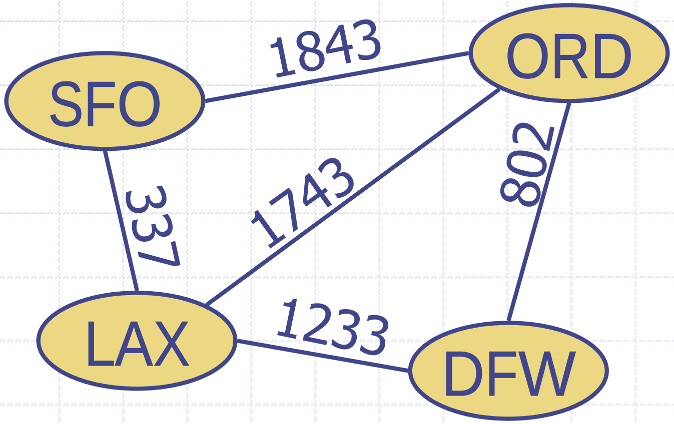
Chapter 6: Graphs

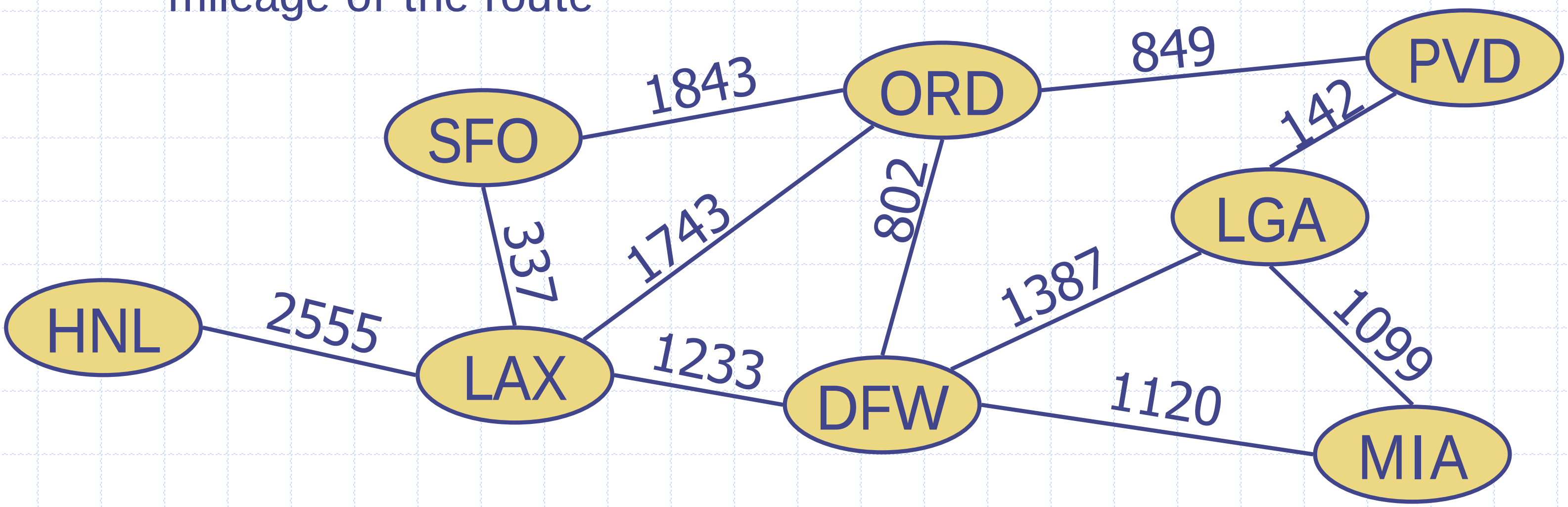


Outline and Reading

- Graphs (§6.1)
- Data Structures for Graphs (§6.2)
- Graph Traversal (§6.3)
- Directed Graph (§6.4)

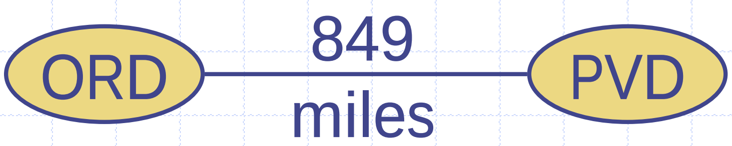
6.1 Graph

- 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



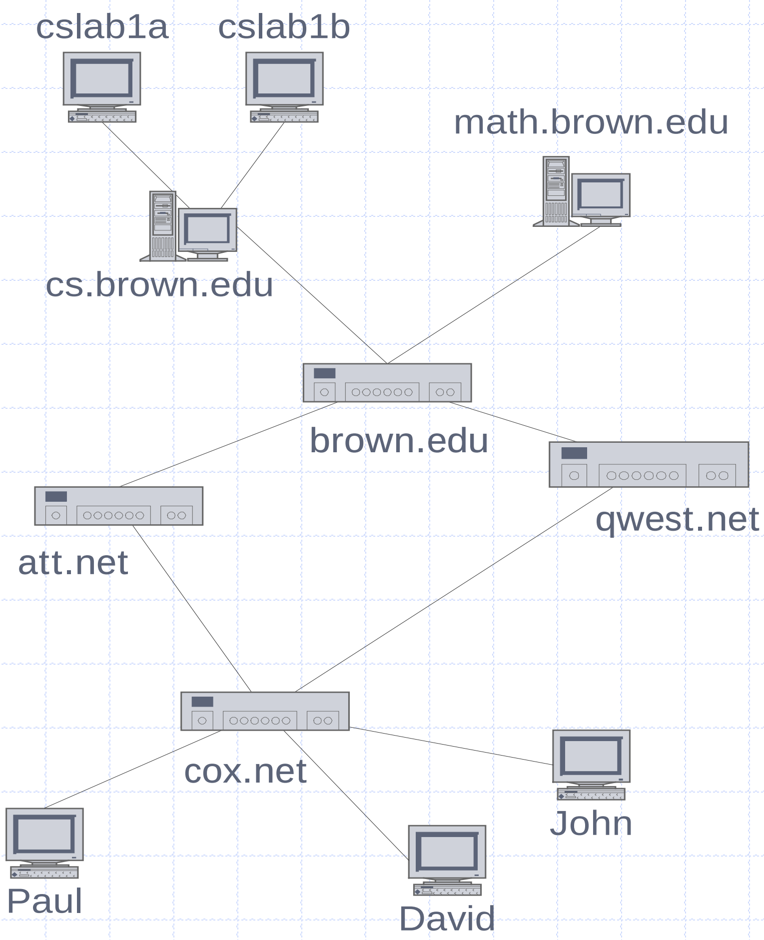
Edge Types

- 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., flight network
- Undirected graph
  - all the edges are undirected
  - e.g., route network



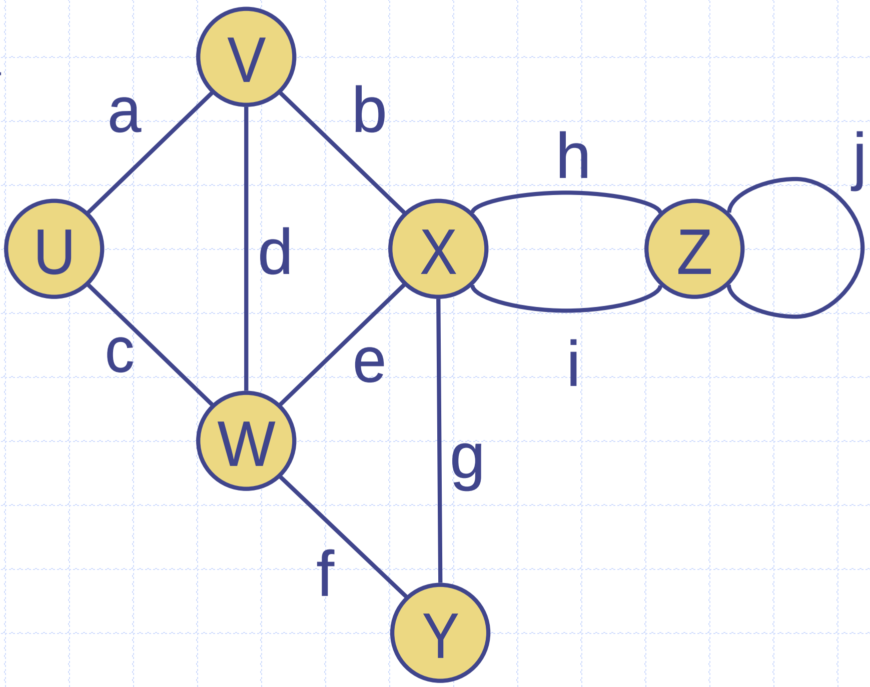
Applications

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



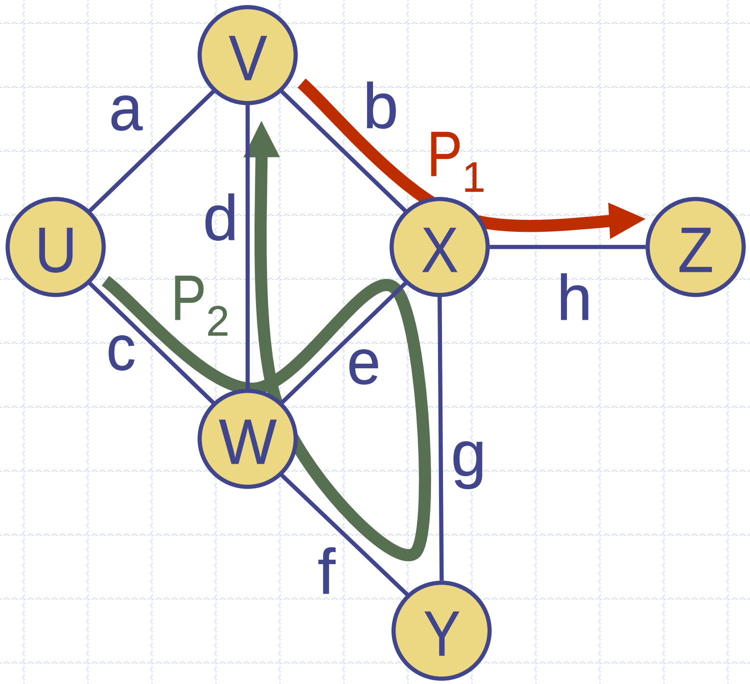
Terminology

- 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



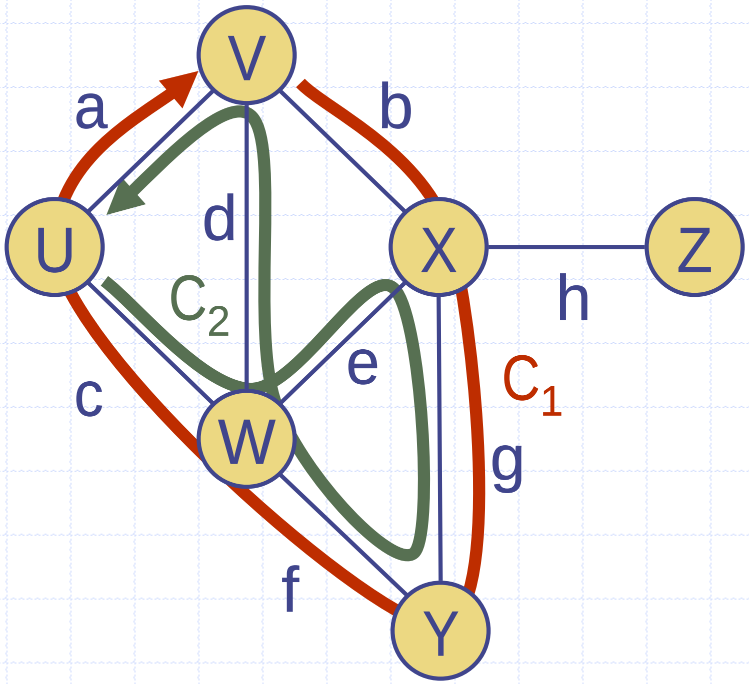
Terminology (cont.)

- 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,↵) 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



Properties

Property 1

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

Proof: each edge is counted twice

Property 2

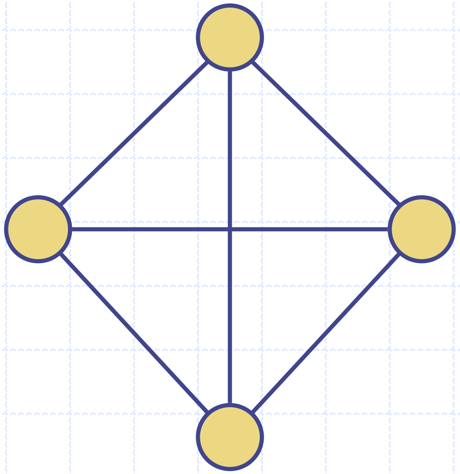
In an undirected graph with no self-loops and no multiple edges

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

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

Notation

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

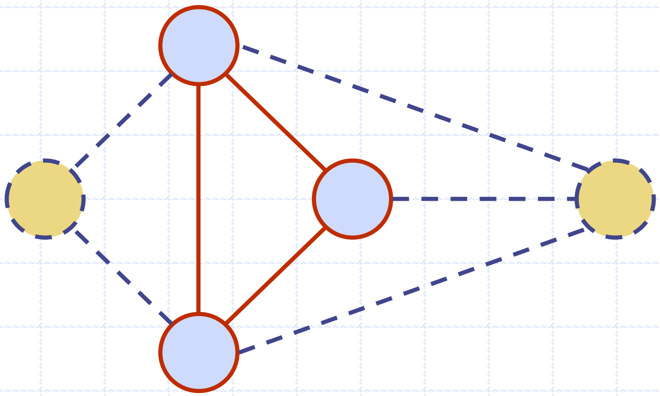


Example

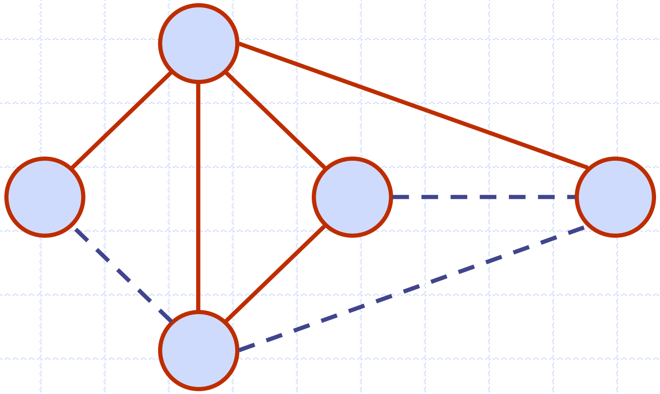
- $n = 4$
- $m = 6$
- $\deg(v) = 3$

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



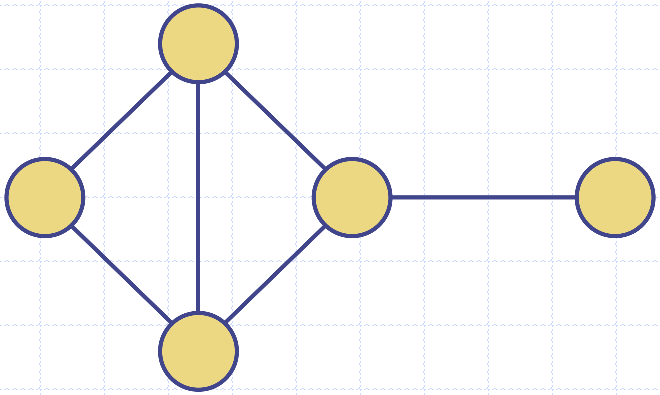
Subgraph



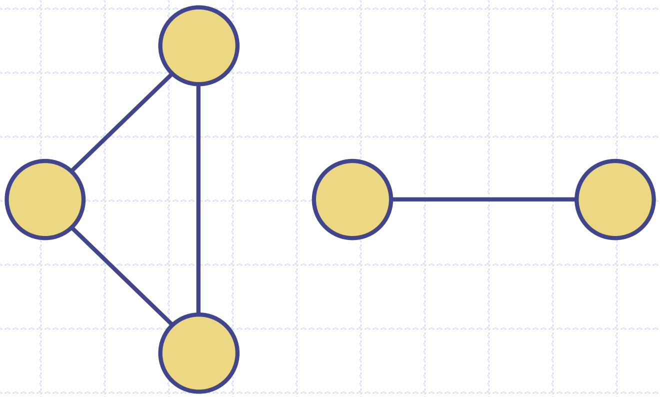
Spanning subgraph

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

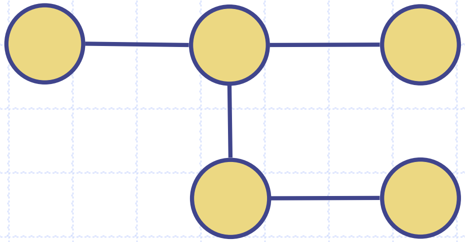


Non connected graph with two connected components

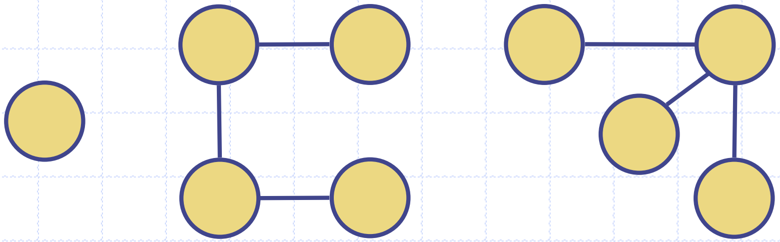
Trees and Forests

- 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



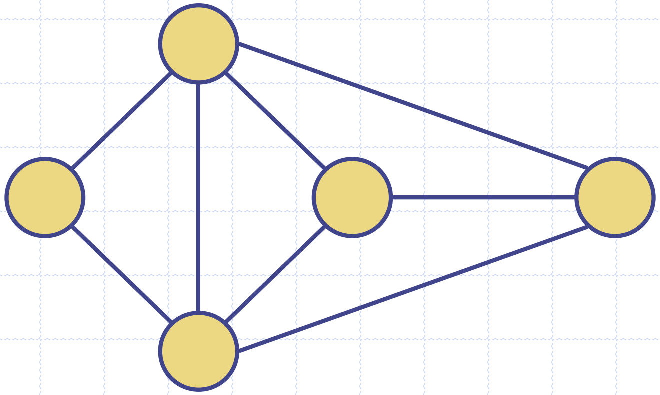
Tree



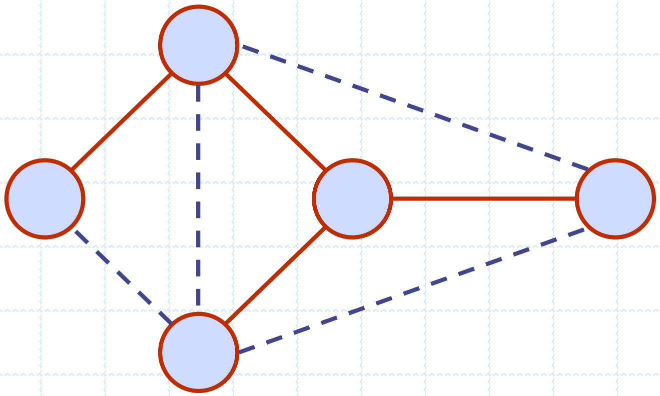
Forest

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

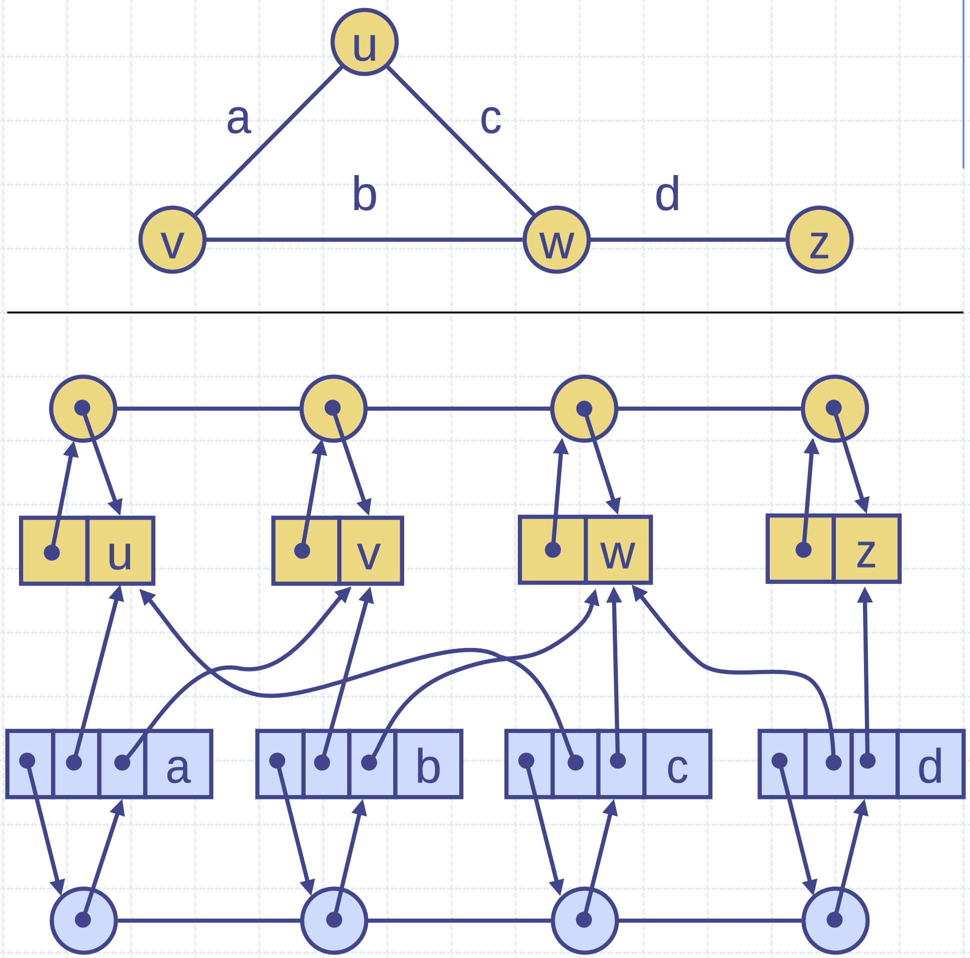
Main Methods of the Graph ADT

- Vertices and edges
  - are positions
  - store elements
- Accessor methods
  - aVertex()
  - incidentEdges(v)
  - endVertices(e)
  - isDirected(e)
  - origin(e)
  - destination(e)
  - opposite(v, e)
  - areAdjacent(v, w)
- Update methods
  - insertVertex(o)
  - insertEdge(v, w, o)
  - insertDirectedEdge(v, w, o)
  - removeVertex(v)
  - removeEdge(e)
- Generic methods
  - numVertices()
  - numEdges()
  - vertices()
  - edges()

6.2 Data Structure for Graphs

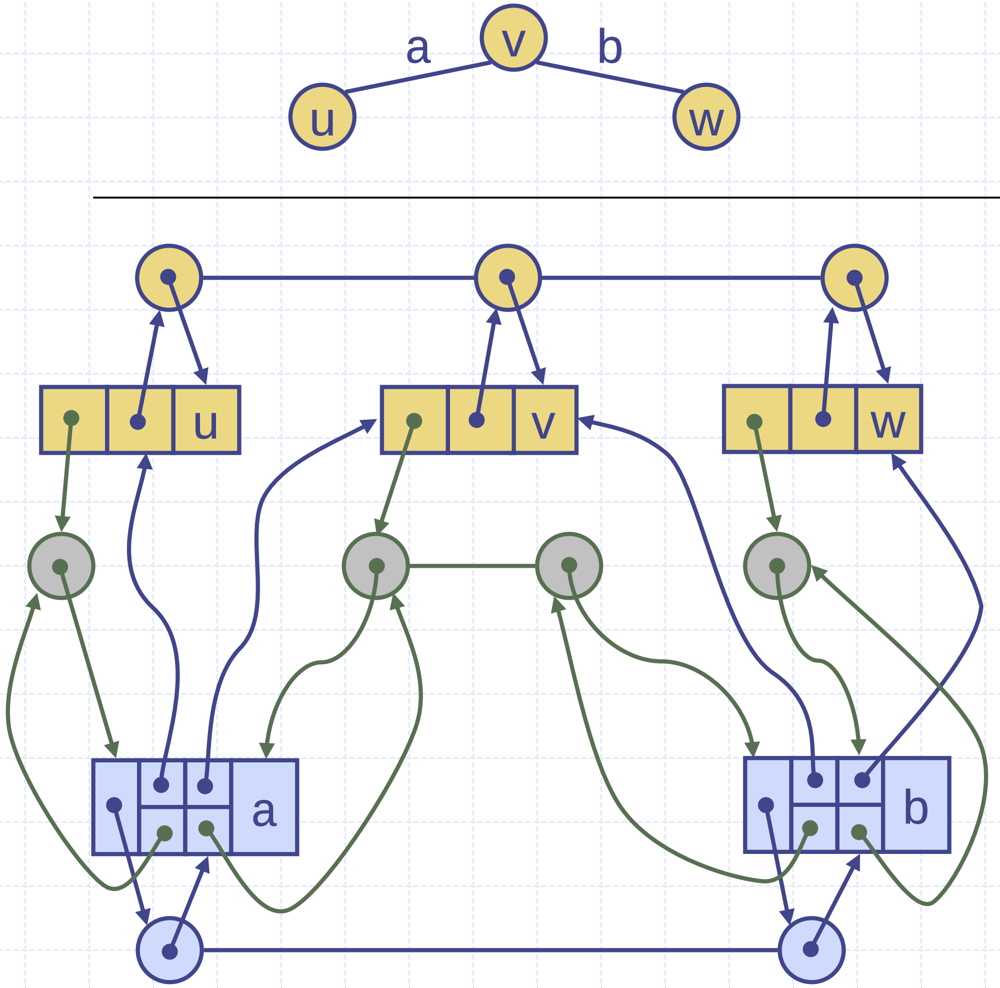
Edge List Structure

- 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



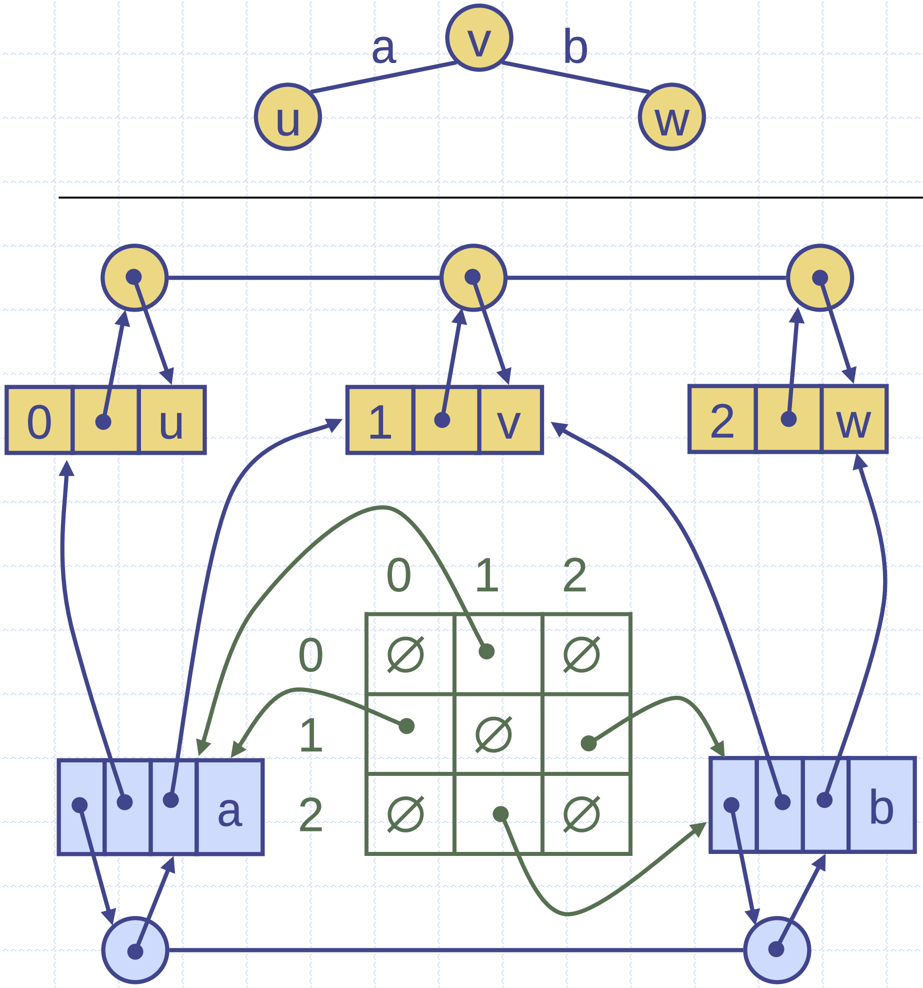
Adjacency List Structure

- Edge 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 Matrix Structure

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- The “old fashioned” version just has 0 for no edge and 1 for edge

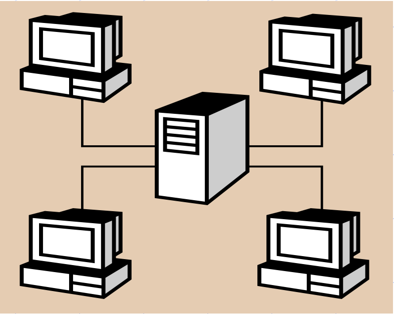


Asymptotic Performance

◆ n vertices, m edges ◆ no parallel edges ◆ no self-loops ◆ Bounds are "big-Oh"	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	$\deg(v)$	n
areAdjacent (v, w)	m	$\min(\deg(v), \deg(w))$	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	$\deg(v)$	n^2
removeEdge(e)	1	1	1

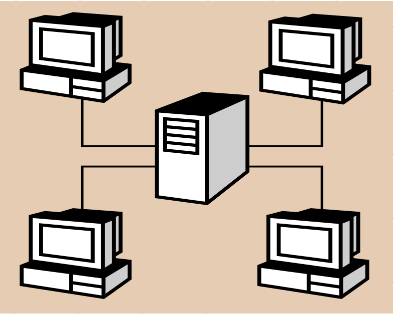
6.3 Graph Traversal

6.3.1 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

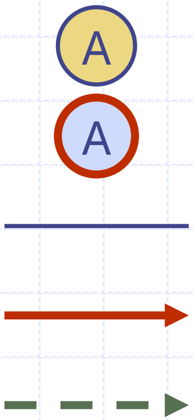


- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

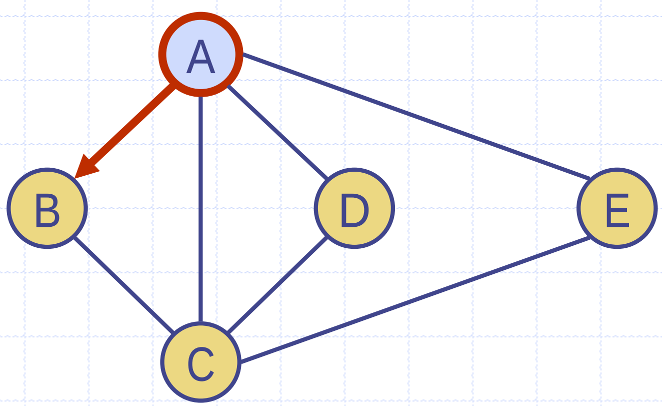
```
Algorithm DFS(G)
    Input graph G
    Output labeling of the edges of G
        as discovery edges and
        back edges
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
       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

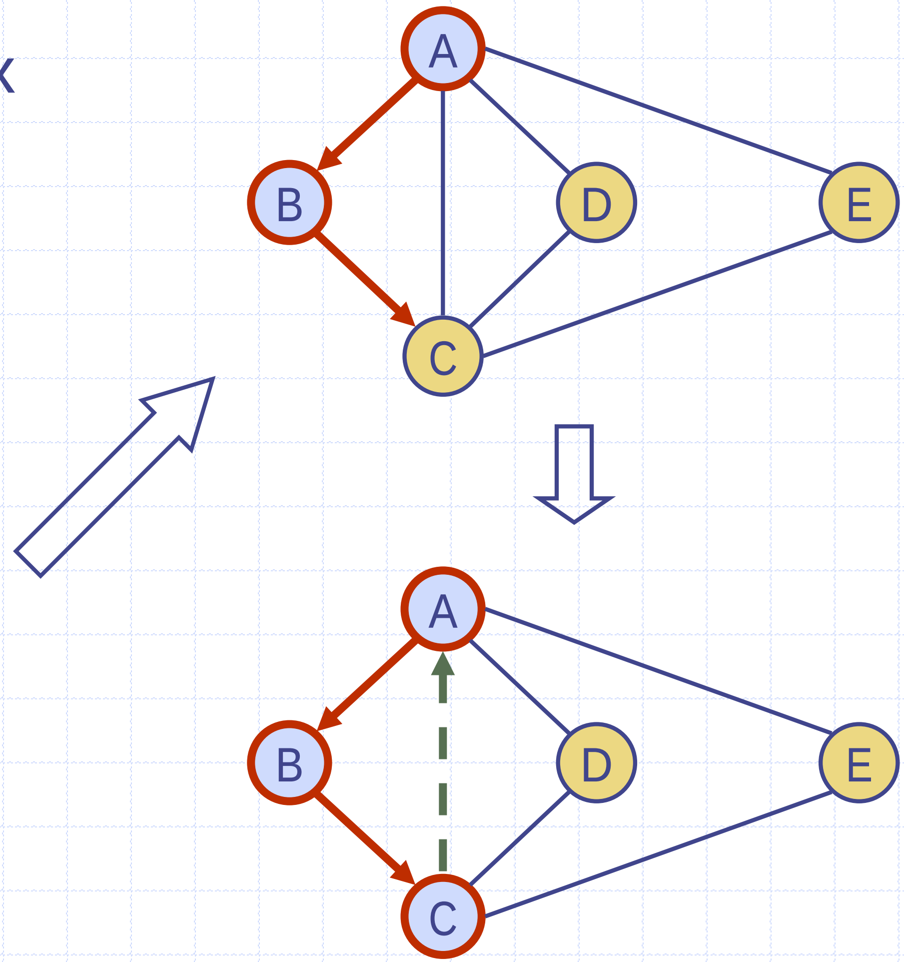
```
Algorithm DFS(G, v)
  Input graph G and a start vertex v of G
  Output labeling of the edges of G
     in the connected component of v
     as discovery edges and back edges
  setLabel(v, VISITED)
  for all e ∈ G.incidentEdges(v)
     if getLabel(e) = UNEXPLORED
        w ← G.opposite(v,e)
     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

Example

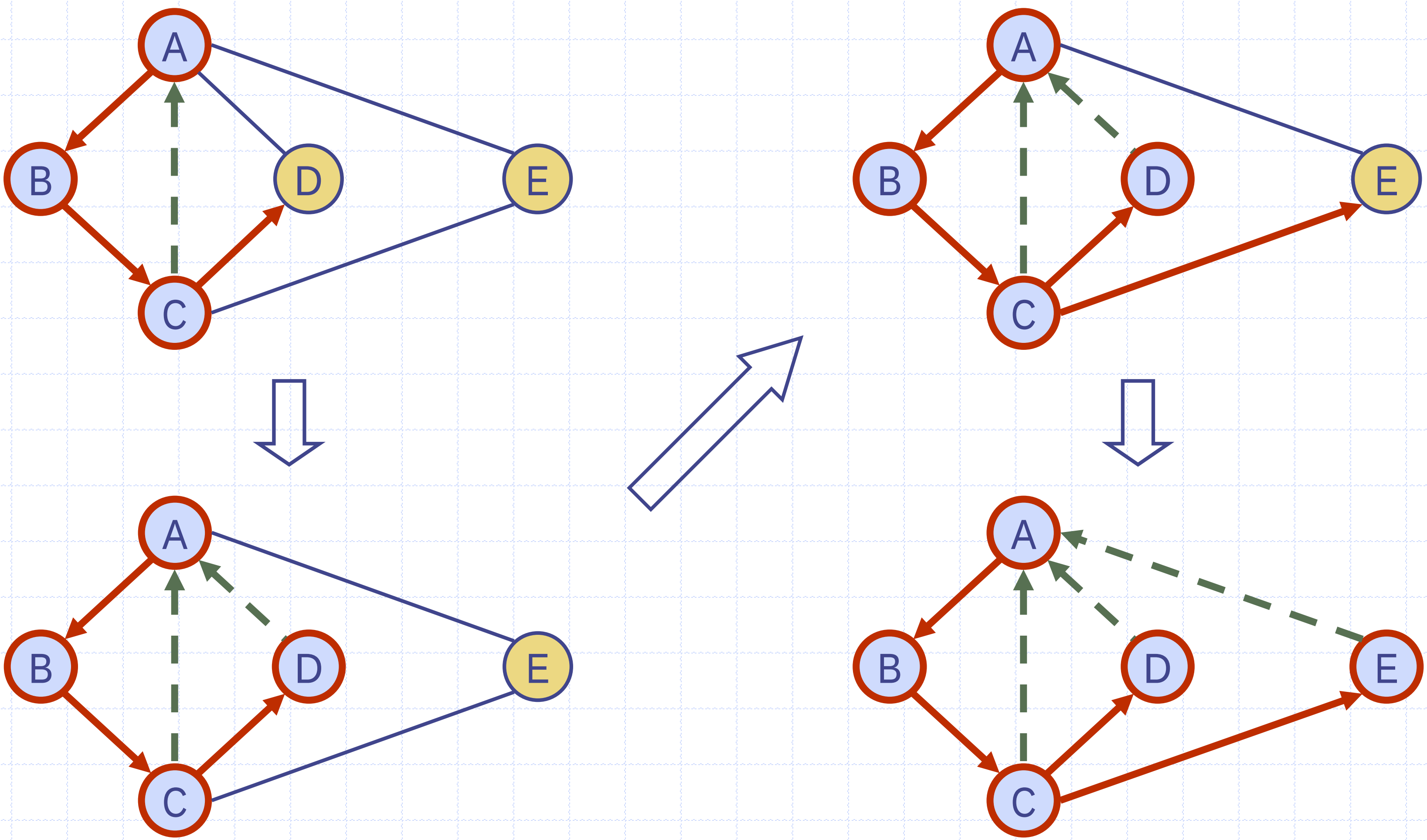


unexplored vertex
visited vertex
unexplored edge
discovery edge
back edge

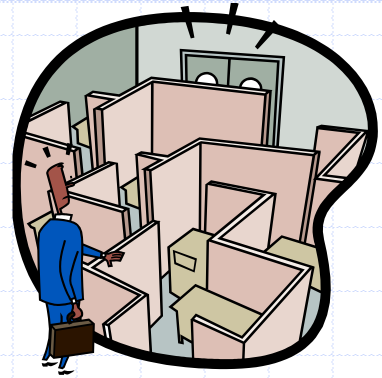




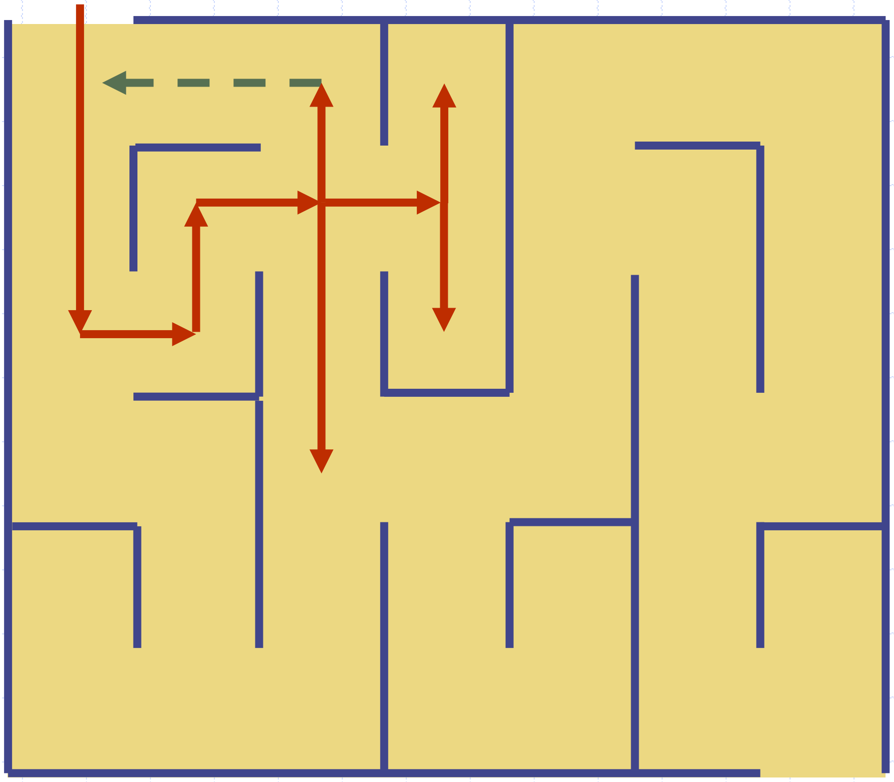
Example (cont.)



DFS and Maze Traversal



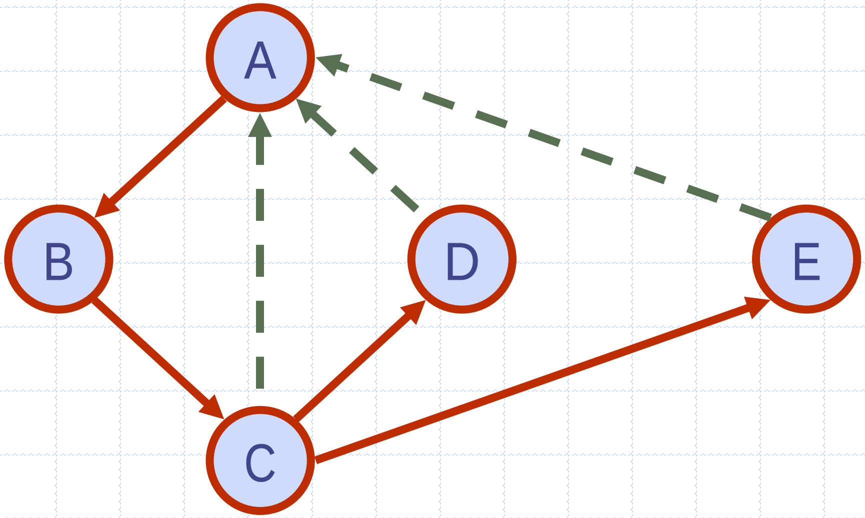
- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



Properties of DFS

Property 1

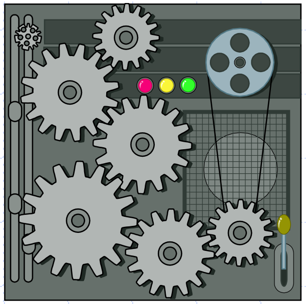
DFS(*G*, *v*) visits all the vertices and edges in the connected component of *v*



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

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

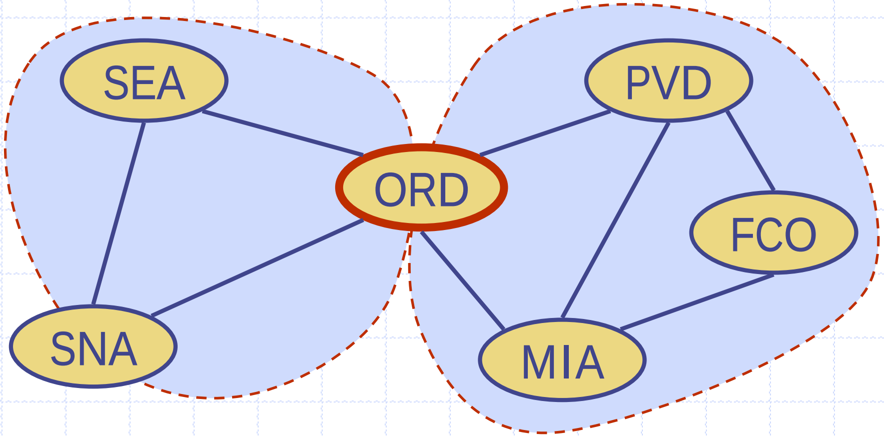
```
Algorithm pathDFS(G, v, z)
   setLabel(v, VISITED)
   S.push(v)
   if v = z
      return S.elements()
   for all e ∈ G.incidentEdges(v)
      if getLabel(e) = UNEXPLORED
         w ← opposite(v, e)
         if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            S.push(e)
    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

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

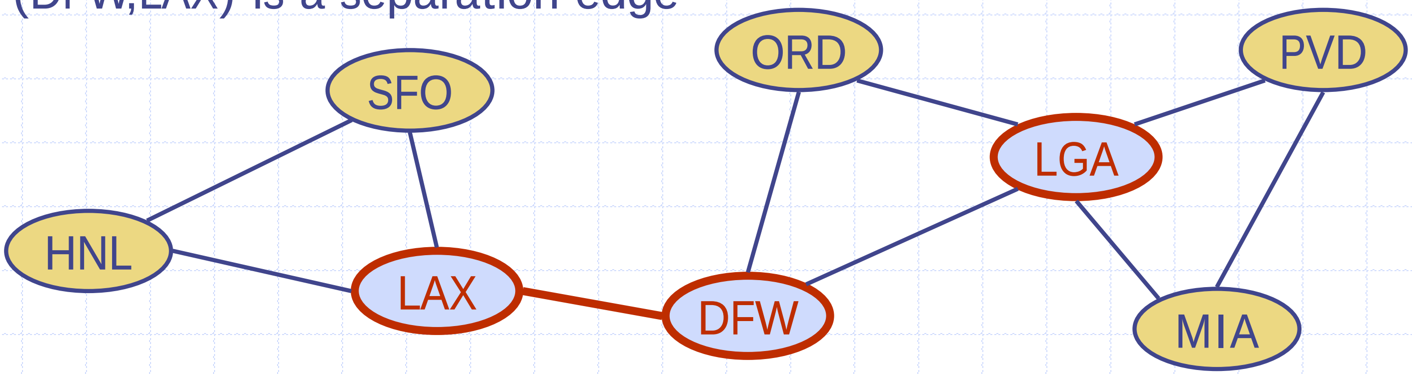
```
Algorithm cycleDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
    for all  e ∈ G.incidentEdges(v)
        if  getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            S.push(e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                pathDFS(G, w, z)
                S.pop()
            else
                C ← new empty stack
                repeat
                    o ← S.pop()
                    C.push(o)
                until o = w
                return C.elements()
    S.pop()
```

6.3.2 Biconnectivity



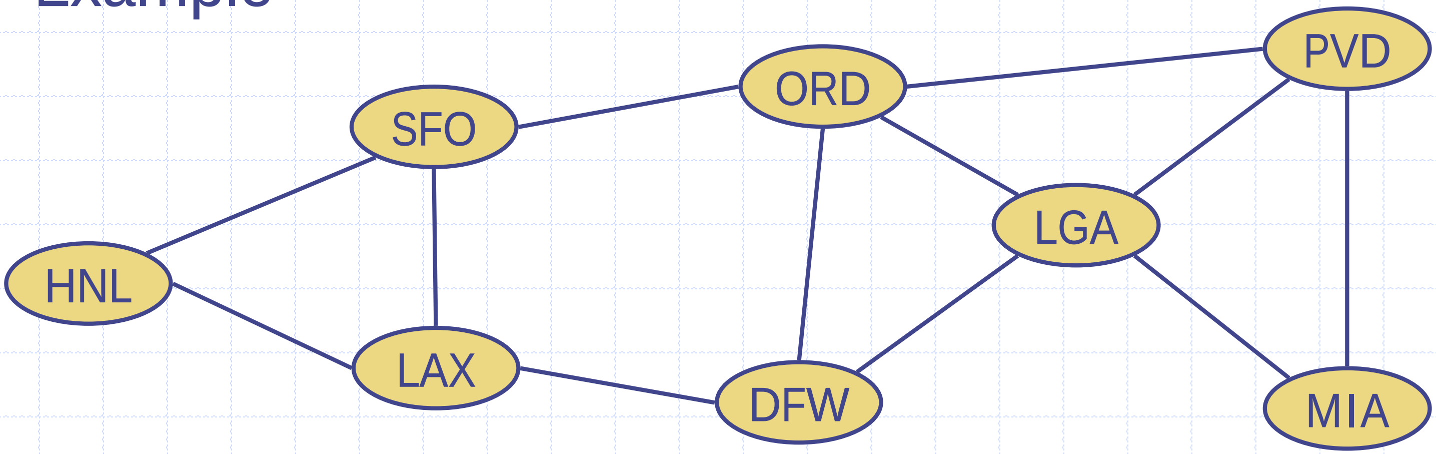
Separation Edges and Vertices

- Definitions
  - Let G be a connected graph
  - A separation edge of G is an edge whose removal disconnects G
  - A separation vertex of G is a vertex whose removal disconnects G
- Applications
  - Separation edges and vertices represent single points of failure in a network and are critical to the operation of the network
- Example
  - DFW, LGA and LAX are separation vertices
  - (DFW,LAX) is a separation edge



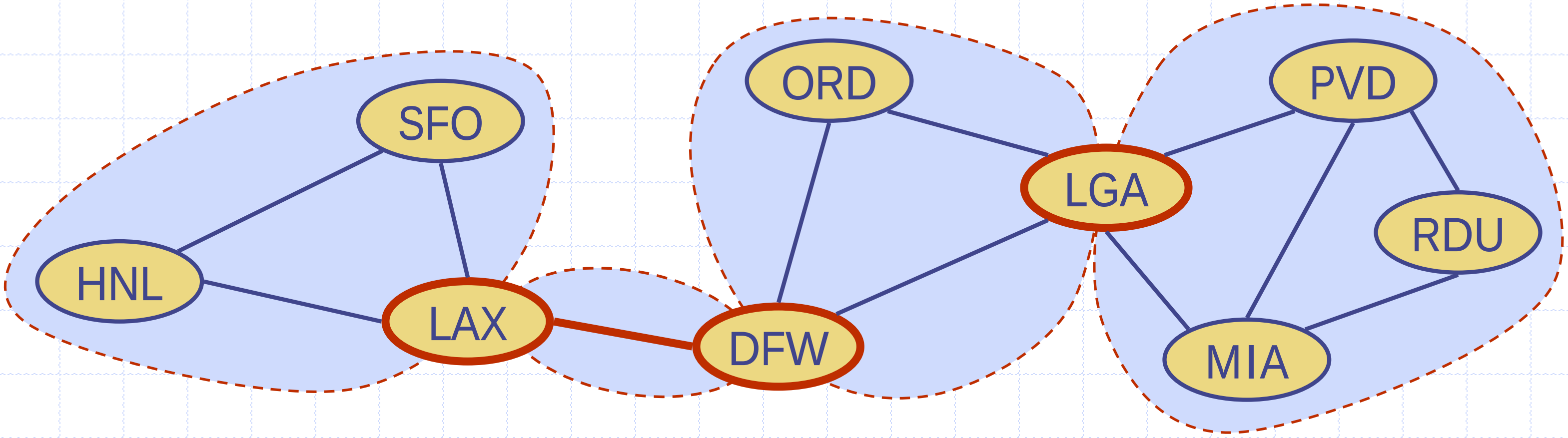
Biconnected Graph

- Equivalent definitions of a biconnected graph G
  - Graph G has no separation edges and no separation vertices
  - For any two vertices u and v of G, there are two disjoint simple paths between u and v (i.e., two simple paths between u and v that share no other vertices or edges)
  - For any two vertices u and v of G, there is a simple cycle containing u and v
- Example



Biconnected Components

- Biconnected component of a graph *G*
  - A maximal biconnected subgraph of *G*, or
  - A subgraph consisting of a separation edge of *G* and its end vertices
- Interaction of biconnected components
  - An edge belongs to exactly one biconnected component
  - A nonseparation vertex belongs to exactly one biconnected component
  - A separation vertex belongs to two or more biconnected components
- Example of a graph with four biconnected components

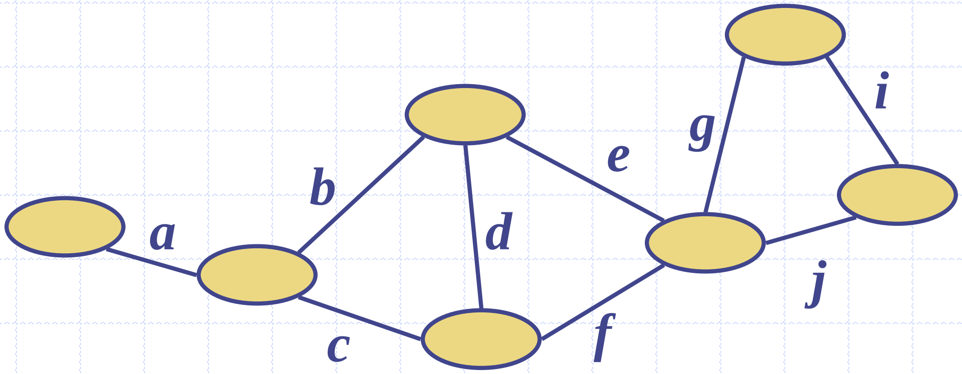


Equivalence Classes

- Given a set S, a relation R on S is a set of ordered pairs of elements of S, i.e., R is a subset of $S \times S$
- An equivalence relation R on S satisfies the following properties
  - Reflexive: $(x,x) \in R$
  - Symmetric: $(x,y) \in R \Rightarrow (y,x) \in R$
  - Transitive: $(x,y) \in R \wedge (y,z) \in R \Rightarrow (x,z) \in R$
- An equivalence relation R on S induces a partition of the elements of S into equivalence classes
- Example (connectivity relation among the vertices of a graph):
  - Let V be the set of vertices of a graph G
  - Define the relation
    $C = \{(v,w) \in V \times V$ such that G has a path from v to $w\}$
  - Relation C is an equivalence relation
  - The equivalence classes of relation C are the vertices in each connected component of graph G

Link Relation

- Edges e and f of connected graph G are linked if
  - $e = f$, or
  - G has a simple cycle containing e and f



Theorem:

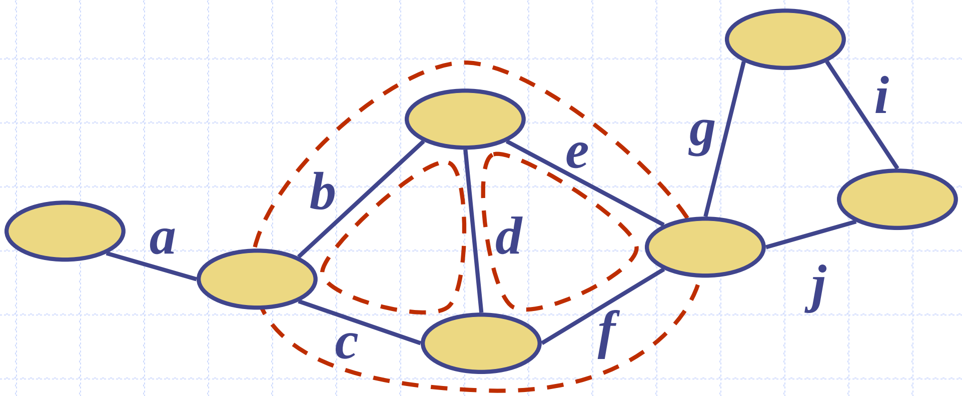
The link relation on the edges of a graph is an equivalence relation

Proof Sketch:

- The reflexive and symmetric properties follow from the definition
- For the transitive property, consider two simple cycles sharing an edge

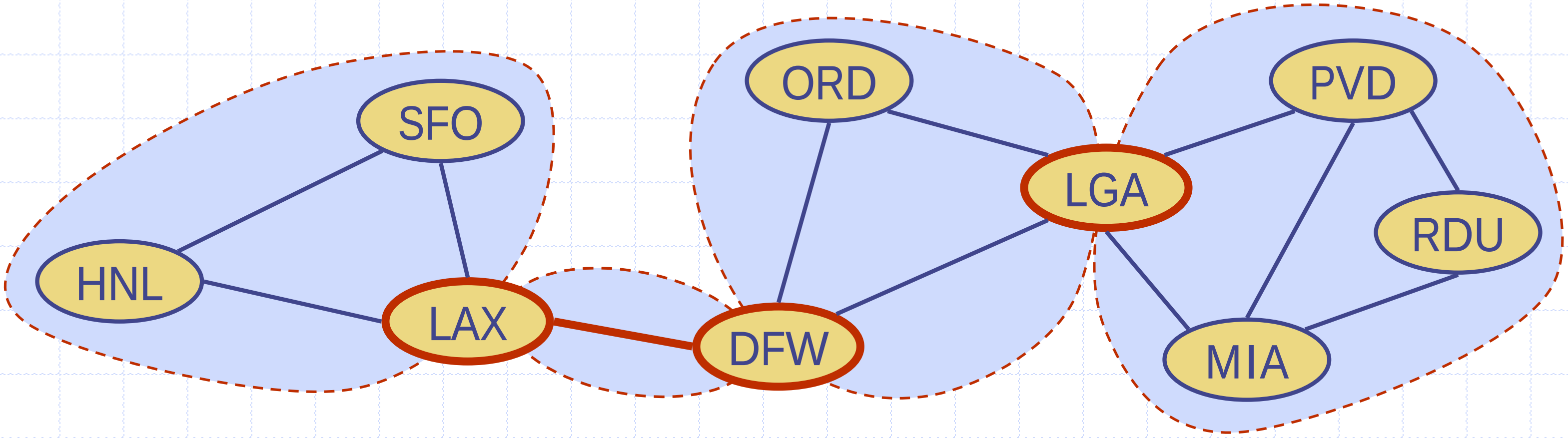
Equivalence classes of linked edges:

$\{a\}$ $\{b, c, d, e, f\}$ $\{g, i, j\}$



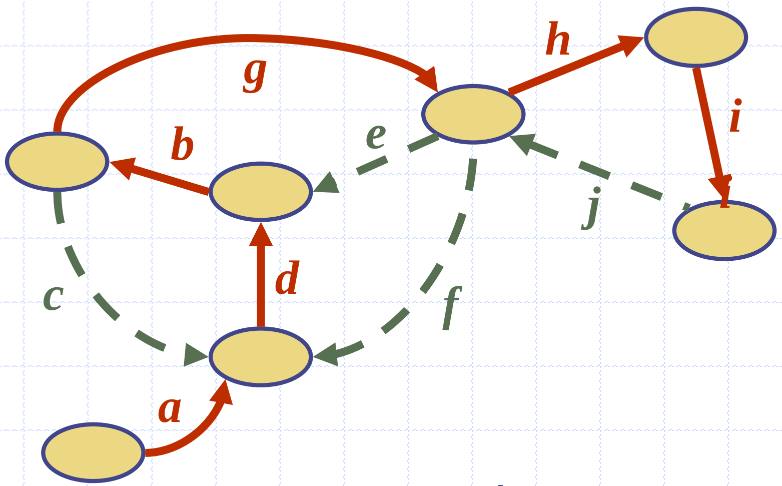
Link Components

- The link components of a connected graph G are the equivalence classes of edges with respect to the link relation
- A biconnected component of G is the subgraph of G induced by an equivalence class of linked edges
- A separation edge is a single-element equivalence class of linked edges
- A separation vertex has incident edges in at least two distinct equivalence classes of linked edge

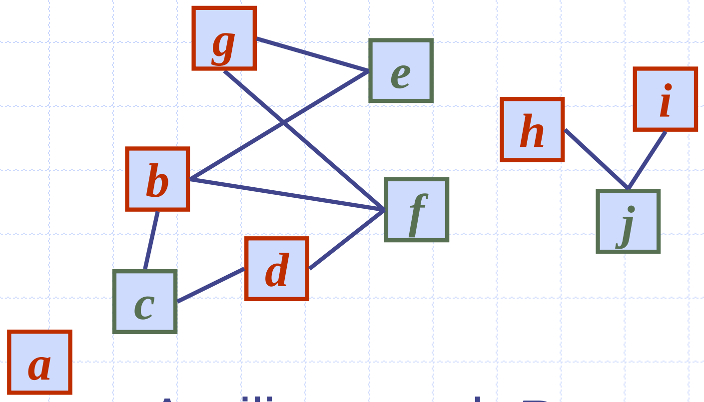


Auxiliary Graph

- Auxiliary graph B for a connected graph G
  - Associated with a DFS traversal of G
  - The vertices of B are the edges of G
  - For each back edge e of G, B has edges (e,f_1), (e,f_2) , …, (e,f_k), where f_1, f_2, …, f_k are the discovery edges of G that form a simple cycle with e
  - Its connected components correspond to the the link components of G



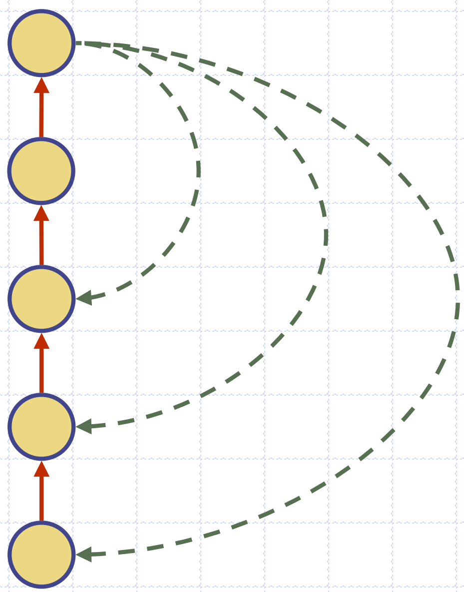
DFS on graph G



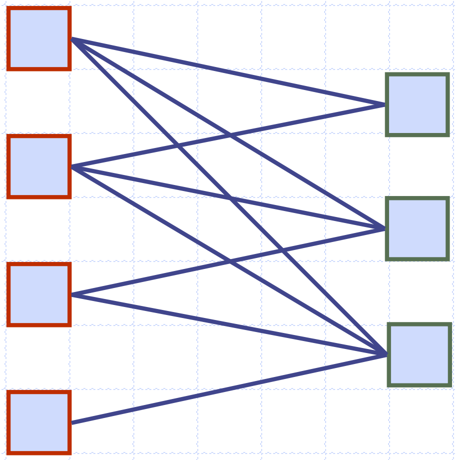
Auxiliary graph B

Auxiliary Graph (cont.)

- In the worst case, the number of edges of the auxiliary graph is proportional to ***nm***



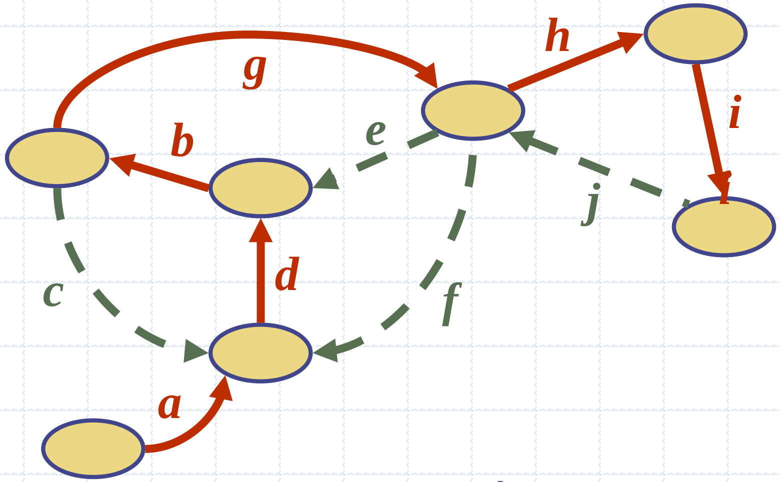
DFS on graph ***G***



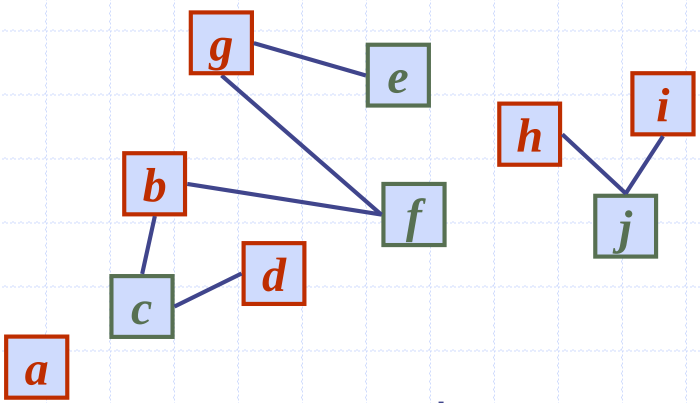
Auxiliary graph ***B***

Proxy Graph

```
Algorithm proxyGraph(G)
    Input connected graph G
    Output proxy graph F for G
    F ← empty graph
    DFS(G, s) { s is any vertex of G}
    for all discovery edges e of G
        F.insertVertex(e)
        setLabel(e, UNLINKED)
    for all vertices v of G in DFS visit order
        for all back edges e = (u,v)
            F.insertVertex(e)
            repeat
                f ← discovery edge with dest. u
                F.insertEdge(e,f,∅)
                if  f getLabel(f) = UNLINKED
                    setLabel(f, LINKED)
                    u ← origin of edge f
                else
                    u ← v { ends the loop }
            until u = v
    return F
```



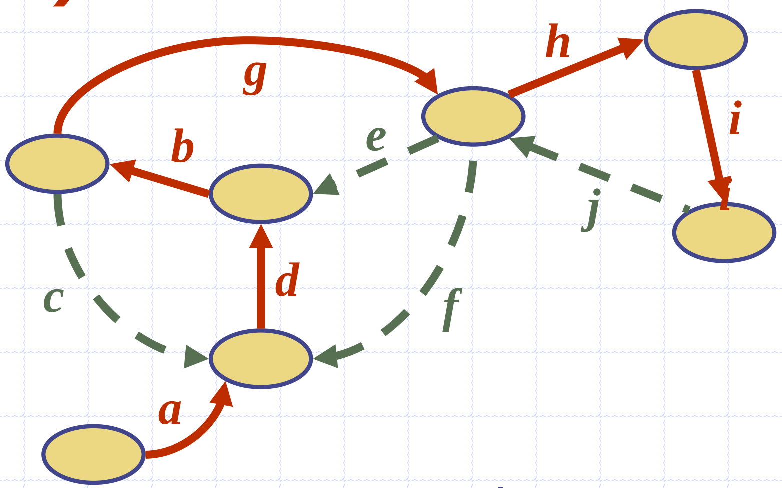
DFS on graph *G*



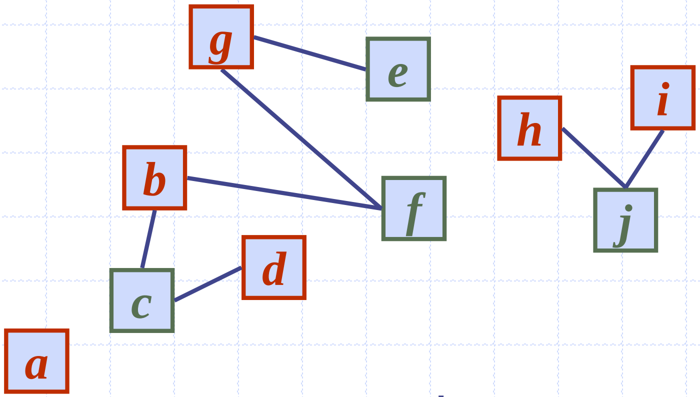
Proxy graph *F*

Proxy Graph (cont.)

- Proxy graph F for a connected graph G
  - Spanning forest of the auxiliary graph B
  - Has m vertices and $O(m)$ edges
  - Can be constructed in $O(n + m)$ time
  - Its connected components (trees) correspond to the the link components of G
- Given a graph G with n vertices and m edges, we can compute the following in $O(n + m)$ time
  - The biconnected components of G
  - The separation vertices of G
  - The separation edges of G

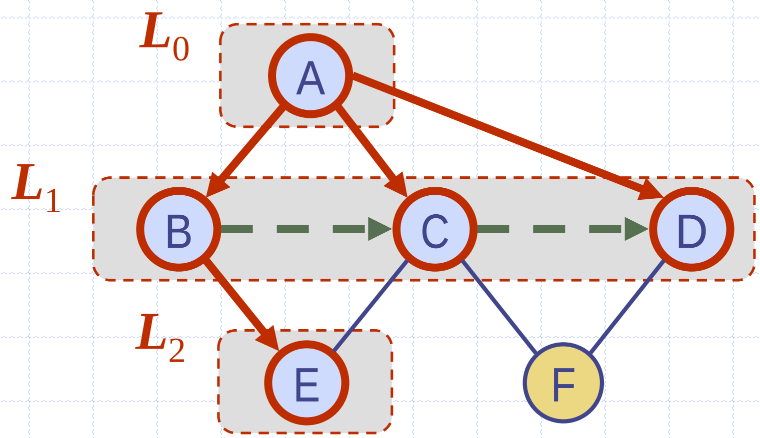


DFS on graph G



Proxy graph F

6.3.3 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

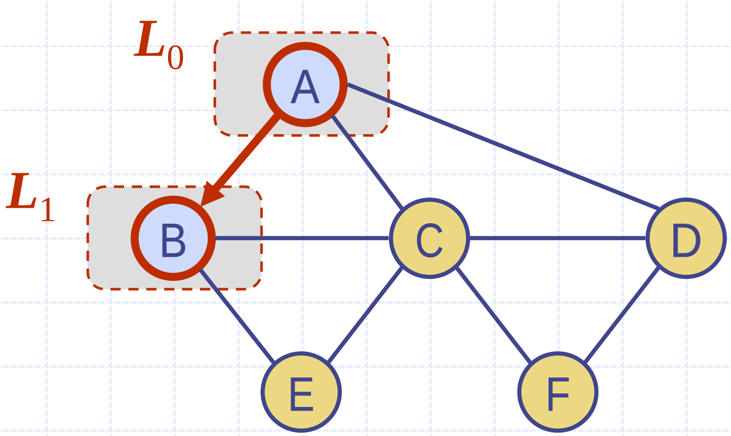
- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

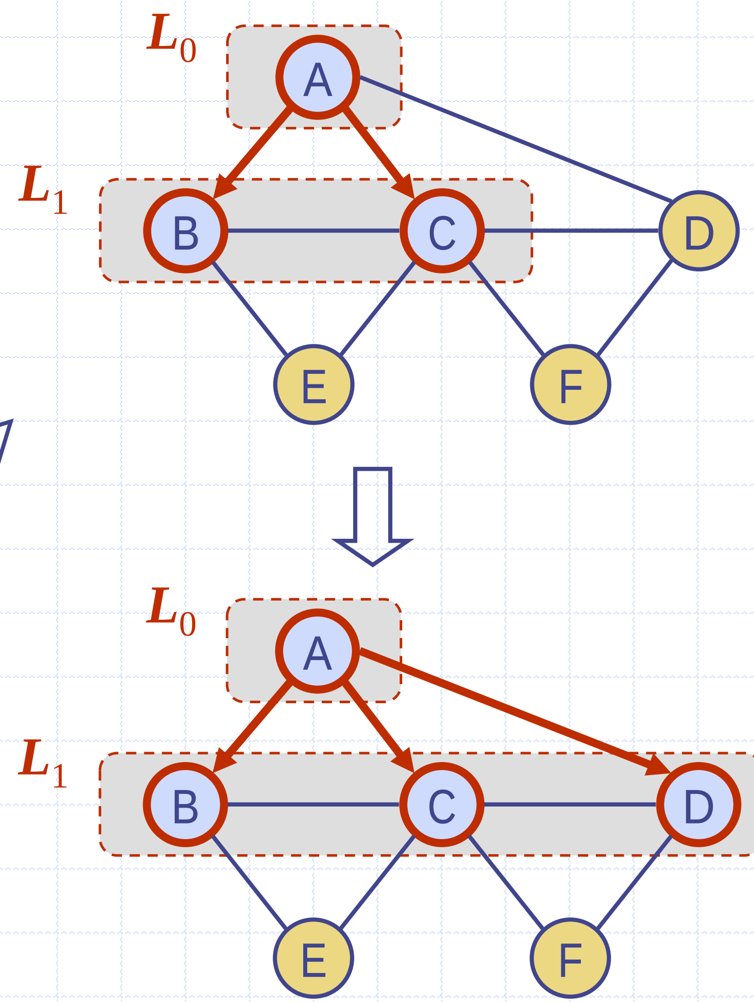
```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
        BFS(G,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

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

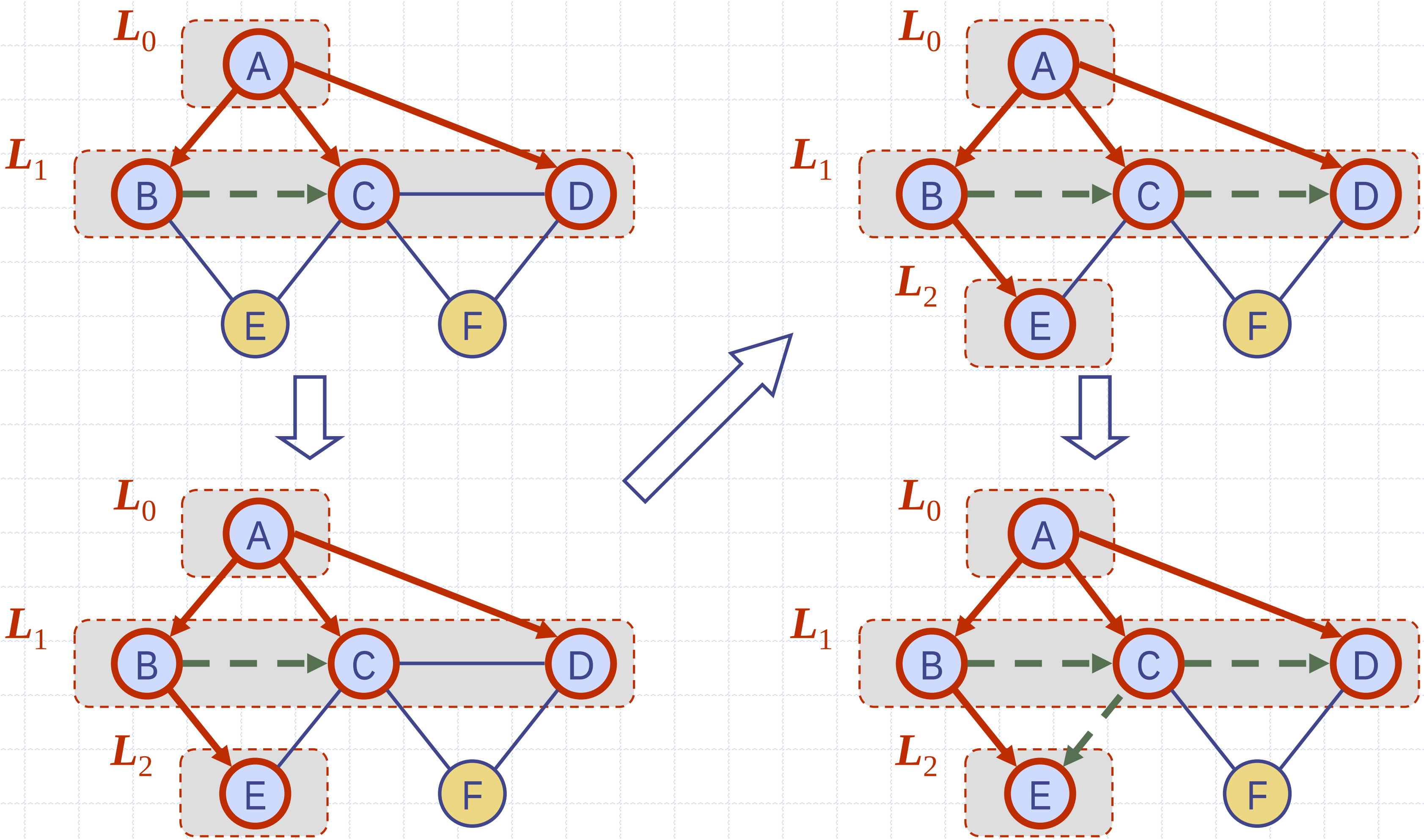
Example

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

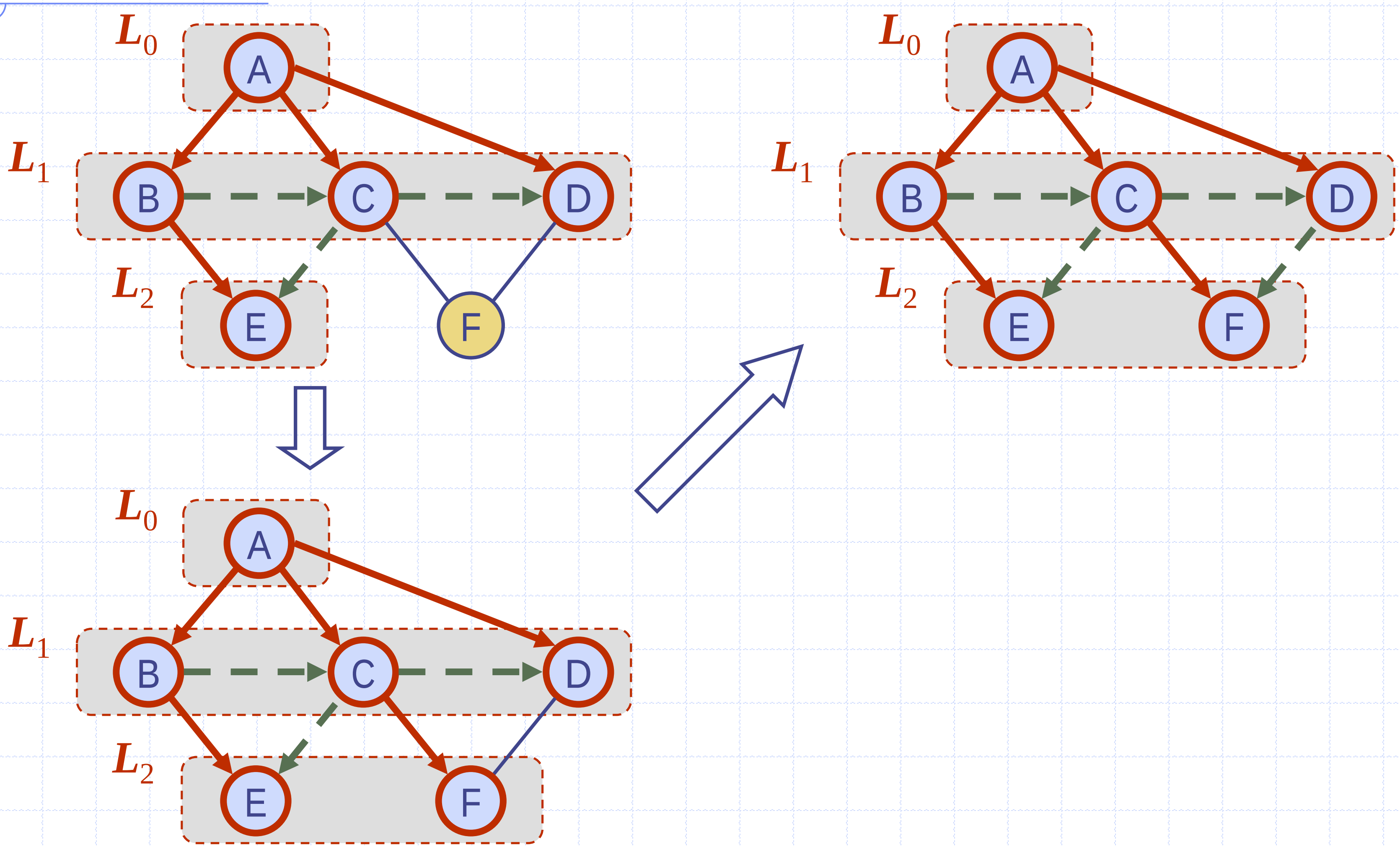




Example (cont.)



Example (cont.)



Properties

Notation

G_s: connected component of s

Property 1

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

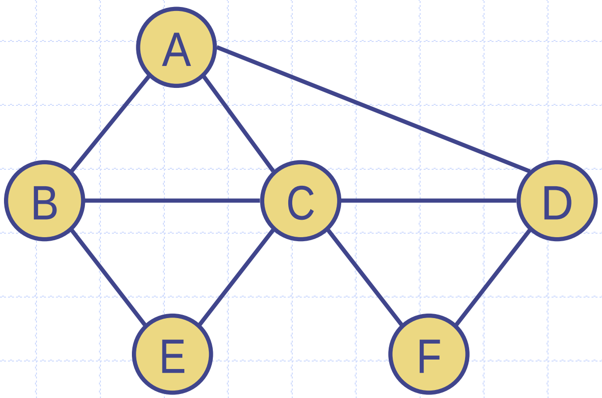
Property 2

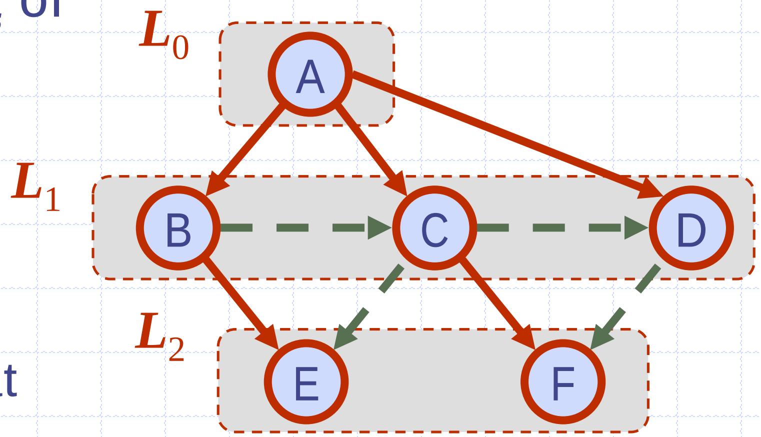
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

Property 3

For each vertex v in L_i

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





Analysis

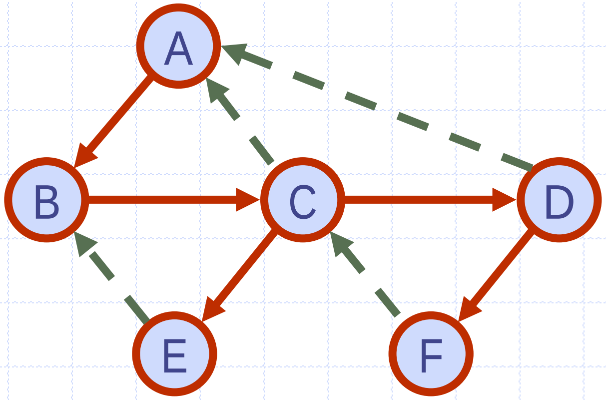
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges is called once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

Applications

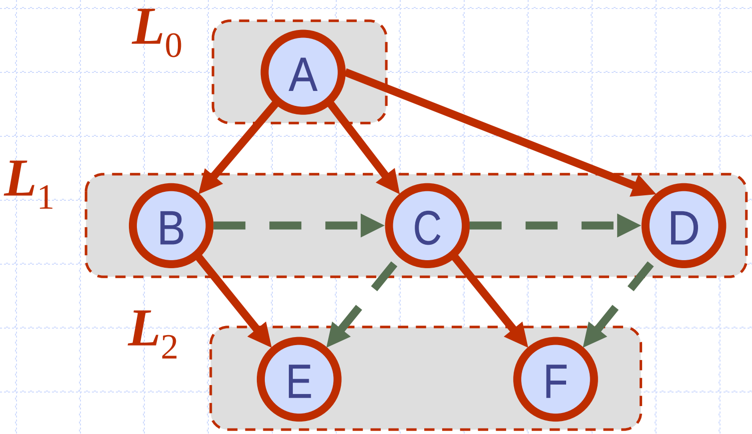
- Using the template method pattern, we can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G,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

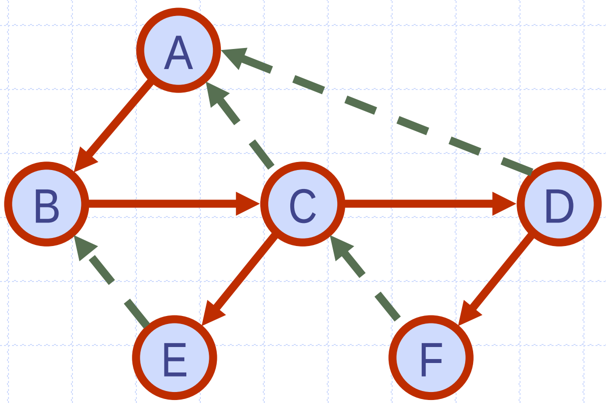
DFS vs. BFS (cont.)

Back edge (v,w)

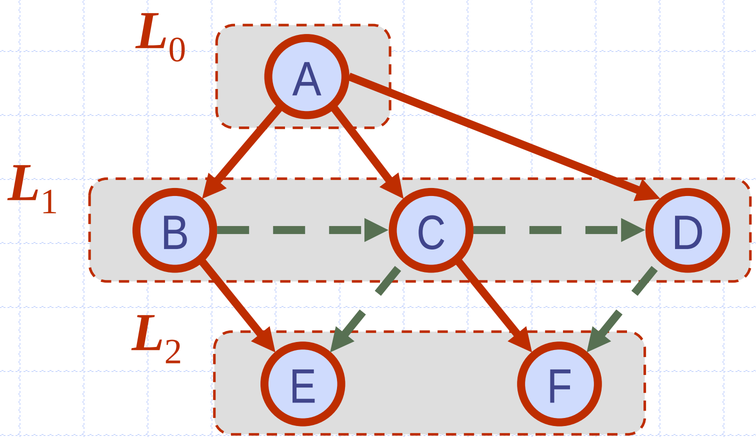
- w is an ancestor of v in the tree of discovery edges

Cross edge (v,w)

- w is in the same level as v or in the next level in the tree of discovery edges

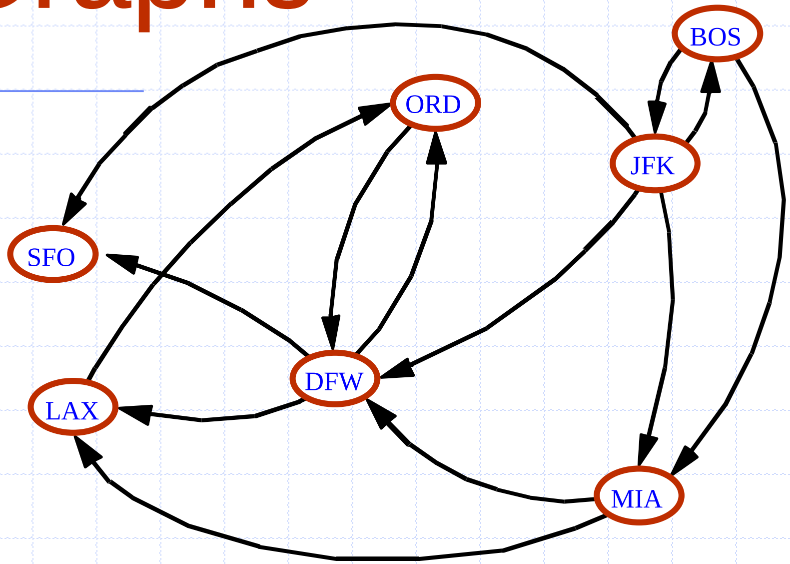


DFS



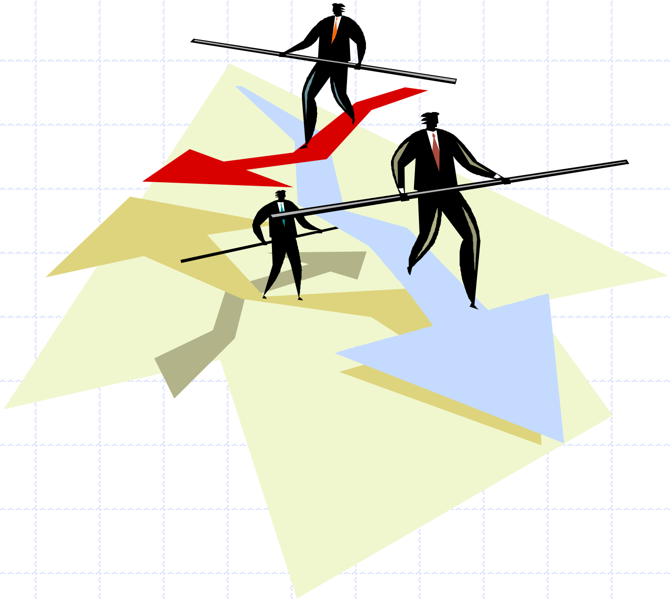
BFS

6.4 Directed Graphs



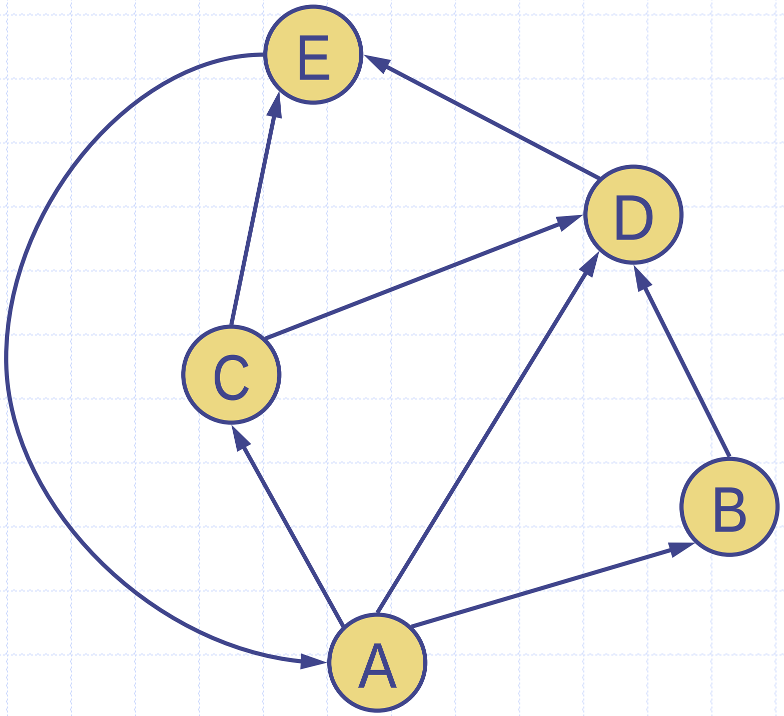
Outline and Reading (§6.4)

- Reachability (§6.4.1)
  - Directed DFS
  - Strong connectivity
- Transitive closure (§6.4.2)
  - The Floyd-Warshall Algorithm
- Directed Acyclic Graphs (DAG's) (§6.4.4)
  - Topological Sorting

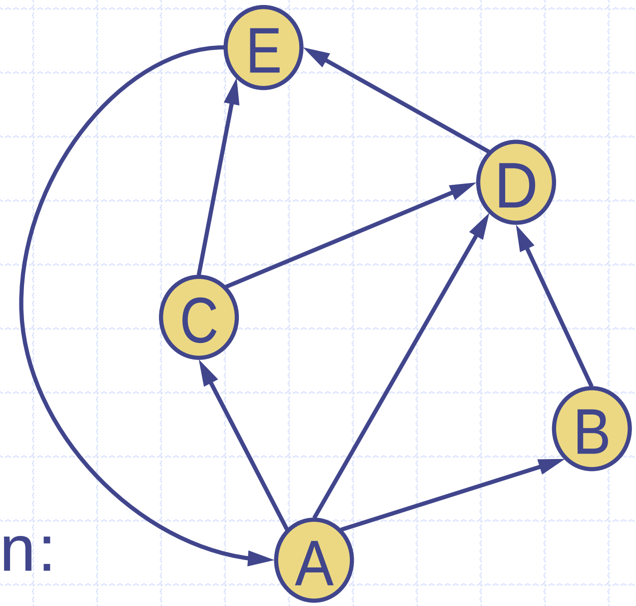


Digraphs

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



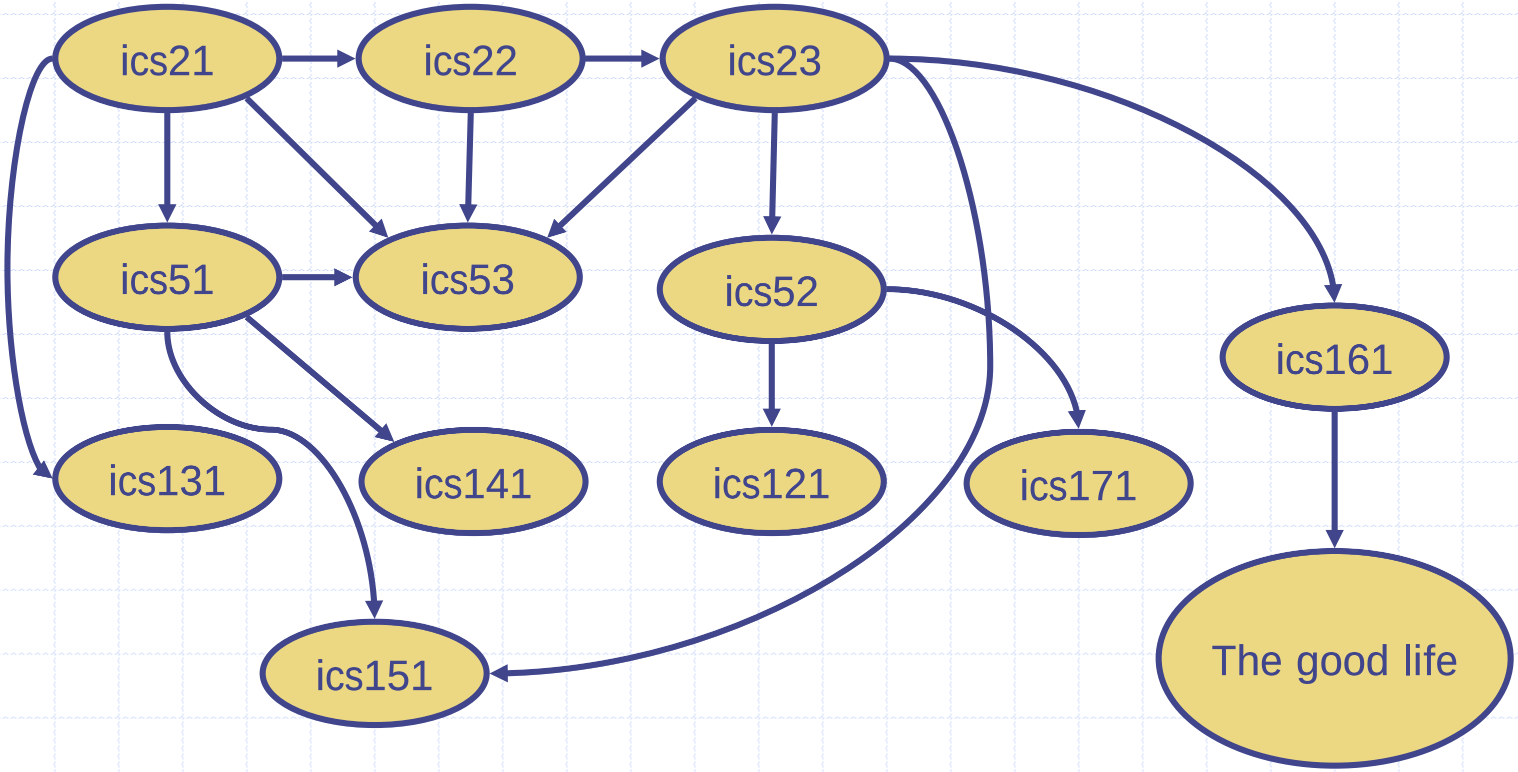
Digraph Properties



- 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(n-1).
- If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of of the sets of in-edges and out-edges in time proportional to their size.

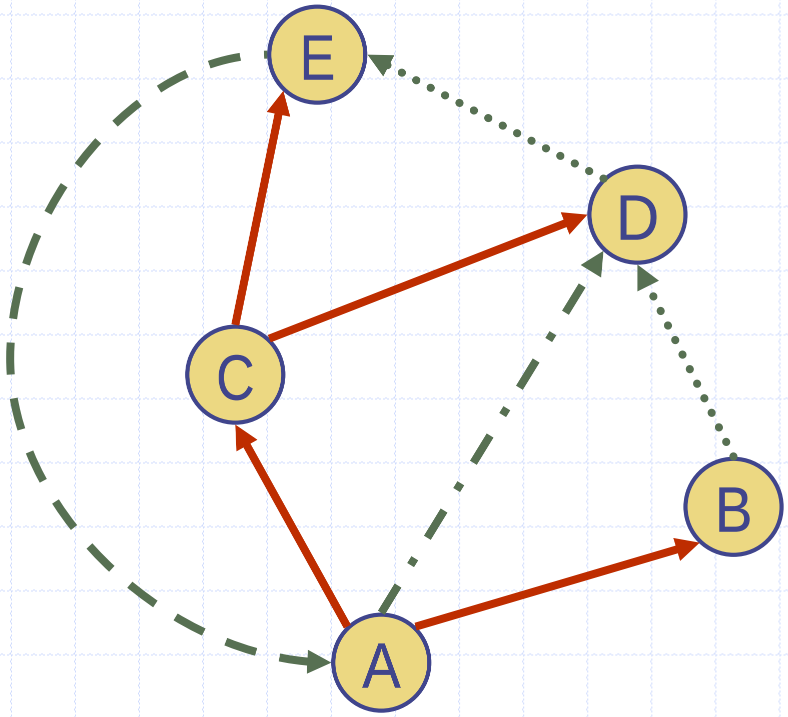
Digraph Application

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



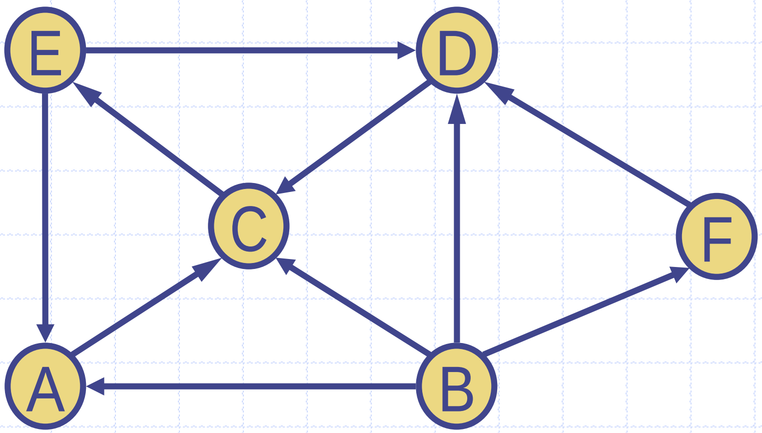
Directed DFS

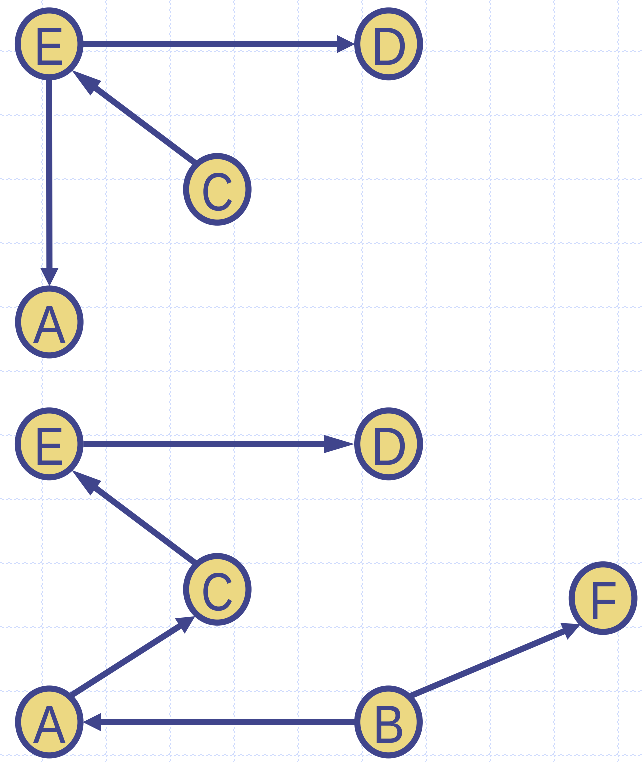
- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction
- In the directed DFS algorithm, we have four types of edges
  - discovery edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex s determines the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



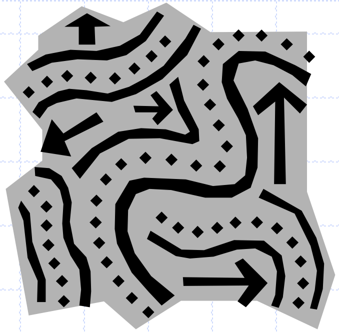
Reachability

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

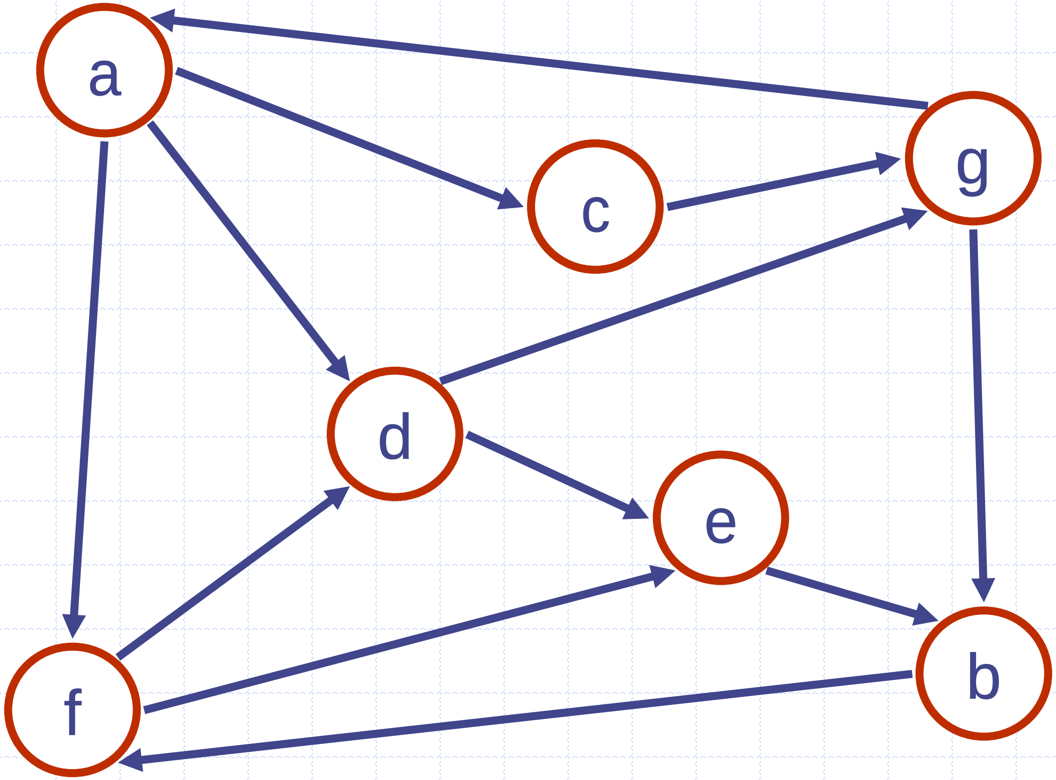




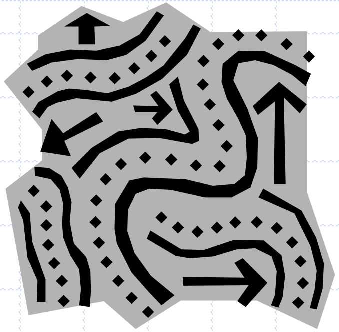
Strong Connectivity



- Each vertex can reach all other vertices

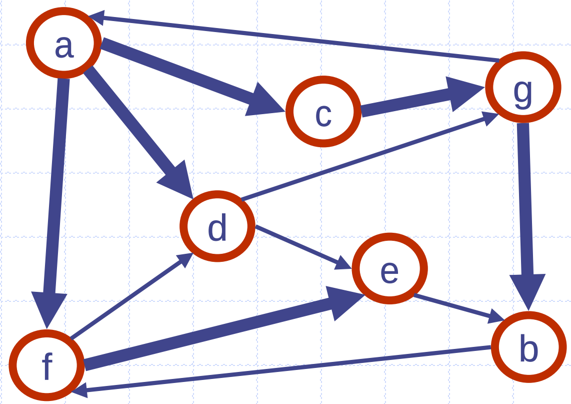


Strong Connectivity Algorithm

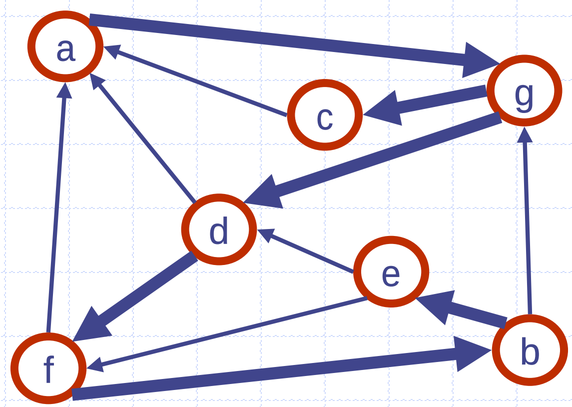


- Pick a vertex v in G.
- Perform a DFS from v in G.
  - If there's a w not visited, print "no".
- Let G' be G with edges reversed.
- Perform a DFS from v in G'.
  - If there's a w not visited, print "no".
  - Else, print "yes".
- Running time: O(n+m).

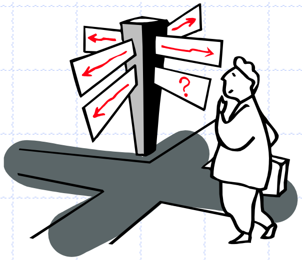
G:



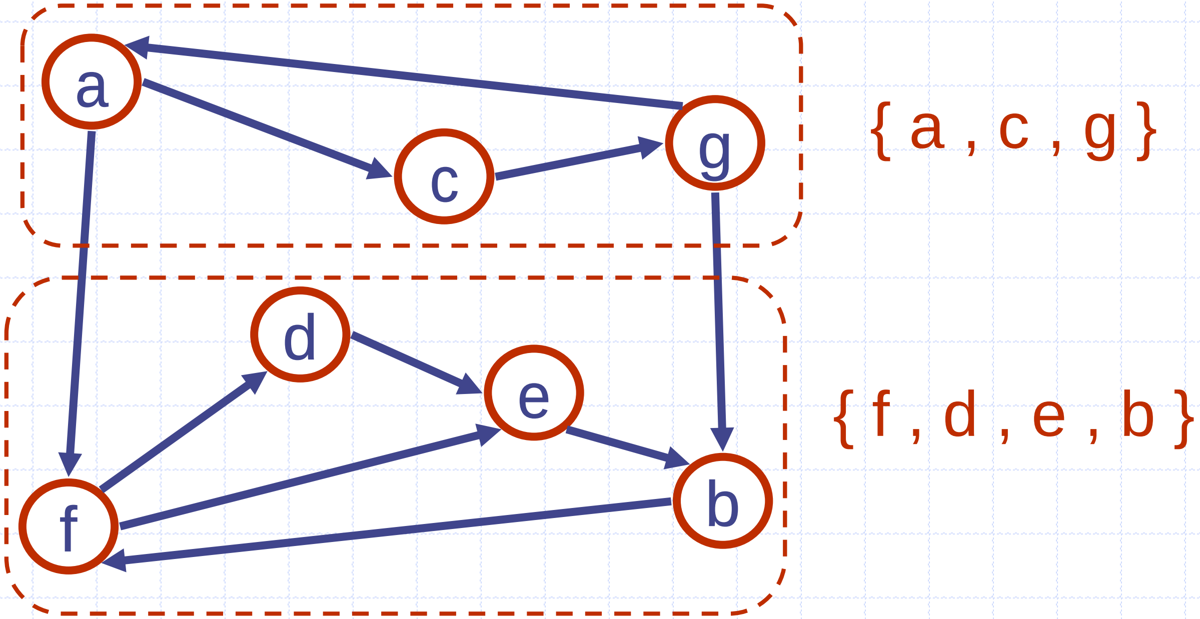
G':



Strongly Connected Components

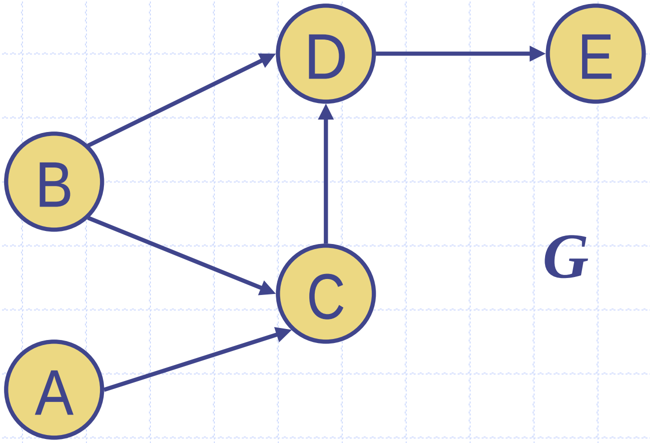


- Maximal subgraphs such that each vertex can reach all other vertices in the subgraph
- Can also be done in O(n+m) time using DFS, but is more complicated (similar to biconnectivity).

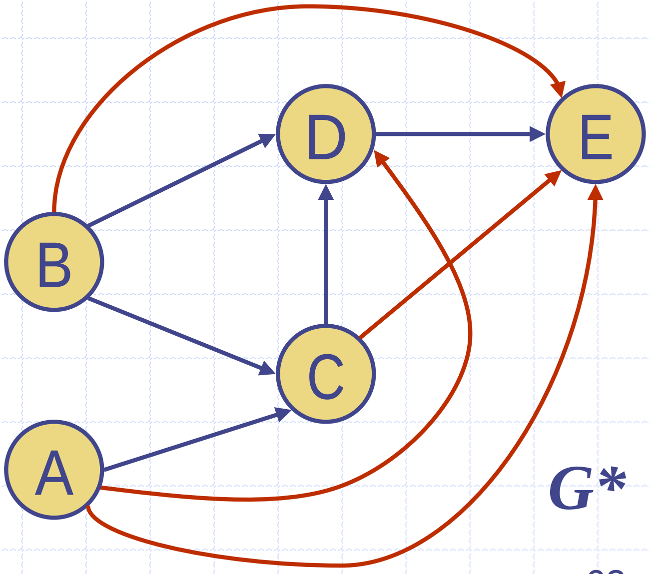


Transitive Closure

- 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



G

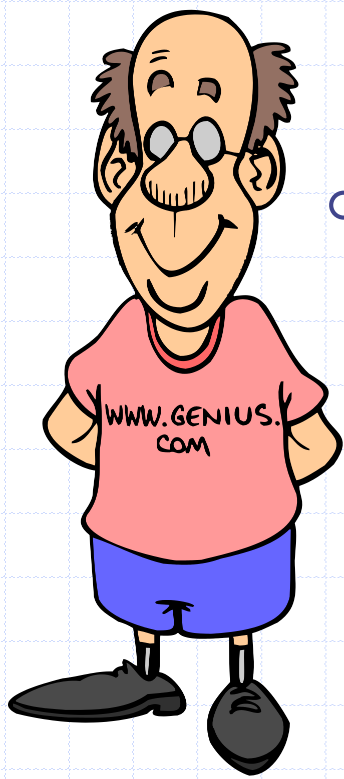


G^*

Computing the Transitive Closure

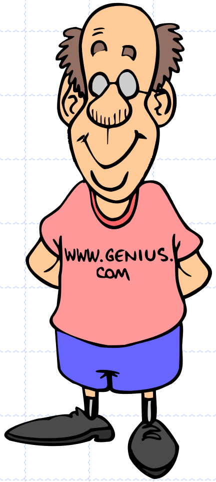
- We can perform DFS starting at each vertex
  - O(n(n+m))

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.

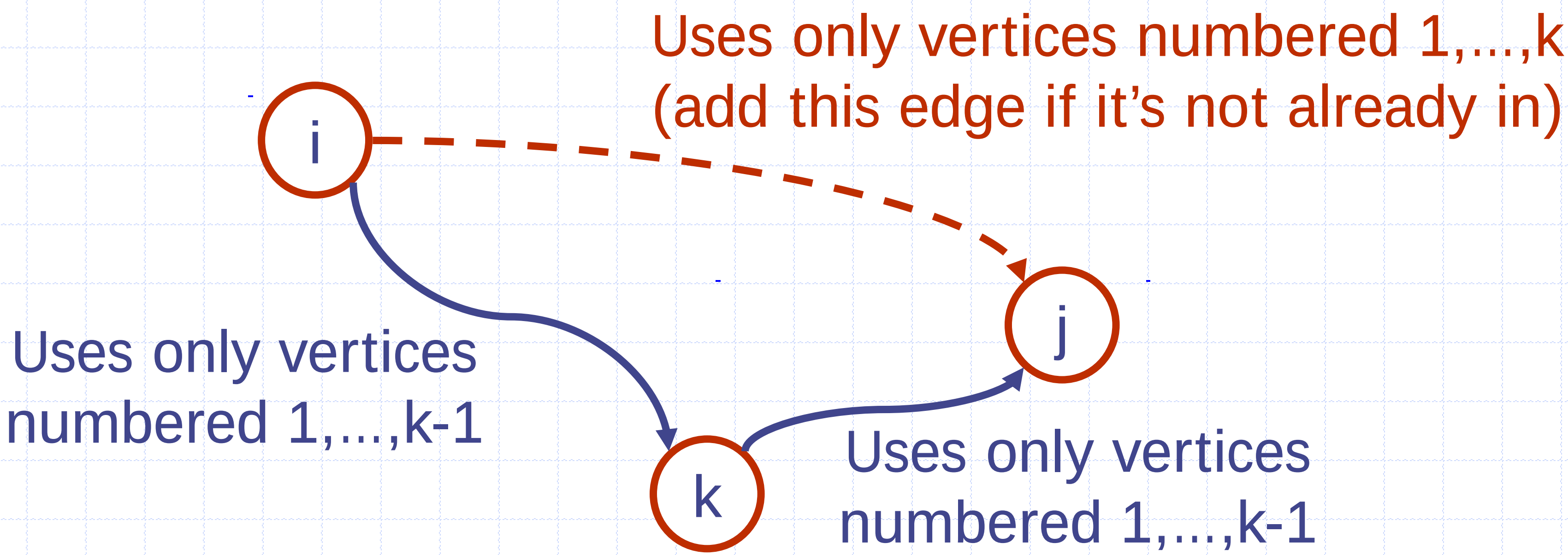


- Alternatively ... Use dynamic programming: the Floyd-Warshall Algorithm

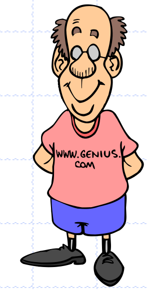
Floyd-Warshall Transitive Closure



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



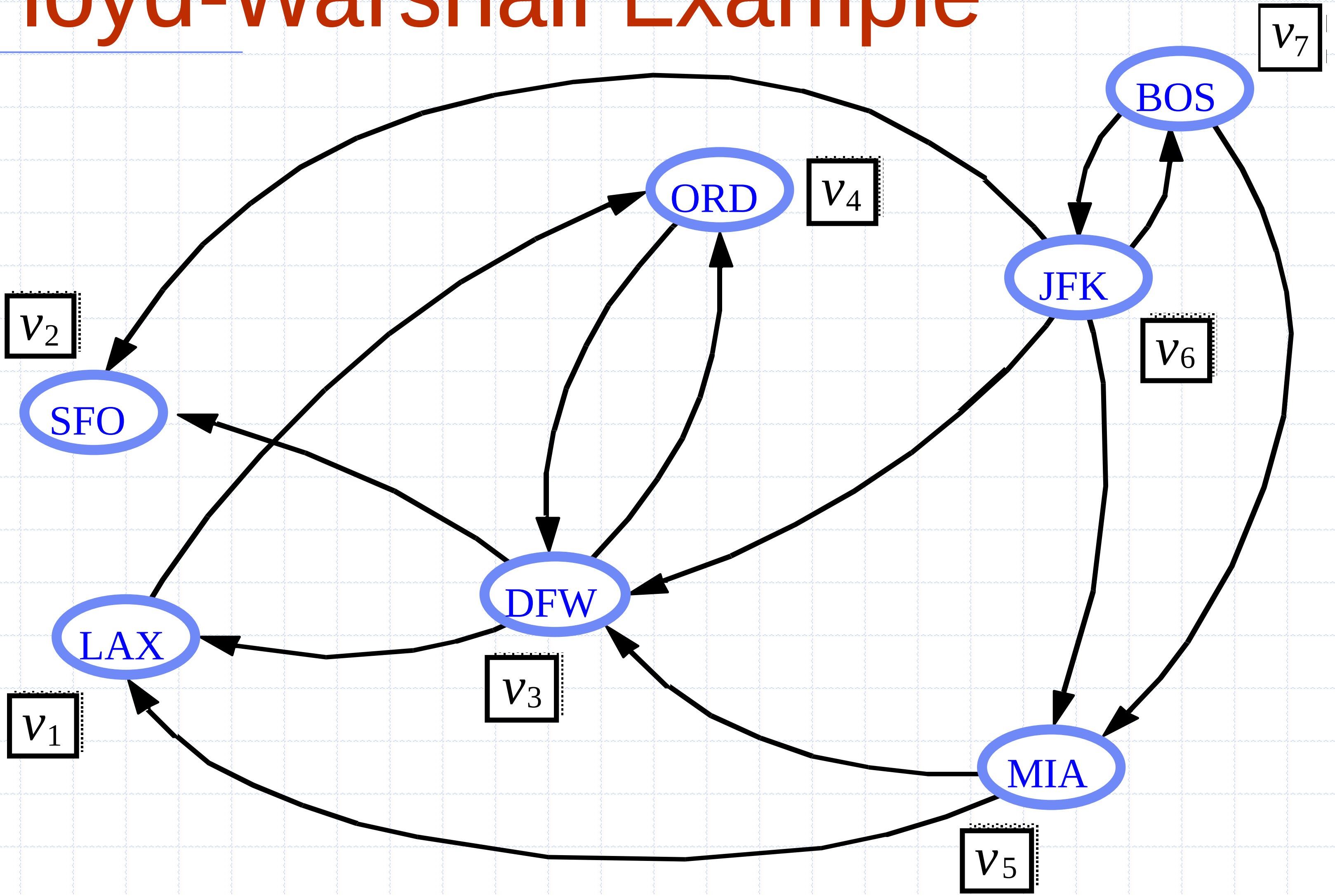
Floyd-Warshall's Algorithm



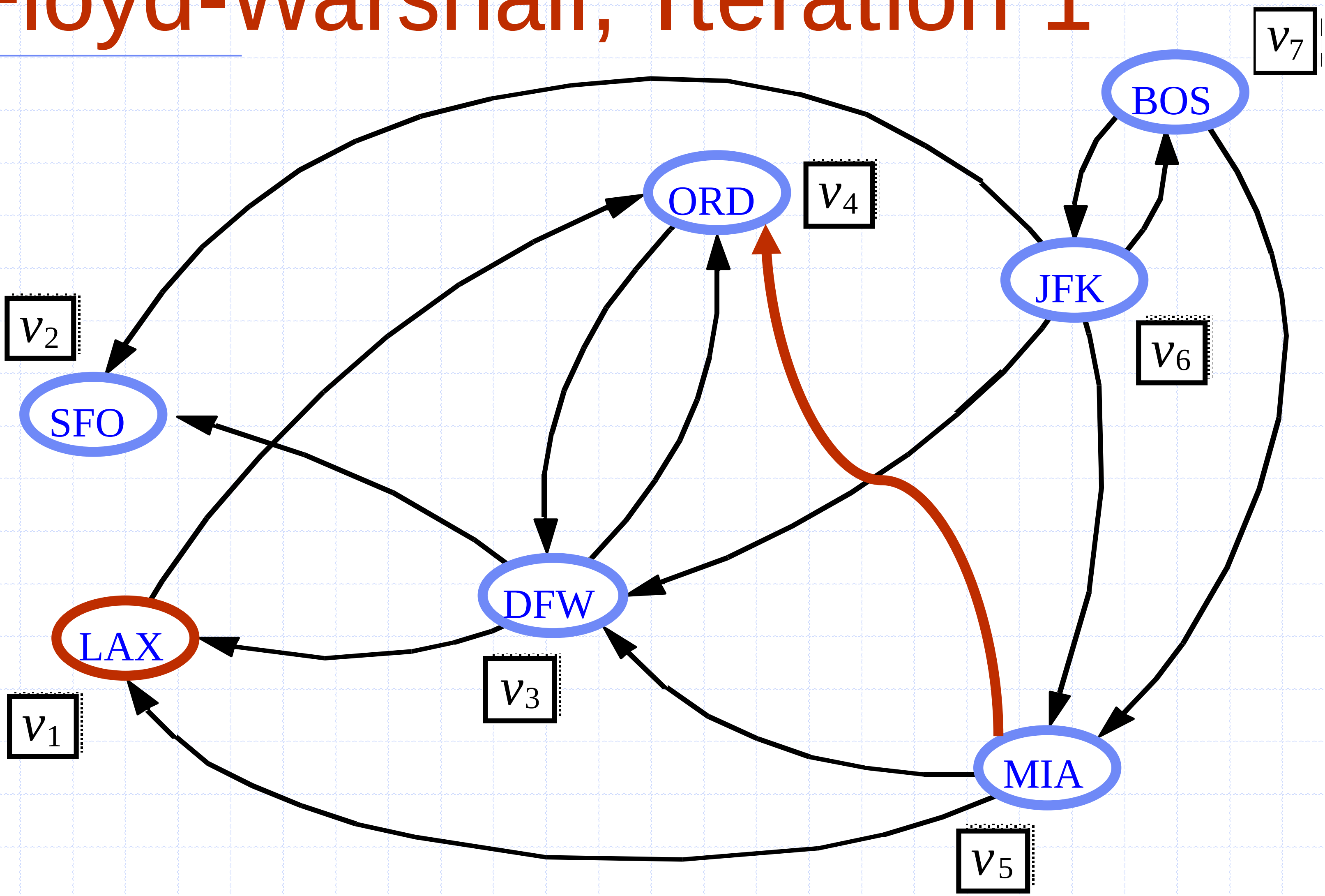
- Floyd-Warshall's algorithm numbers the vertices of G as $v_1, \ldots, v_n$ and computes a series of digraphs $G_0, \ldots, G_n$
  - $G_0 = G$
  - G_k has a directed edge (v_i, v_j) if G has a directed path from v_i to v_j with intermediate vertices in the set $\{v_1, \ldots, v_k\}$
- We have that $G_n = G^*$
- 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

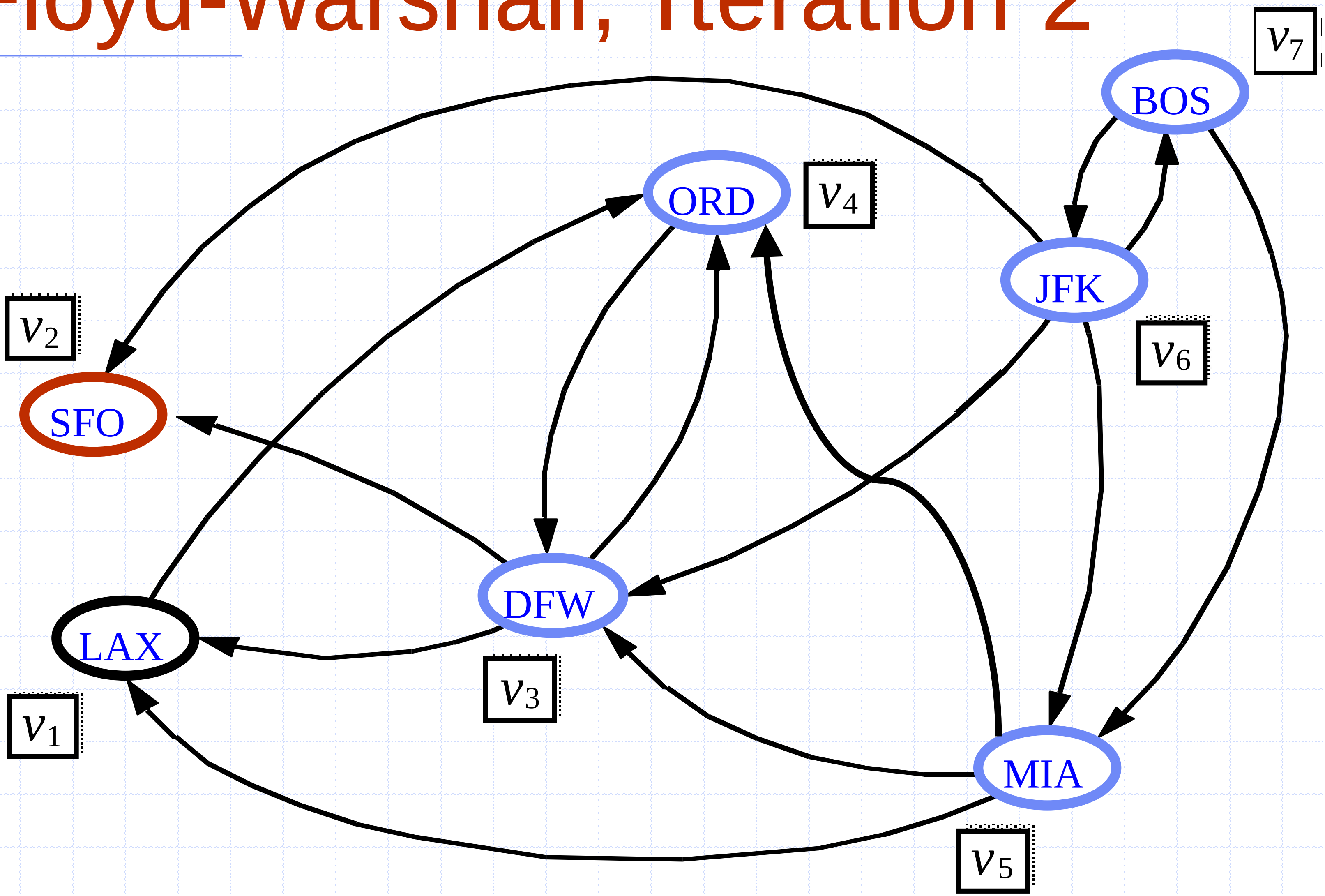
Floyd-Warshall Example



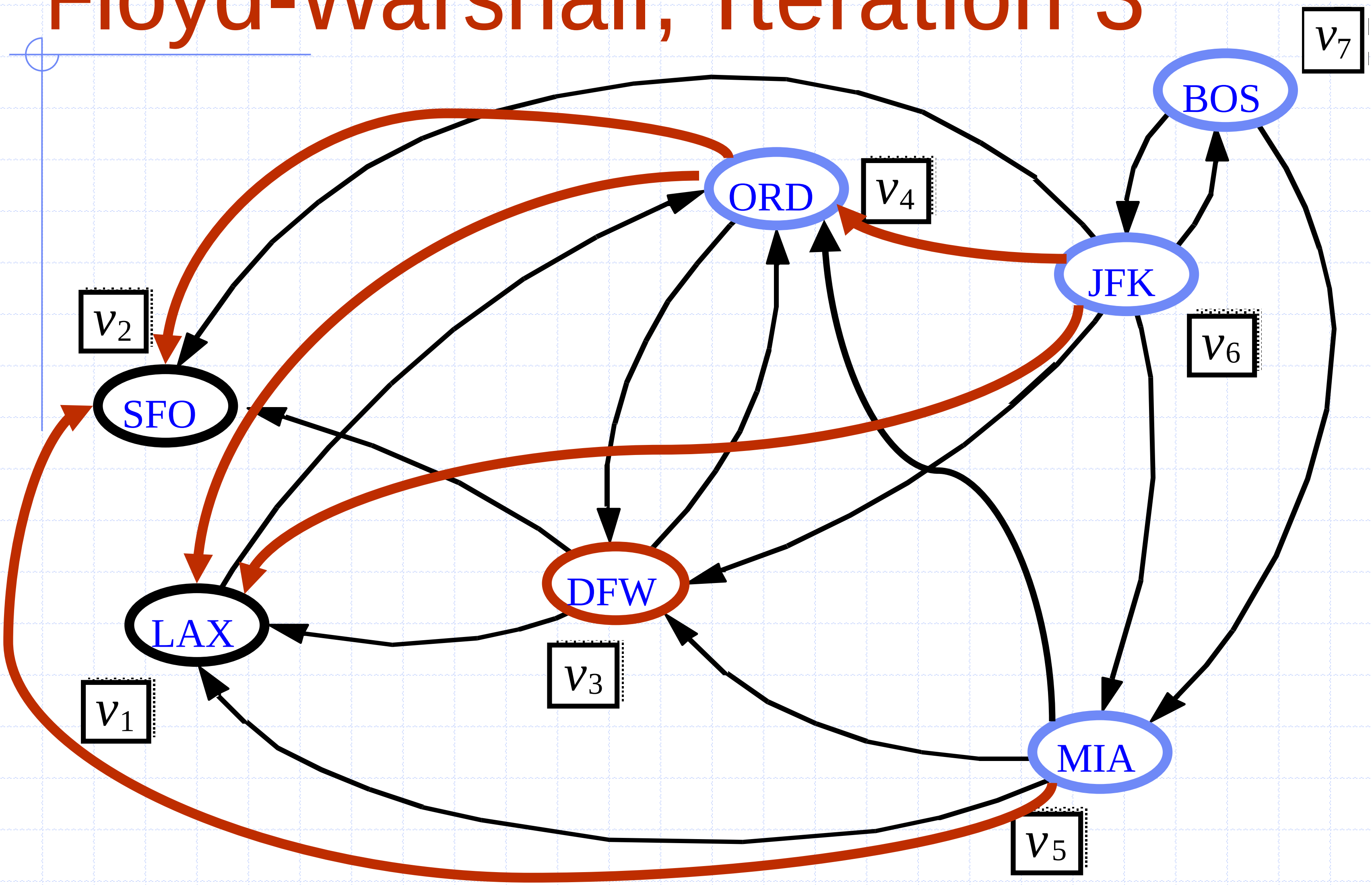
Floyd-Warshall, Iteration 1



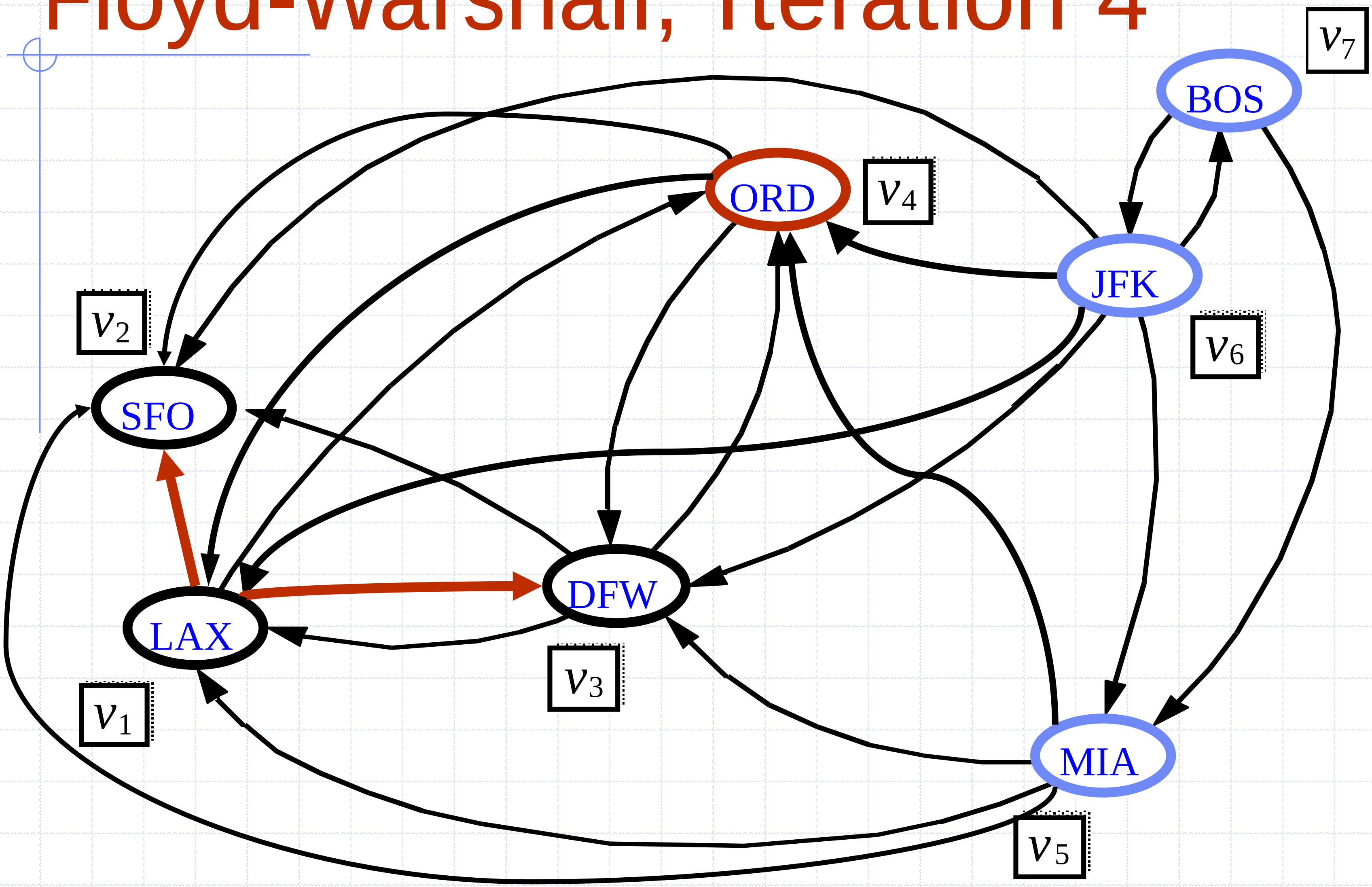
Floyd-Warshall, Iteration 2



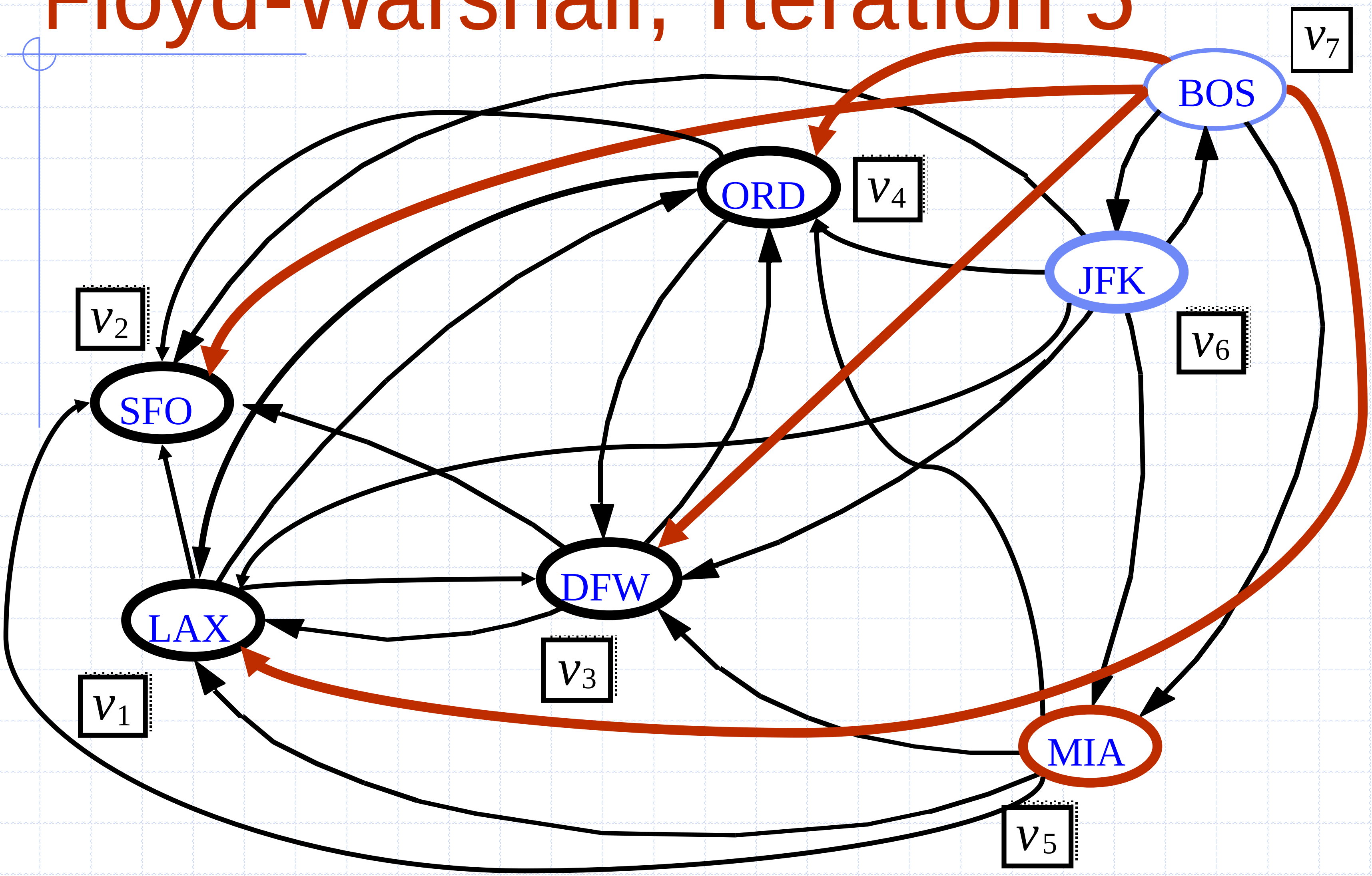
Floyd-Warshall, Iteration 3



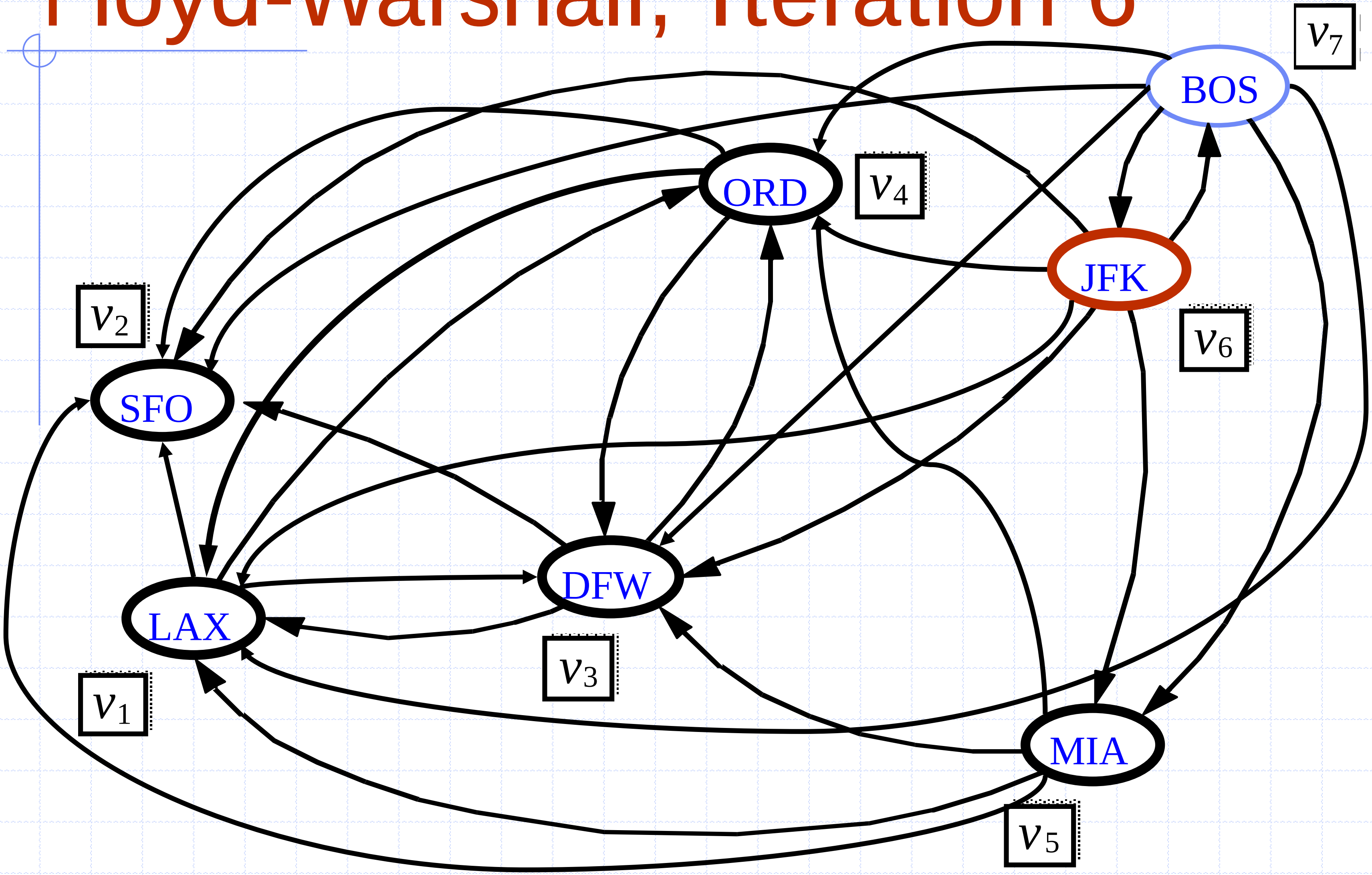
Floyd-Warshall, Iteration 4



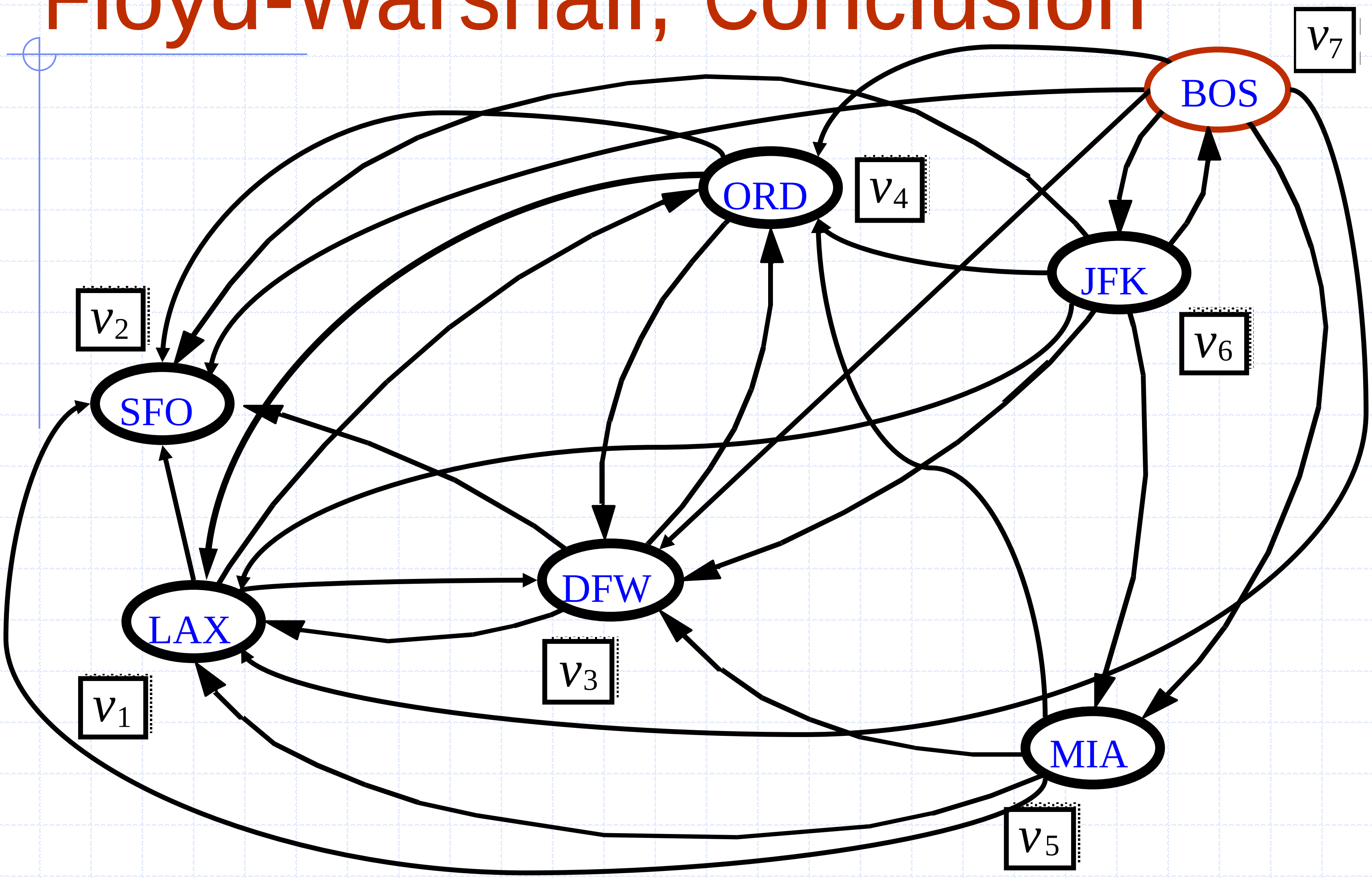
Floyd-Warshall, Iteration 5



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

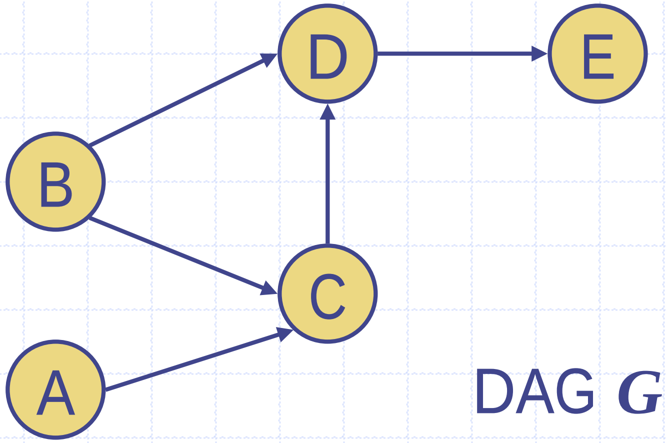
$$v_1, \ldots, v_n$$

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

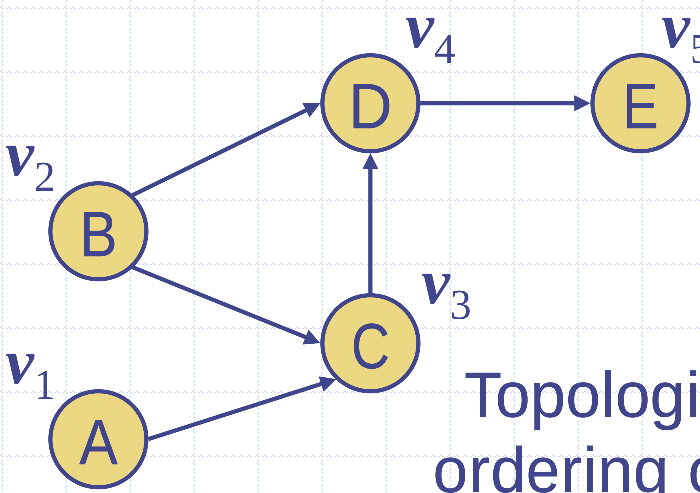
- Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints

Theorem

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

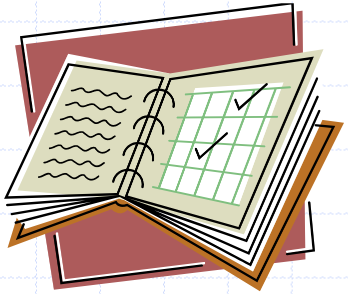


DAG G

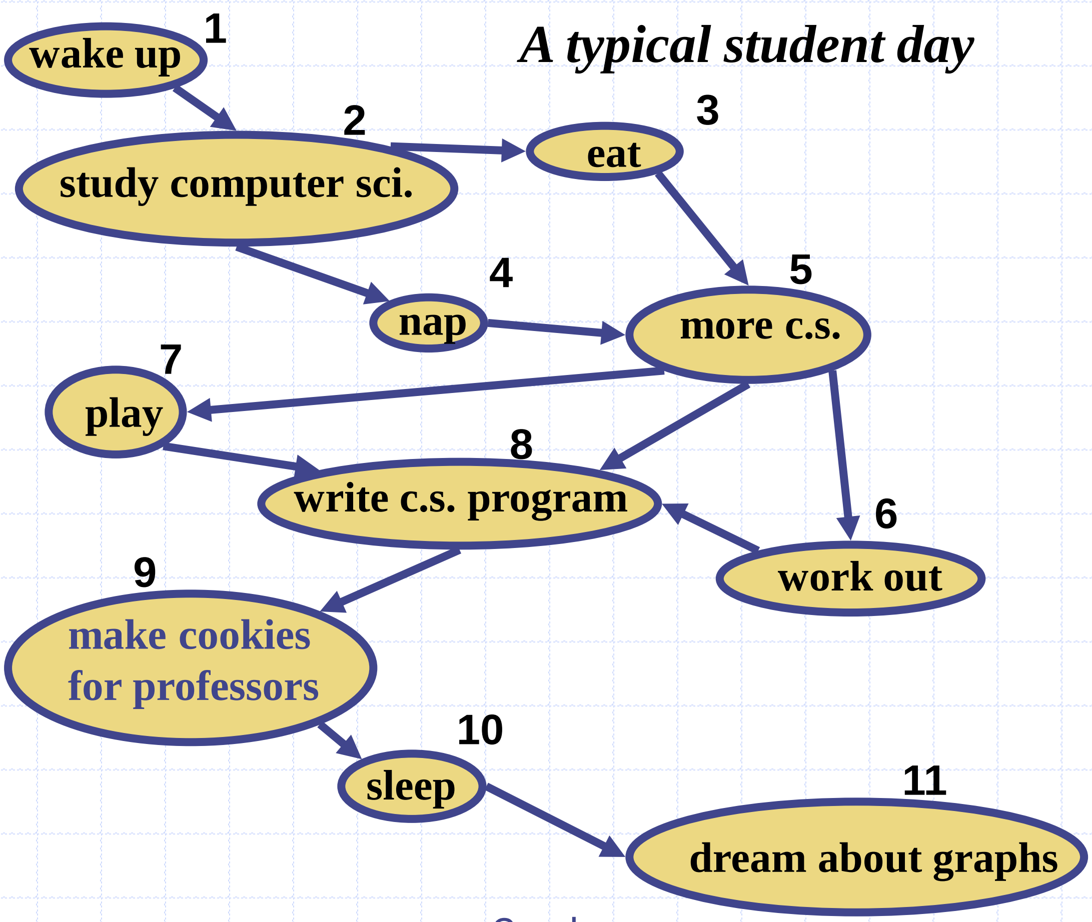


Topological ordering of G

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 Goodrich-Tamassia

```
Method TopologicalSort(G)
    H ← G            // Temporary copy of G
    n ← G.numVertices()
    while H is not empty do
        Let v be a vertex with no outgoing edges
        Label v ← n
        n ← n - 1
        Remove v from H
```

- Running time: O(n + m). How...?

Topological Sorting Algorithm using DFS

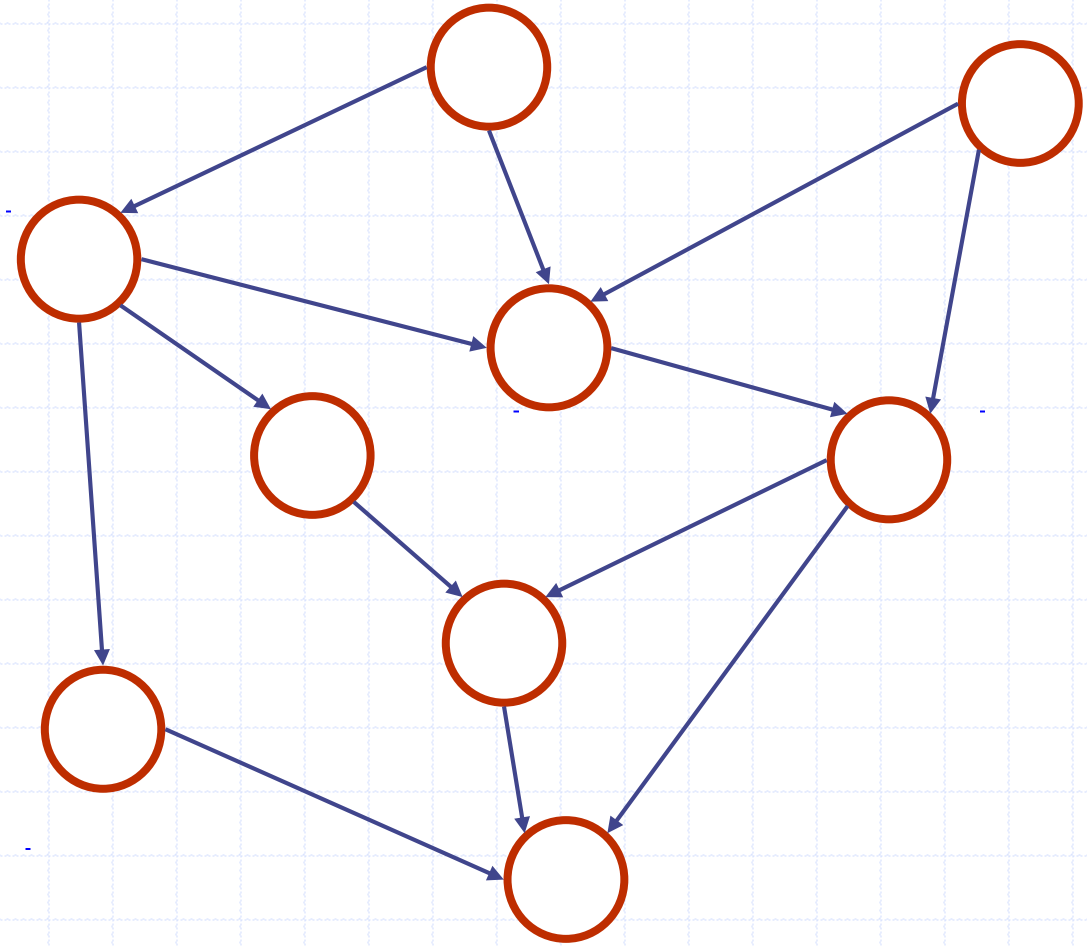
- Simulate the algorithm by using depth-first search

Algorithm *topologicalDFS*(*G*)
    Input dag *G*
    Output topological ordering of *G*
  $n \leftarrow$ *G.numVertices*()
  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*
      topologicalDFS(*G, v*)

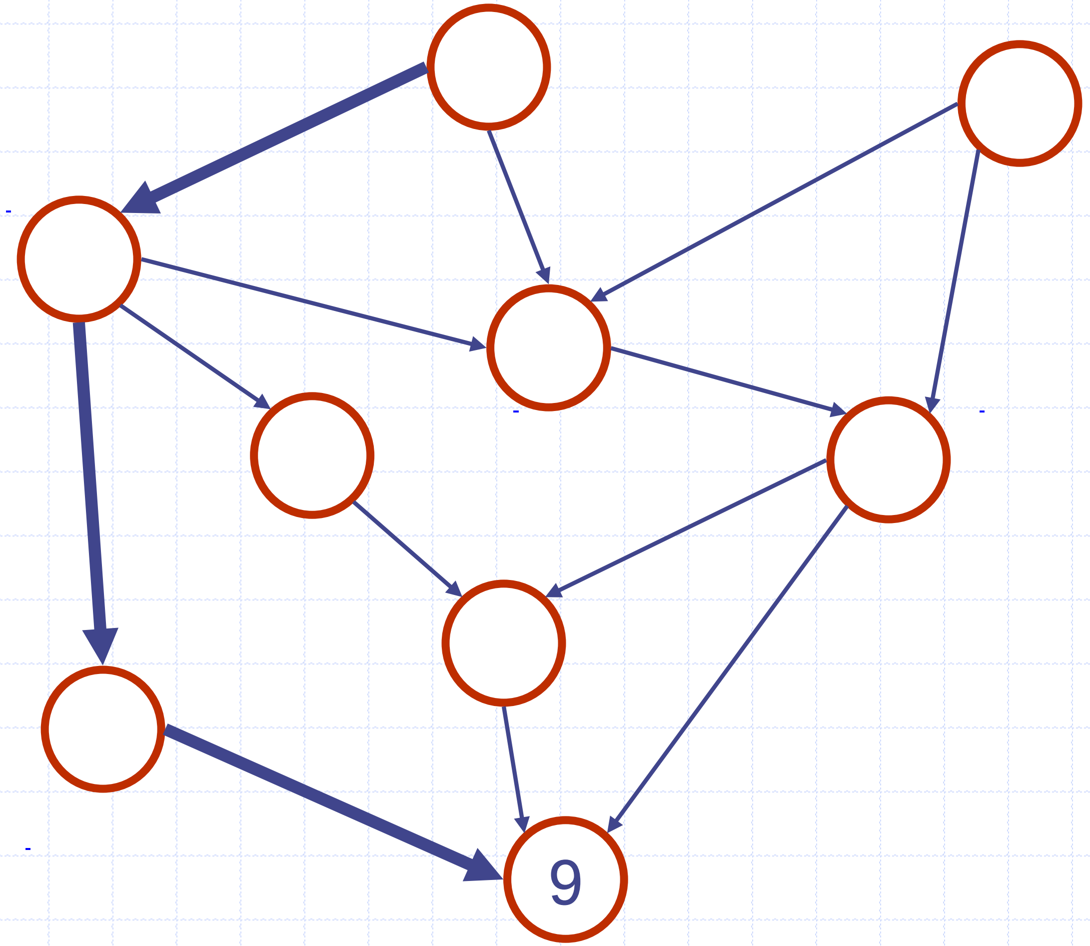
- O(n+m) time.

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*
  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*)
        topologicalDFS(*G, w*)
      else
        {*e* is a forward or cross edge}
  Label *v* with topological number *n*
  $n \leftarrow n - 1$

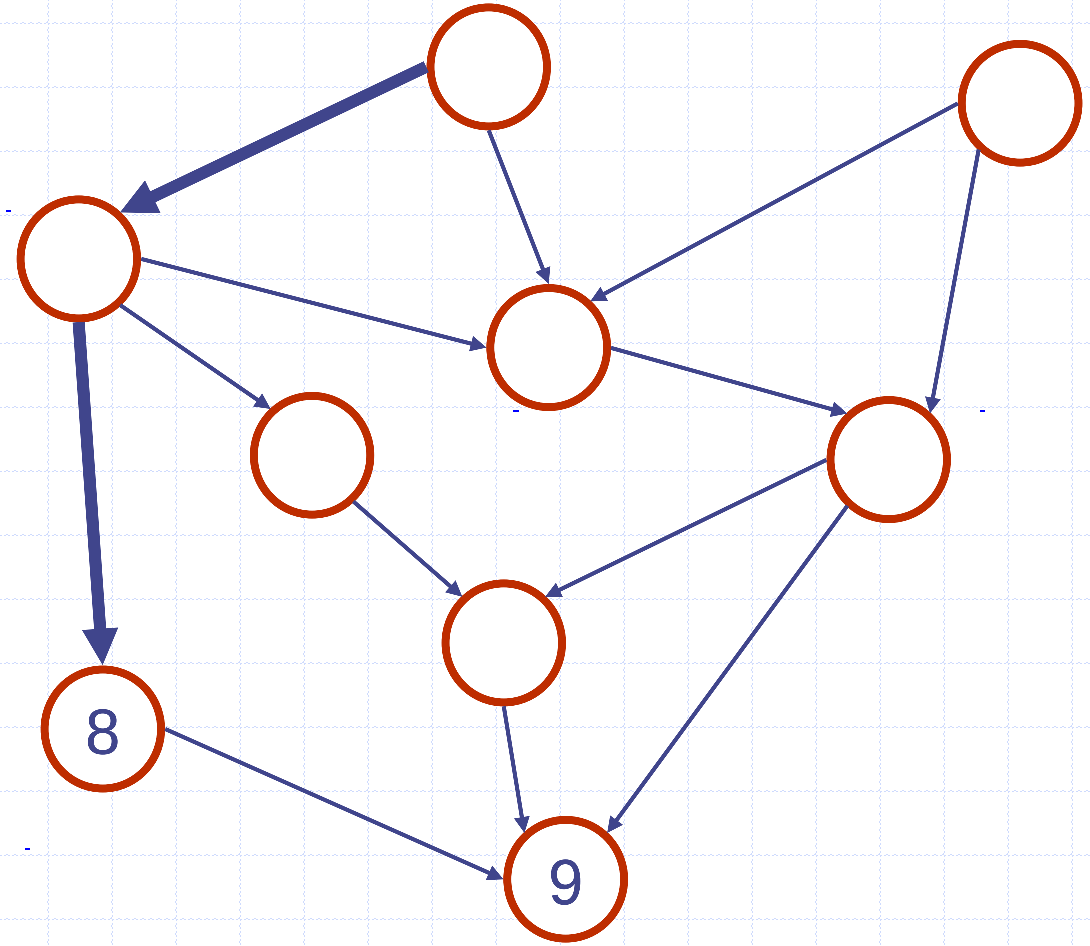
Topological Sorting Example



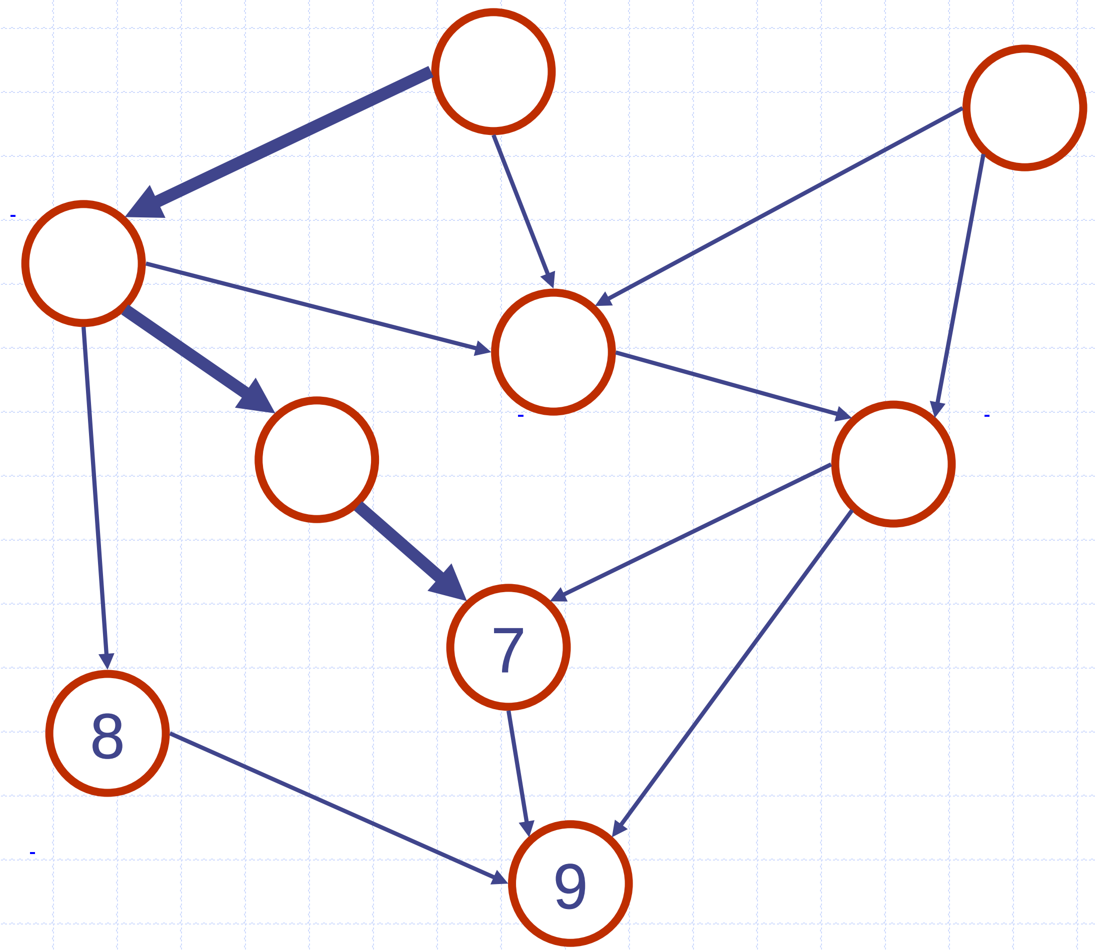
Topological Sorting Example



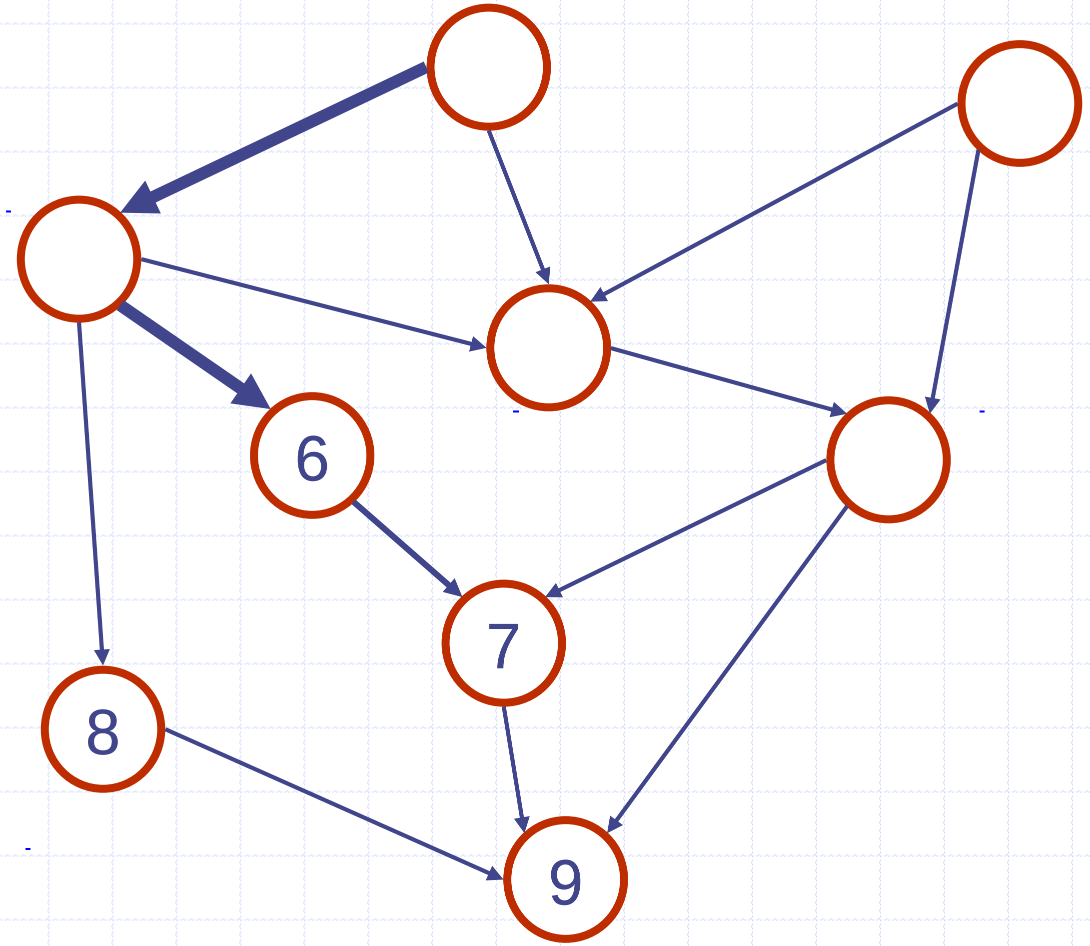
Topological Sorting Example



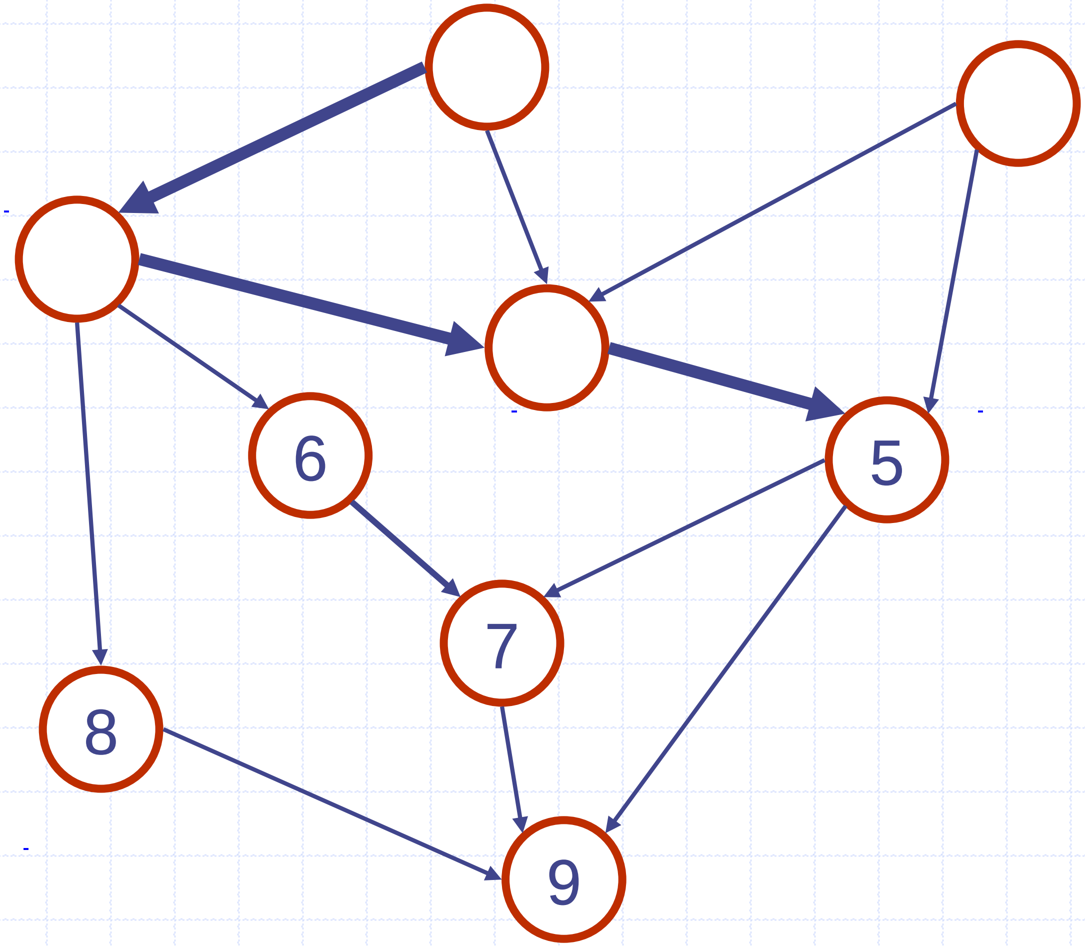
Topological Sorting Example



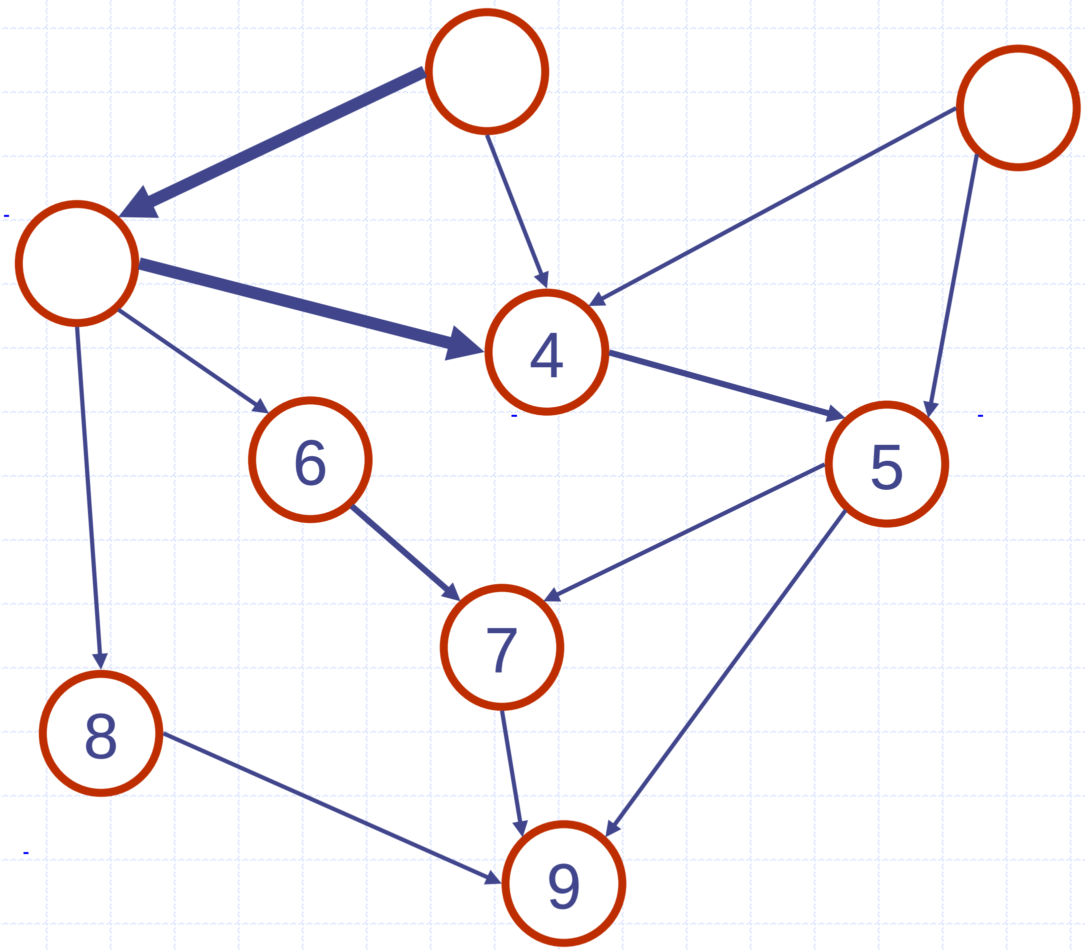
Topological Sorting Example



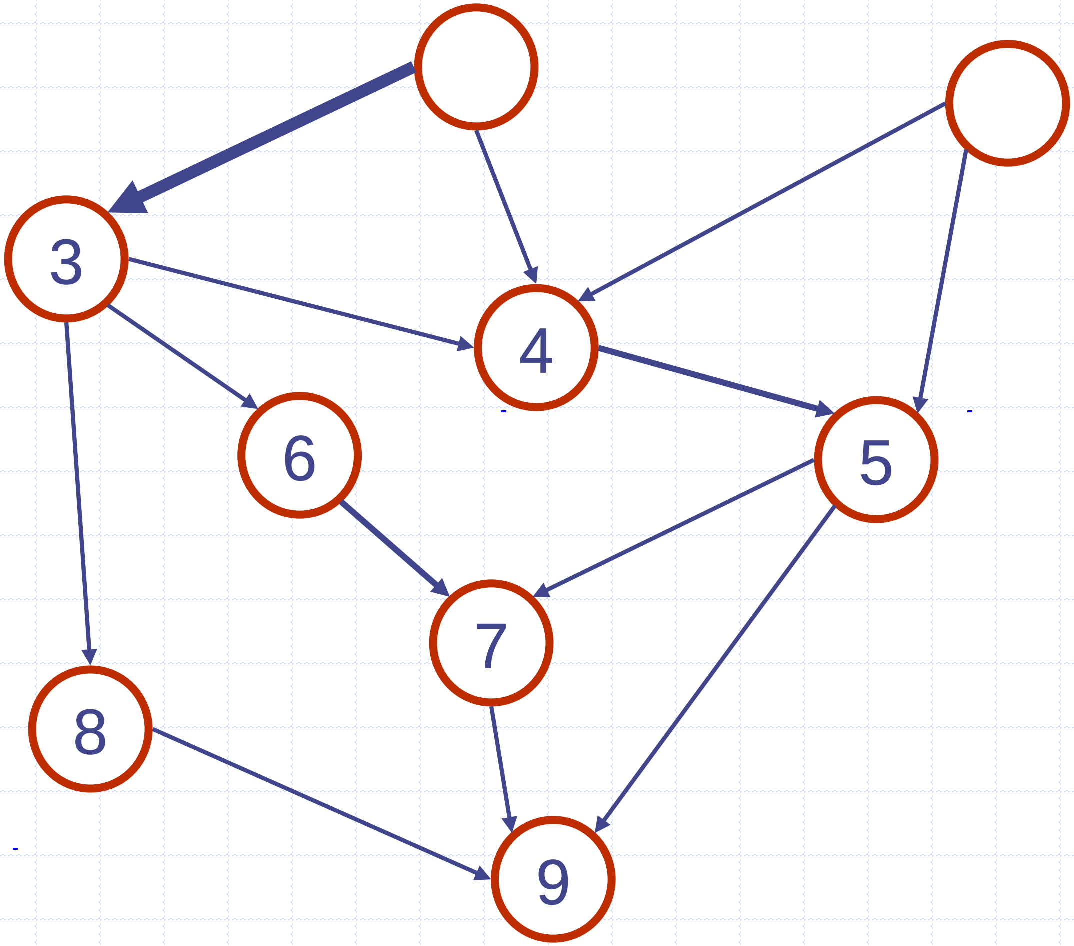
Topological Sorting Example



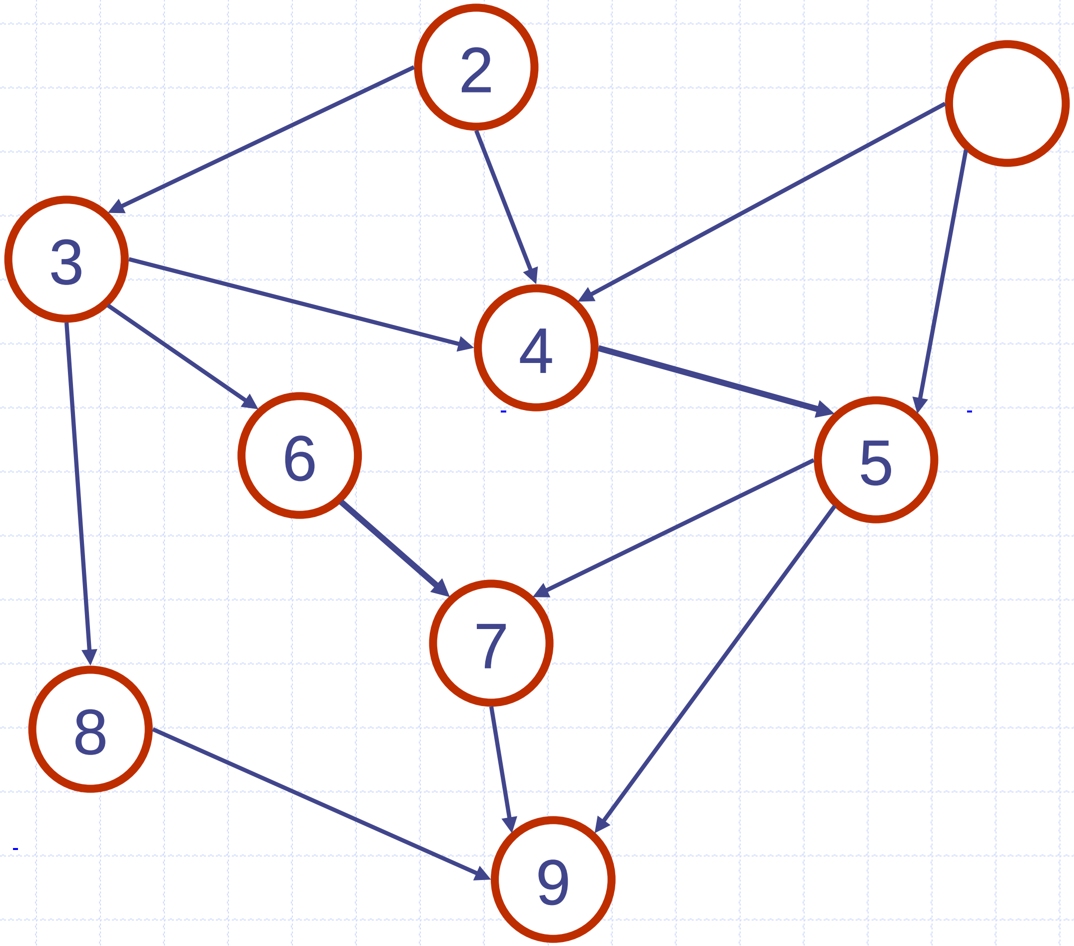
Topological Sorting Example



Topological Sorting Example



Topological Sorting Example



Topological Sorting Example