

# **INFO1105/1905**

## **Data Structures**

### **Week 8: Graphs (rest)**

see textbook section 14.4, 14.5, 14.6

Professor Alan Fekete

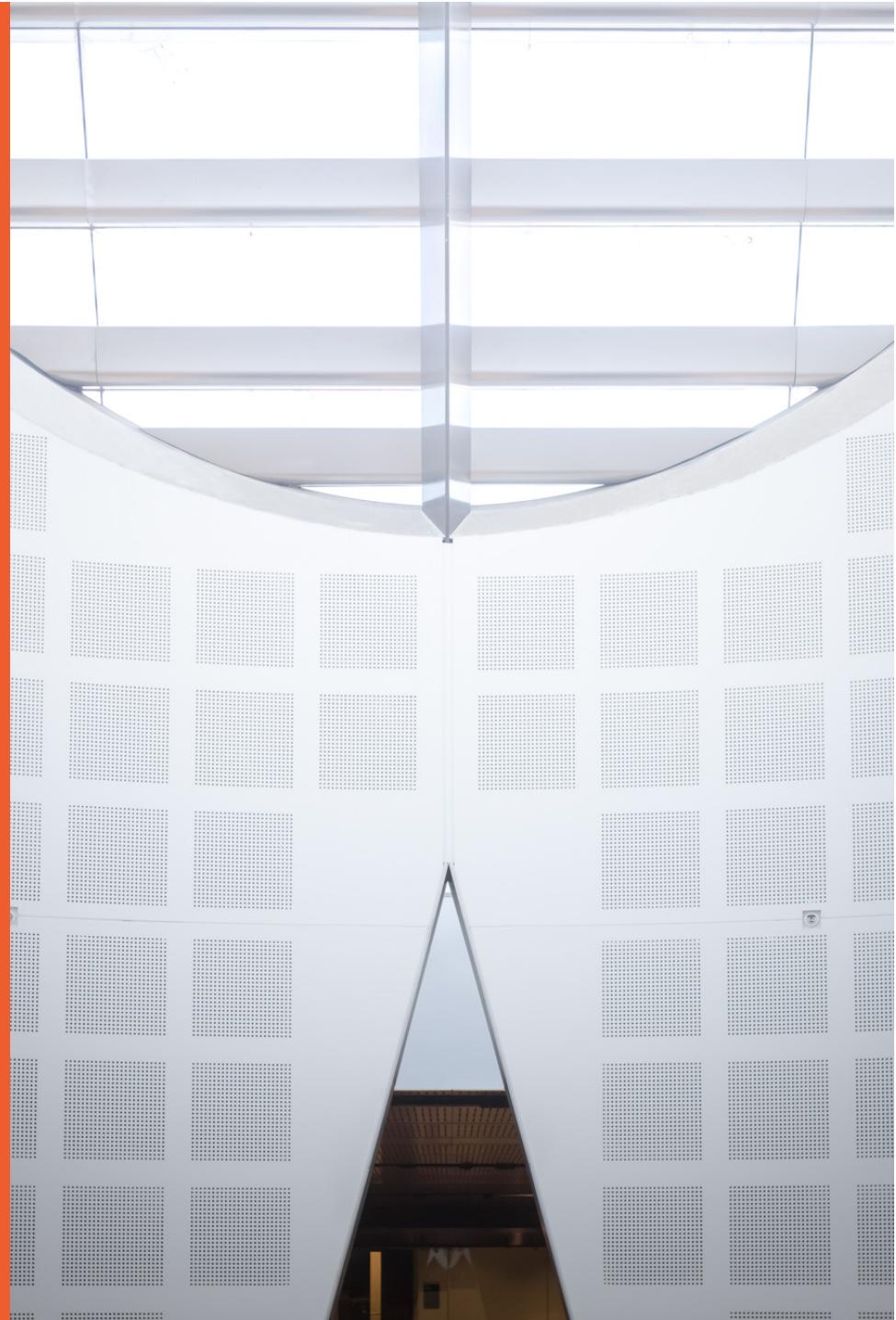
Dr John Stavrakakis

School of Information Technologies

using material from the textbook  
and A/Prof Kalina Yacef, Dr Taso Viglas



THE UNIVERSITY OF  
**SYDNEY**



# Copyright warning

**COMMONWEALTH OF AUSTRALIA**

**Copyright Regulations 1969**

**WARNING**

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

**Do not remove this notice.**

- These slides contain material from the textbook (Goodrich, Tamassia & Goldwasser)
  - Data structures and algorithms in Java (5<sup>th</sup> & 6<sup>th</sup> edition)
- With modifications and additions from the University of Sydney
- The slides are a guide or overview of some big ideas
  - Students are responsible for knowing what is in the referenced sections of the textbook, not just what is in the slides

## Reminder! Asst 1

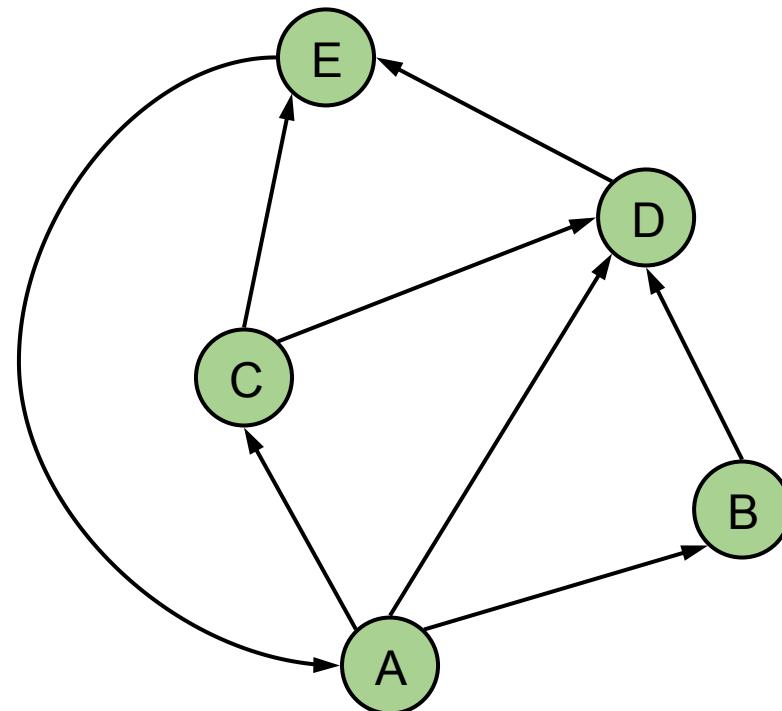
- Asst 1 is due 5pm Friday Sept 16 (this week)
- Two aspects to submit:
  - Report (in pdf, not hand-written), submit via Turnitin link on eLearning site
  - Code (including Junit tests), submit via edstem
- Make sure that you follow exactly the interfaces provided
- Important advice: start by ignoring “distinction” criteria, and focus on reaching a pass
  - in particular, don’t worry about efficiency or corner cases; keep it simple
  - get that to pass the public tests, and write the report, and submit it all
  - after that, you can try to make changes (fix failing hidden tests, use different data structure for efficiency, do more comments, refactor, write justification for costs, etc) to improve the grade, and resubmit frequently

# Outline

- Directed graphs
  - Directed DFS
  - Strong connectivity
  - Transitive closure (Floyd-Warshall algorithm)
  - Topological ordering
- Weighted graphs
  - Shortest paths (Dijkstra's algorithm)

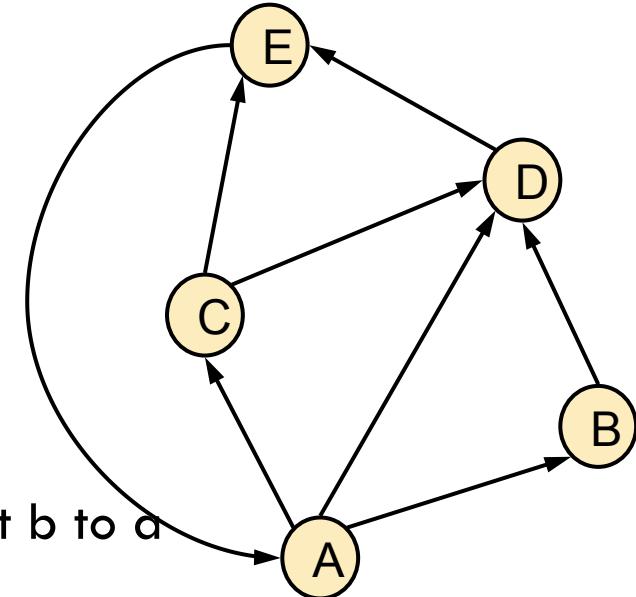
# Digraphs

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



## Digraph Properties

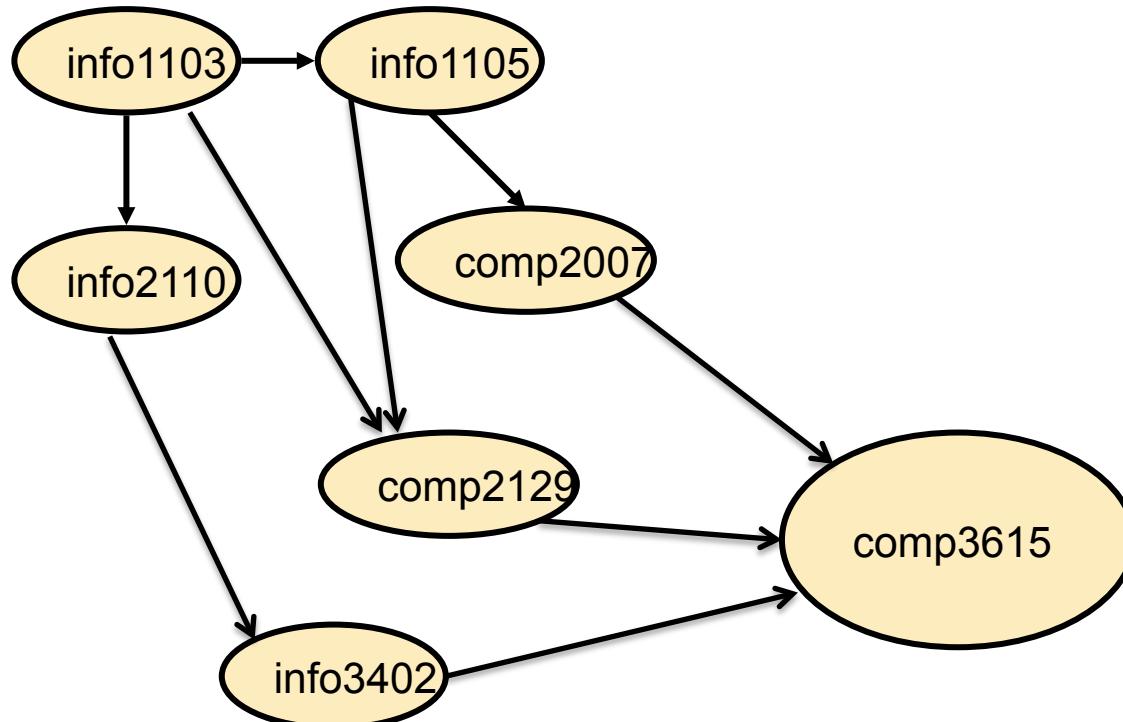
- A graph  $G=(V,E)$  such that
  - Each edge goes in **one direction**:
    - Edge  $(a,b)$  goes from  $a$  to  $b$ , but not  $b$  to  $a$
- If  $G$  is simple,  $m \leq n \cdot (n - 1)$
- If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of incoming edges and outgoing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



Recall  $m = \text{number of edges}$ ,  $n = \text{number of nodes}$

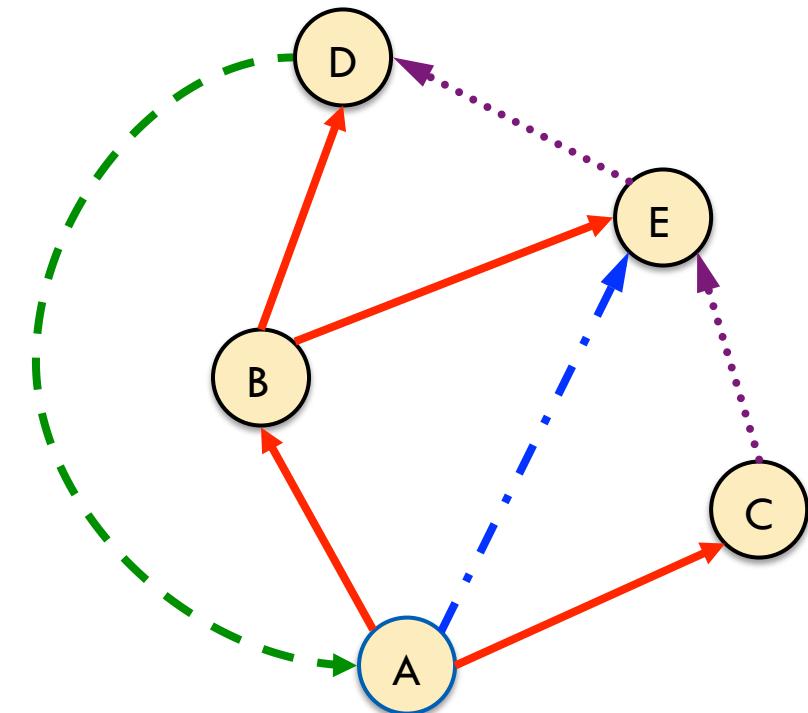
## Digraph Application

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



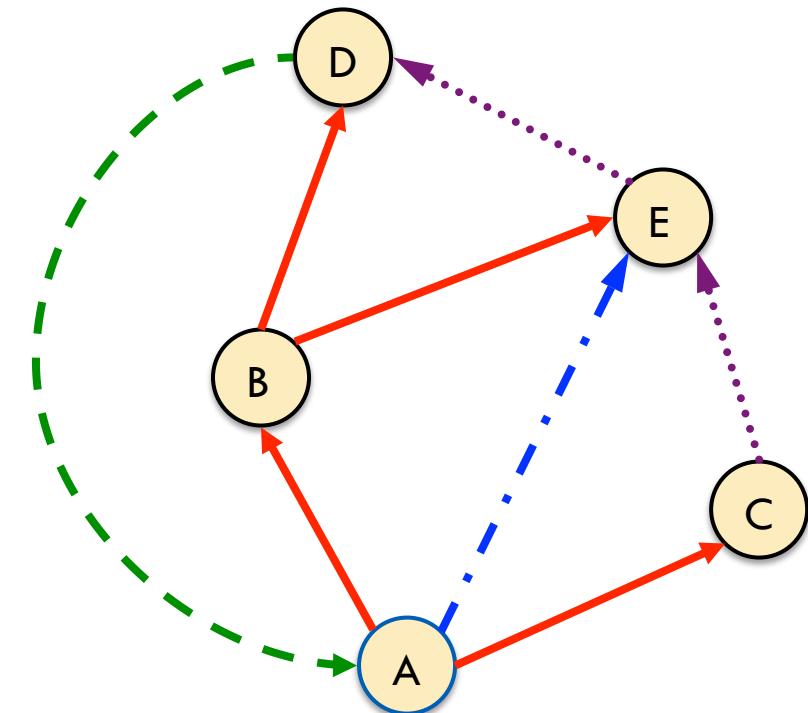
## Directed DFS

- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction
- A directed DFS starting at a vertex  $s$  determines the vertices **reachable** from  $s$



## Directed DFS

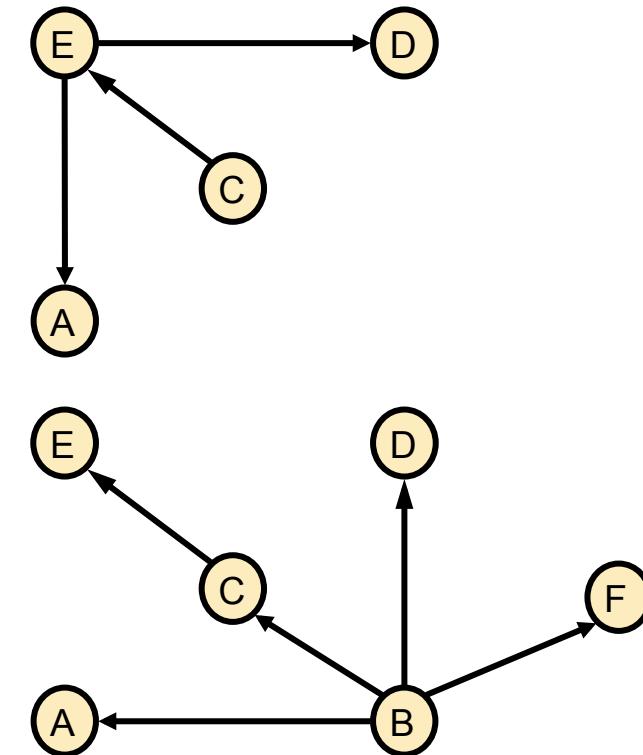
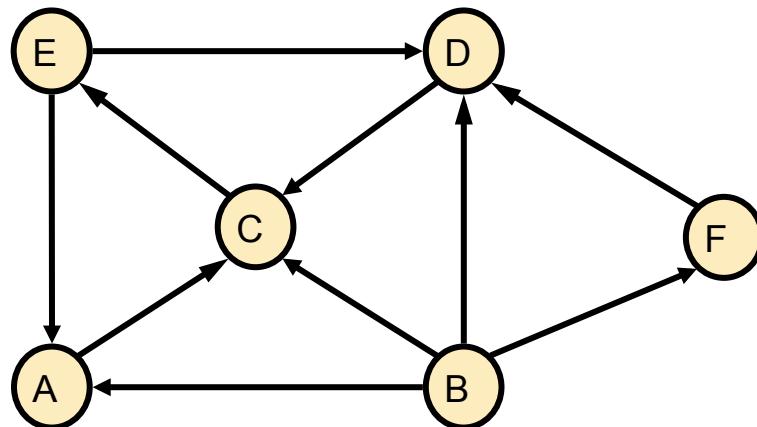
- In the directed DFS algorithm, we have four types of edges
  - discovery edges
    - These are the DFS tree edges
  - back edges
    - Connect a vertex to an ancestor in the DFS tree
  - forward edges
    - Connect a vertex to a descendant in the DFS tree
  - cross edges
    - Connect to vertex that is not an ancestor/descendant
- A directed DFS starting at a vertex  $s$  determines the vertices **reachable** from  $s$



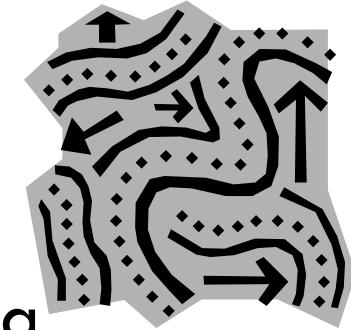
## Reachability



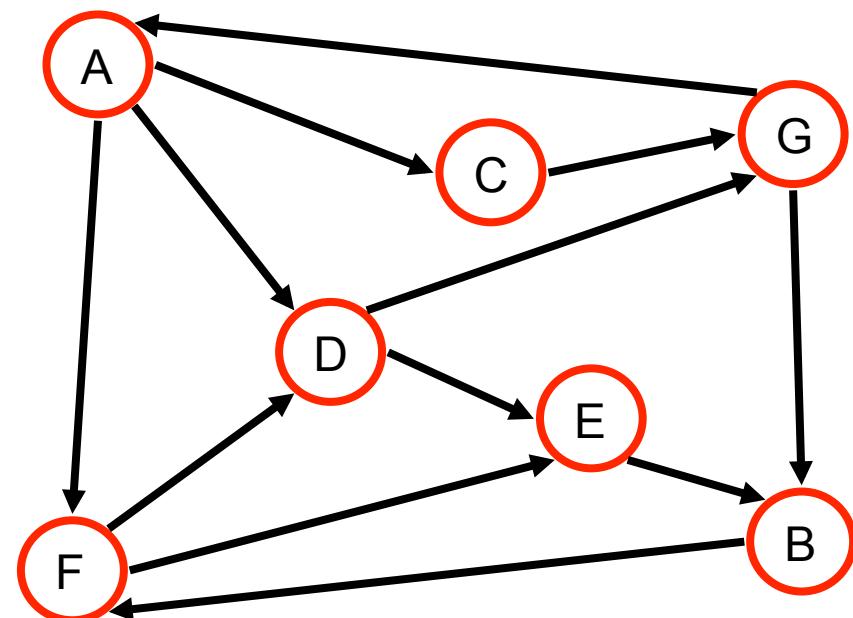
- A reaches B if there is a directed path from A to B
- **DFS tree** rooted at v: vertices reachable from v via directed paths



## Strong Connectivity



- **Strongly connected graph:** each vertex has a directed path to every other vertex



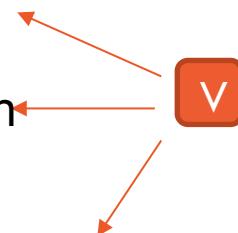
## Strong connectivity by DFS

- Option 1 (naïve) : Run DFS from each vertex  $u$ , and check that each vertex is reached
- Runtime?
  - $n$  DFS runs, so  $O(n.(n+m))$
- There is a faster algorithm

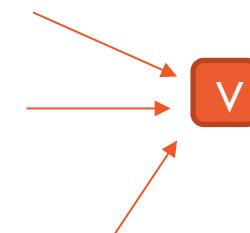
# Strong Connectivity Algorithm

A better way:

DFS from v:  
Testing if v can reach everyone



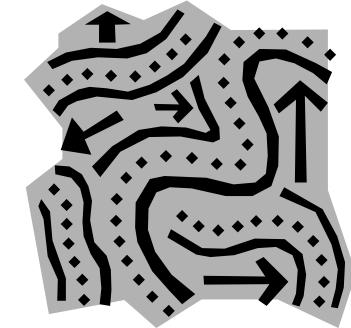
Reverse edge DFS from v:  
Testing if v can be reached from everyone



If v can reach everyone and everyone can reach v then...

...every node x can reach any other node y.

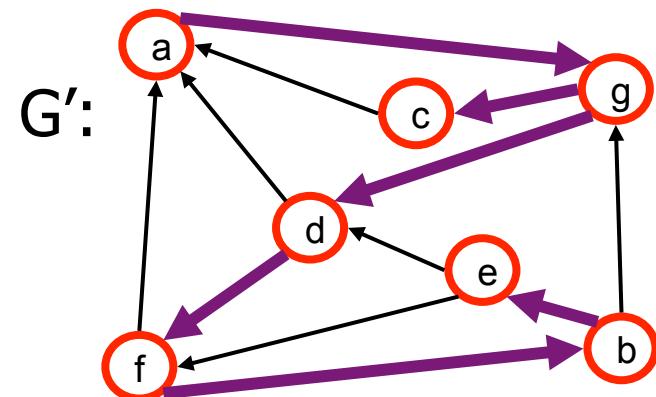
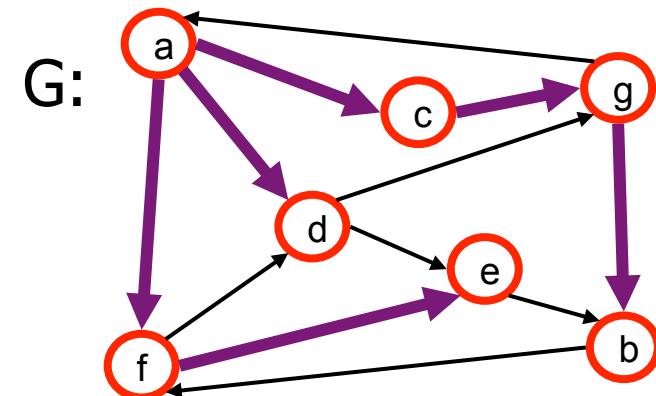
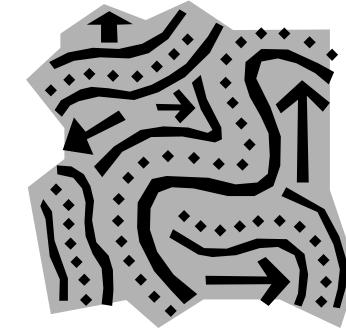
Just go through v:  
x has a path to v (since everyone can reach v)  
and  
v has a path to y (since v can reach everyone)



# Strong Connectivity Algorithm

## Option 2: With only 2 DFS

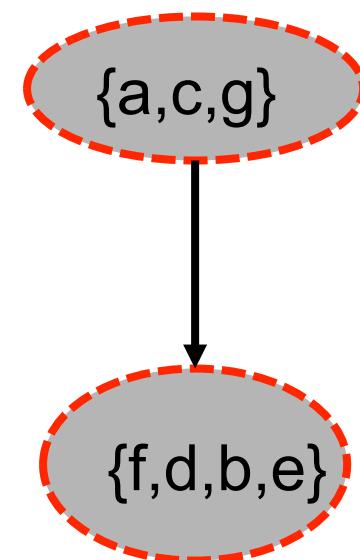
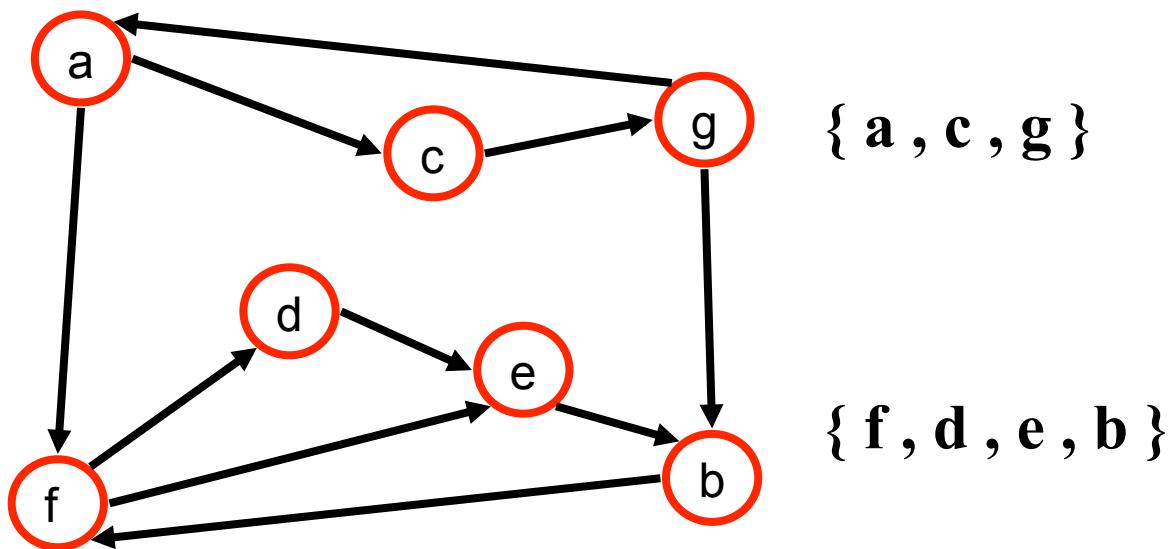
- Pick a vertex  $v$  in  $G$
- Perform a DFS from  $v$  in  $G$ 
  - If there's a  $w$  not visited, return "no"
- Let  $G'$  be  $G$  with edges **reversed**
- Perform a DFS from  $v$  in  $G'$ 
  - If there's a  $w$  not visited, return "no"
  - Else, return "yes"
- Running time:  $O(n+m)$



# Strongly Connected Components

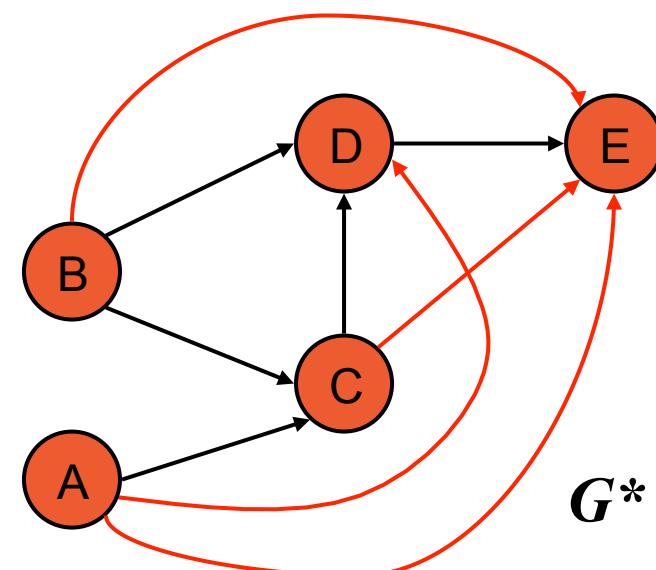
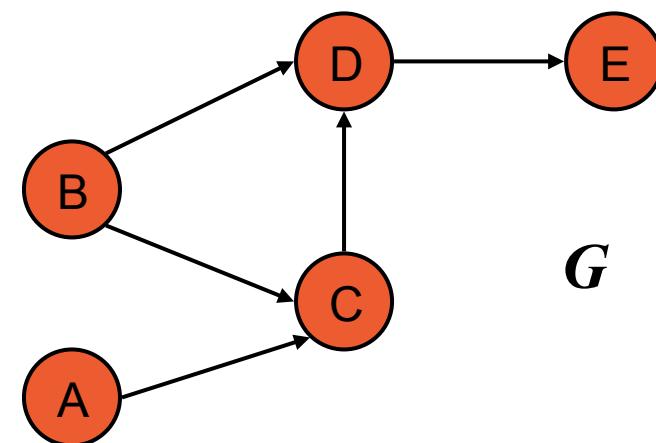


- Strongly connected component = Maximal subgraph such that each vertex can reach all other vertices in the subgraph
  - Maximal: no additional edges or vertices from G can be included in the subgraph without breaking its property of being strongly connected
- How many strongly connected components here?



## Transitive Closure

- For pre-computing all reachable vertices from each vertex
- 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
  - $O(n(n+m))$



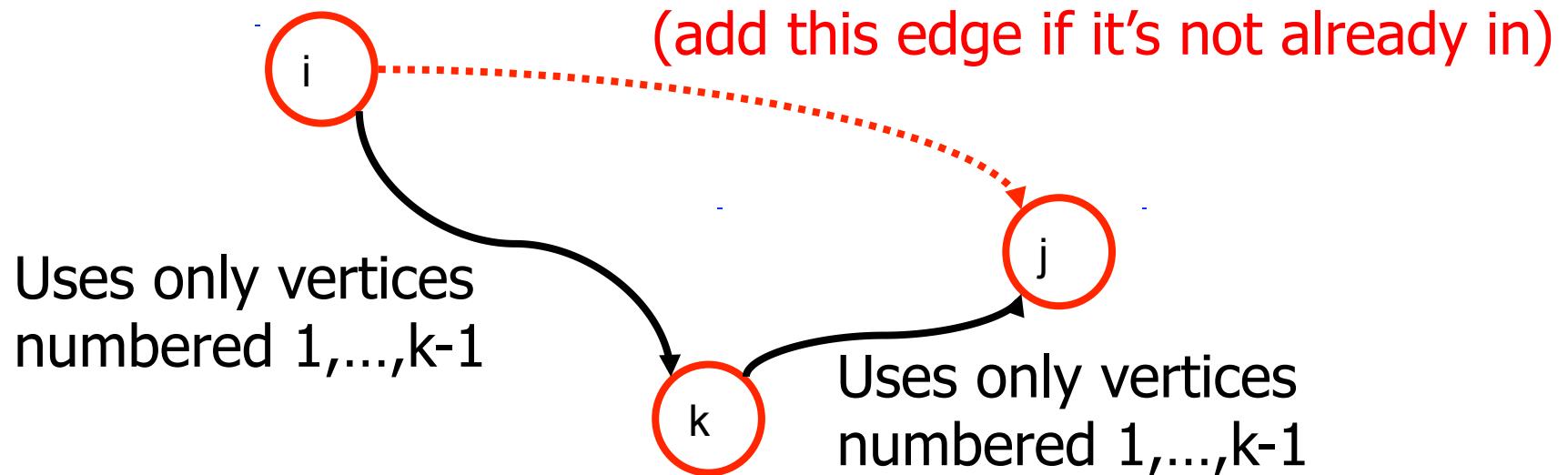
If there's a way to get from **A** to **B** and from **B** to **C**, then there's a way to get from **A** to **C**.

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



## Floyd-Warshall Transitive Closure

- Idea #1: Number the vertices 1, 2, ..., n (arbitrarily).
- Idea #2: Construct transitive closure digraph incrementally. At round  $k+1$ , consider paths that use only vertices numbered 1, 2, ..., 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  $G_n = G^*$
- 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**  $v \in G.vertices()$  {number the 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**  $G_{k-1}.areAdjacent(v_i, v_k)$  and

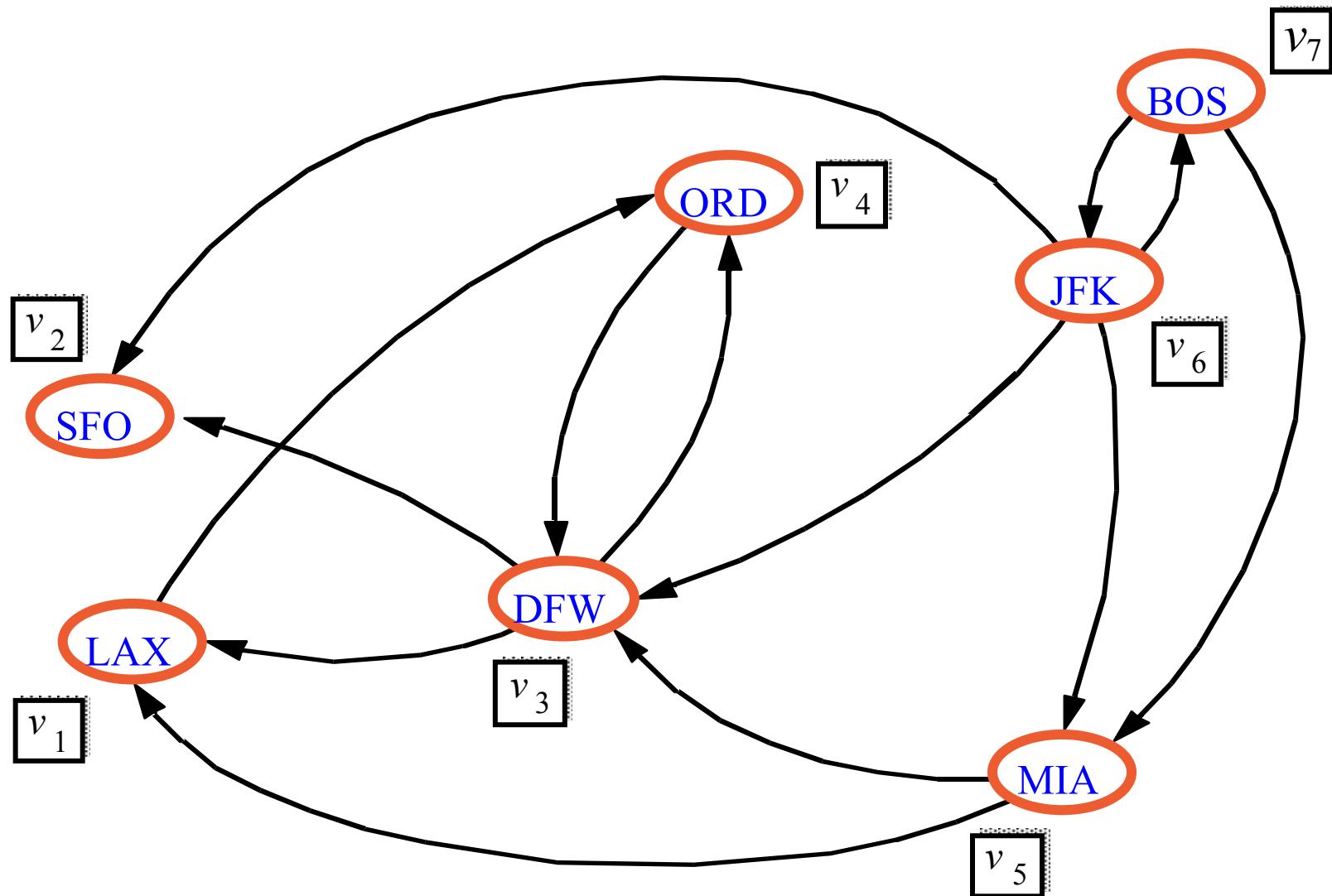
$G_{k-1}.areAdjacent(v_k, v_j)$

**if not**  $G_k.areAdjacent(v_i, v_j)$

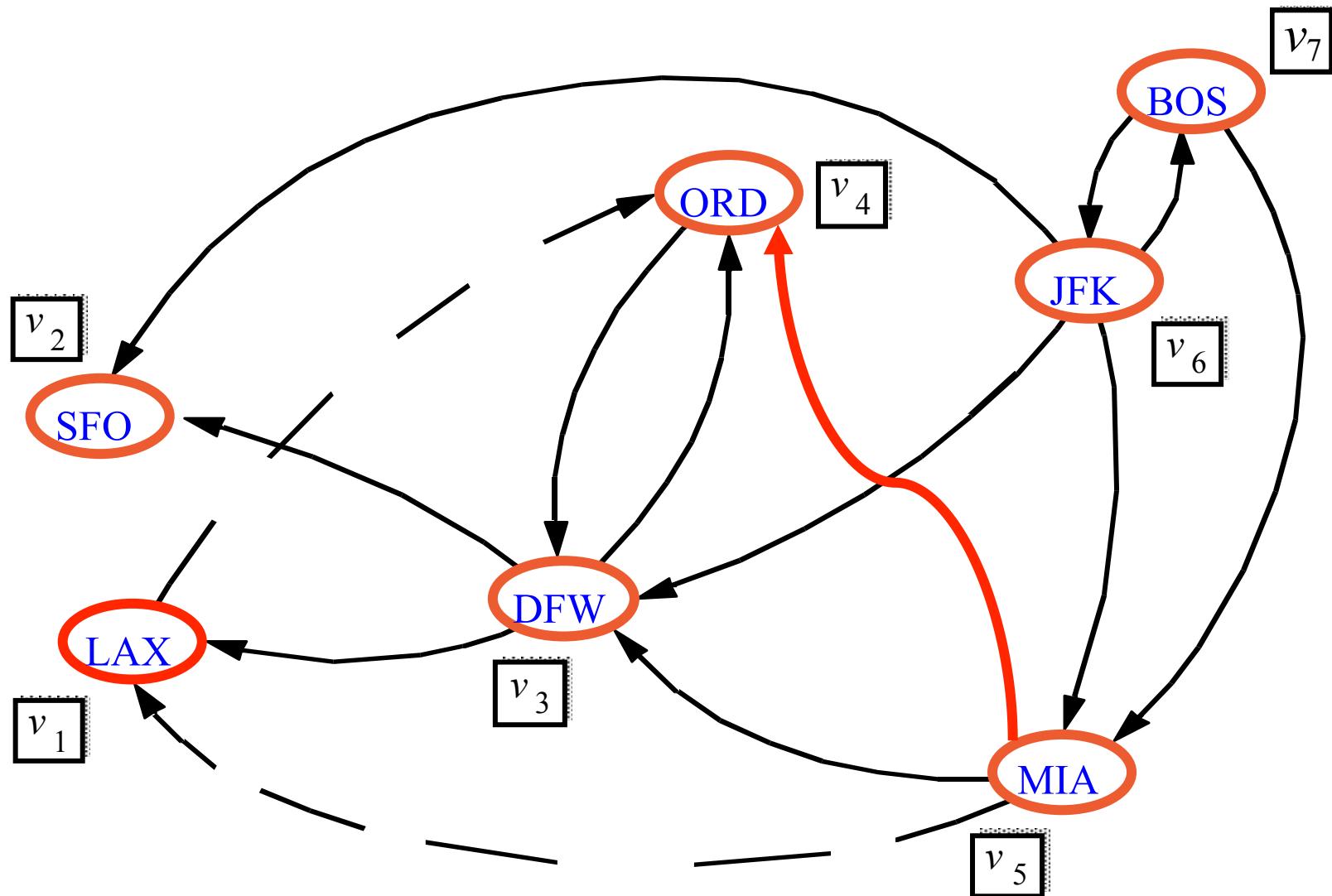
$G_k.insertDirectedEdge(v_i, v_j, k)$

**return**  $G_n$

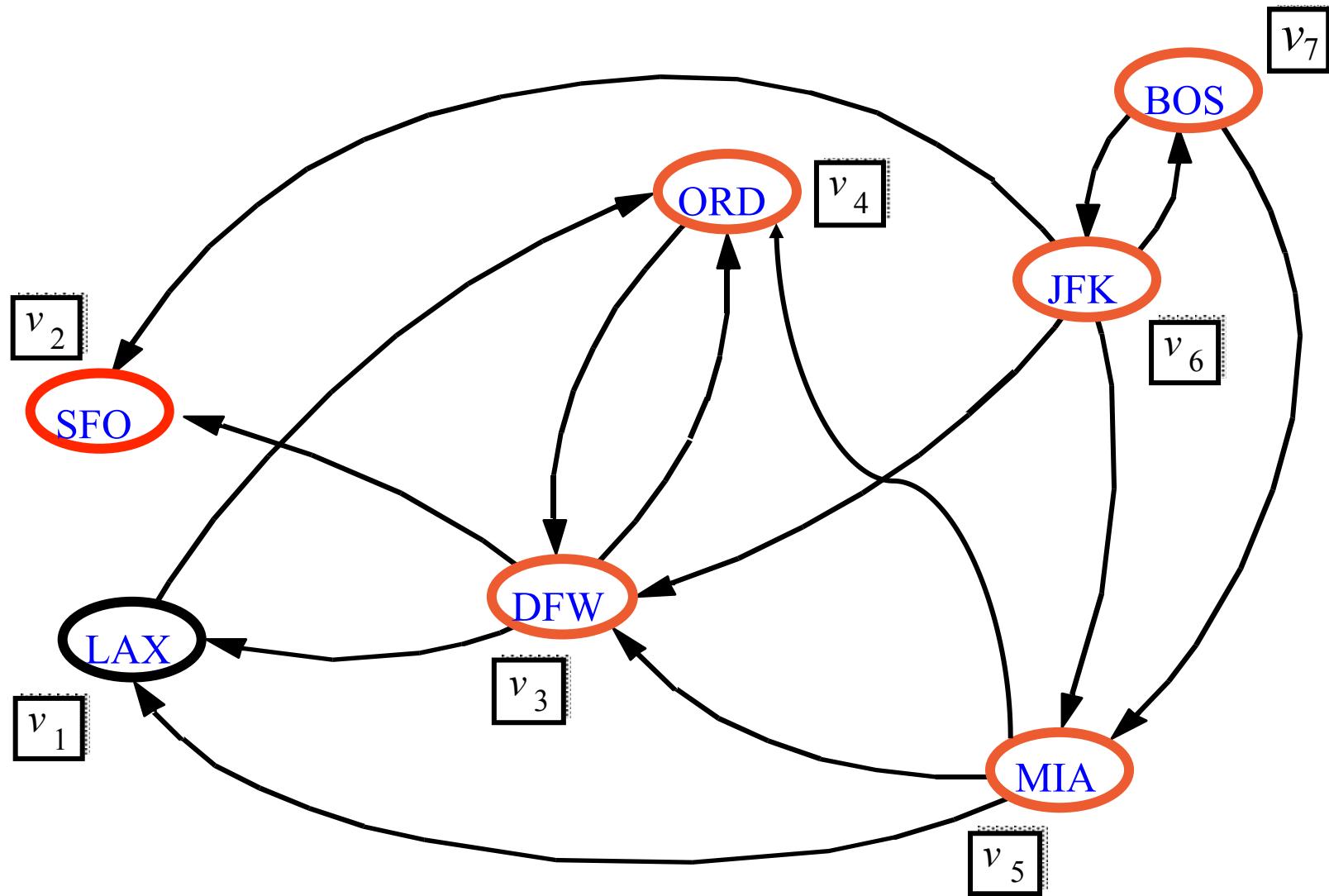
## Floyd-Warshall Example



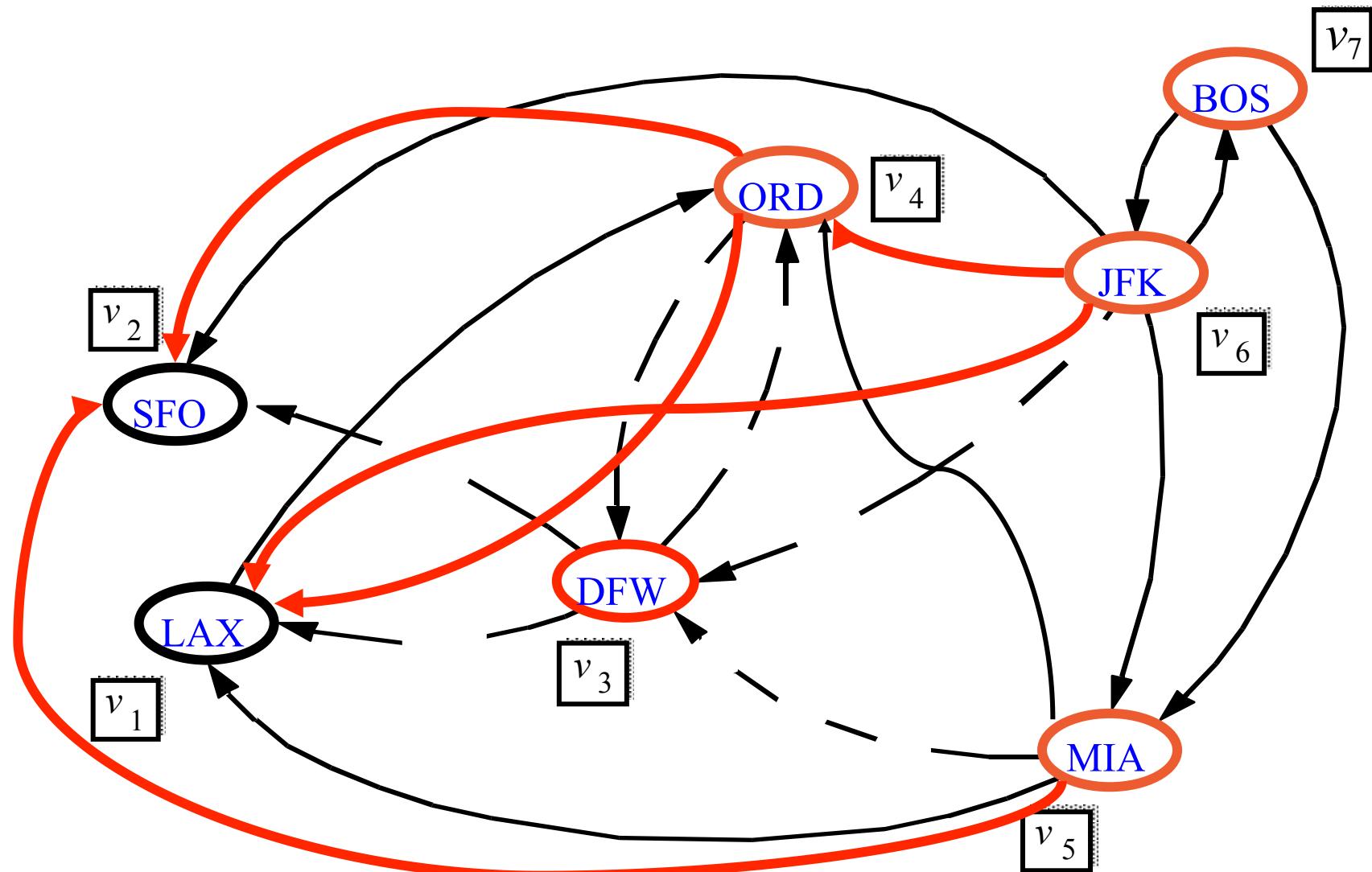
## Floyd-Warshall, Iteration 1



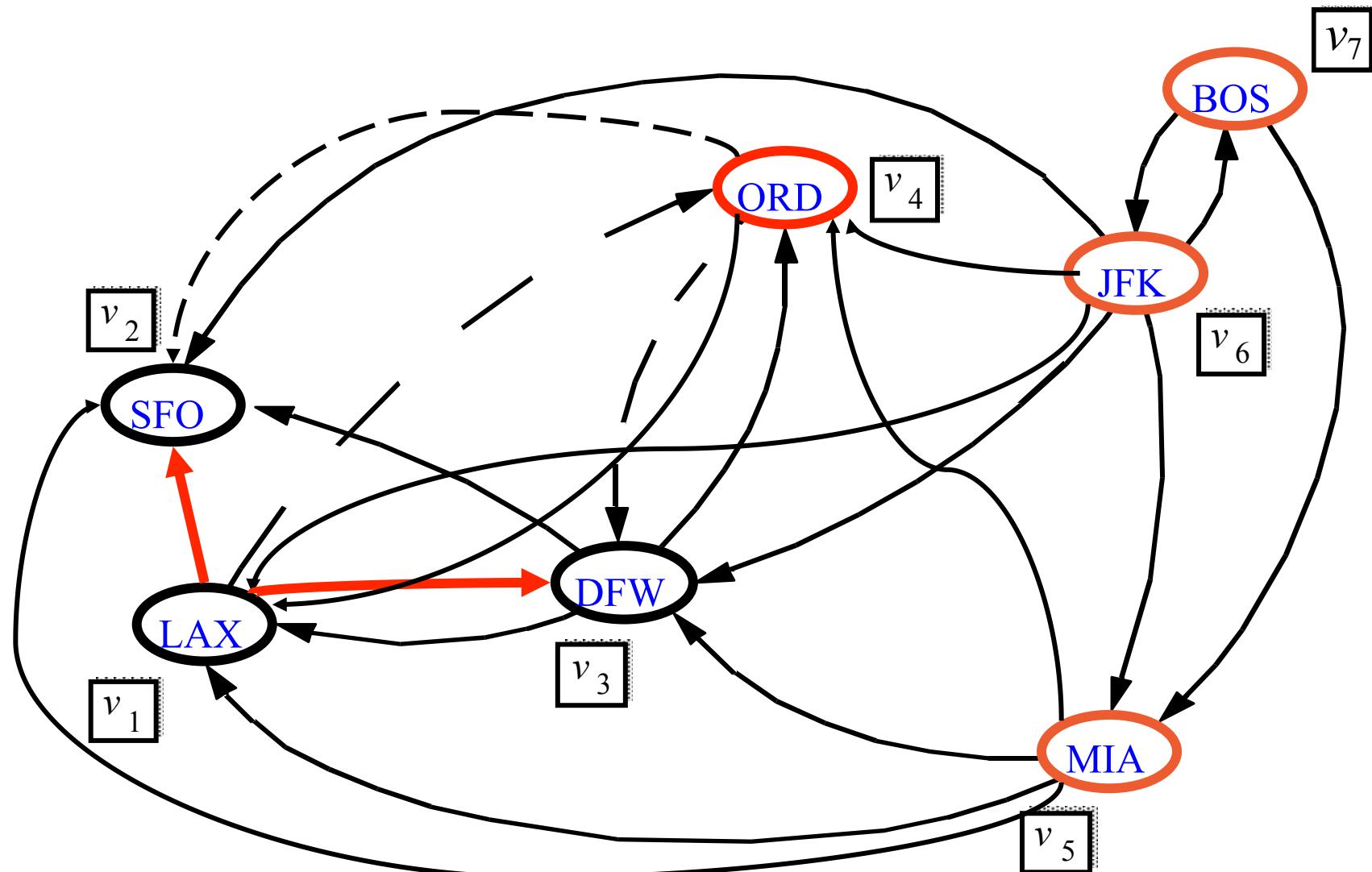
## Floyd-Warshall, Iteration 2



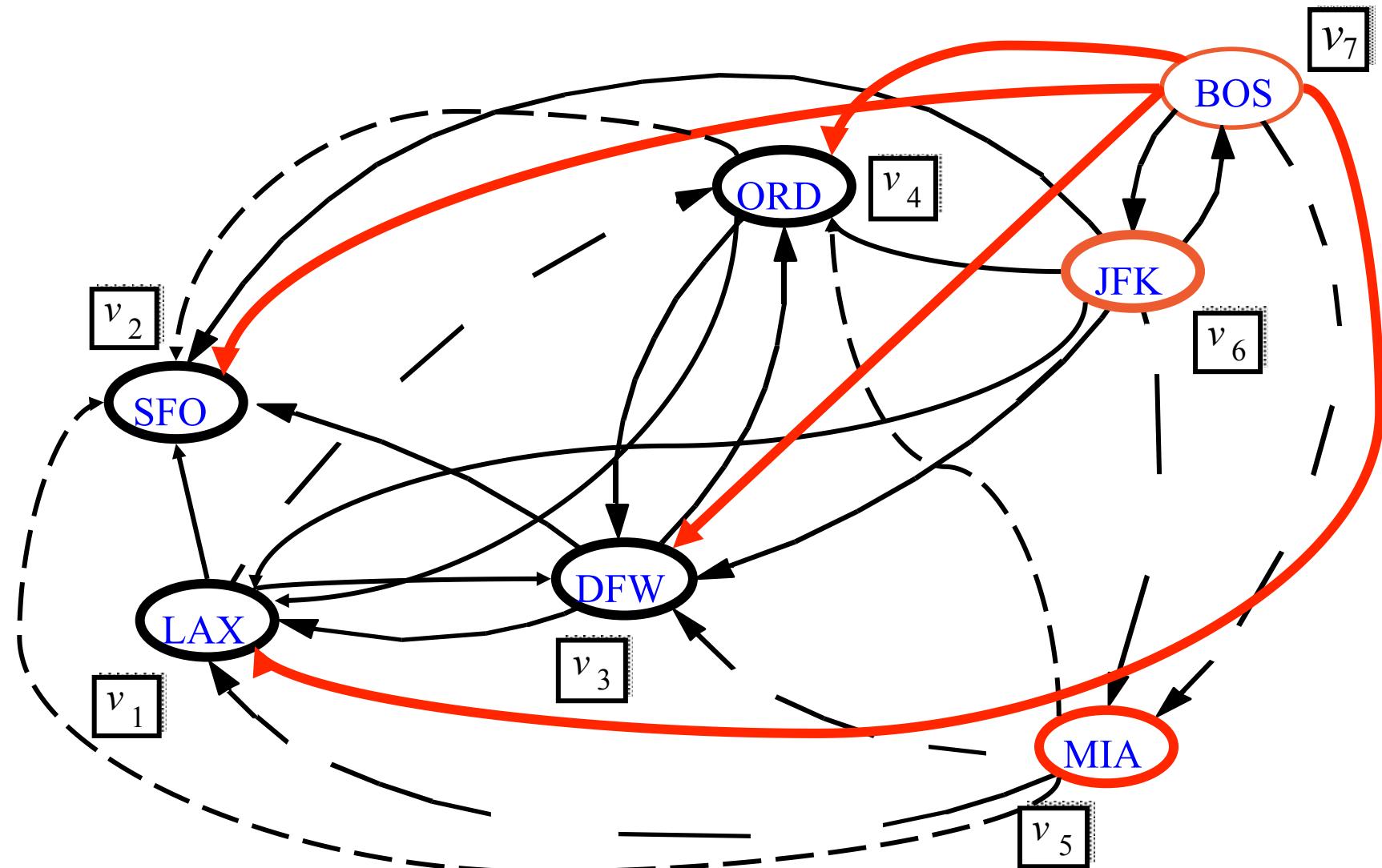
## Floyd-Warshall, Iteration 3



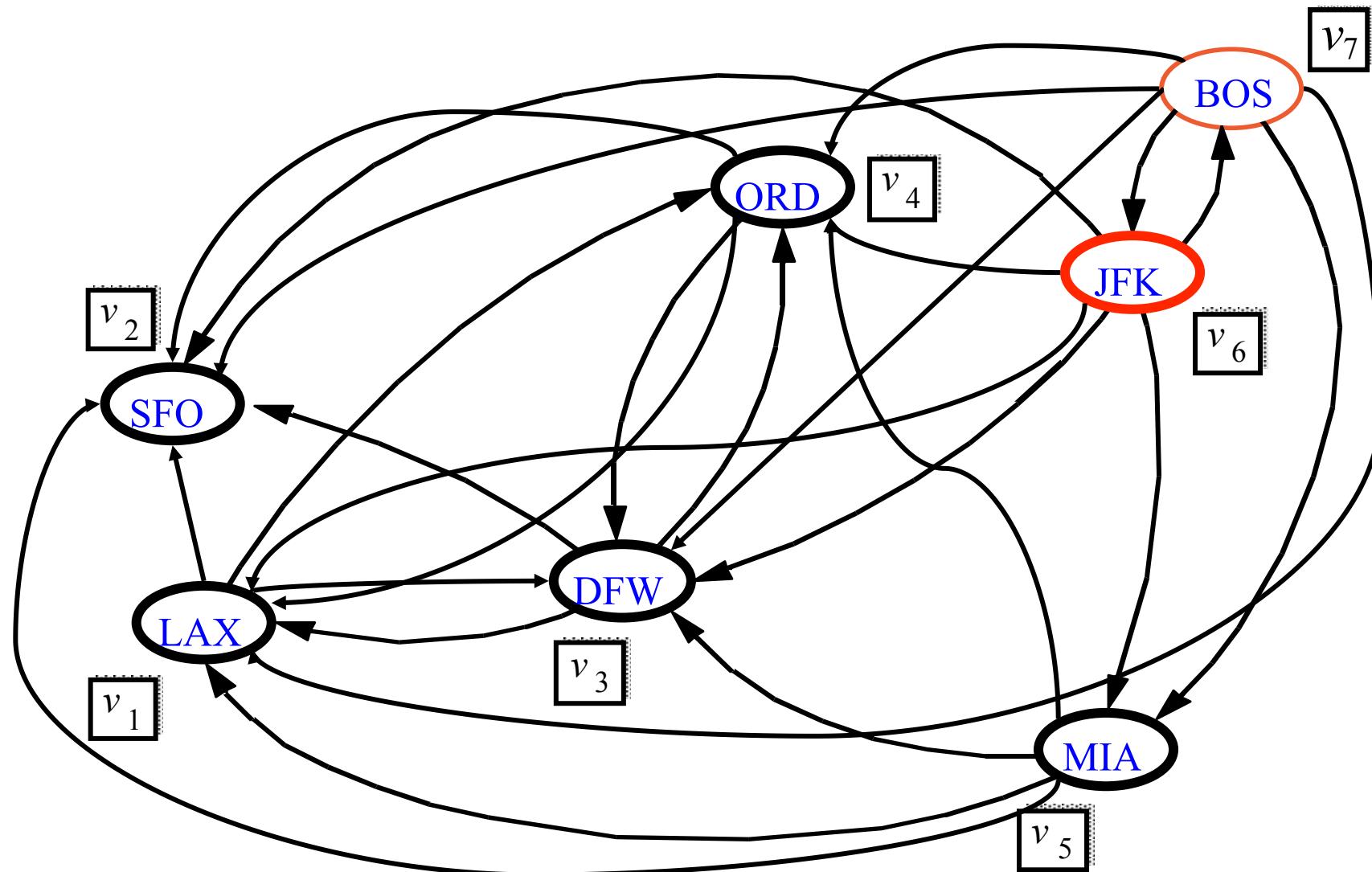
## Floyd-Warshall, Iteration 4



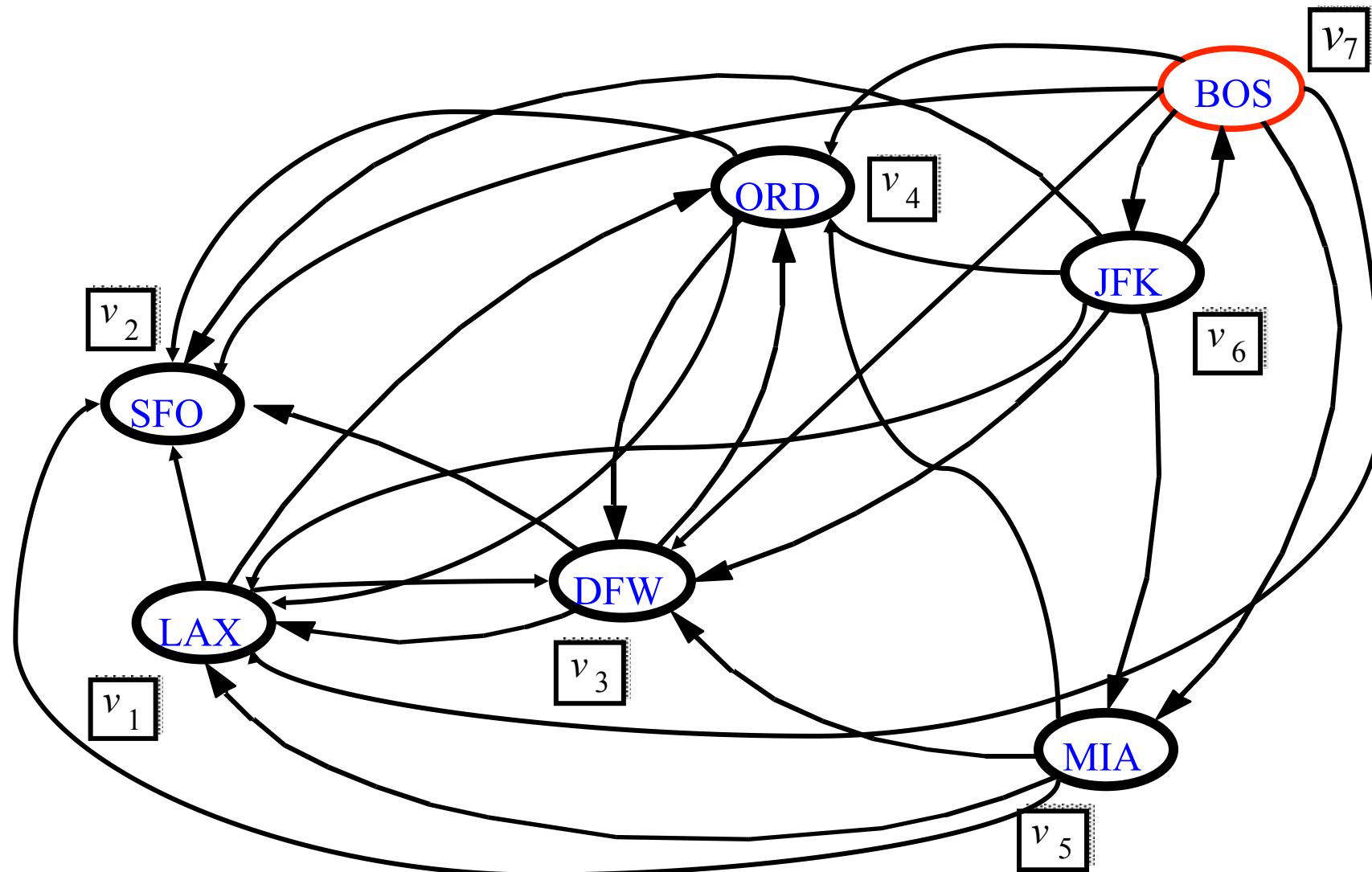
## Floyd-Warshall, Iteration 5



## Floyd-Warshall, Iteration 6



## Floyd-Warshall, Conclusion

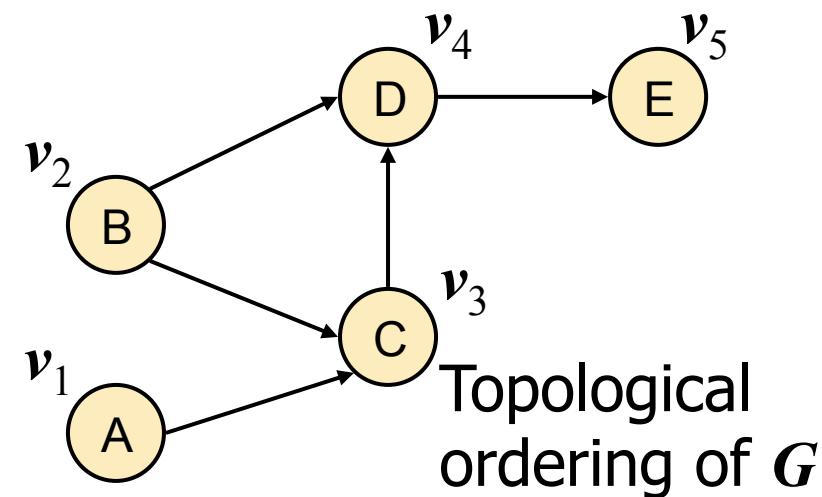
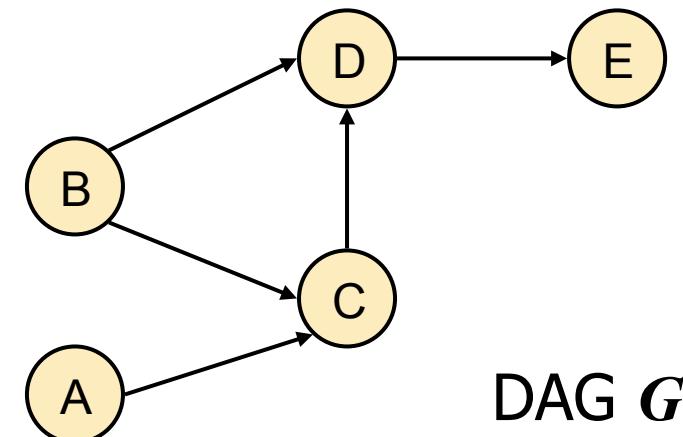


# Java Implementation

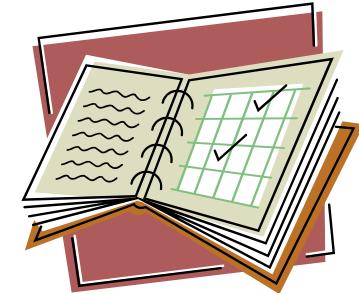
```
1  /** Converts graph g into its transitive closure. */
2  public static <V,E> void transitiveClosure(Graph<V,E> g) {
3      for (Vertex<V> k : g.vertices())
4          for (Vertex<V> i : g.vertices())
5              // verify that edge (i,k) exists in the partial closure
6              if (i != k && g.getEdge(i,k) != null)
7                  for (Vertex<V> j : g.vertices())
8                      // verify that edge (k,j) exists in the partial closure
9                      if (i != j && j != k && g.getEdge(k,j) != null)
10                         // if (i,j) not yet included, add it to the closure
11                         if (g.getEdge(i,j) == null)
12                             g.insertEdge(i, j, null);
13 }
```

## DAGs and Topological Ordering

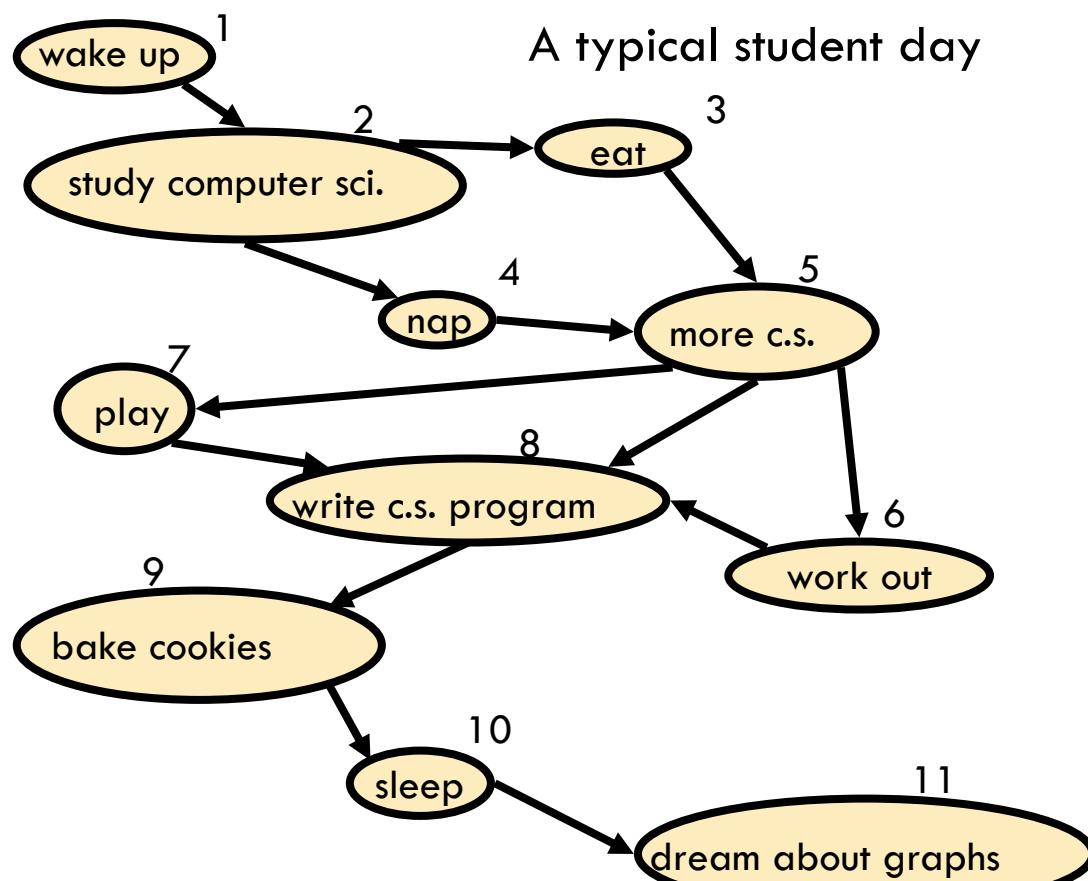
- A **directed acyclic graph (DAG)** is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering  $v_1, \dots, v_n$  of the vertices such that for every edge  $(v_i, v_j)$ , we have  $i < j$
- Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints
- **Theorem**  
A digraph has a topological ordering if and only if it is a DAG



# Topological Sorting



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



# Topological sorting

**Algorithm** TopologicalSort( $\vec{G}$ ):

**Input:** A digraph  $\vec{G}$  with  $n$  vertices.

**Output:** A topological ordering  $v_1, \dots, v_n$  of  $\vec{G}$ .

$S \leftarrow$  an initially empty stack.

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

    Let  $\text{incounter}(u)$  be the in-degree of  $u$ .

**if**  $\text{incounter}(u) = 0$  **then**

$S.\text{push}(u)$

$i \leftarrow 1$

**while**  $\text{!}S.\text{isEmpty}()$  **do**

$u \leftarrow S.\text{pop}()$

        Let  $u$  be vertex number  $i$  in the topological ordering.

$i \leftarrow i + 1$

**for all** outgoing edge  $(u, w)$  of  $u$  **do**

$\text{incounter}(w) \leftarrow \text{incounter}(w) - 1$

**if**  $\text{incounter}(w) = 0$  **then**

$S.\text{push}(w)$

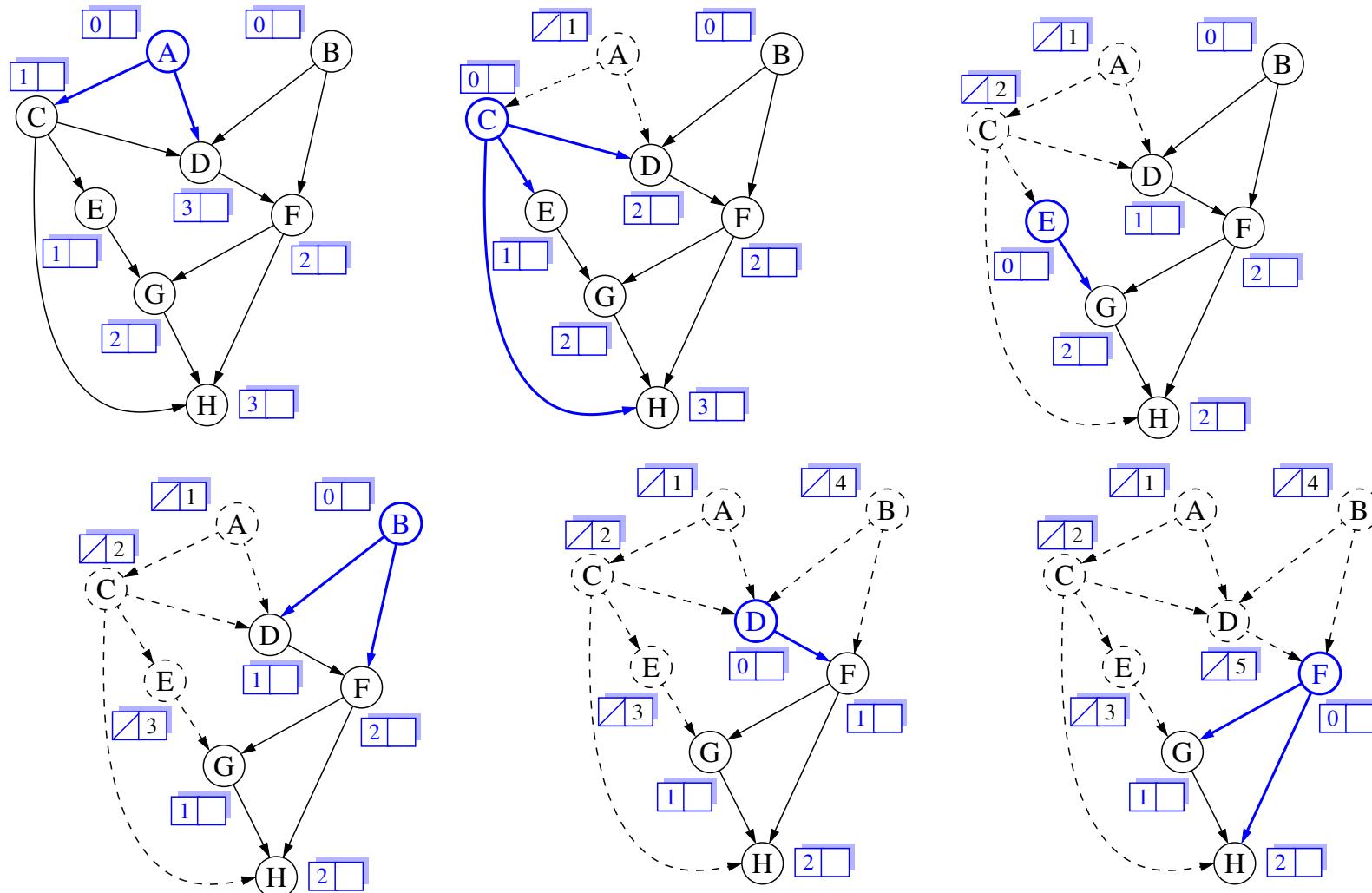
## Topological sorting analysis

- Uses an auxiliary stack and a map
  - this could take  $O(n)$  extra space
- Considers each vertex, and each adjacent outgoing edge (once)
  - Running time is  $O(n+m)$  provided graph has  $O(1)$  operations

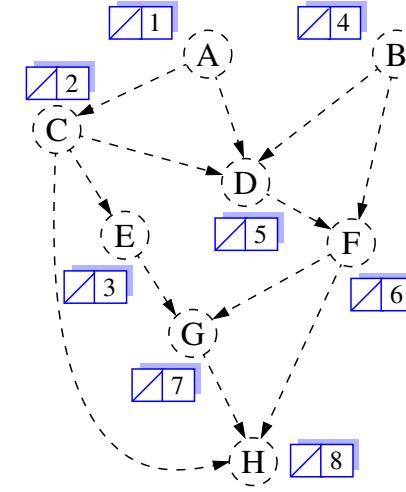
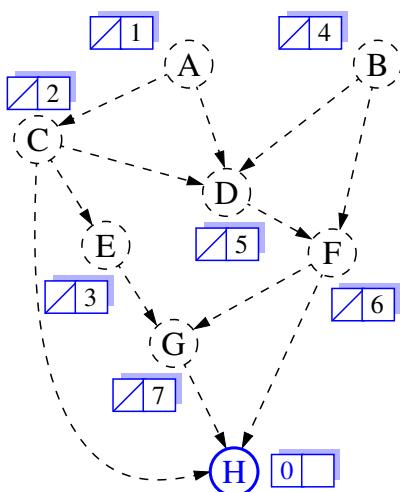
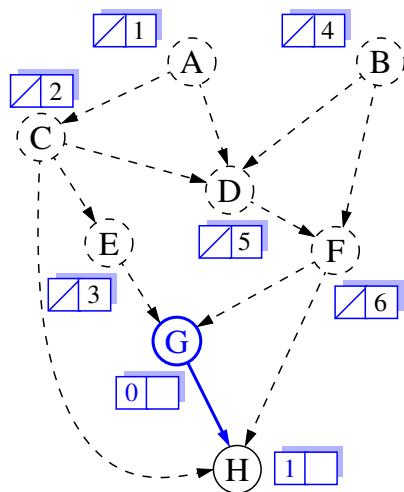
# Java Implementation

```
1  /** Returns a list of vertices of directed acyclic graph g in topological order. */
2  public static <V,E> PositionalList<Vertex<V>> topologicalSort(Graph<V,E> g) {
3      // list of vertices placed in topological order
4      PositionalList<Vertex<V>> topo = new LinkedPositionalList<>();
5      // container of vertices that have no remaining constraints
6      Stack<Vertex<V>> ready = new LinkedStack<>();
7      // map keeping track of remaining in-degree for each vertex
8      Map<Vertex<V>, Integer> inCount = new ProbeHashMap<>();
9      for (Vertex<V> u : g.vertices()) {
10          inCount.put(u, g.inDegree(u));           // initialize with actual in-degree
11          if (inCount.get(u) == 0)                 // if u has no incoming edges,
12              ready.push(u);                   // it is free of constraints
13      }
14      while (!ready.isEmpty()) {
15          Vertex<V> u = ready.pop();
16          topo.addLast(u);
17          for (Edge<E> e : g.outgoingEdges(u)) { // consider all outgoing neighbors of u
18              Vertex<V> v = g.opposite(u, e);
19              inCount.put(v, inCount.get(v) - 1);    // v has one less constraint without u
20              if (inCount.get(v) == 0)
21                  ready.push(v);
22      }
23  }
24  return topo;
25 }
```

# Topological sorting



## Topological sorting (cont)



## Non-examinable enrichment: A different algorithm for Topological Sorting

- Note: This algorithm is not the one in the book

```
Algorithm TopologicalSort2( $G$ )
 $H \leftarrow G$  // Temporary copy of  $G$ 
 $n \leftarrow G.numVertices()$ 
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 v2 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 }

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

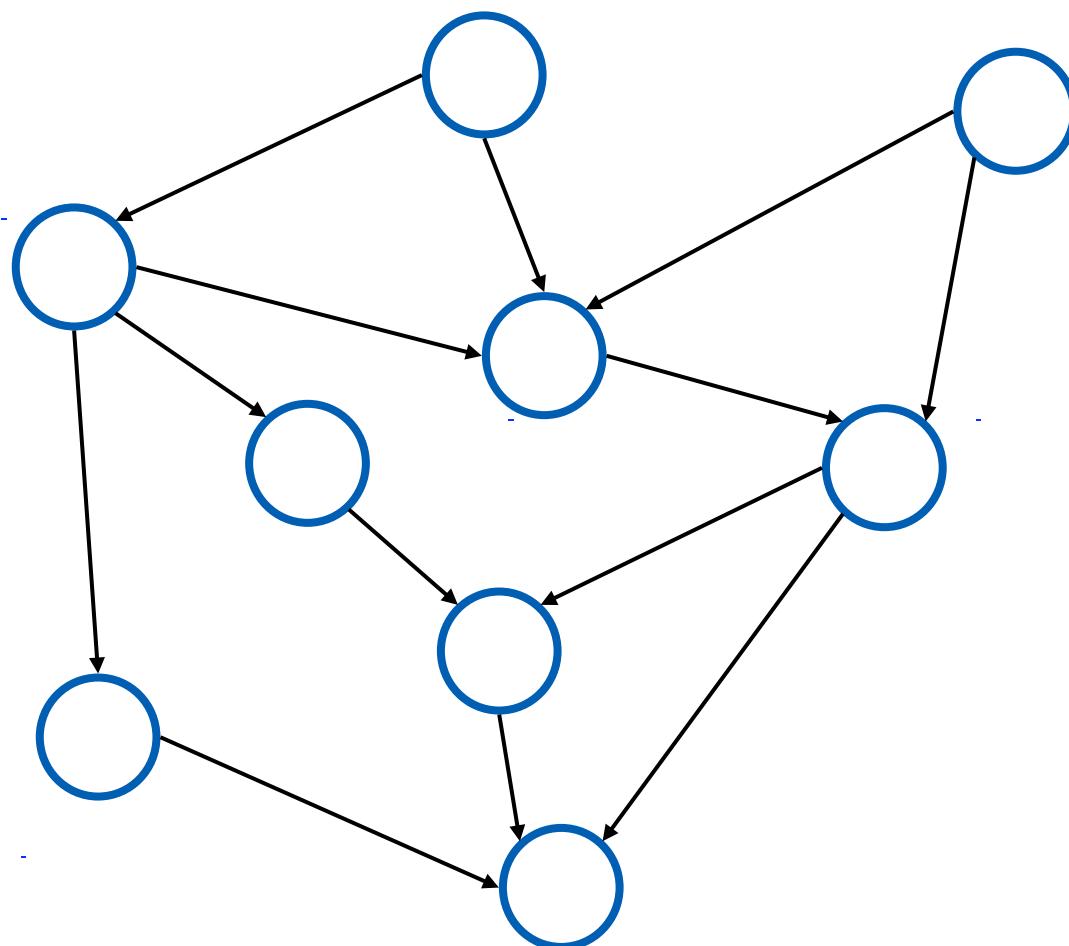
**else**

{  $e$  is a forward or cross edge }

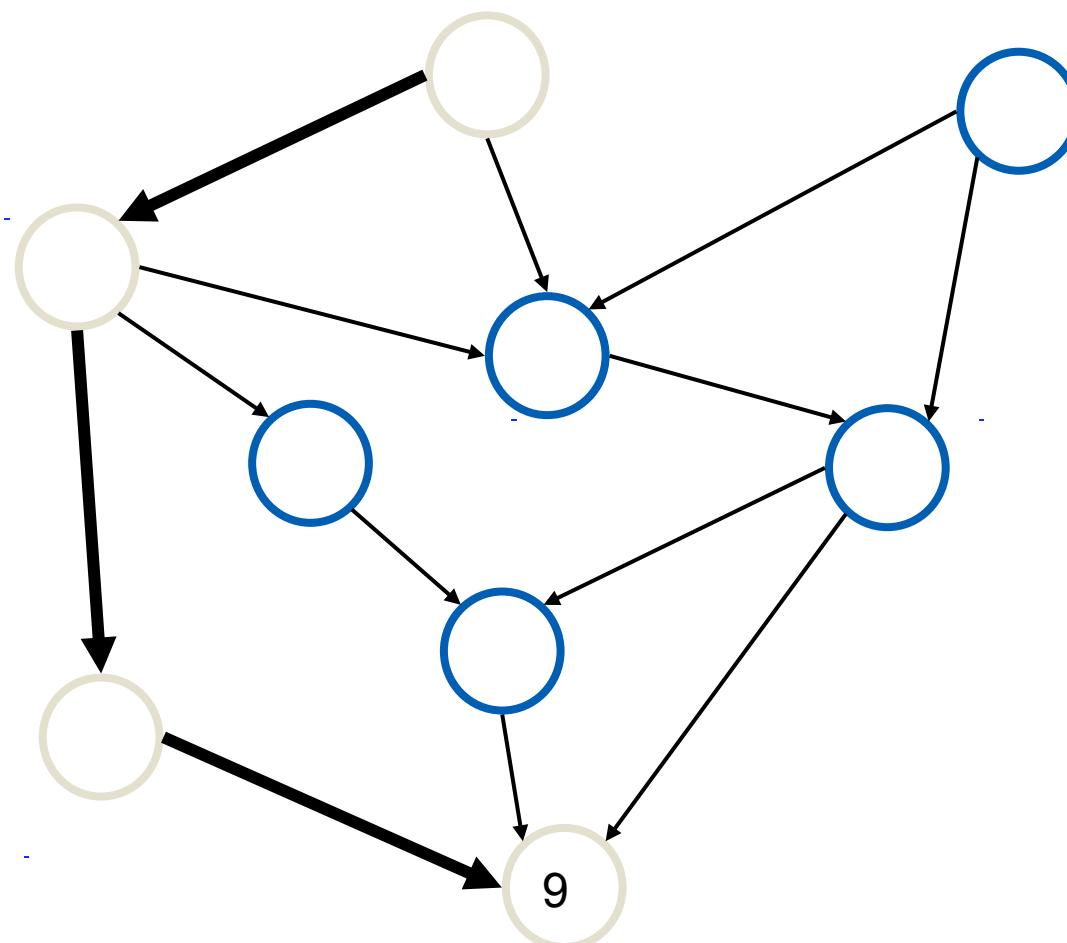
Label  $v$  with topological number  $n$

$n \leftarrow n - 1$

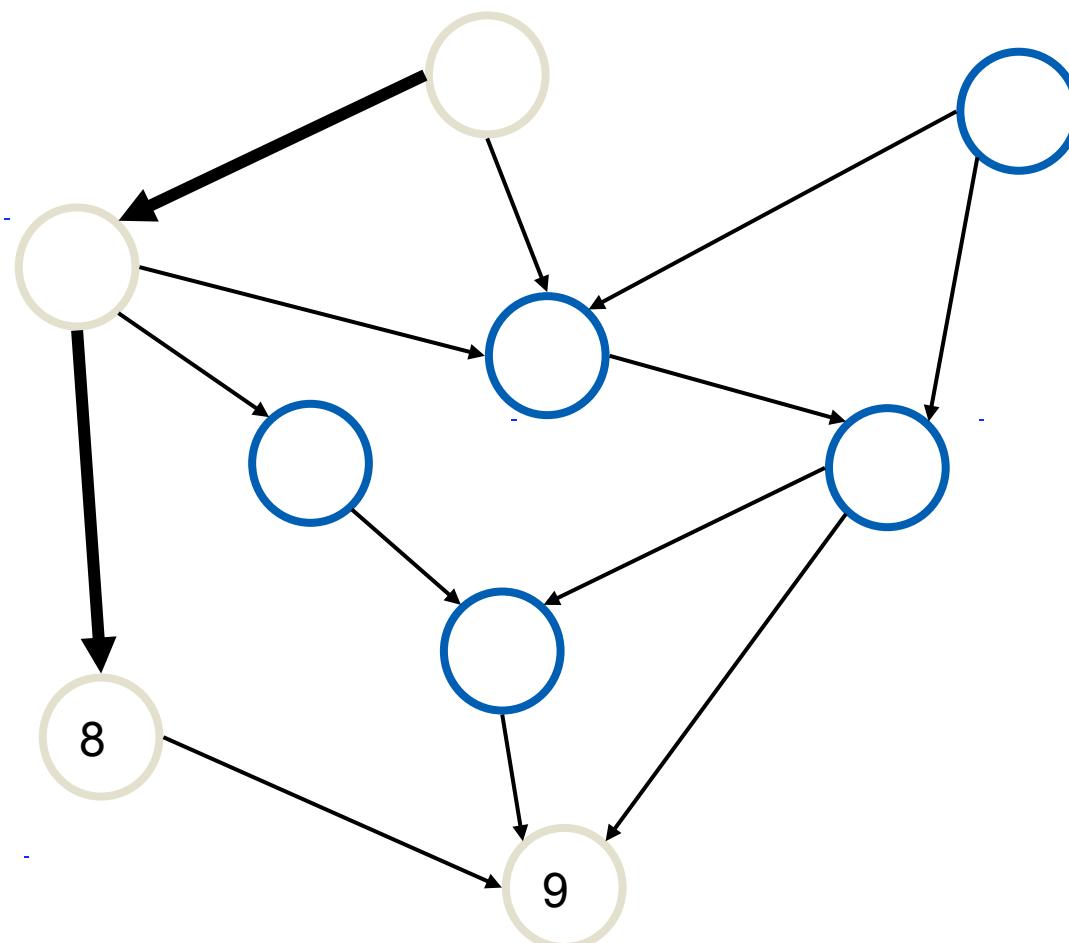
## Topological Sorting v2 Example



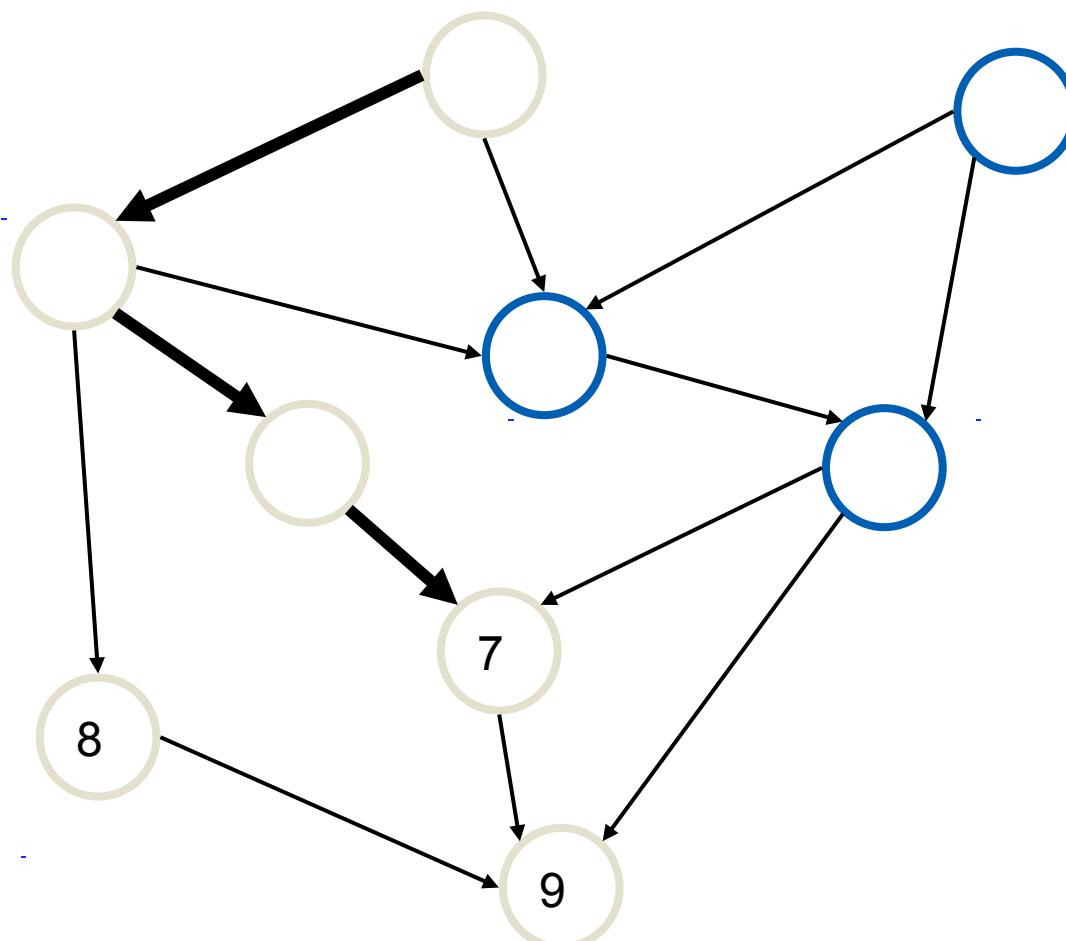
## Topological Sorting v2 Example



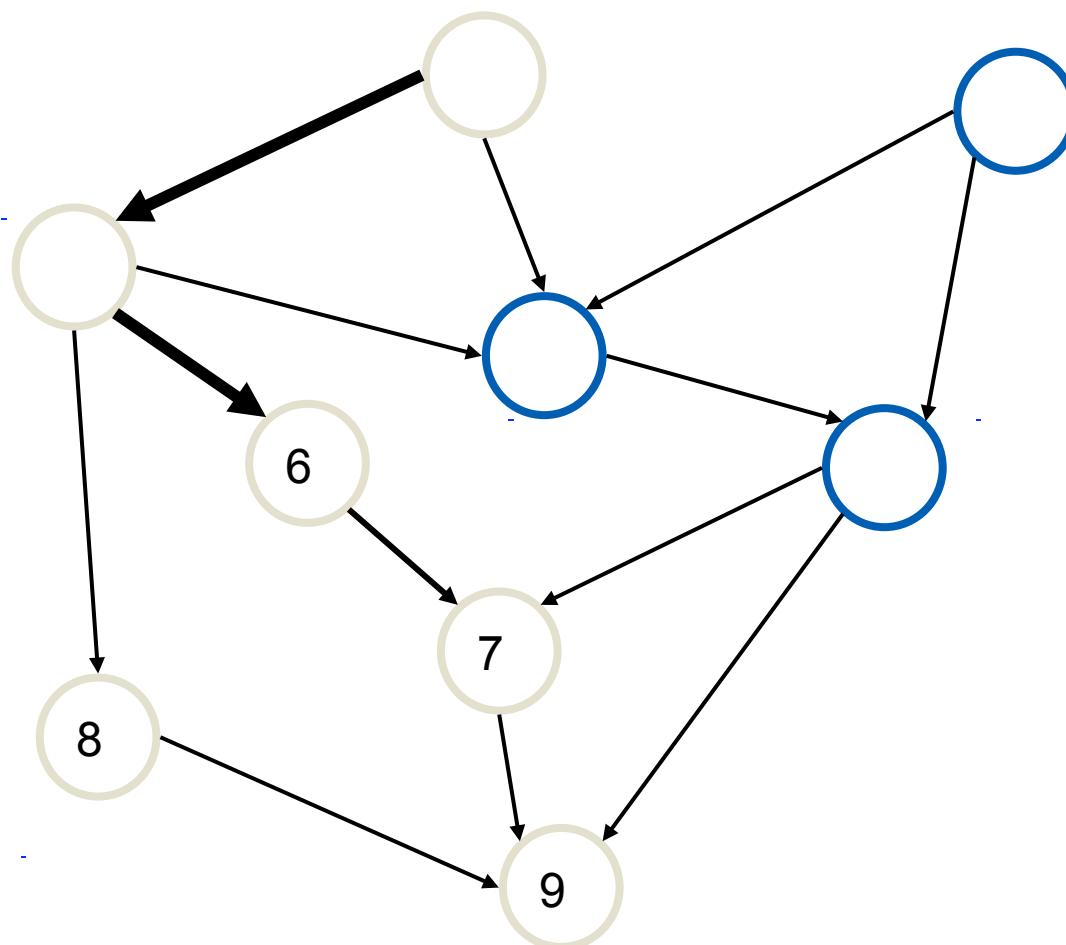
## Topological Sorting v2 Example



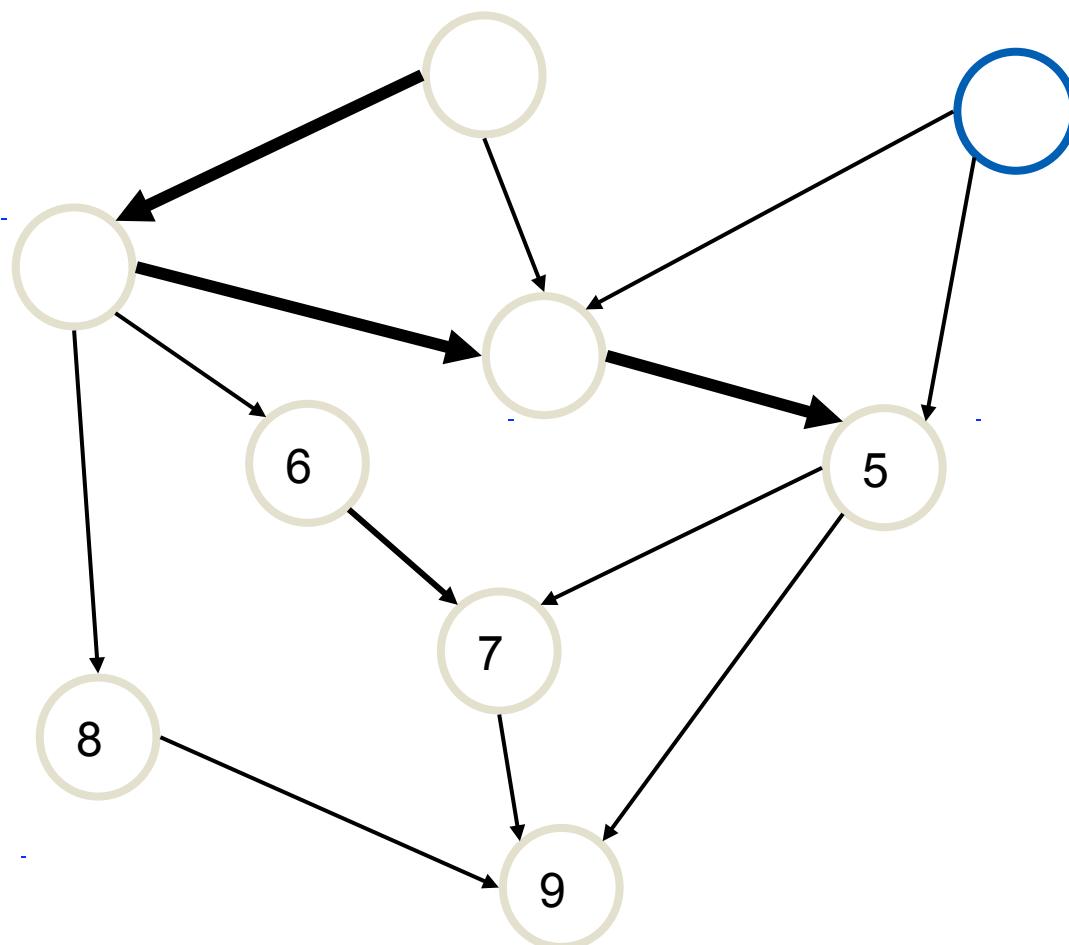
## Topological Sorting v2 Example



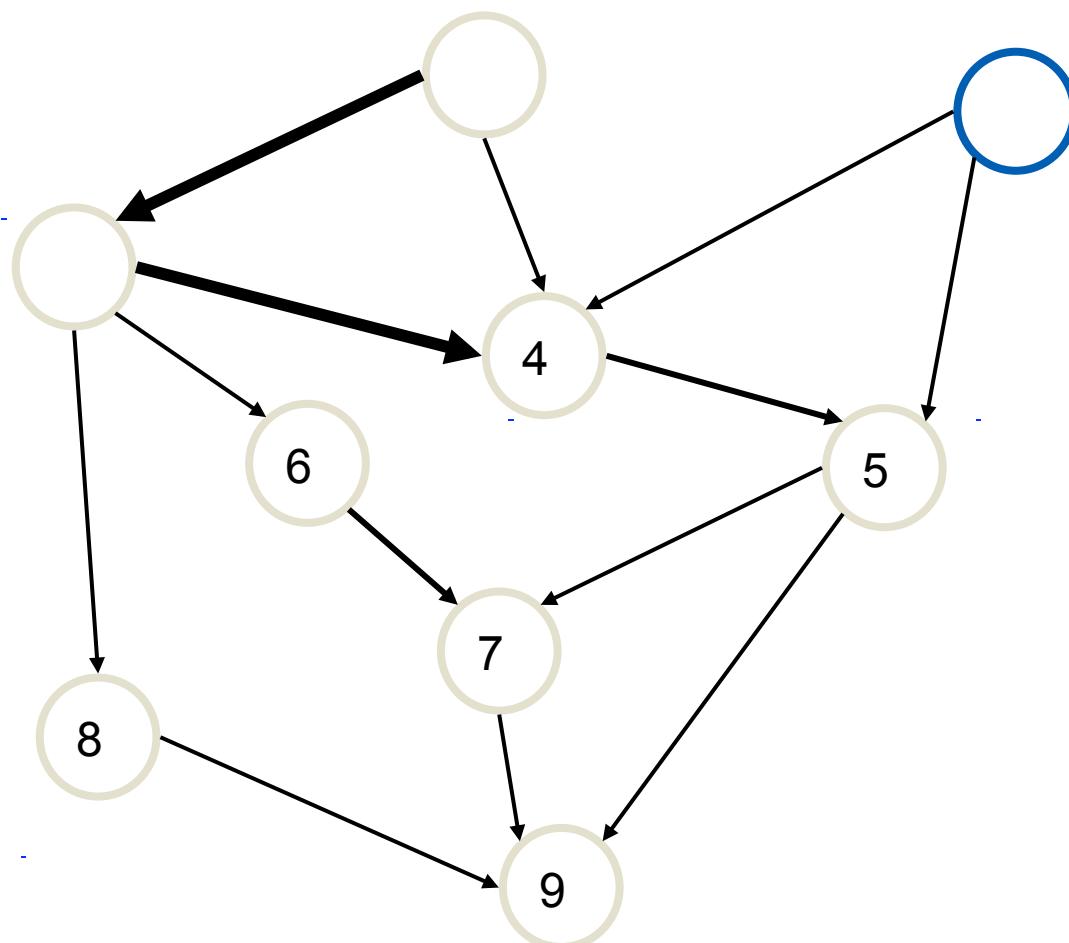
## Topological Sorting v2 Example



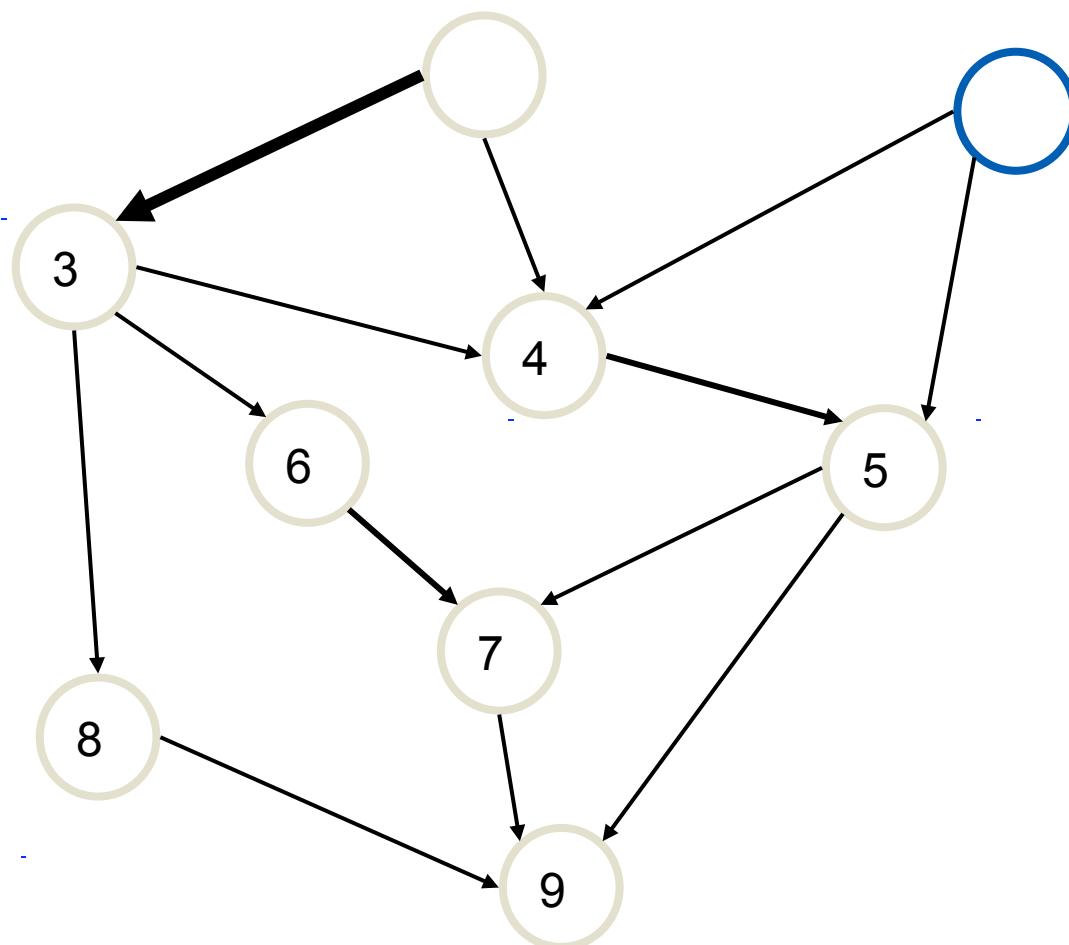
## Topological Sorting v2 Example



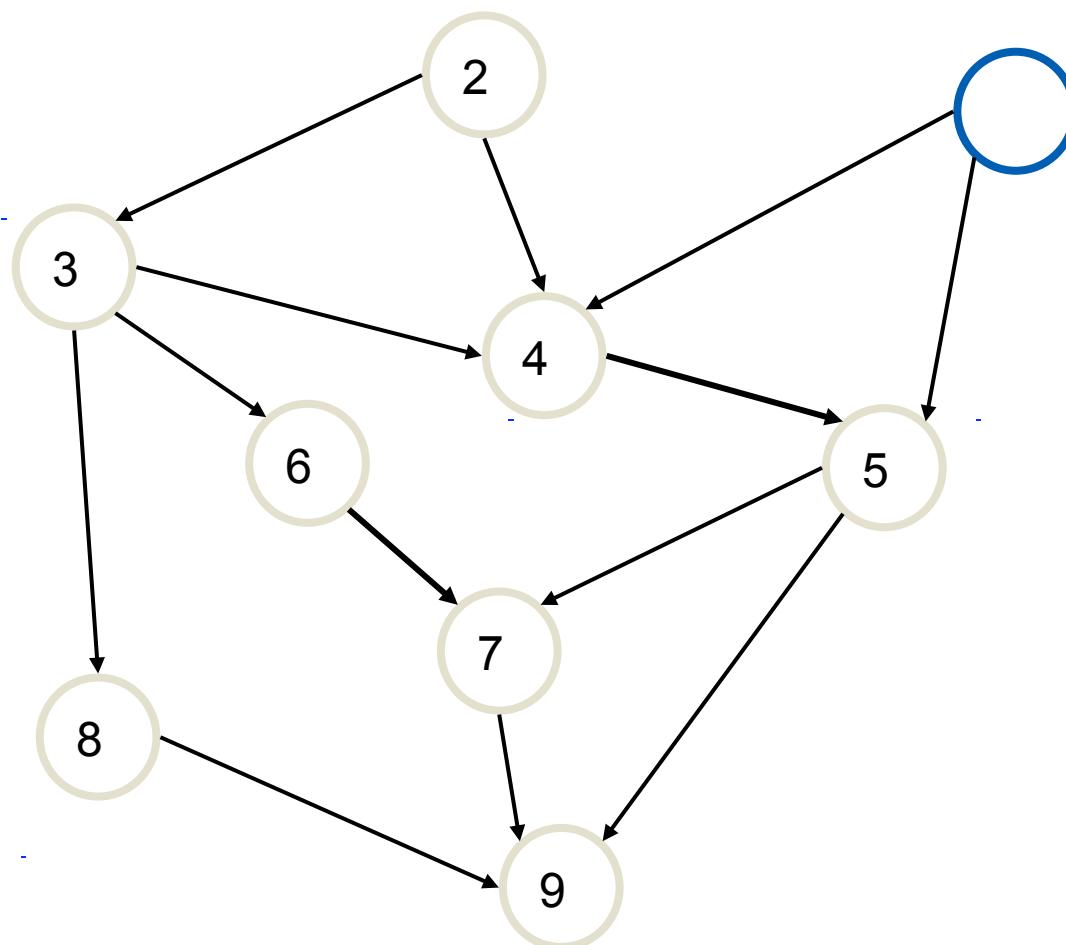
## Topological Sorting v2 Example



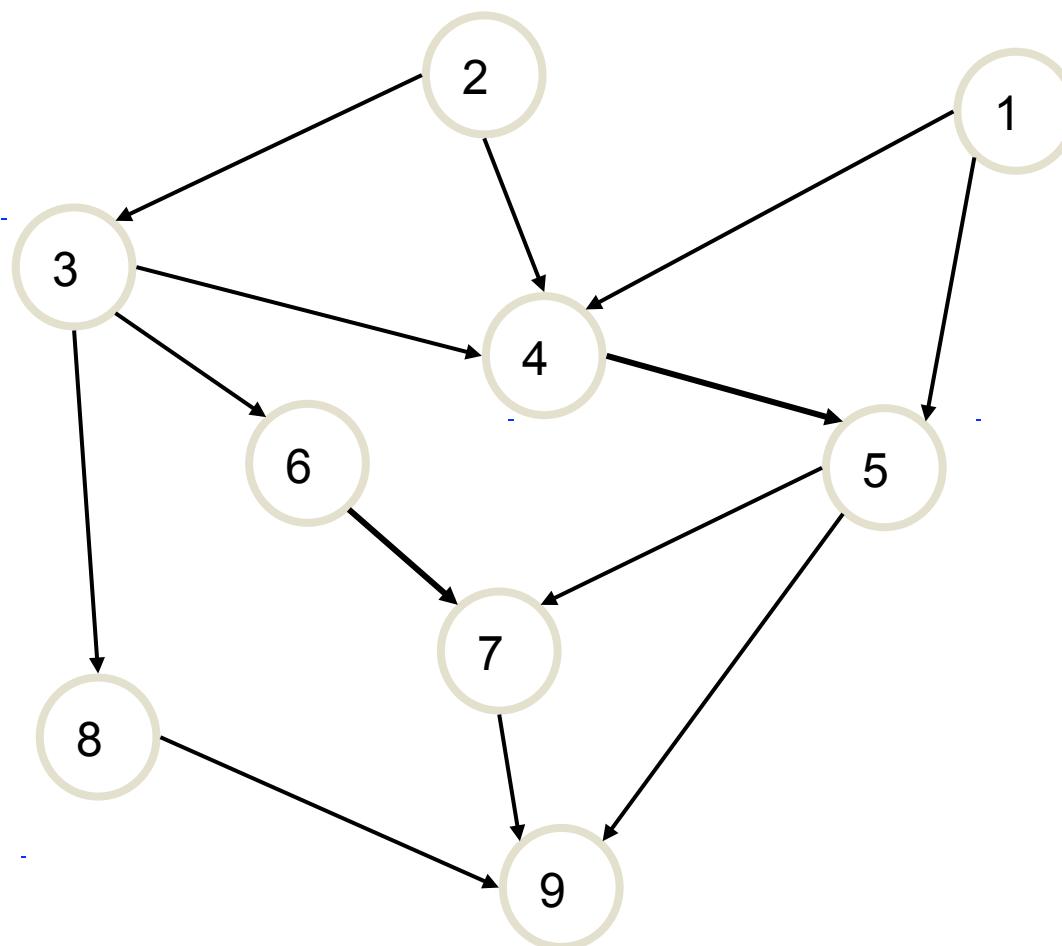
## Topological Sorting v2 Example



## Topological Sorting v2 Example



## Topological Sorting v2 Example

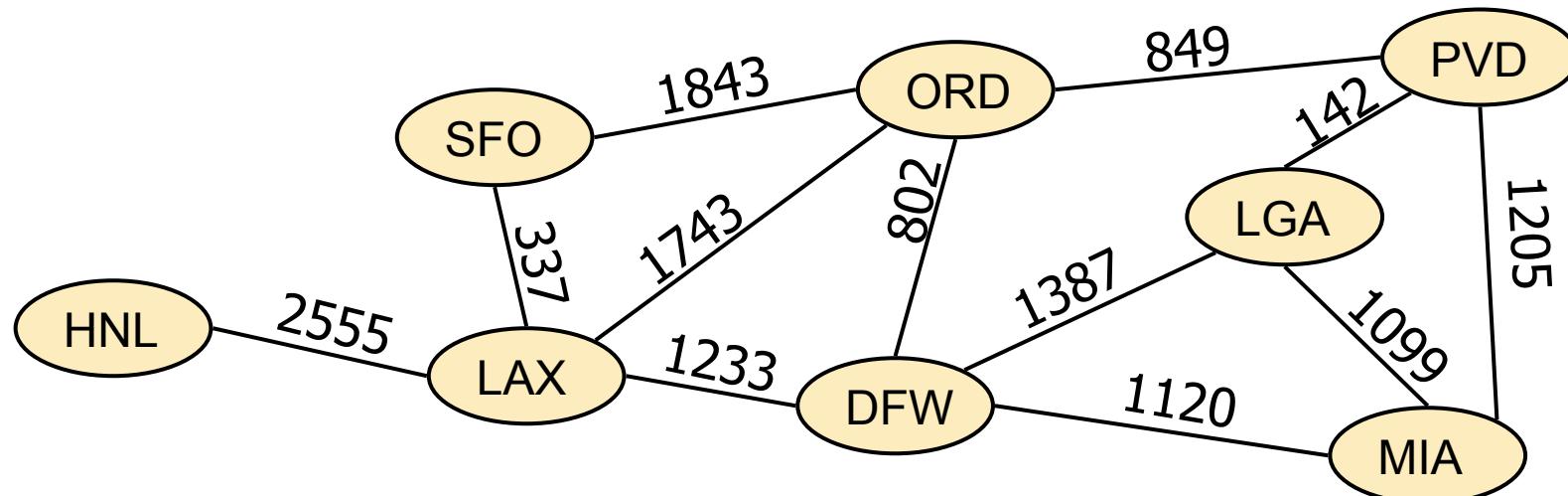


# Outline

- Directed graphs
  - Directed DFS
  - Strong connectivity
  - Transitive closure (Floyd-Warshall algorithm)
  - Topological ordering
- Weighted graphs
  - Shortest paths (Dijkstra's algorithm)

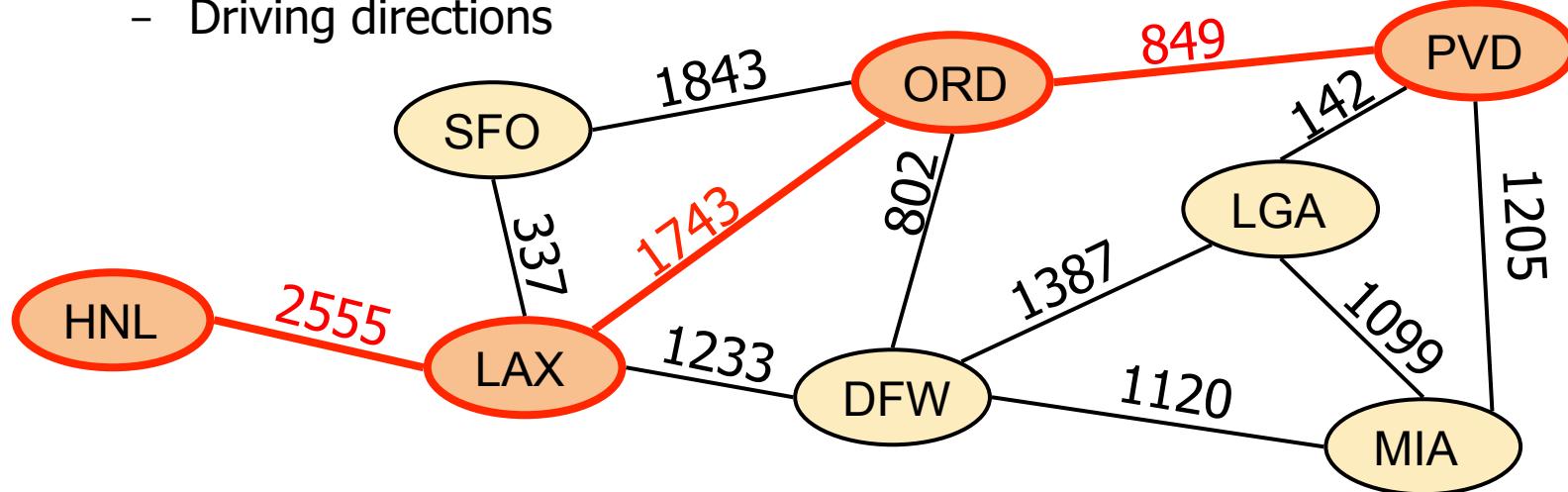
# Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Positive or negative
- 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:

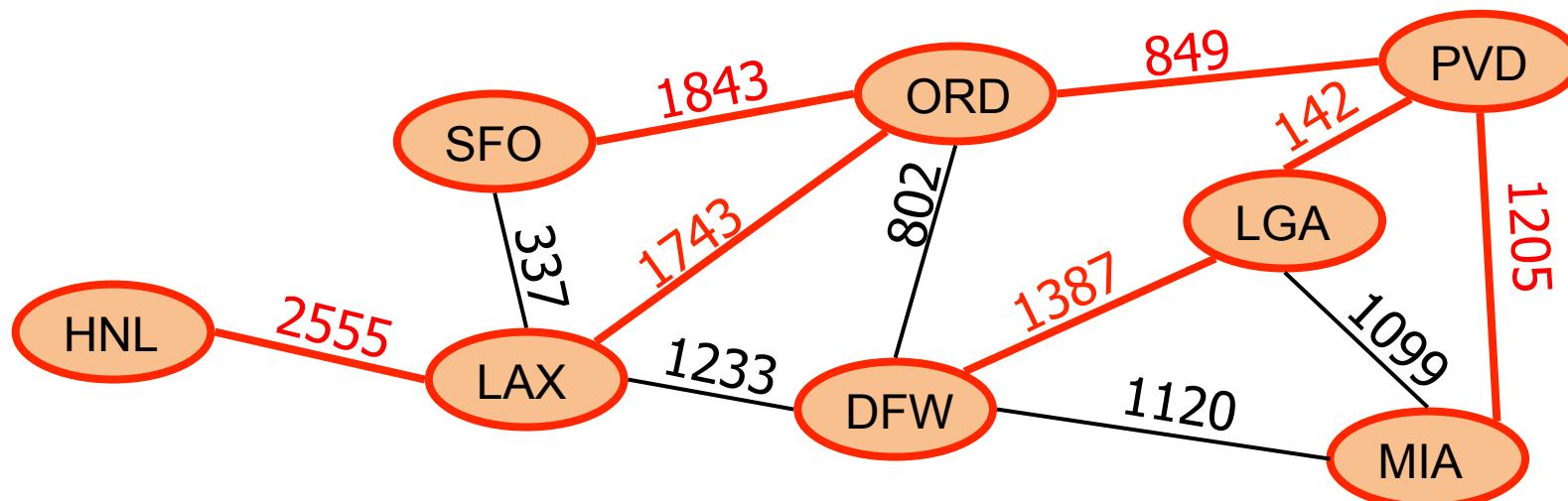
A subpath of a shortest path is itself a shortest path

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 ( PVD )



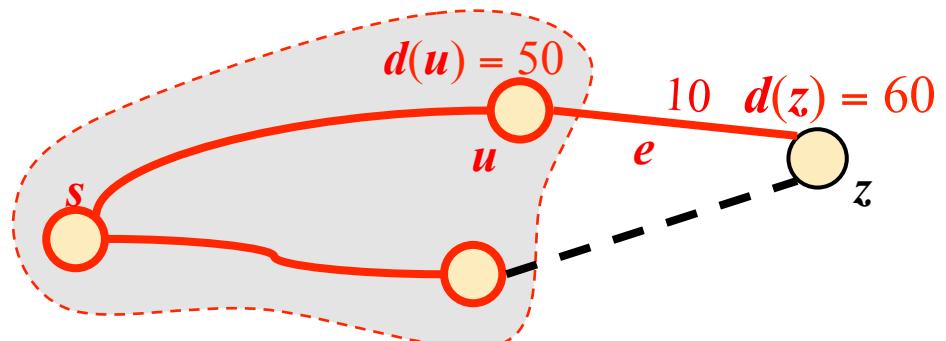
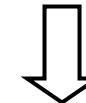
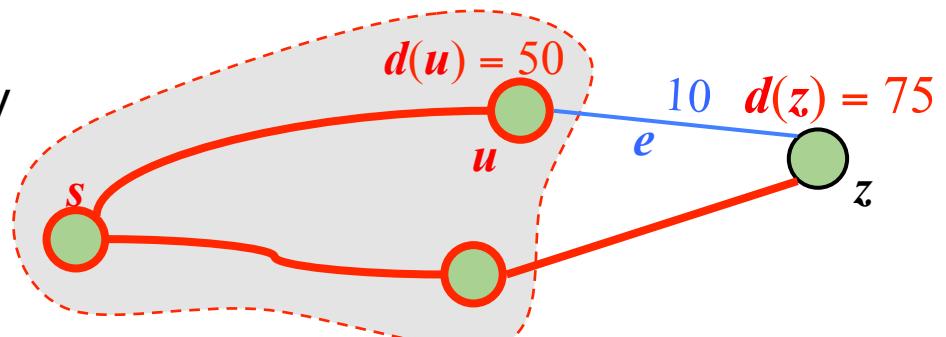
# Dijkstra's Algorithm

- 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  $u$
- *Greedy method*

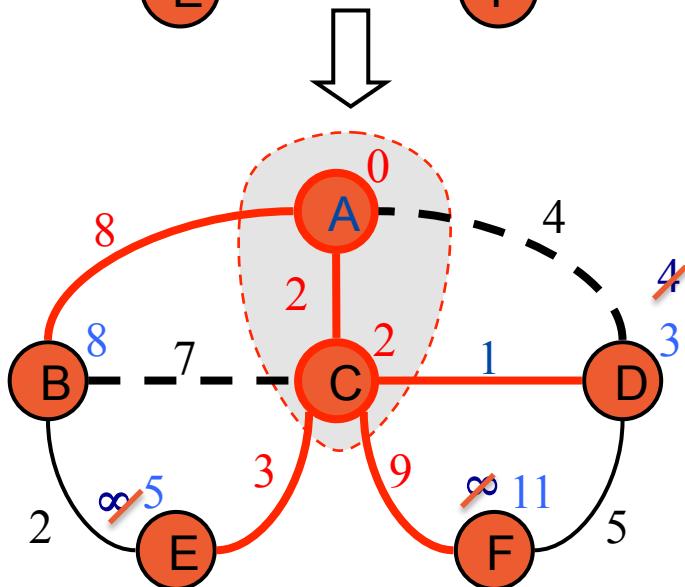
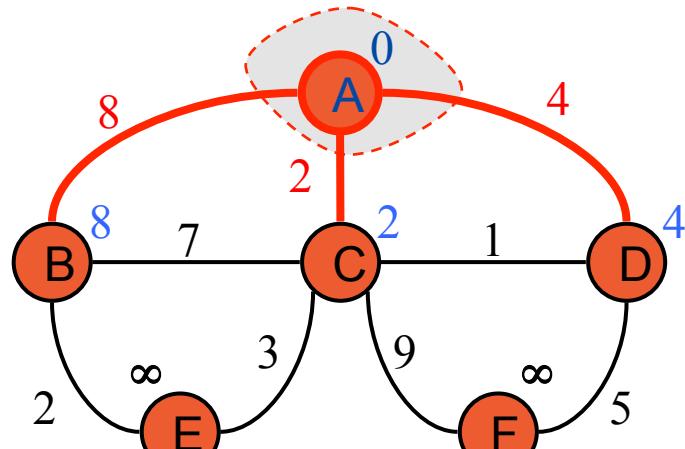
# Edge Relaxation

- 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:

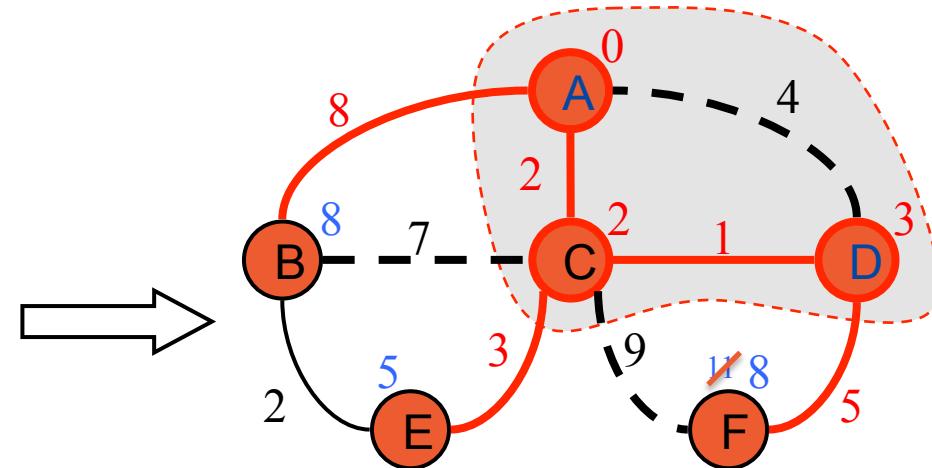
$$d(z) \leftarrow \min\{d(z), d(u) + \text{weight}(e)\}$$



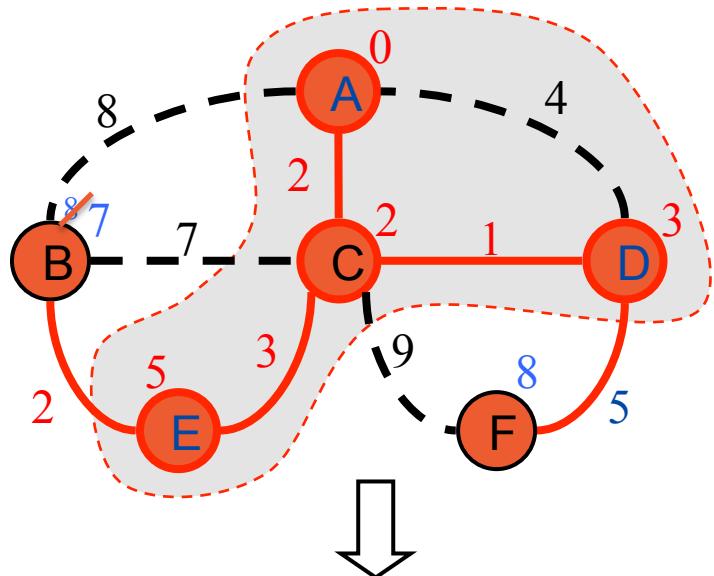
## Example



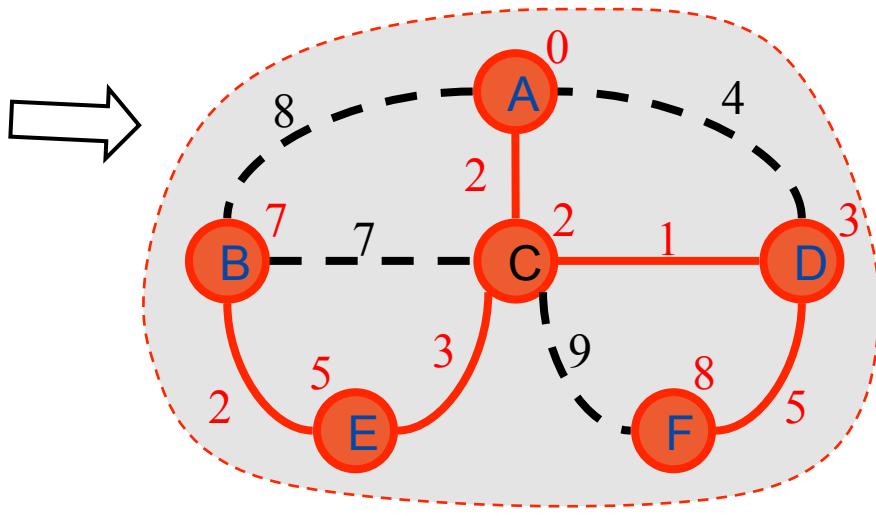
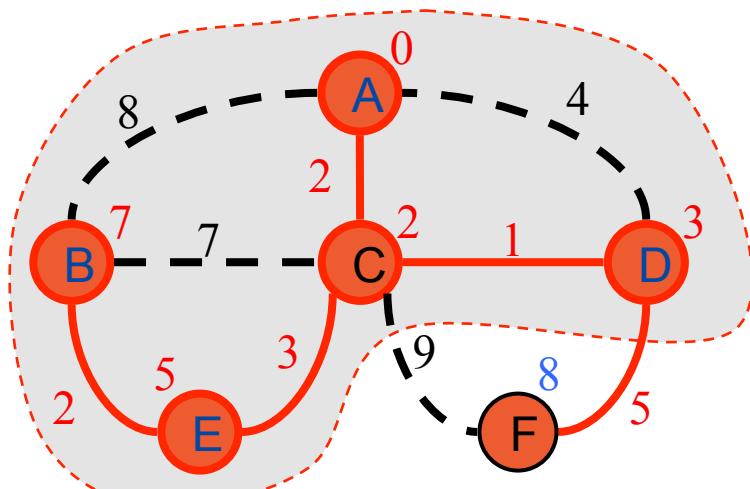
Node	Distance from A	Predecessor node
A	0	--
B	$\infty$	A
C	2	A
D	3	A, C
E	5	C
F	8	C, D



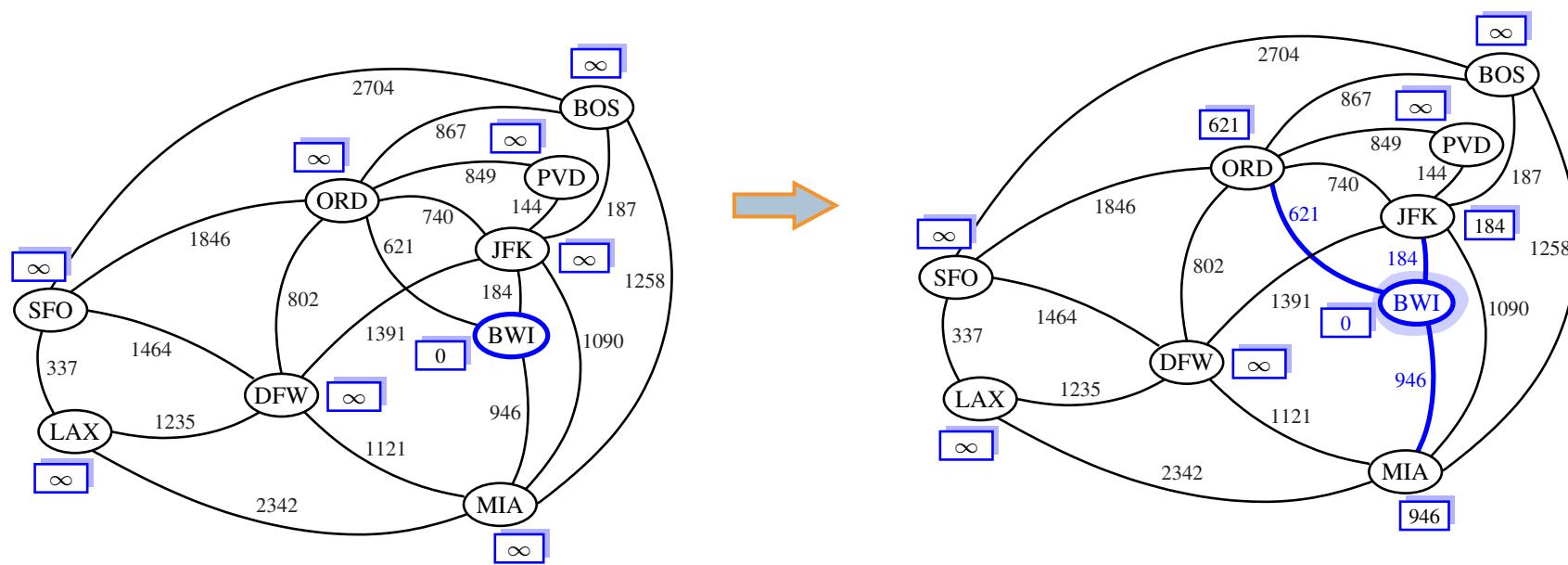
## Example



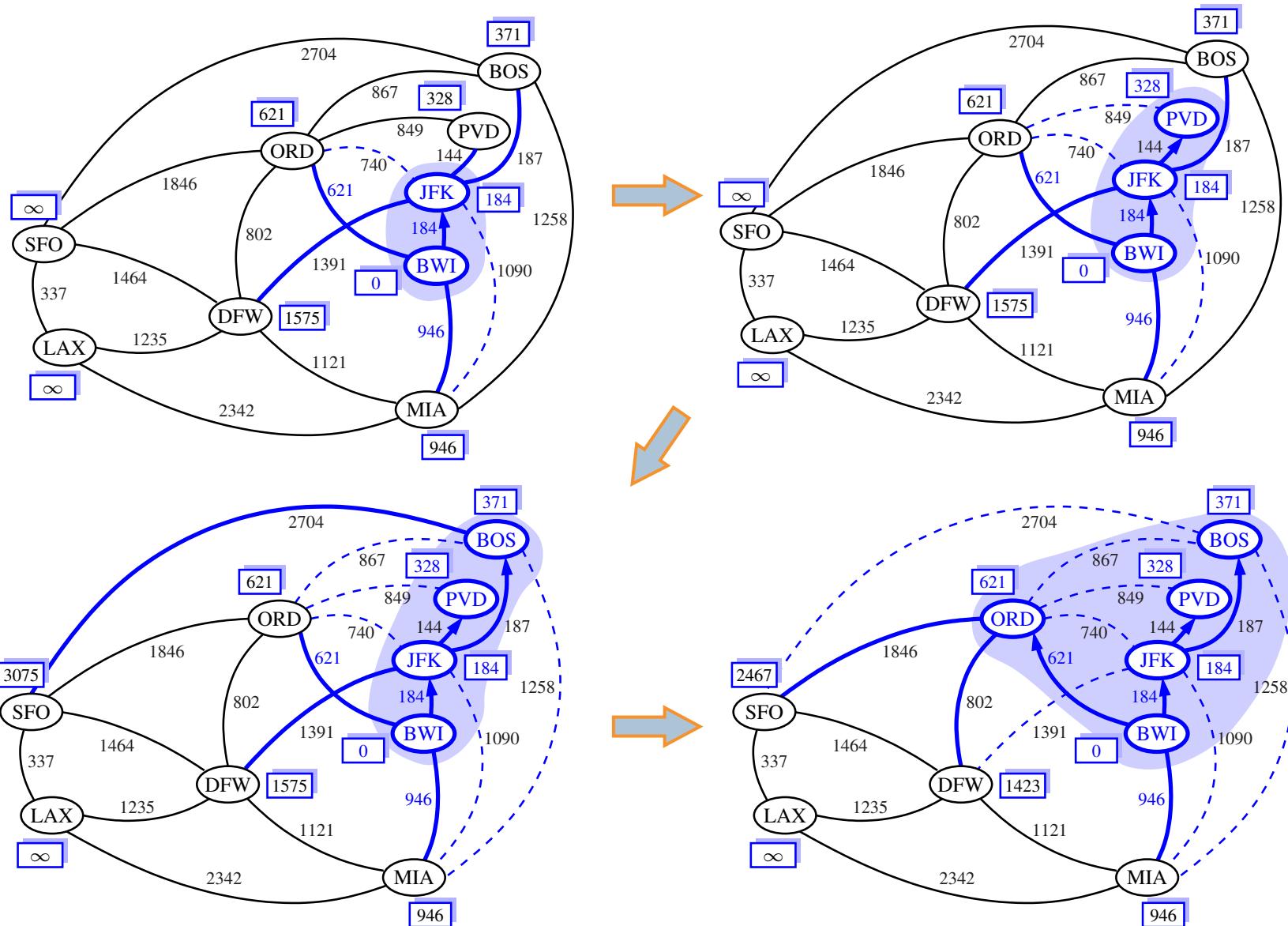
Node	Distance from A	Predecessor node
A	0	--
B	$\infty$	A E
C	$\infty$	A
D	$\infty$	A C
E	$\infty$	C
F	$\infty$	C D



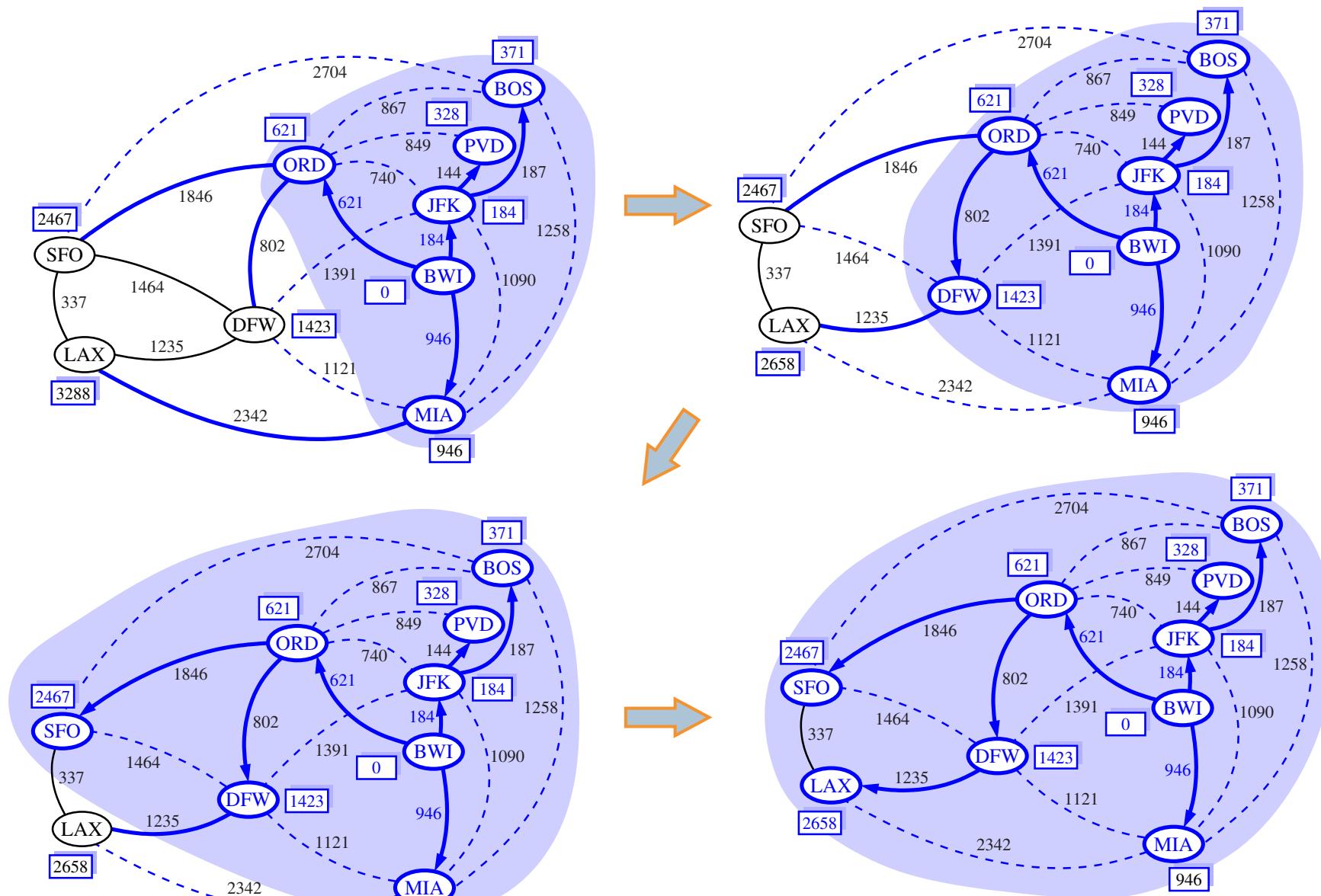
## Dijkstra: Example 2



## Dijkstra: Example 2 (cont)



## Dijkstra: Example 2 (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  $Q.\text{remove\_min}()$

**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
  - Method `incidentEdges` is called 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

# Java Implementation

```
1  /** Computes shortest-path distances from src vertex to all reachable vertices of g. */
2  public static <V> Map<Vertex<V>, Integer>
3      shortestPathLengths(Graph<V, Integer> g, Vertex<V> src) {
4      // d.get(v) is upper bound on distance from src to v
5      Map<Vertex<V>, Integer> d = new ProbeHashMap<>();
6      // map reachable v to its d value
7      Map<Vertex<V>, Integer> cloud = new ProbeHashMap<>();
8      // pq will have vertices as elements, with d.get(v) as key
9      AdaptablePriorityQueue<Integer, Vertex<V>> pq;
10     pq = new HeapAdaptablePriorityQueue<>();
11     // maps from vertex to its pq locator
12     Map<Vertex<V>, Entry<Integer, Vertex<V>>> pqTokens;
13     pqTokens = new ProbeHashMap<>();
14
15     // for each vertex v of the graph, add an entry to the priority queue, with
16     // the source having distance 0 and all others having infinite distance
17     for (Vertex<V> v : g.vertices()) {
18         if (v == src)
19             d.put(v, 0);
20         else
21             d.put(v, Integer.MAX_VALUE);
22         pqTokens.put(v, pq.insert(d.get(v), v));           // save entry for future updates
23     }
```

# Java Implementation (continued)

```
24 // now begin adding reachable vertices to the cloud
25 while (!pq.isEmpty()) {
26     Entry<Integer, Vertex<V>> entry = pq.removeMin();
27     int key = entry.getKey();
28     Vertex<V> u = entry.getValue();
29     cloud.put(u, key);                                // this is actual distance to u
30     pqTokens.remove(u);                             // u is no longer in pq
31     for (Edge<Integer> e : g.outgoingEdges(u)) {
32         Vertex<V> v = g.opposite(u,e);
33         if (cloud.get(v) == null) {
34             // perform relaxation step on edge (u,v)
35             int wgt = e.getElement();
36             if (d.get(u) + wgt < d.get(v)) {          // better path to v?
37                 d.put(v, d.get(u) + wgt);           // update the distance
38                 pq.replaceKey(pqTokens.get(v), d.get(v)); // update the pq entry
39             }
40         }
41     }
42 }
43 return cloud;           // this only includes reachable vertices
44 }
```

# Shortest Paths Tree

- Using the template method pattern, we can extend Dijkstra's algorithm to return a **tree of shortest paths** from the start vertex to all other vertices
- We store with each vertex a third label:
  - parent edge in the shortest path tree
- In the edge relaxation step, we update the parent label

**Algorithm *DijkstraShortestPathsTree(G, s)***

...

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

...

***setParent(v, Ø)***

...

**for all  $e \in G.incidentEdges(u)$**

{ relax edge  $e$  }

$z \leftarrow G.opposite(u,e)$

$r \leftarrow getDistance(u) + weight(e)$

if  $r < getDistance(z)$

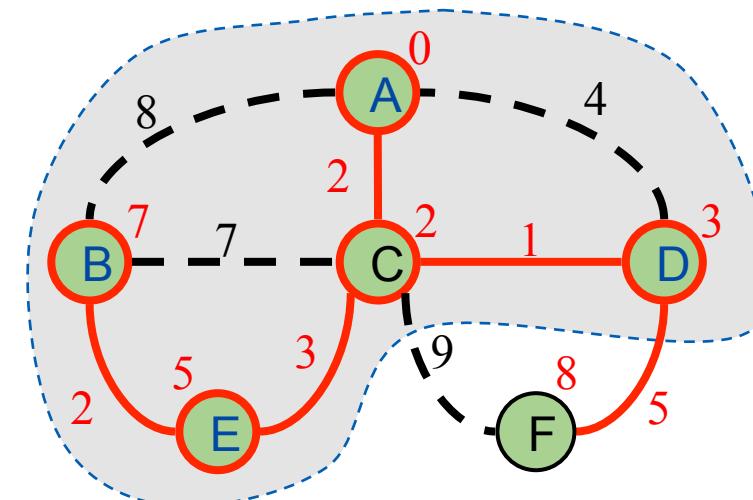
***setDistance(z, r)***

***setParent(z,e)***

***Q.replaceKey(getEntry(z),r)***

# 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

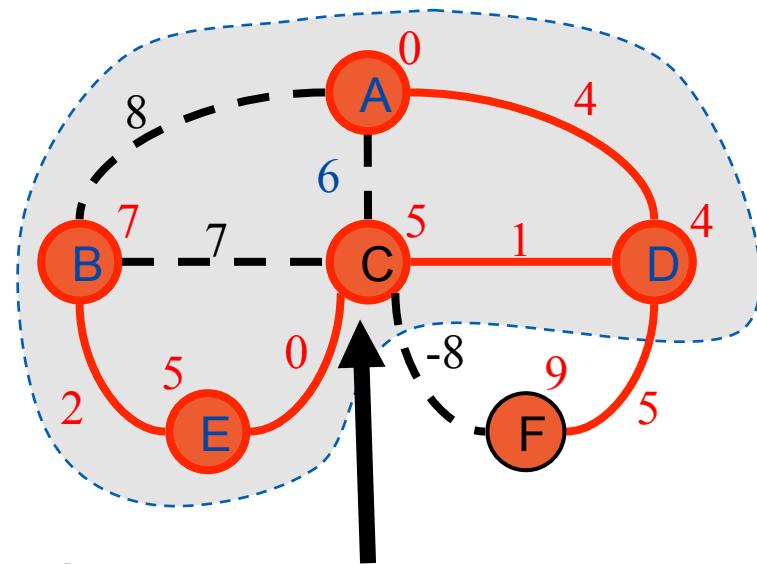


# Why It Doesn't Work for Negative-Weight Edges



- ◆ Dijkstra's algorithm is a "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  $d(C)=5$ !

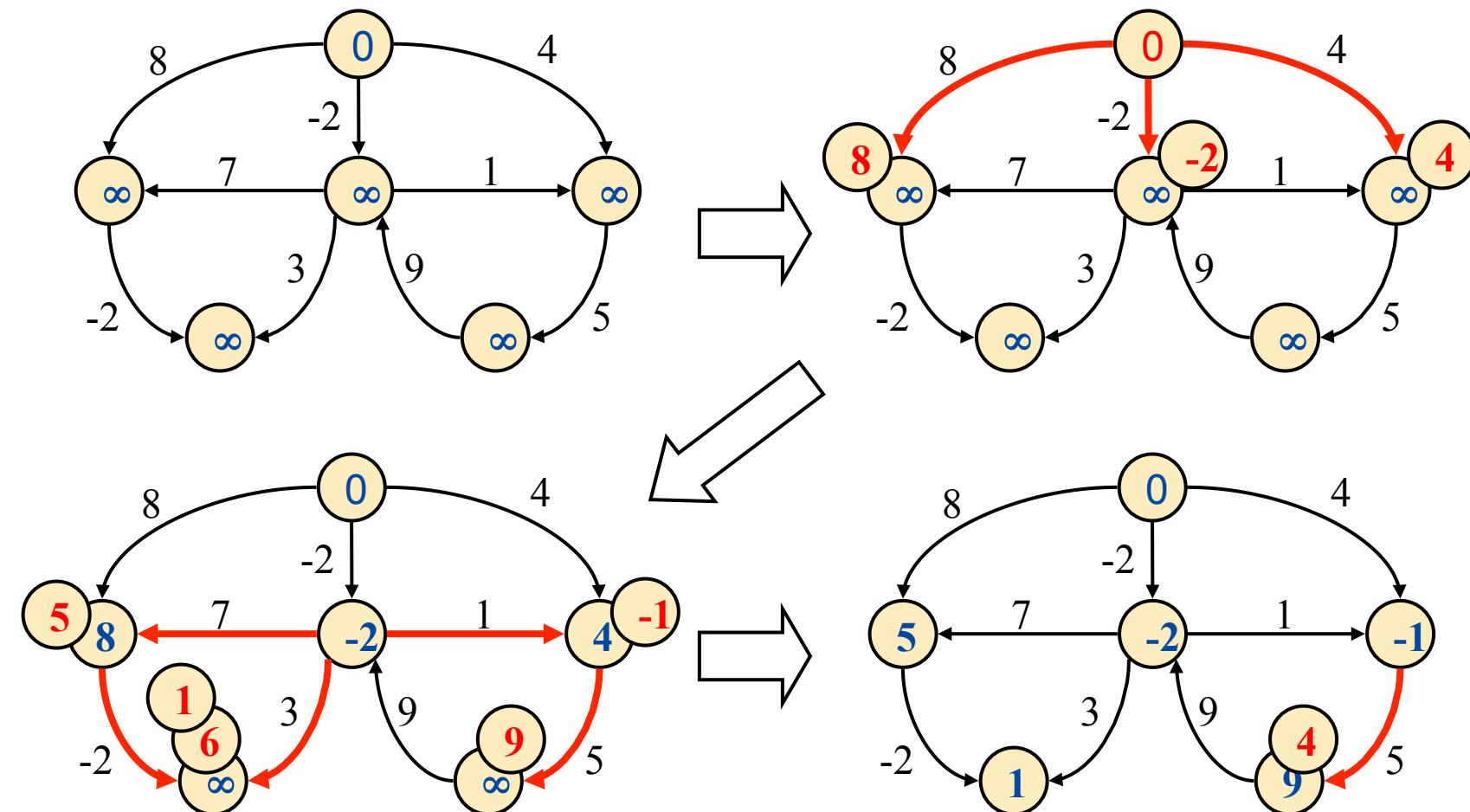
## Bellman-Ford Algorithm (non-examinable enrichment, not in textbook)

- 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



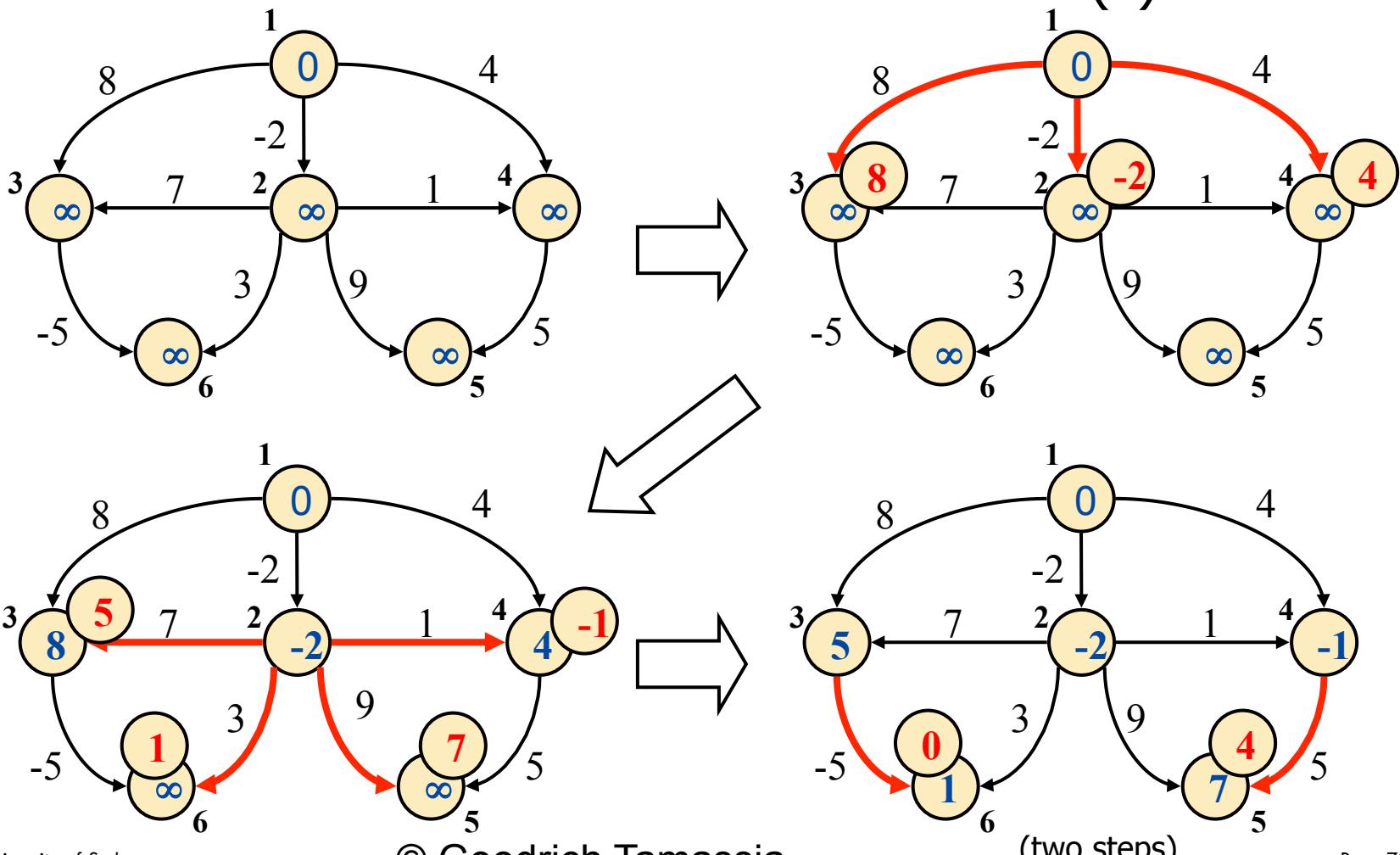
## DAG-based Algorithm (non-examinable enrichment, not in textbook)

- 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-based shortest paths Example

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



# Summary

- Directed graphs
  - Directed DFS (section 14.3)
  - Strong connectivity (section 14.3)
  - Transitive closure (Floyd-Warshall algorithm) (section 14.4)
  - Topological ordering (section 14.5)
- Weighted graphs (Section 14.6)
  - Shortest paths (Dijkstra's algorithm)