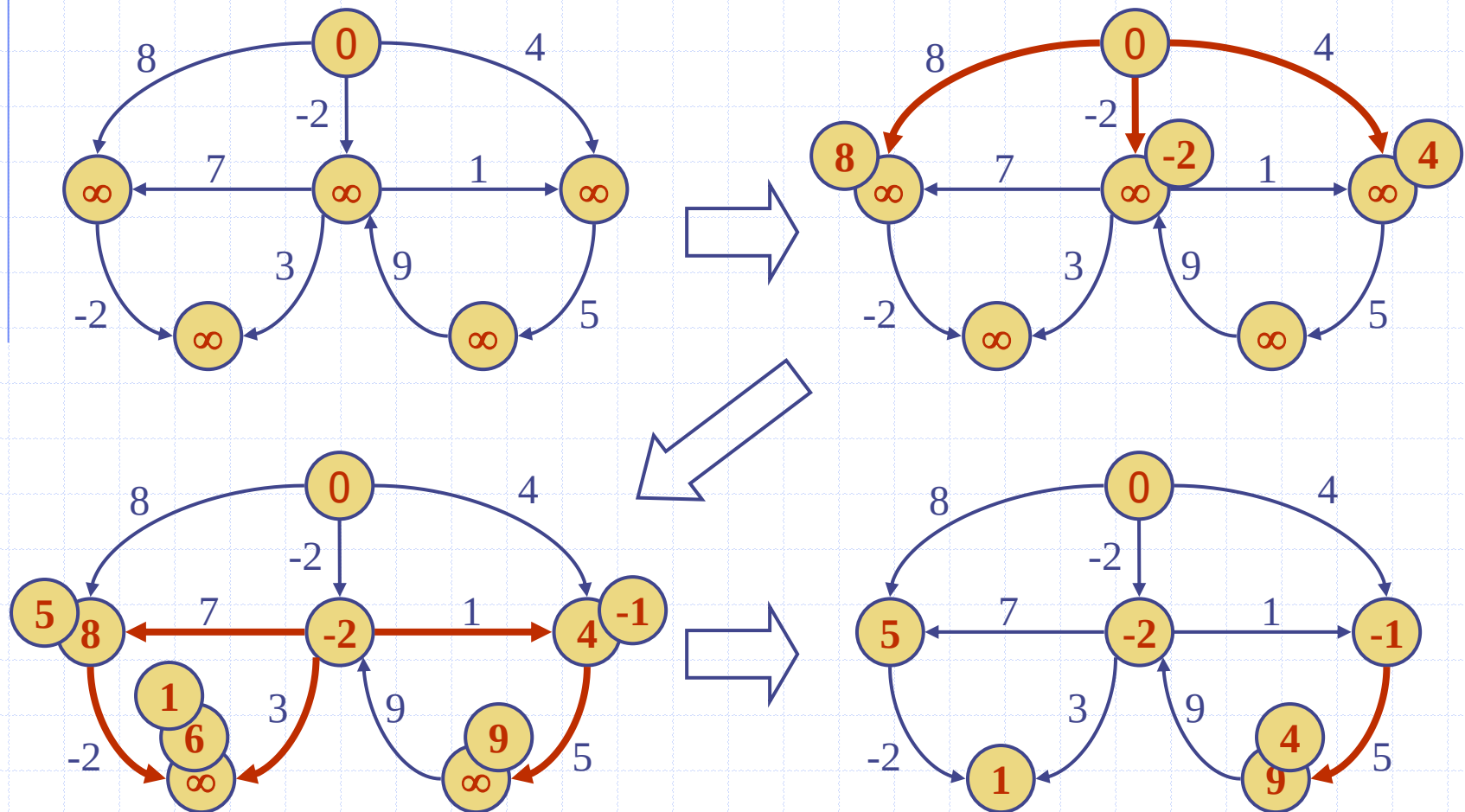
Bellman-Ford Algorithm (not in book)

- ❑ Works even with negative-weight edges
- ❑ Must assume directed edges (for otherwise we would have negative-weight cycles)
- ❑ Iteration i finds all shortest paths that use i edges.
- ❑ Running time: $O(nm)$.
- ❑ Can be extended to detect a negative-weight cycle if it exists
 - How?

```
Algorithm BellmanFord( $G, s$ )  
  for all  $v \in G.vertices()$   
    if  $v = s$   
      setDistance( $v, 0$ )  
    else  
      setDistance( $v, \infty$ )  
  for  $i \leftarrow 1$  to  $n - 1$  do  
    for each  $e \in G.edges()$   
      { relax edge  $e$  }  
       $u \leftarrow G.origin(e)$   
       $z \leftarrow G.opposite(u, e)$   
       $r \leftarrow getDistance(u) + weight(e)$   
      if  $r < getDistance(z)$   
        setDistance( $z, r$ )
```

Bellman-Ford Example

Nodes are labeled with their $d(v)$ values



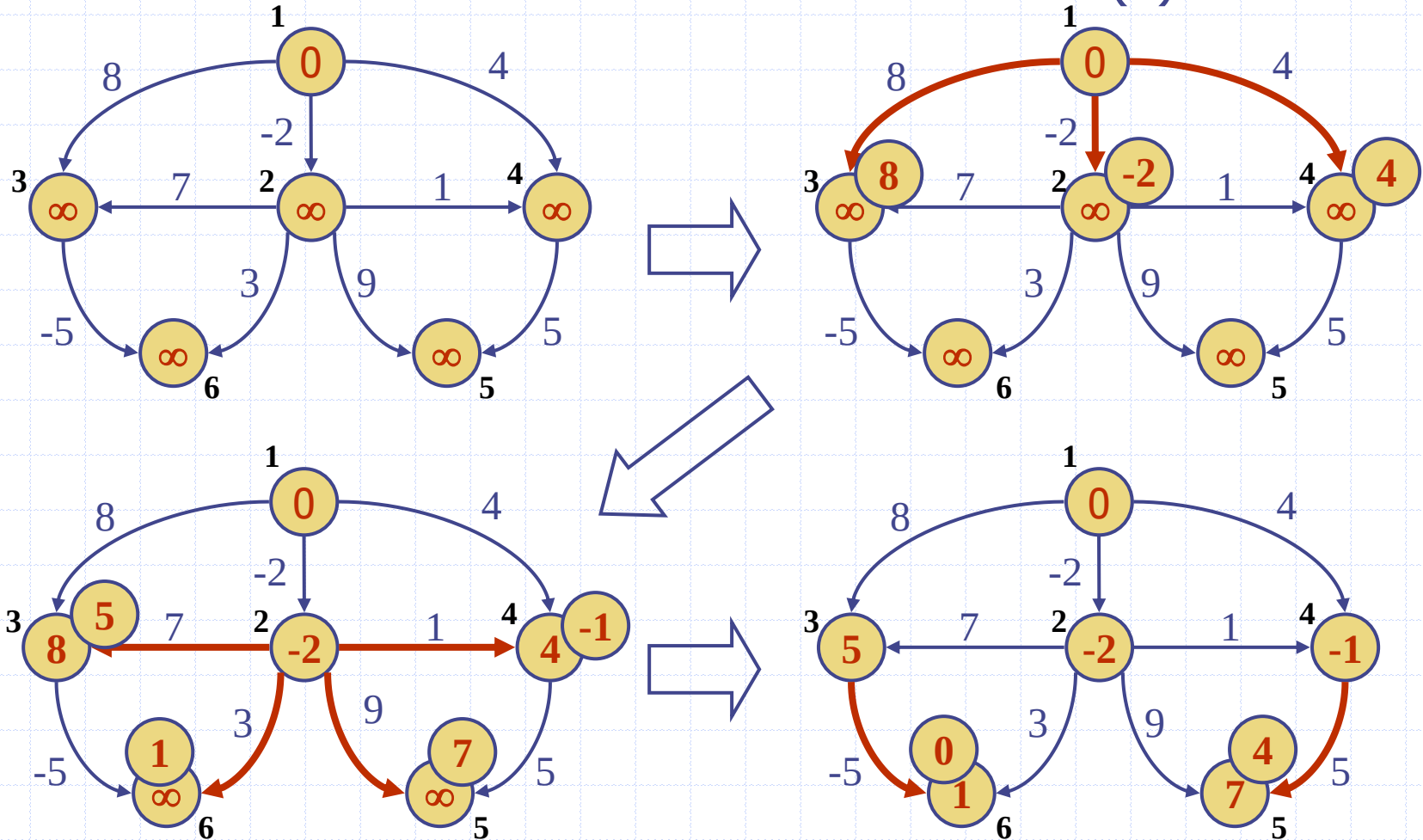
DAG-based Algorithm (not in book)

- ❑ Works even with negative-weight edges
- ❑ Uses topological order
- ❑ Doesn't use any fancy data structures
- ❑ Is much faster than Dijkstra's algorithm
- ❑ Running time: $O(n+m)$.

```
Algorithm DagDistances( $G, s$ )  
  for all  $v \in G.vertices()$   
    if  $v = s$   
      setDistance( $v, 0$ )  
    else  
      setDistance( $v, \infty$ )  
  { Perform a topological sort of the vertices }  
  for  $u \leftarrow 1$  to  $n$  do {in topological order}  
    for each  $e \in G.outEdges(u)$   
      { relax edge  $e$  }  
       $z \leftarrow G.opposite(u, e)$   
       $r \leftarrow getDistance(u) + weight(e)$   
      if  $r < getDistance(z)$   
        setDistance( $z, r$ )
```

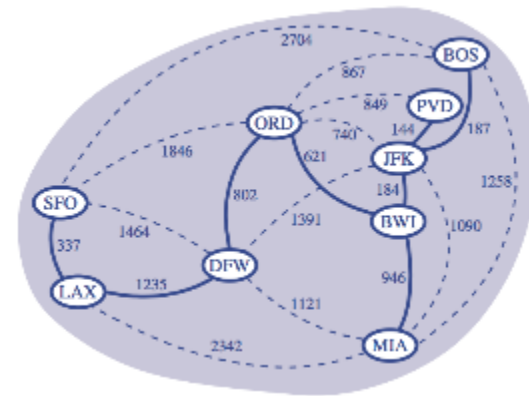
DAG Example

Nodes are labeled with their $d(v)$ values



Presentation for use with the textbook **Data Structures and Algorithms in Java, 6th edition**, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Minimum Spanning Trees



Minimum Spanning Trees

Spanning subgraph

- Subgraph of a graph G containing all the vertices of G

Spanning tree

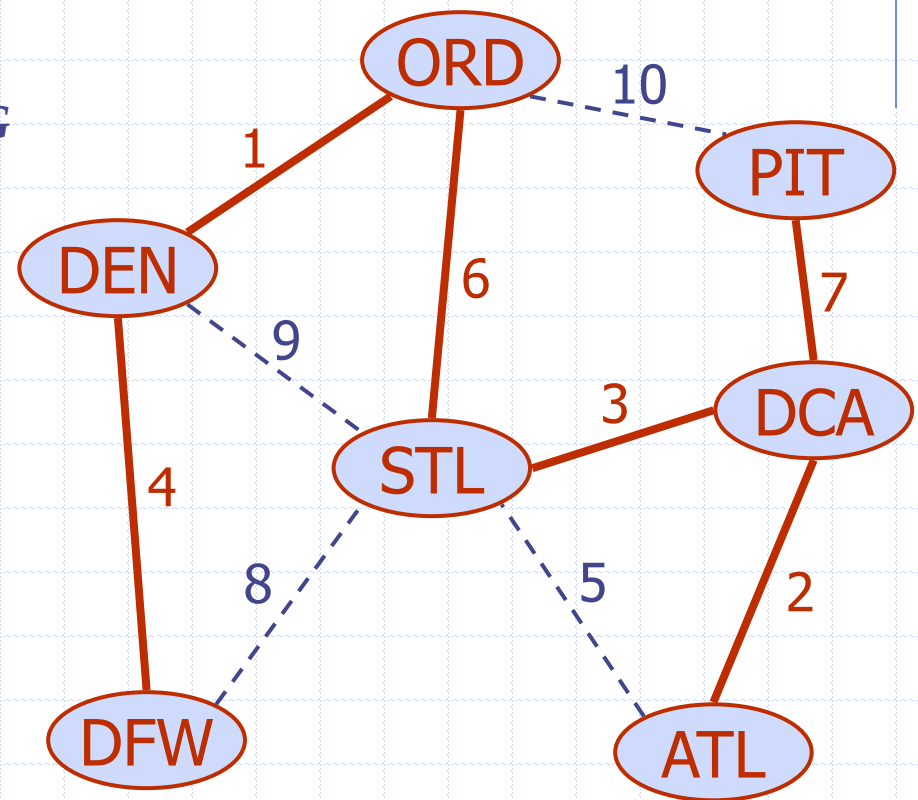
- Spanning subgraph that is itself a (free) tree

Minimum spanning tree (MST)

- Spanning tree of a weighted graph with minimum total edge weight

Applications

- Communications networks
- Transportation networks



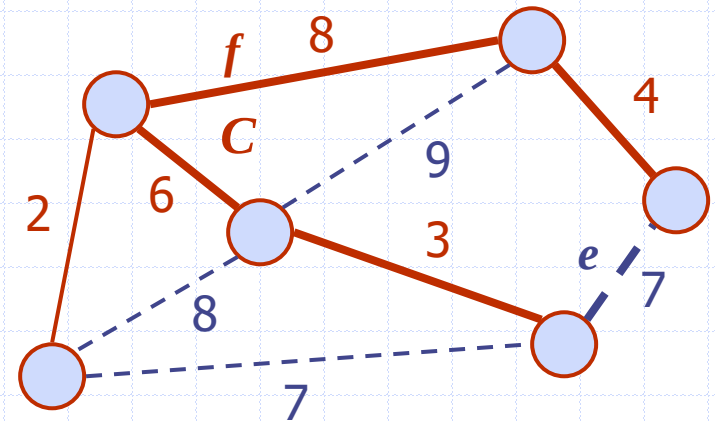
Cycle Property

Cycle Property:

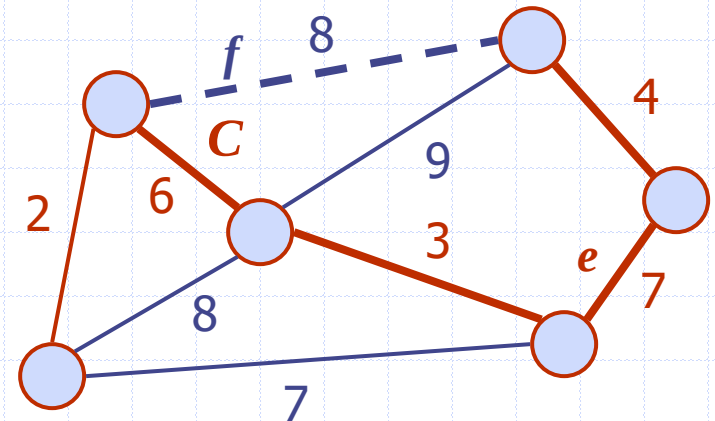
- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and C let be the cycle formed by e with T
- For every edge f of C , $\text{weight}(f) \leq \text{weight}(e)$

Proof:

- By contradiction
- If $\text{weight}(f) > \text{weight}(e)$ we can get a spanning tree of smaller weight by replacing e with f



Replacing f with e yields a better spanning tree



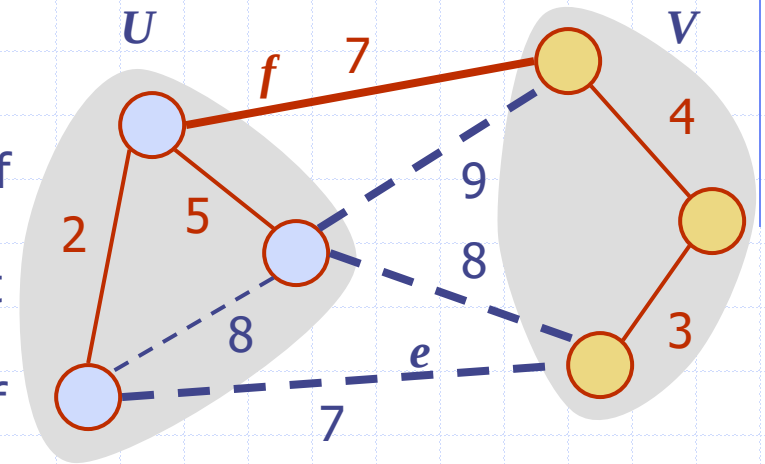
Partition Property

Partition Property:

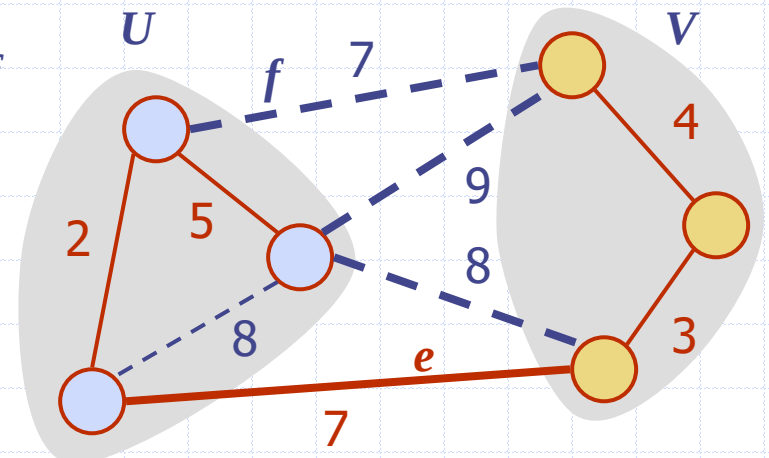
- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

Proof:

- Let T be an MST of G
- If T does not contain e , consider the cycle C formed by e with T and let f be an edge of C across the partition
- By the cycle property,
 $\text{weight}(f) \leq \text{weight}(e)$
- Thus, $\text{weight}(f) = \text{weight}(e)$
- We obtain another MST by replacing f with e



Replacing f with e yields another MST



Prim-Jarnik's Algorithm

- Similar to Dijkstra's algorithm
- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- We store with each vertex v label $d(v)$ representing the smallest weight of an edge connecting v to a vertex in the cloud
- At each step:
 - We add to the cloud the vertex u outside the cloud with the smallest distance label
 - We update the labels of the vertices adjacent to u

Prim-Jarnik Pseudo-code

Algorithm PrimJarnik(G):

Input: An undirected, weighted, connected graph G with n vertices and m edges

Output: A minimum spanning tree T for G

Pick any vertex s of G

$D[s] = 0$

for each vertex $v \neq s$ **do**

$D[v] = \infty$

Initialize $T = \emptyset$.

Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v , where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.

while Q is not empty **do**

$(u, e) = \text{value returned by } Q.\text{remove_min}()$

 Connect vertex u to T using edge e .

for each edge $e' = (u, v)$ such that v is in Q **do**

 {check if edge (u, v) better connects v to T }

if $w(u, v) < D[v]$ **then**

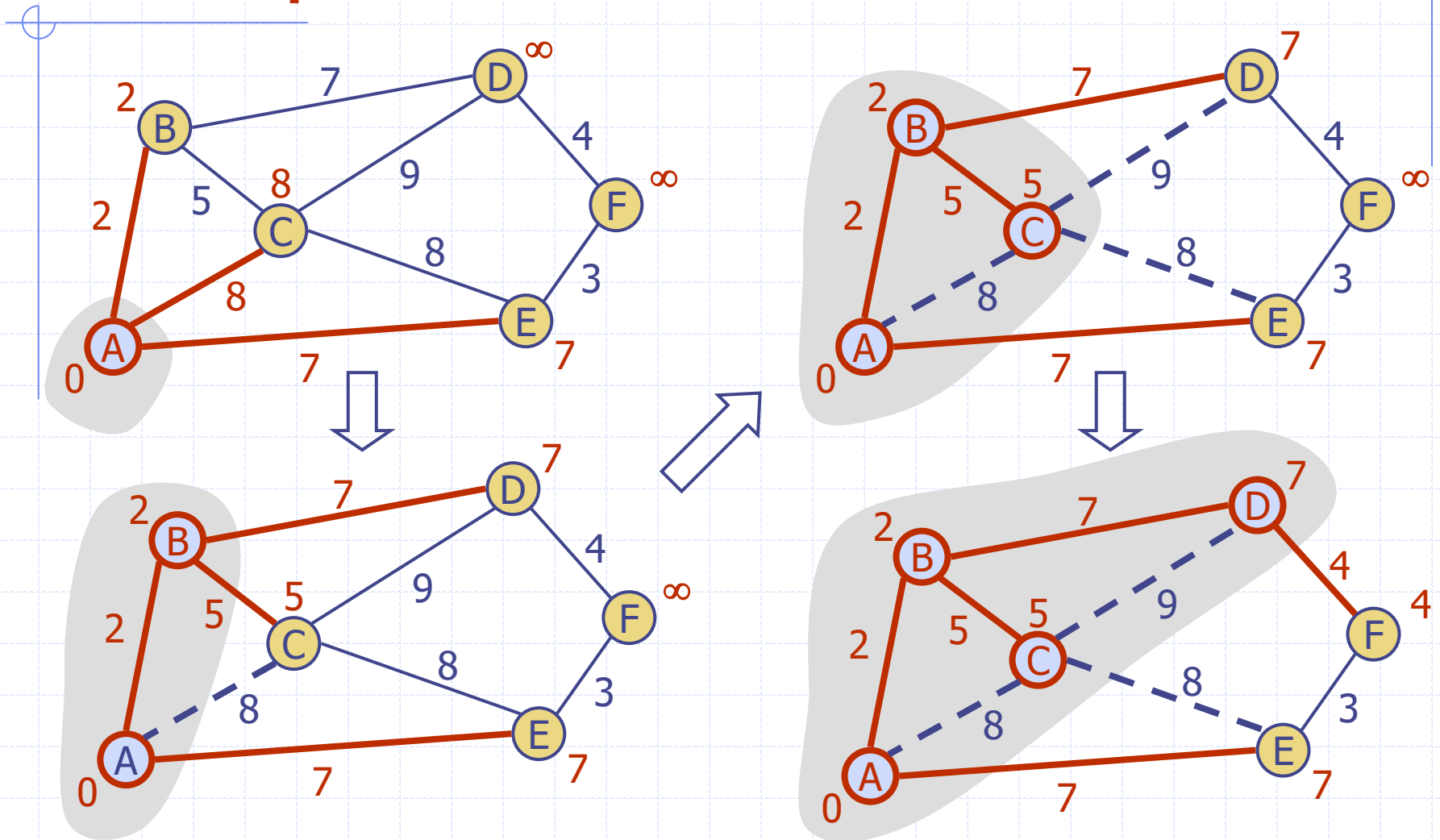
$D[v] = w(u, v)$

 Change the key of vertex v in Q to $D[v]$.

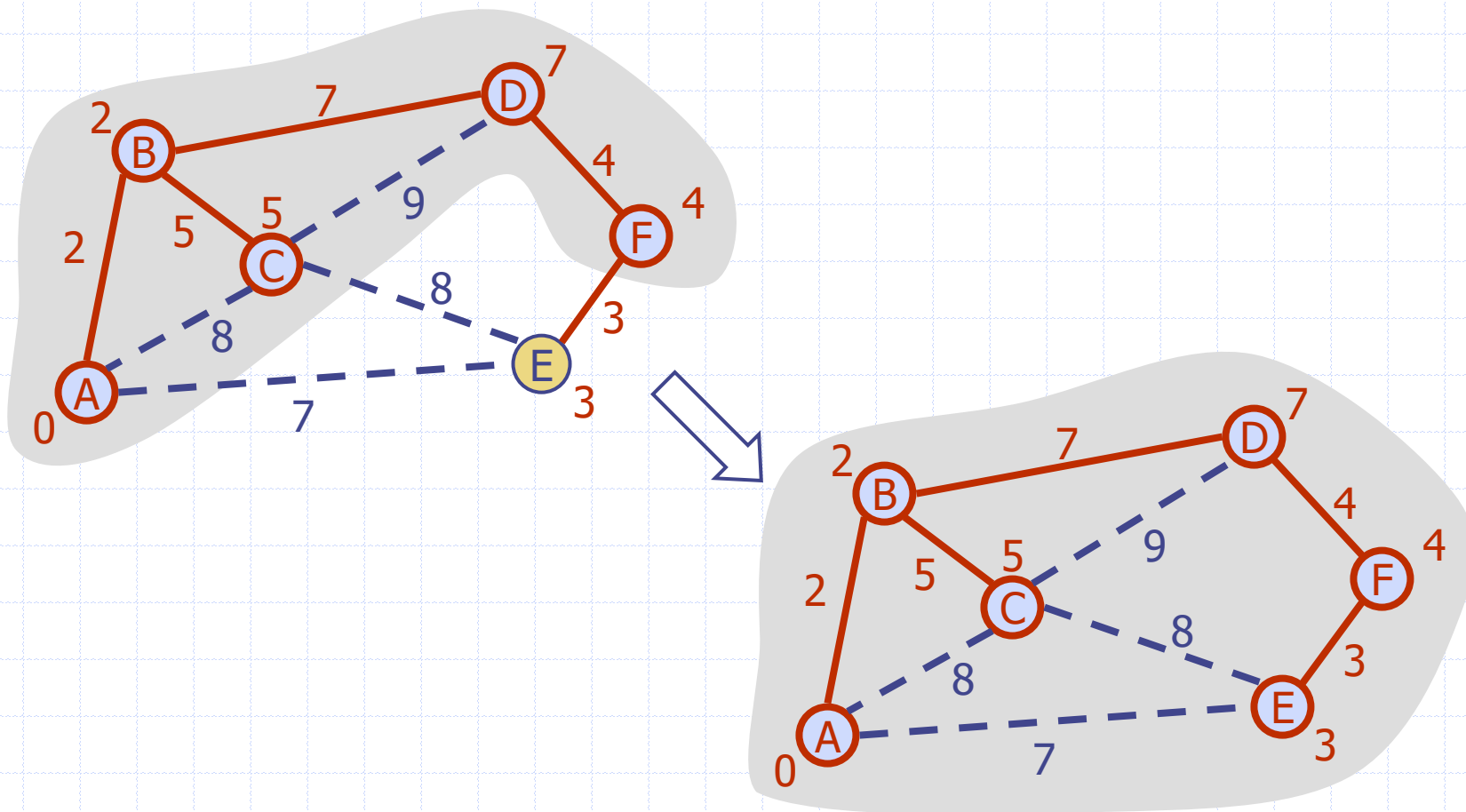
 Change the value of vertex v in Q to (v, e') .

return the tree T

Example



Example (contd.)



Analysis

- Graph operations
 - We cycle through the incident edges once for each vertex
- Label operations
 - We set/get the distance, parent and locator labels of vertex z $O(\deg(z))$ times
 - Setting/getting a label takes $O(1)$ time
- Priority queue operations
 - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
 - The key of a vertex w in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Prim-Jarnik's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
 - Recall that $\sum_v \deg(v) = 2m$
- The running time is $O(m \log n)$ since the graph is connected

Kruskal's Approach

- Maintain a partition of the vertices into clusters
 - Initially, single-vertex clusters
 - Keep an MST for each cluster
 - Merge “closest” clusters and their MSTs
- A priority queue stores the edges outside clusters
 - Key: weight
 - Element: edge
- At the end of the algorithm
 - One cluster and one MST

Algorithm Kruskal(G):

Input: A simple connected weighted graph G with n vertices and m edges

Output: A minimum spanning tree T for G

for each vertex v in G do

Define an elementary cluster $C(v) = \{v\}$.

Initialize a priority queue Q to contain all edges in G , using the weights as keys.

$T = \emptyset$	$\{T \text{ will ultimately contain the edges of the MST}\}$
-----------------	--

while T has fewer than $n - 1$ edges do

(u, v) = value returned by $Q.remove_min()$

Let $C(u)$ be the cluster containing u , and let $C(v)$ be the cluster containing v .

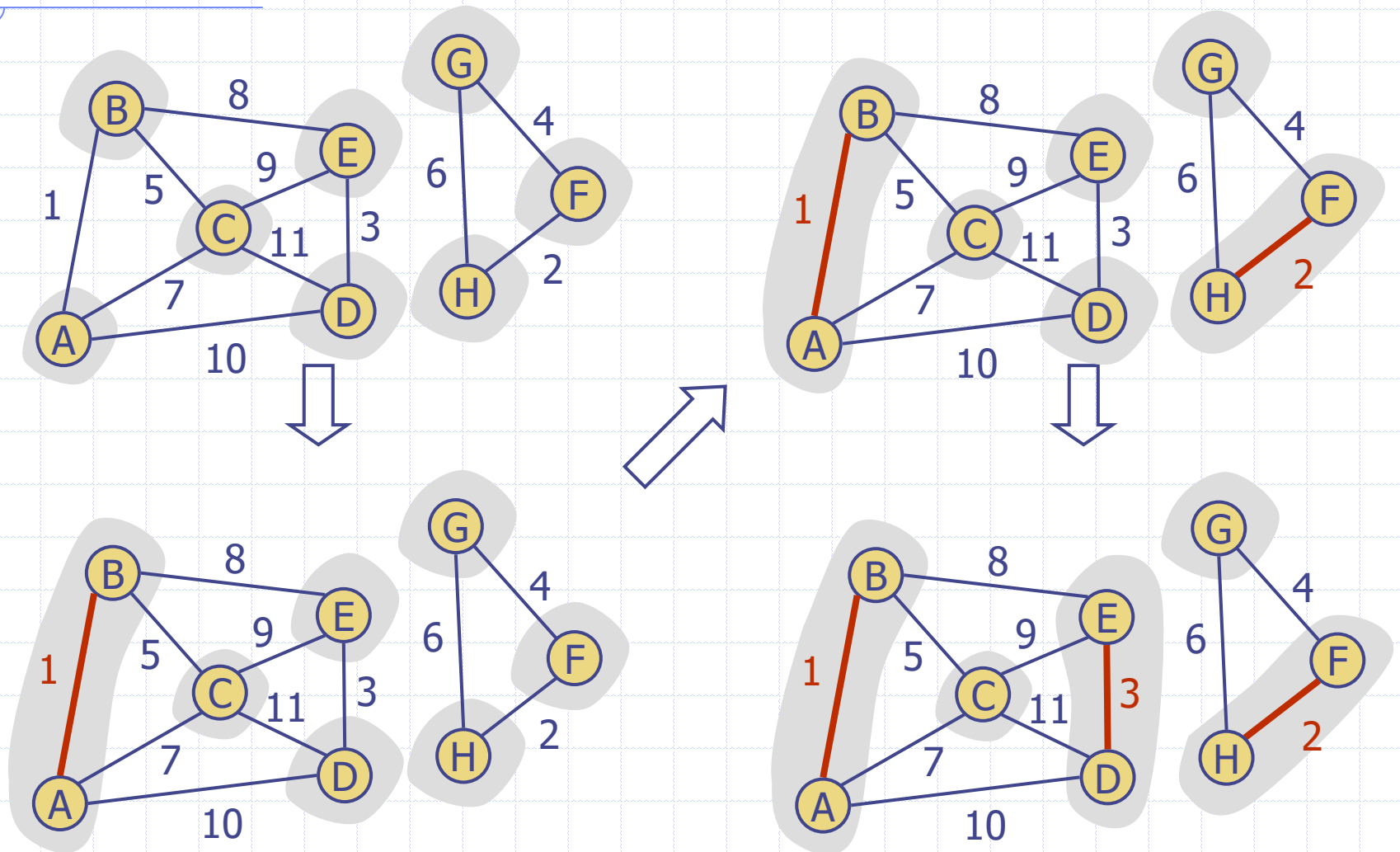
if $C(u) \neq C(v)$ then

Add edge (u, v) to T .

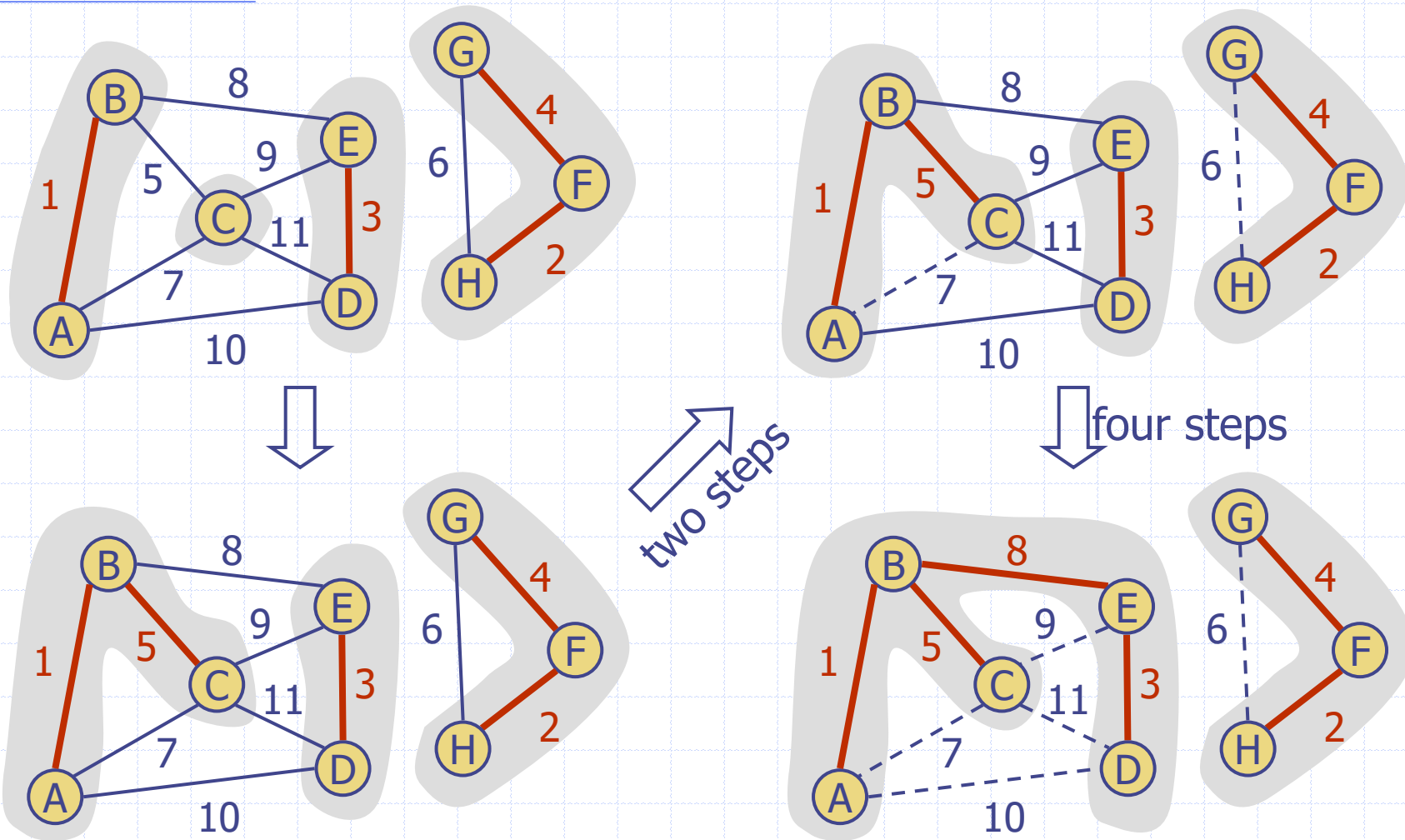
Merge $C(u)$ and $C(v)$ into one cluster.

return tree T

Example



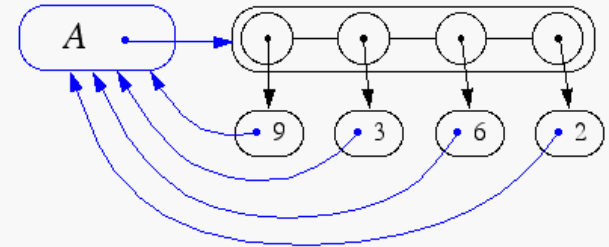
Example (contd.)



Data Structure for Kruskal's Algorithm

- ❑ The algorithm maintains a forest of trees
- ❑ A priority queue extracts the edges by increasing weight
- ❑ An edge is accepted if it connects distinct trees
- ❑ We need a data structure that maintains a **partition**, i.e., a collection of disjoint sets, with operations:
 - **makeSet**(u): create a set consisting of u
 - **find**(u): return the set storing u
 - **union**(A, B): replace sets A and B with their union

List-based Partition



- ❑ Each set is stored in a sequence
- ❑ Each element has a reference back to the set
 - operation **find**(u) takes $O(1)$ time, and returns the set of which u is a member.
 - in operation **union**(A, B), we move the elements of the smaller set to the sequence of the larger set and update their references
 - the time for operation **union**(A, B) is $\min(|A|, |B|)$
- ❑ Whenever an element is processed, it goes into a set of size at least double, hence each element is processed at most $\log n$ times

Partition-Based Implementation

- Partition-based version of Kruskal's Algorithm
 - Cluster merges as unions
 - Cluster locations as finds
- Running time $O((n + m) \log n)$
 - Priority Queue operations: $O(m \log n)$
 - Union-Find operations: $O(n \log n)$

Baruvka's Algorithm (Exercise)

- Like Kruskal's Algorithm, Baruvka's algorithm grows many clusters at once and maintains a forest T
- Each iteration of the while loop halves the number of connected components in forest T
- The running time is $O(m \log n)$

Algorithm *BaruvkaMST*(G)

```
 $T \leftarrow V$  {just the vertices of  $G$ }  
while  $T$  has fewer than  $n - 1$  edges do  
  for each connected component  $C$  in  $T$  do  
    Let edge  $e$  be the smallest-weight edge from  $C$  to another component in  $T$   
    if  $e$  is not already in  $T$  then  
      Add edge  $e$  to  $T$   
return  $T$ 
```

Example of Baruvka's Algorithm (animated)

