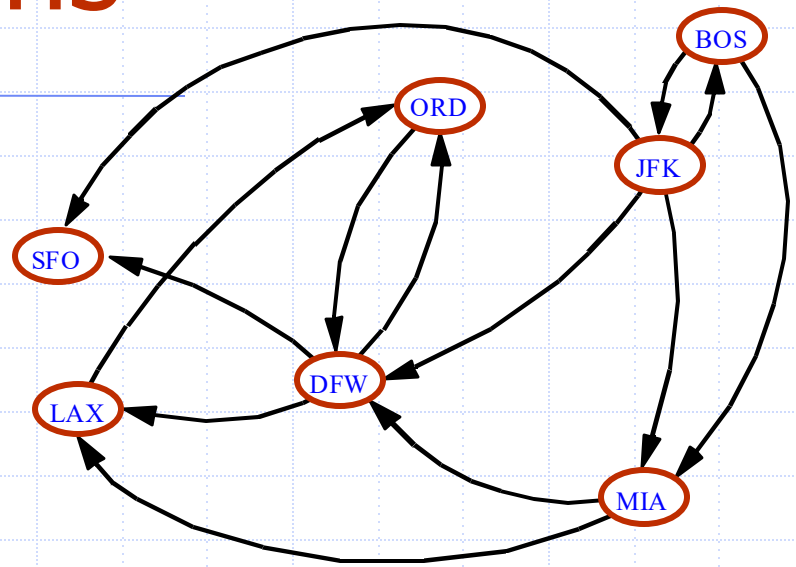
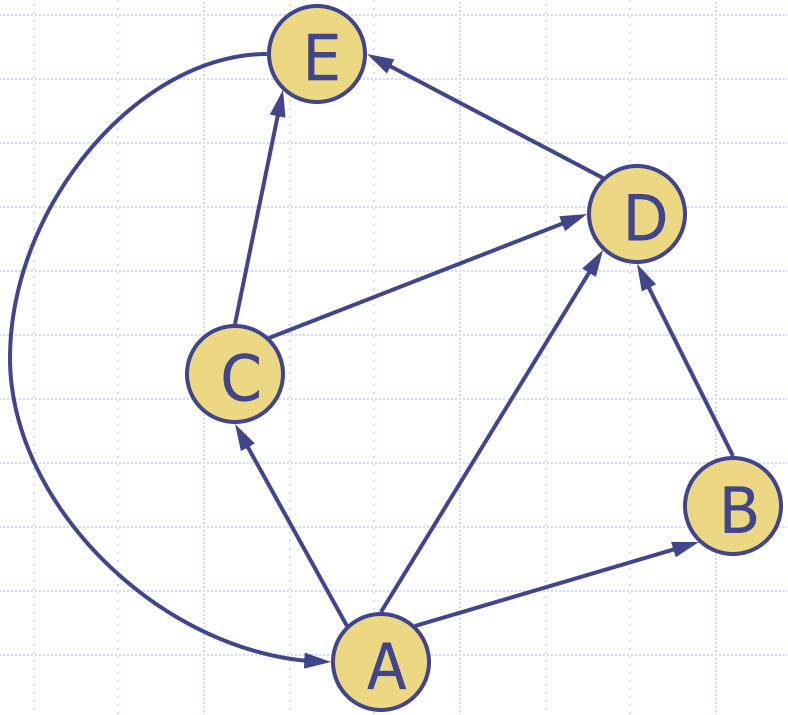
Directed Graphs

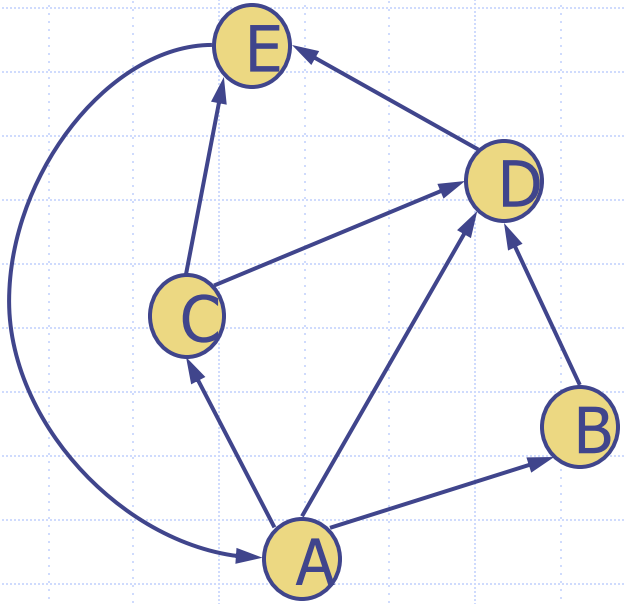


Digraphs

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



Digraph Properties

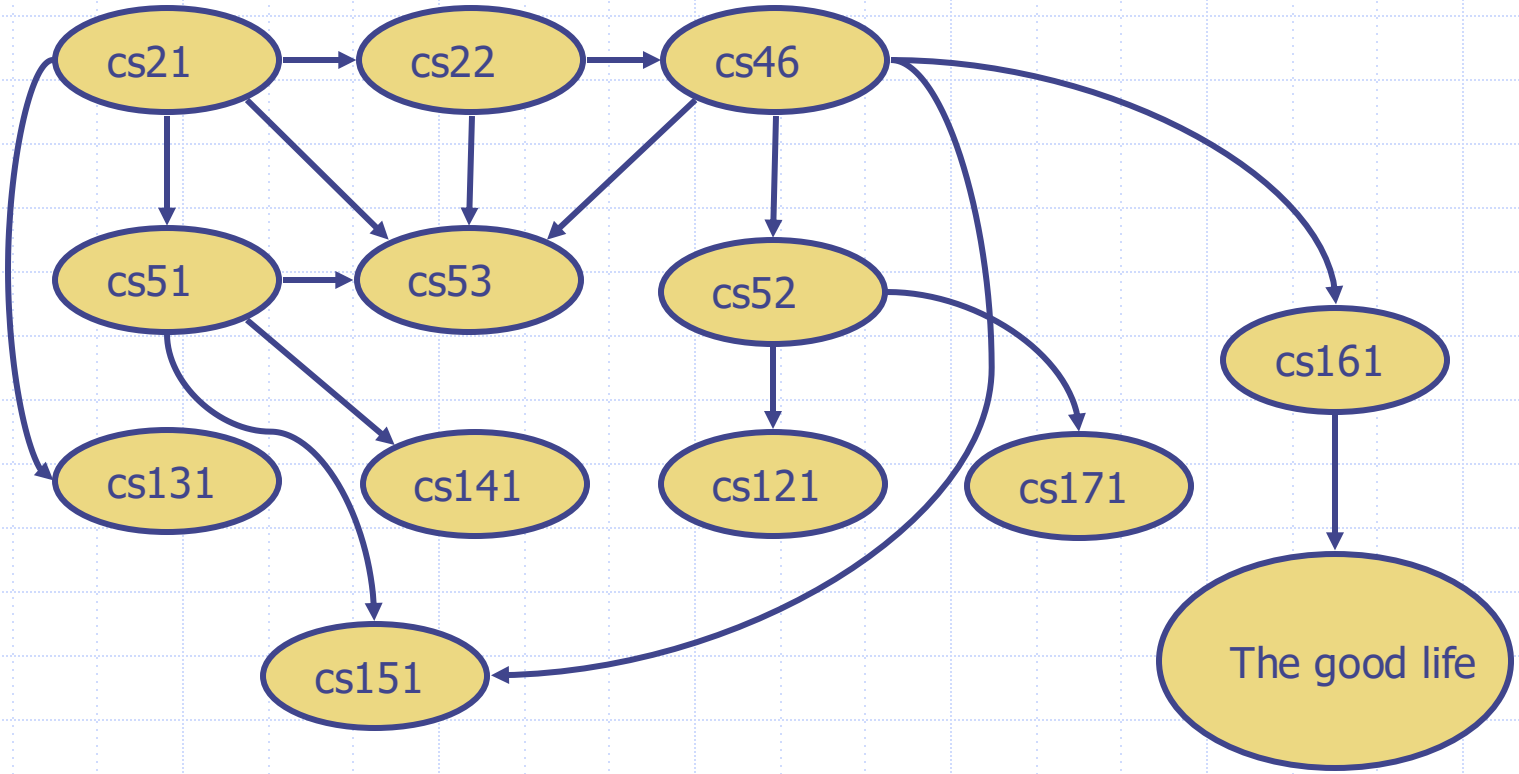


- A graph G=(V,E) such that
  - Each edge goes in one direction:
  - Edge (a,b) goes from a to b,

  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

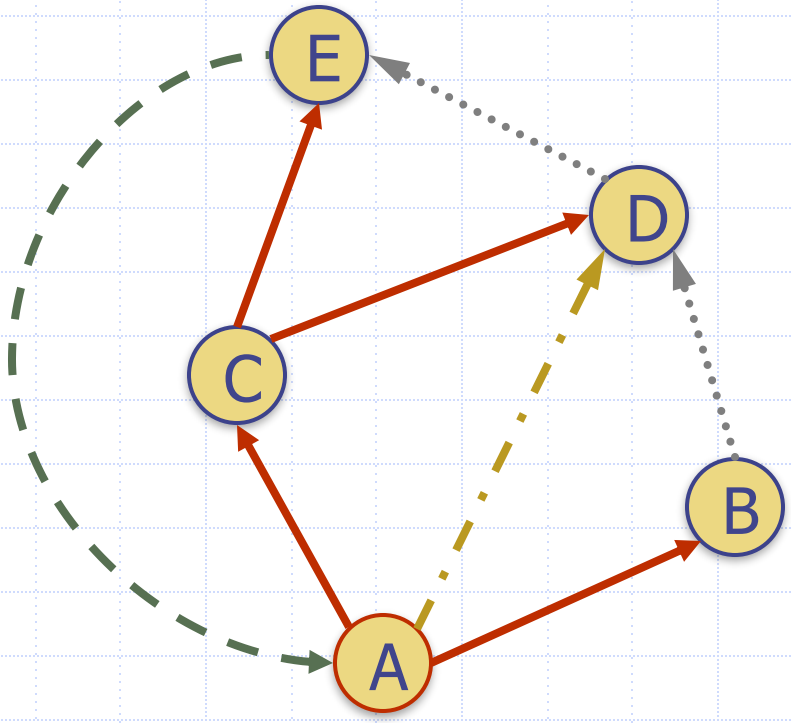
Digraph Application

- 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/tree edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex s determines the vertices reachable from s



Directed DFS

- Forward edge (u, v)

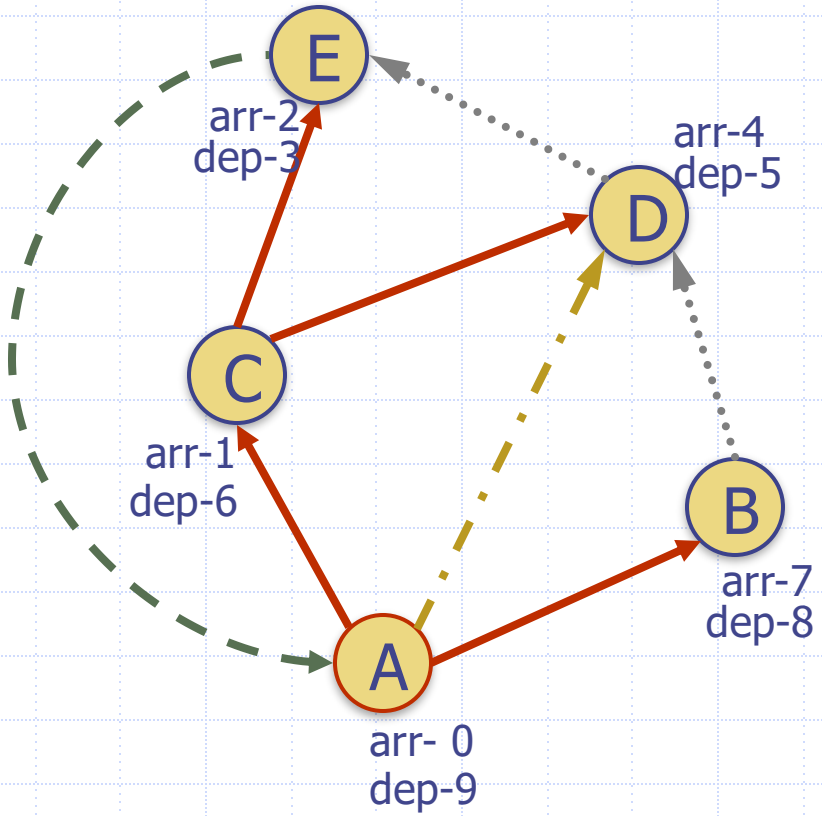
arr(u)<arr(v)<dep(v)<dep(u)

- Back edge (u, v)

arr(v)<arr(u)<dep(u)<dep(v)

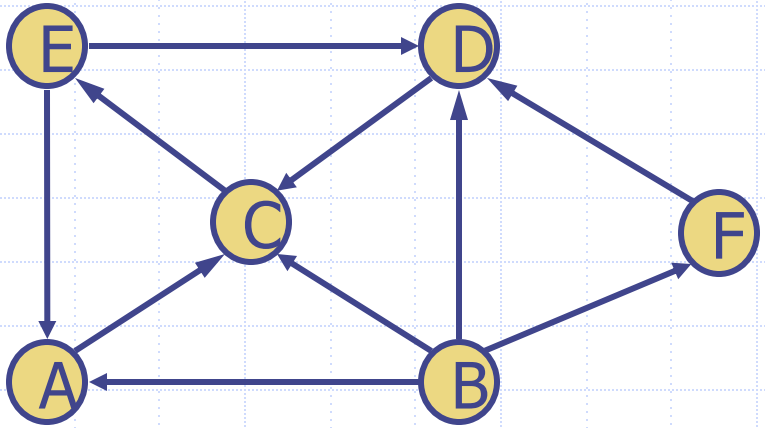
- Cross edge (u, v)

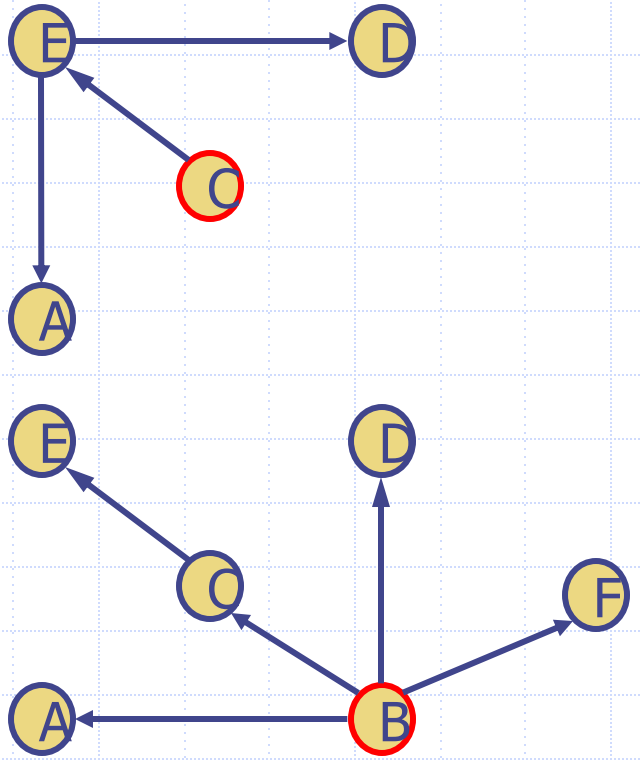
arr(v)<dep(v)<arr(u)<dep(u)



Reachability

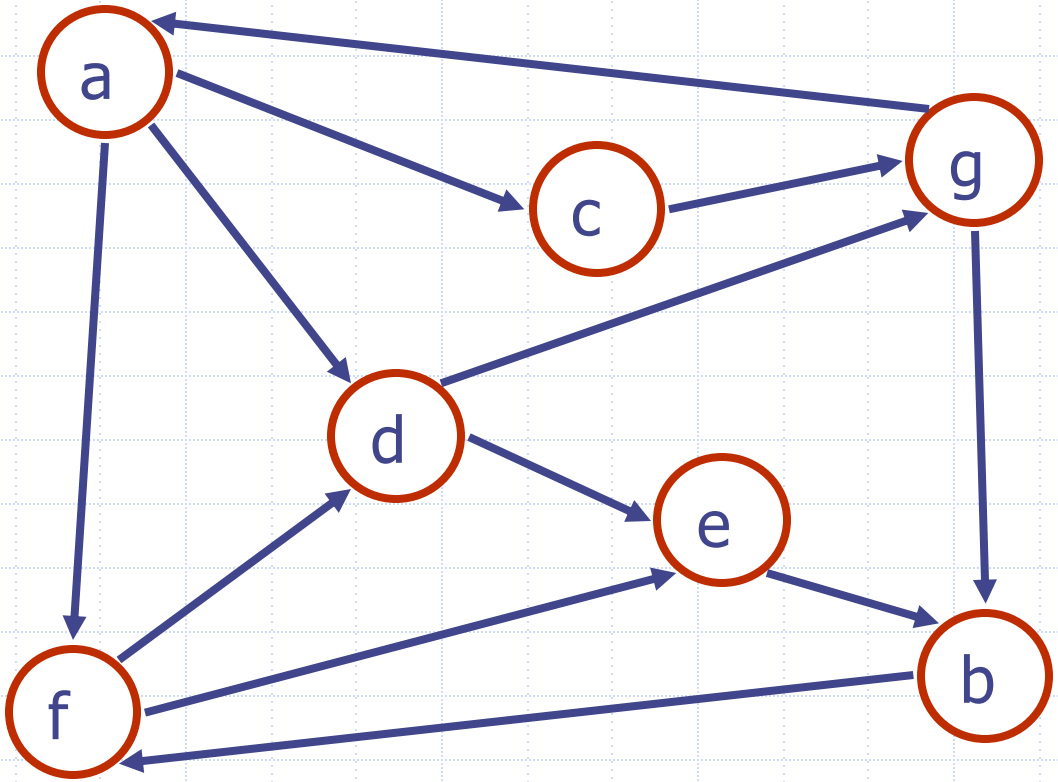
- DFS tree rooted at v: vertices reachable from v via directed paths





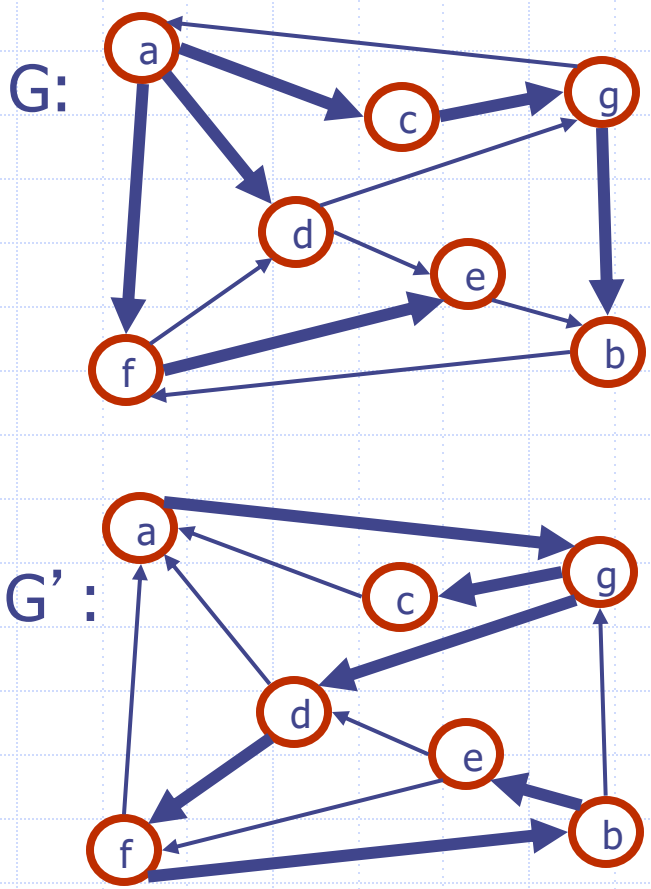
Strong Connectivity

- Each vertex can reach all other vertices



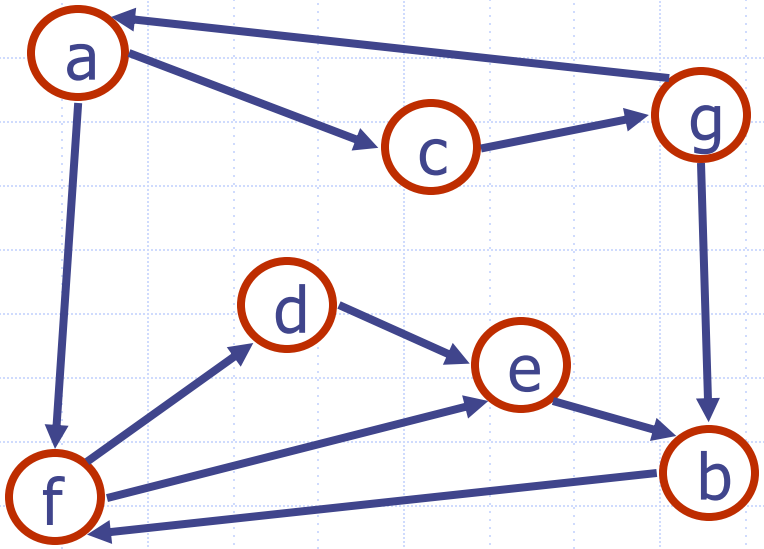
Strong Connectivity Algorithm

- 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.

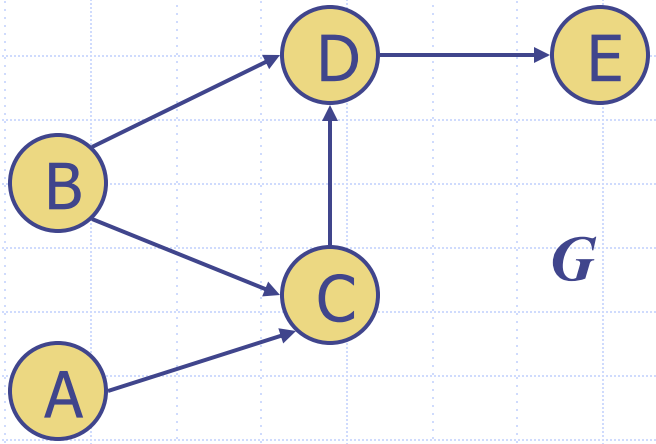


{ a , c , g }

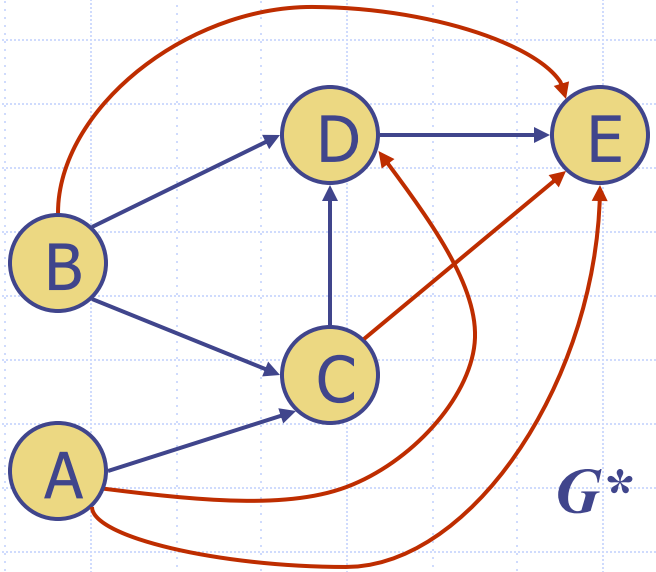
{ f , d , e , b }

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



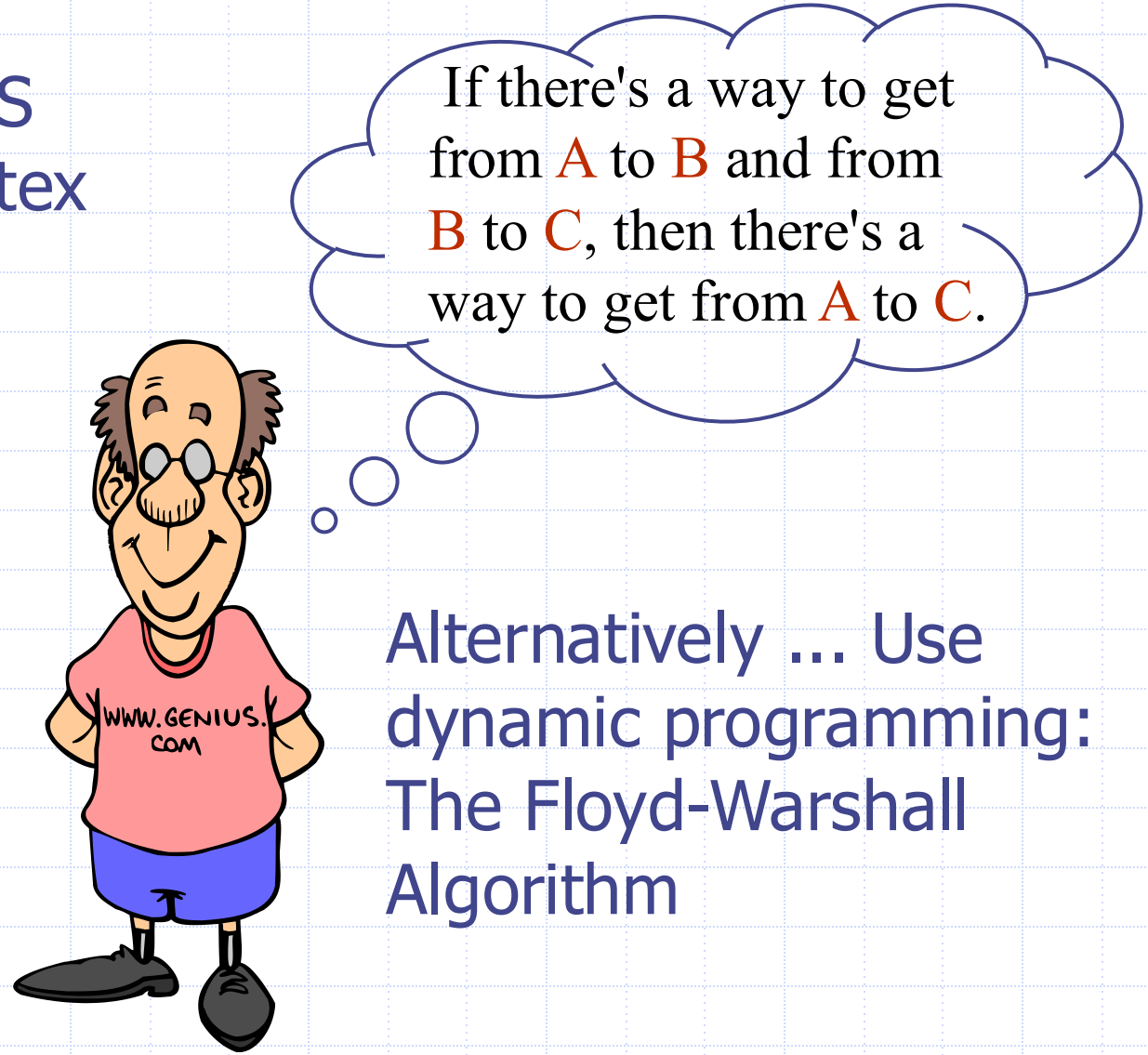
G



G^*

Computing the Transitive Closure

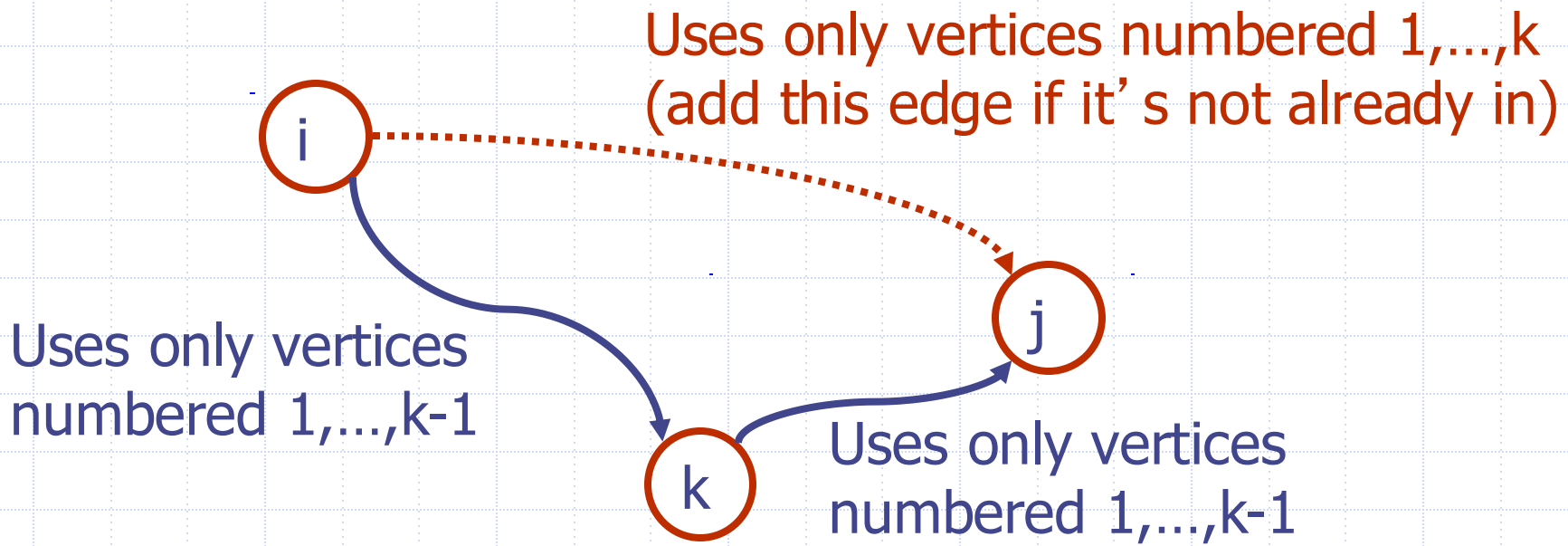
- We can perform DFS starting at each vertex
  - O(n(n+m))



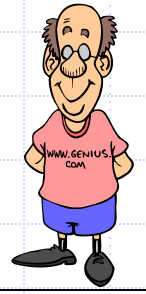
Alternatively ... Use dynamic programming: The Floyd-Warshall Algorithm

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:



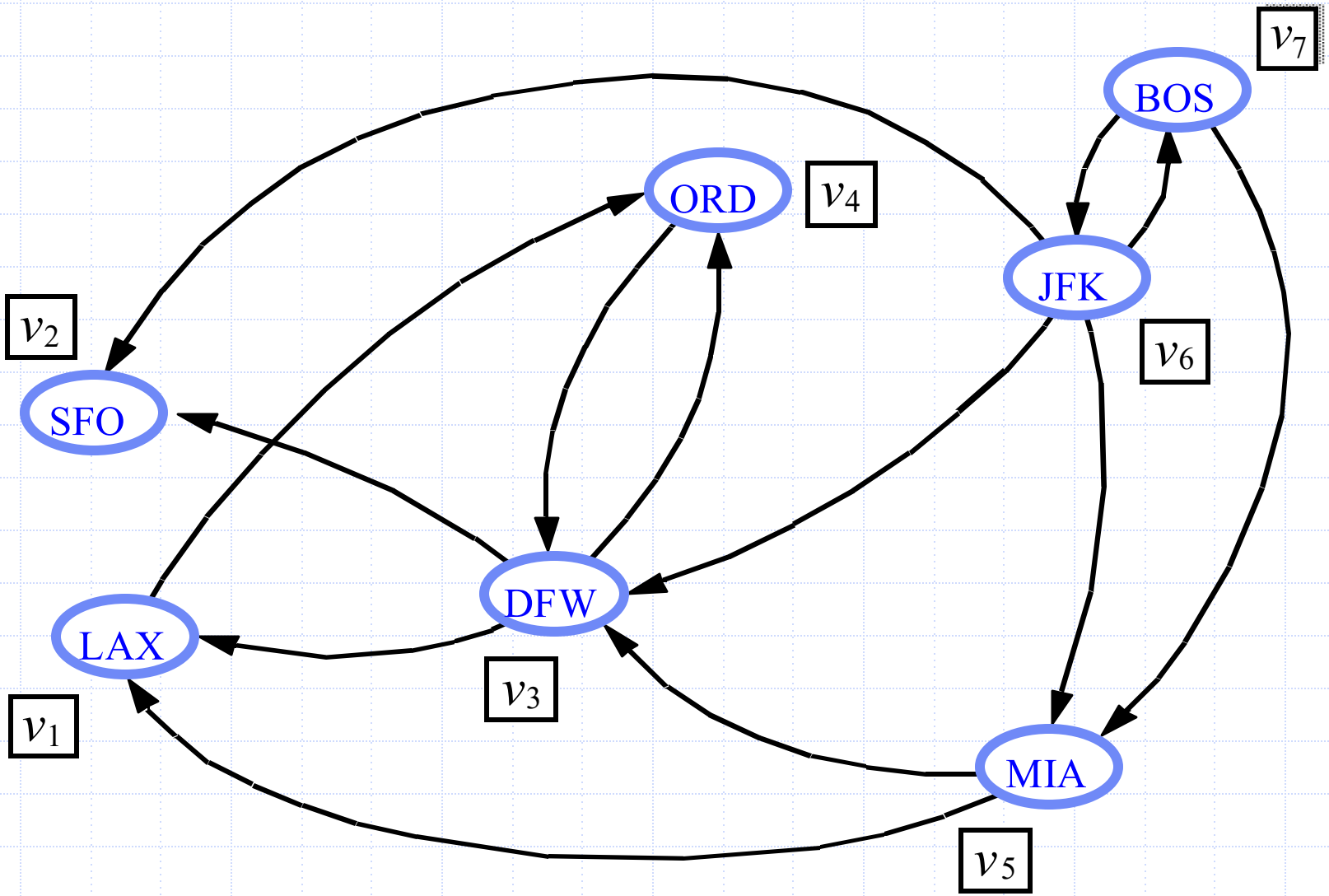
Floyd-Warshall's Algorithm



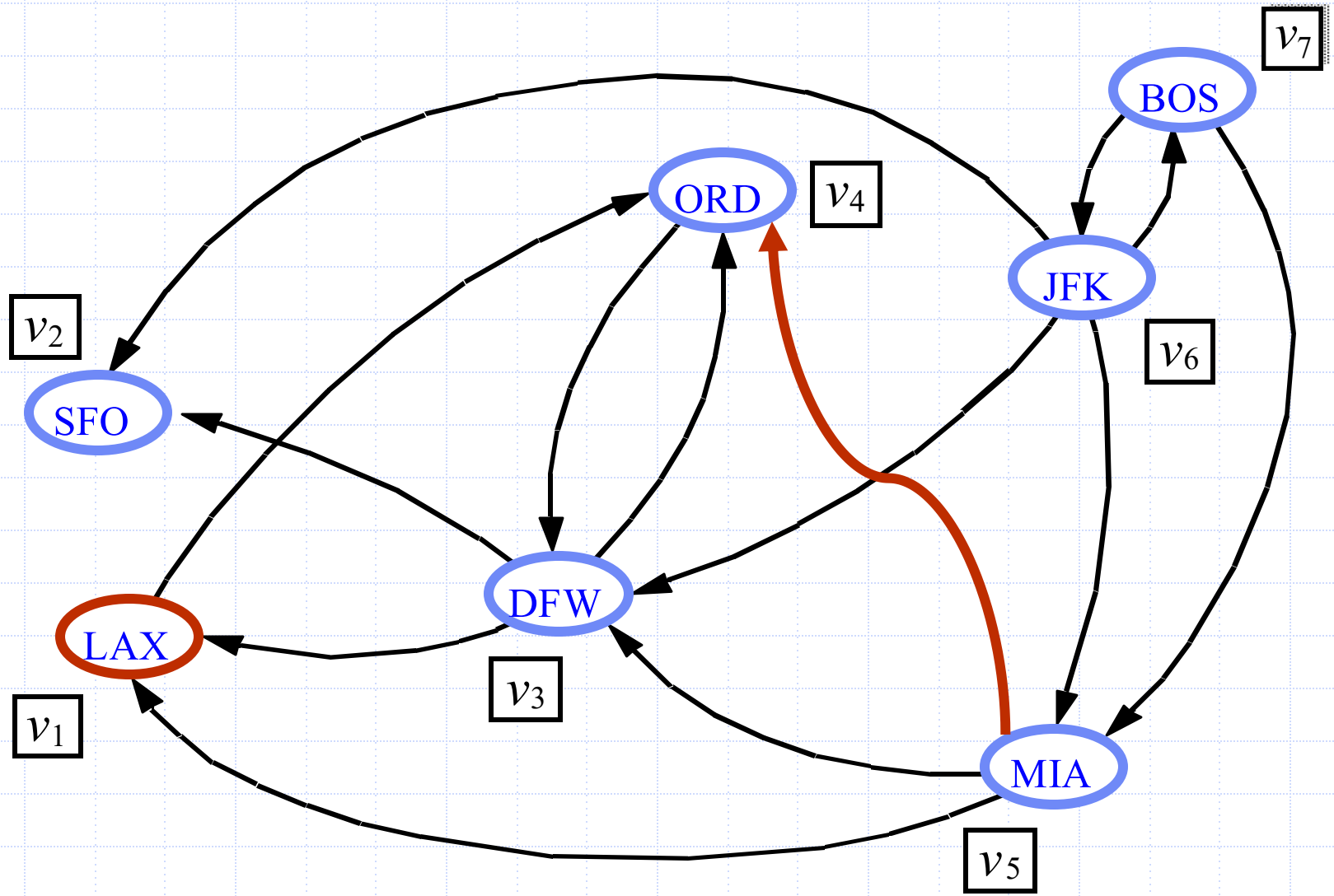
- Number vertices $v_1, \ldots, v_n$
- Compute digraphs $G_0, \ldots, G_n$
  - $G_0 = G$
  - G_k has directed edge (v_i, v_j) if G has a directed path from v_i to v_j with intermediate vertices in $\{v_1, \ldots, v_k\}$
- We have that $G_n = G^*$
- In phase k, digraph G_k is computed from G_{k-1}
- Running 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 ← 1
    for all v ∈ G.vertices()
        denote v as v_i
        i ← i + 1
    G_0 ← G
    for k ← 1 to n do
        G_k ← G_{k-1}
        for i ← 1 to n (i ≠ k) do
            for j ← 1 to n (j ≠ i, k) do
                if G_{k-1}.areAdjacent(v_i, v_k) ∧
                        G_{k-1}.areAdjacent(v_k, v_j)
                    if ¬G_k.areAdjacent(v_i, v_j)
                        G_k.insertDirectedEdge(v_i, v_j, k)
    return G_n
```

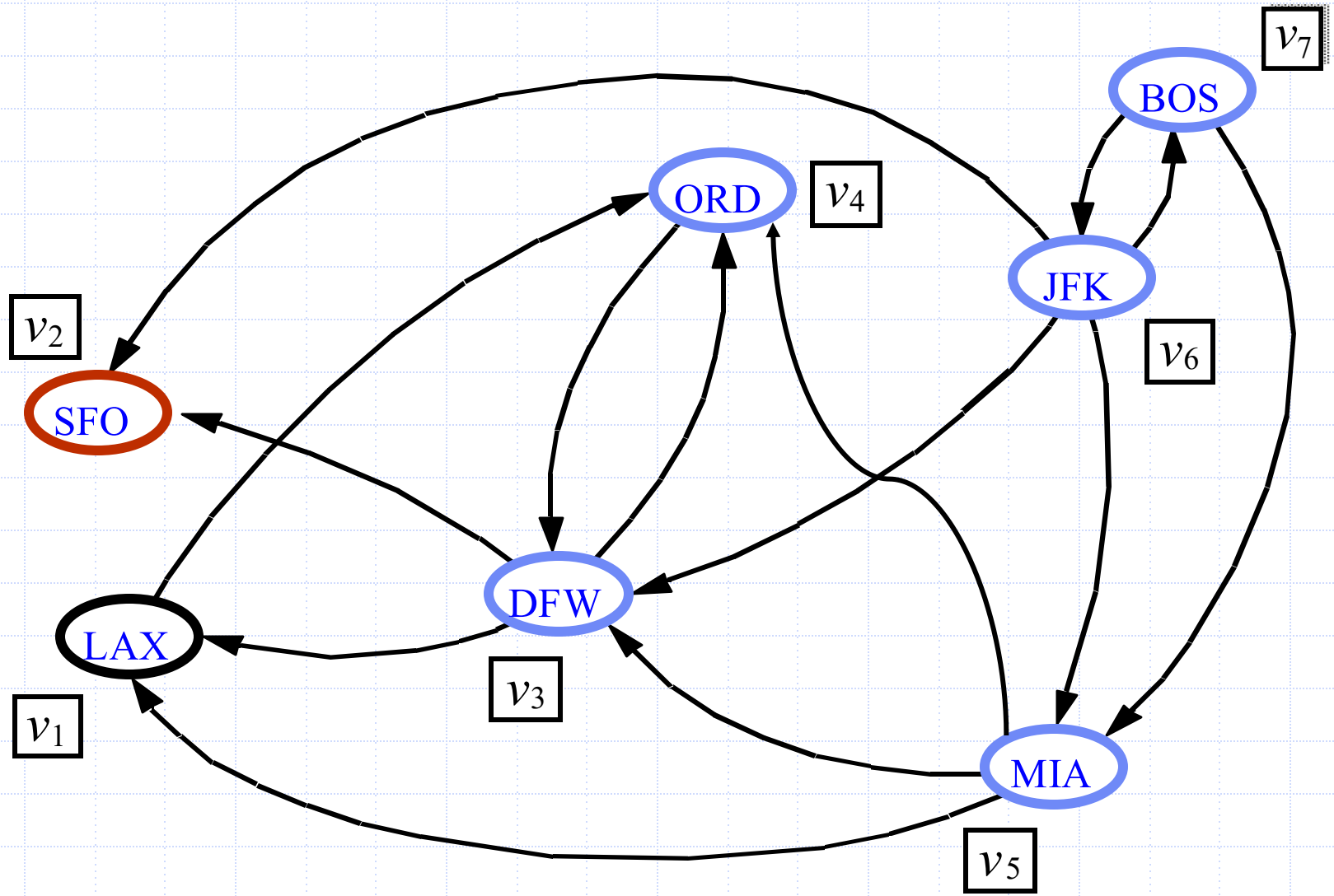
Floyd-Warshall Example



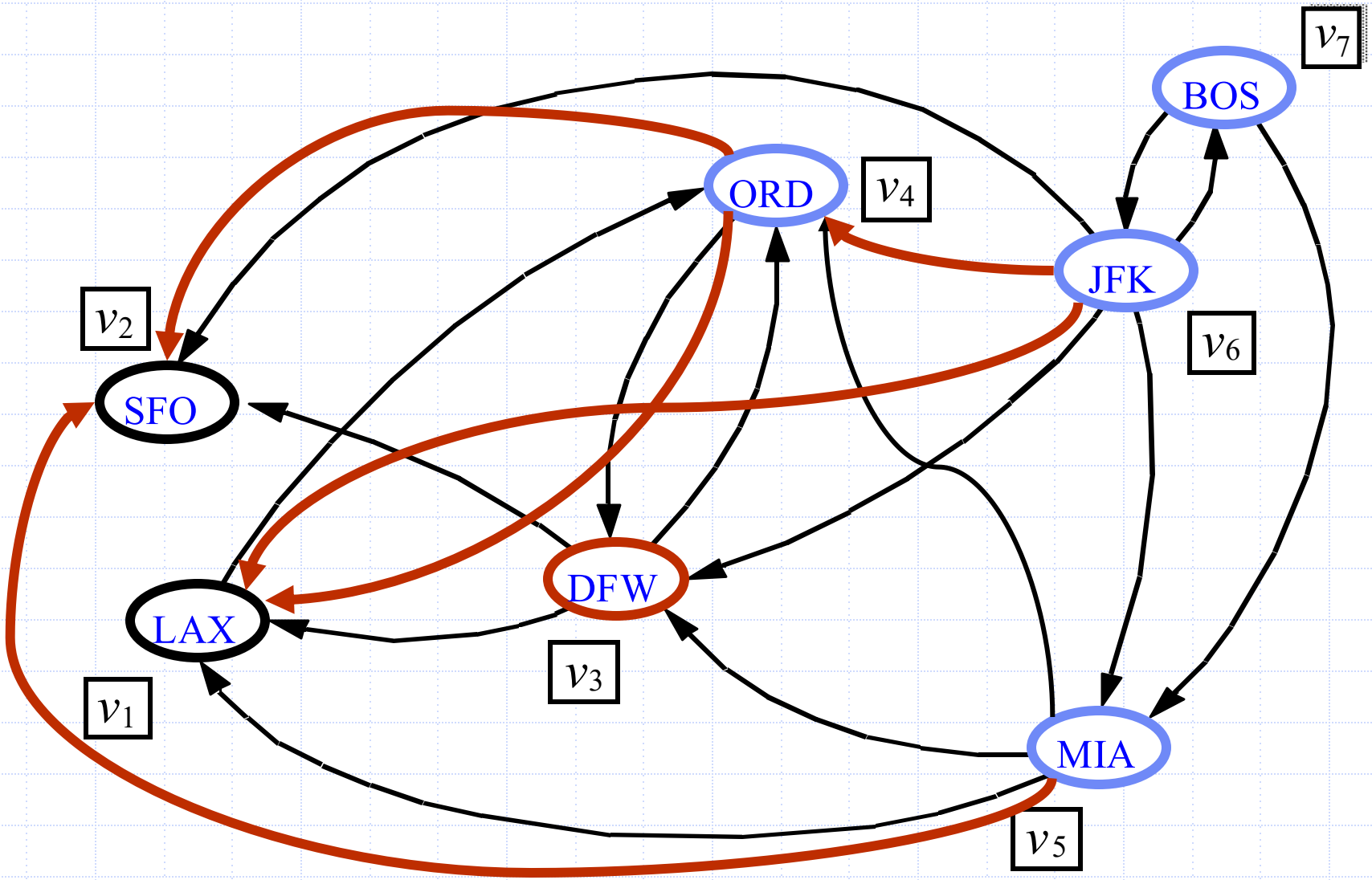
Floyd-Warshall, Iteration 1



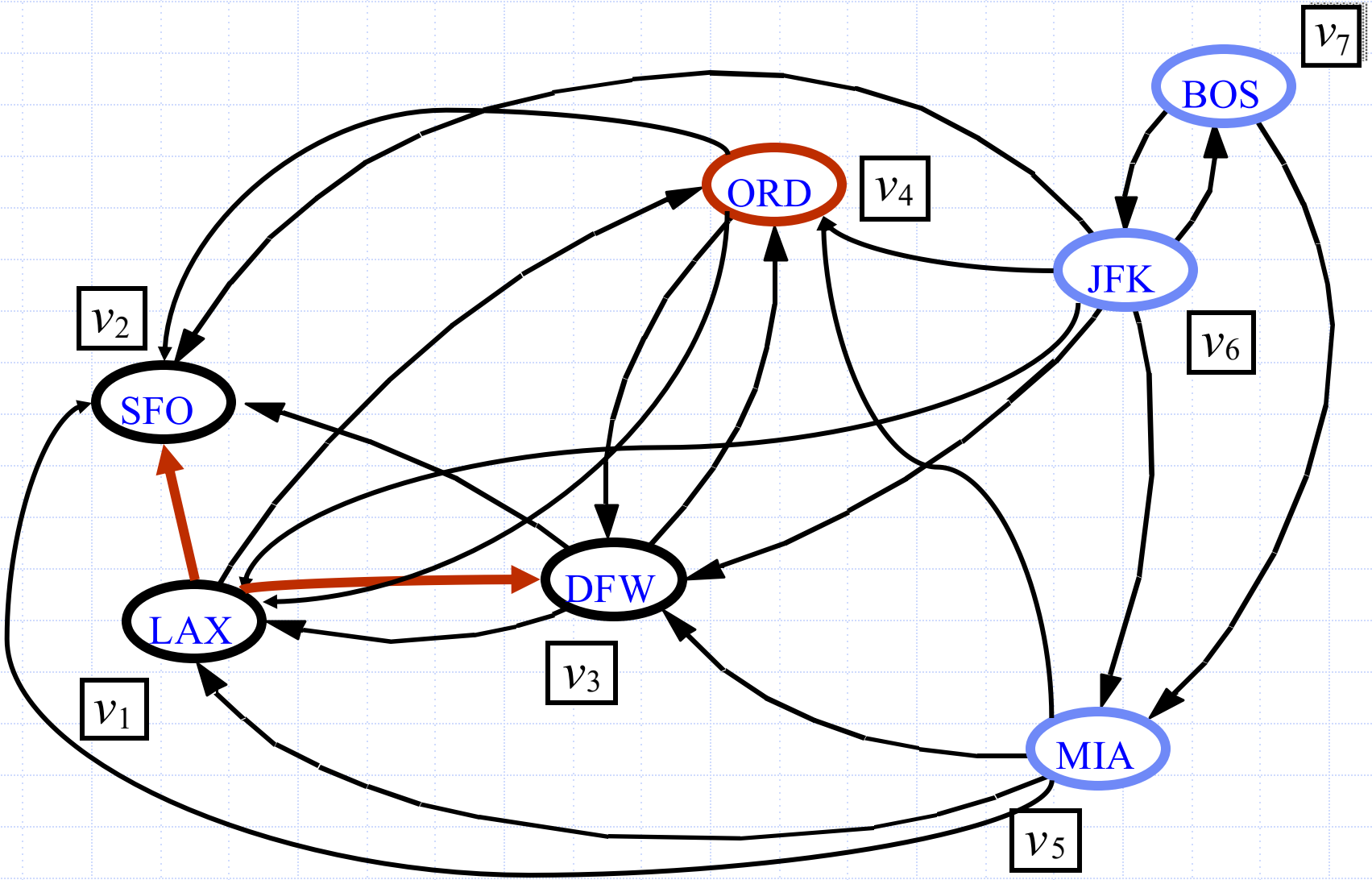
Floyd-Warshall, Iteration 2



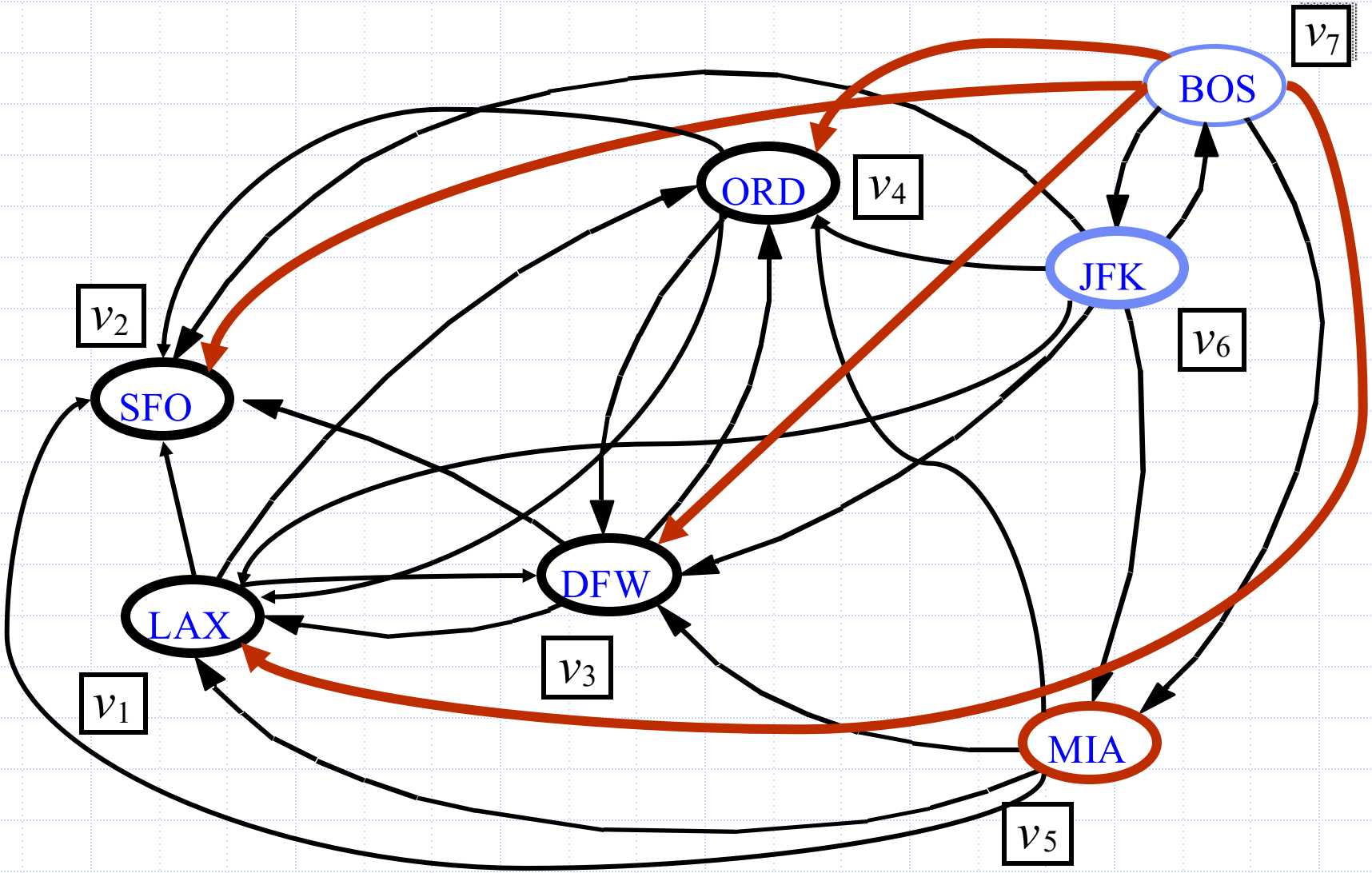
Floyd-Warshall, Iteration 3



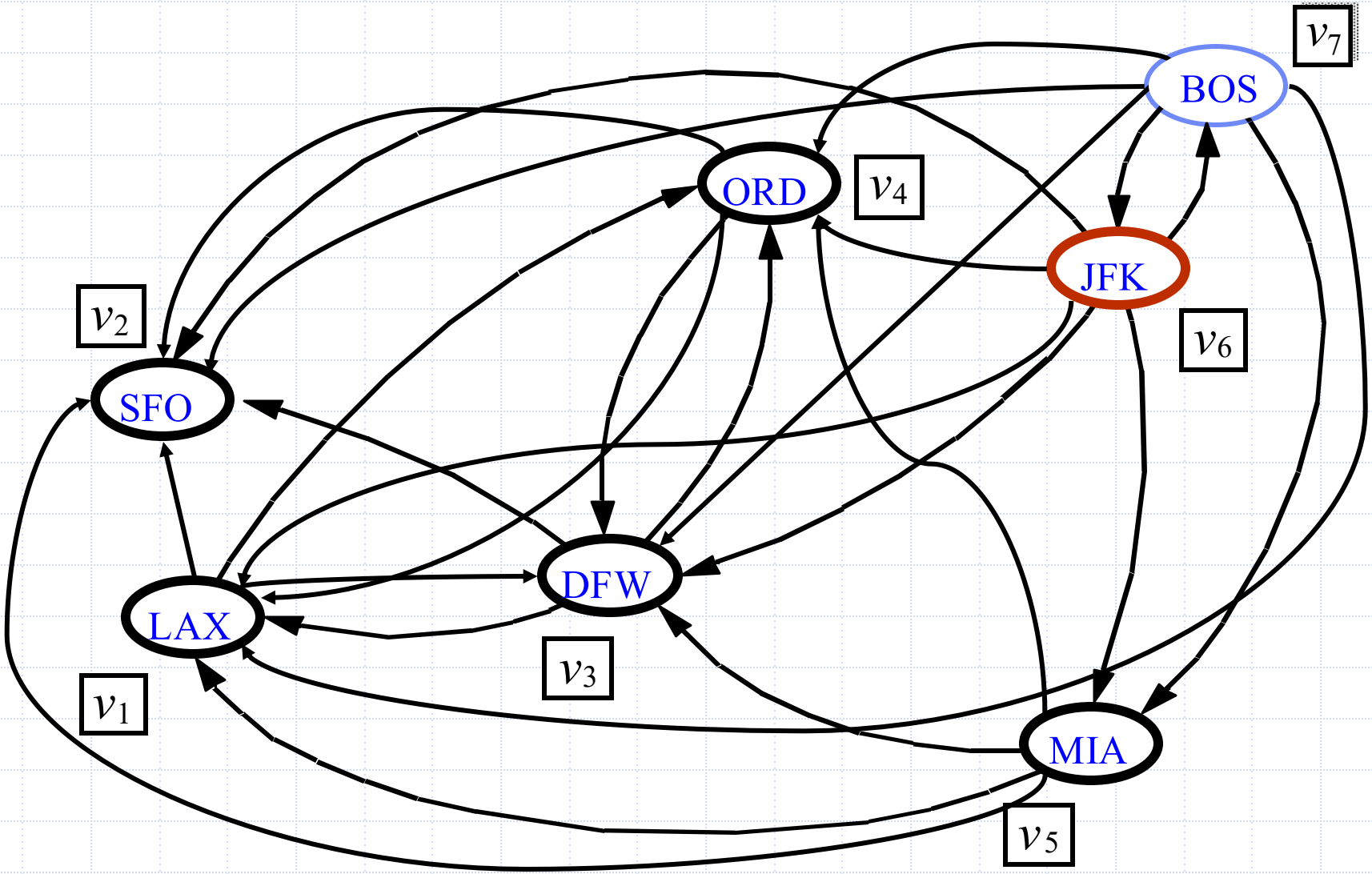
Floyd-Warshall, Iteration 4



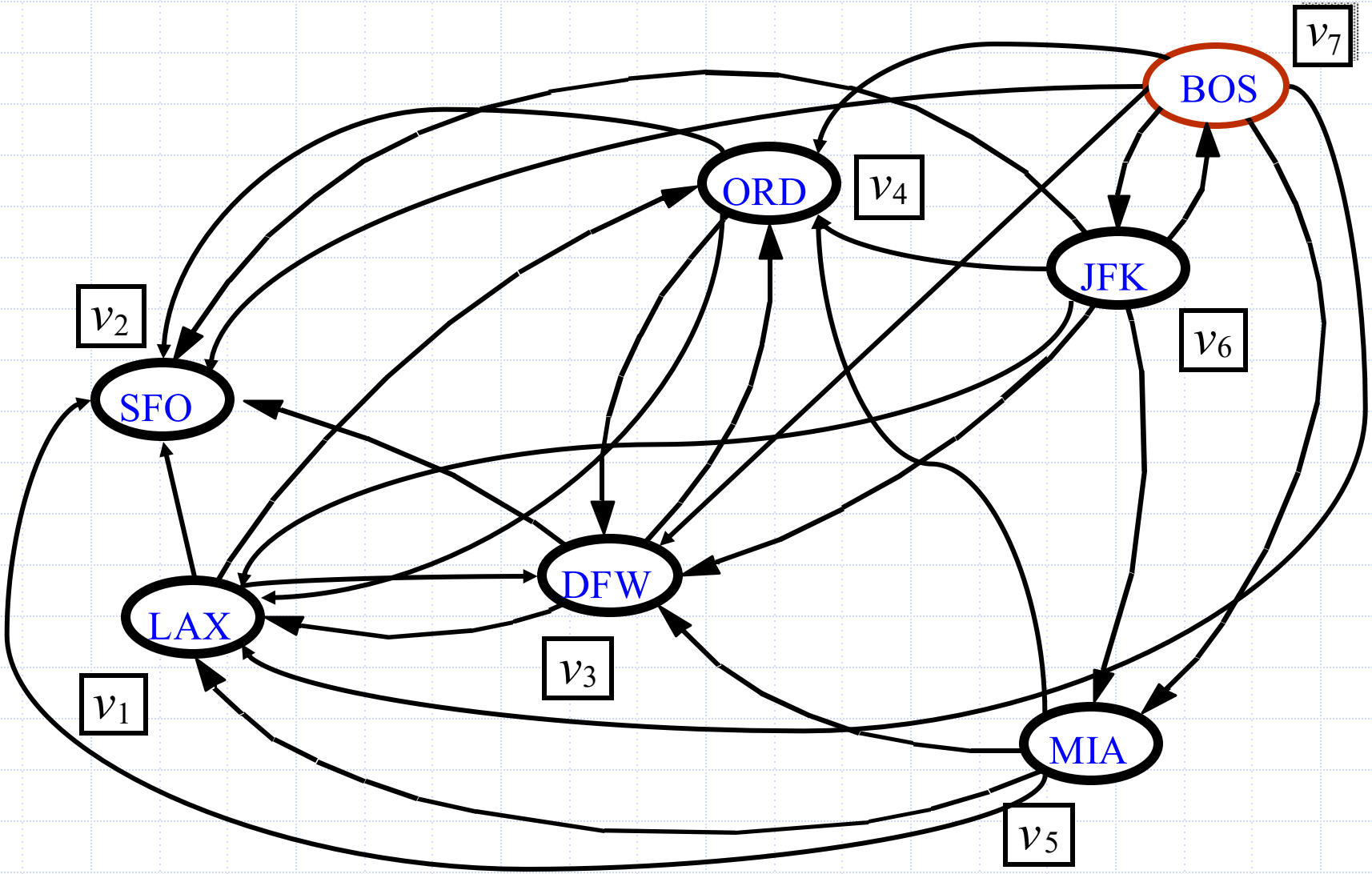
Floyd-Warshall, Iteration 5



Floyd-Warshall, Iteration 6



Floyd-Warshall, Conclusion



Directed Acyclic Graphs

- A directed graph that has no cycle
  - DAG
- If there is a cycle, then there will be a back edge
- If there is no back edge in a graph G, then G is acyclic
  - Do DFS
  - Order vertices by departure time

Directed DFS

- Forward edge (u, v)

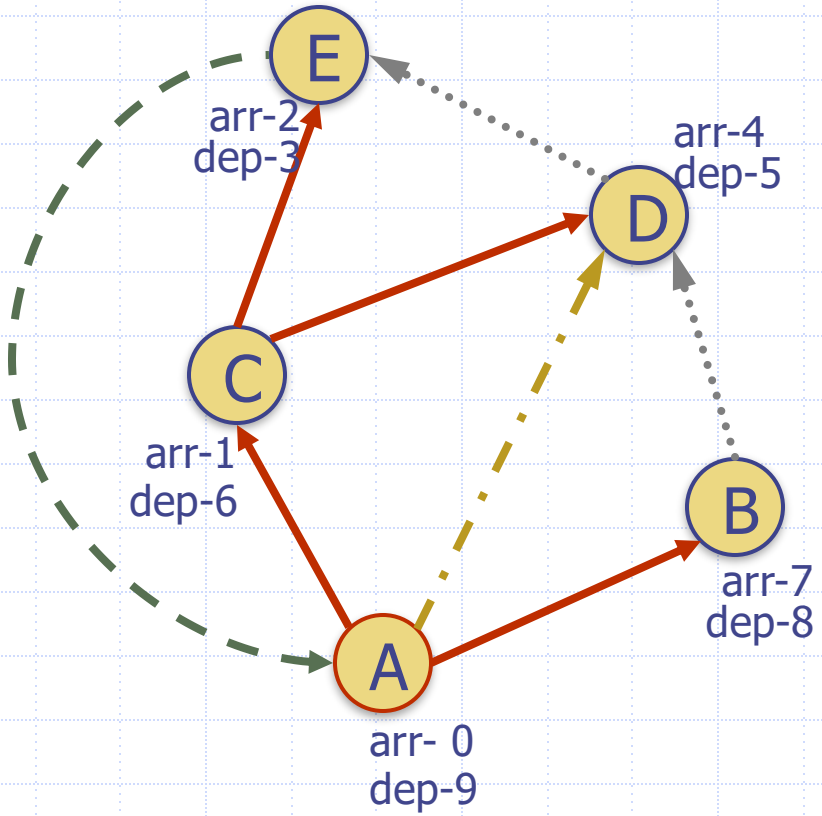
arr(u)<arr(v)<dep(v)<dep(u)

- Back edge (u, v)

arr(v)<arr(u)<dep(u)<dep(v)

- Cross edge (u, v)

arr(v)<dep(v)<arr(u)<dep(u)



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

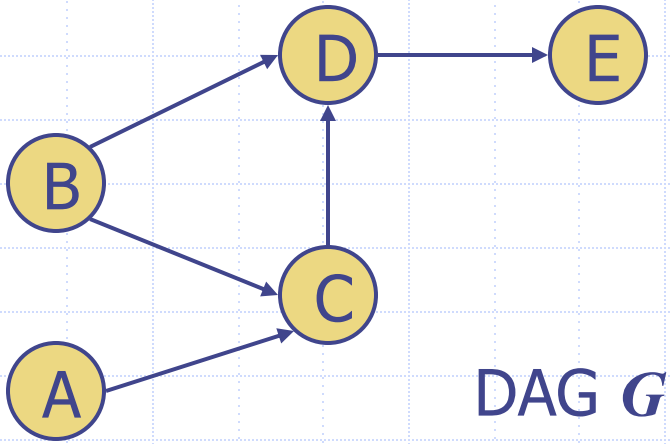
$$v_1, \ldots, v_n$$

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

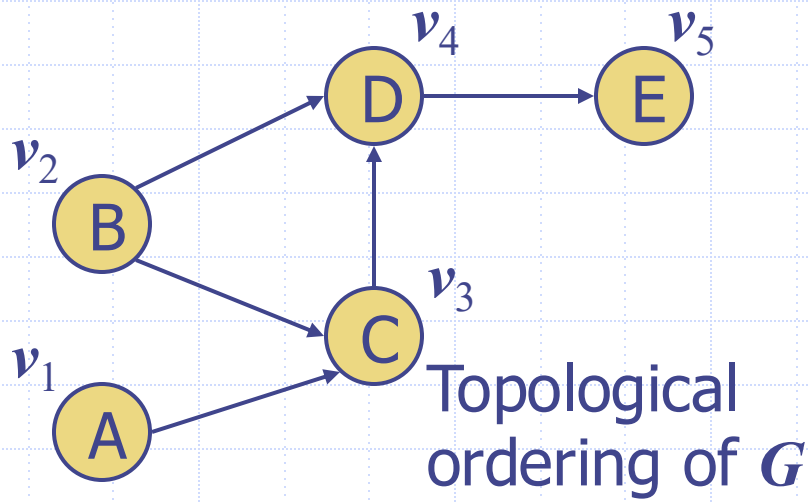
- Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints

Theorem

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



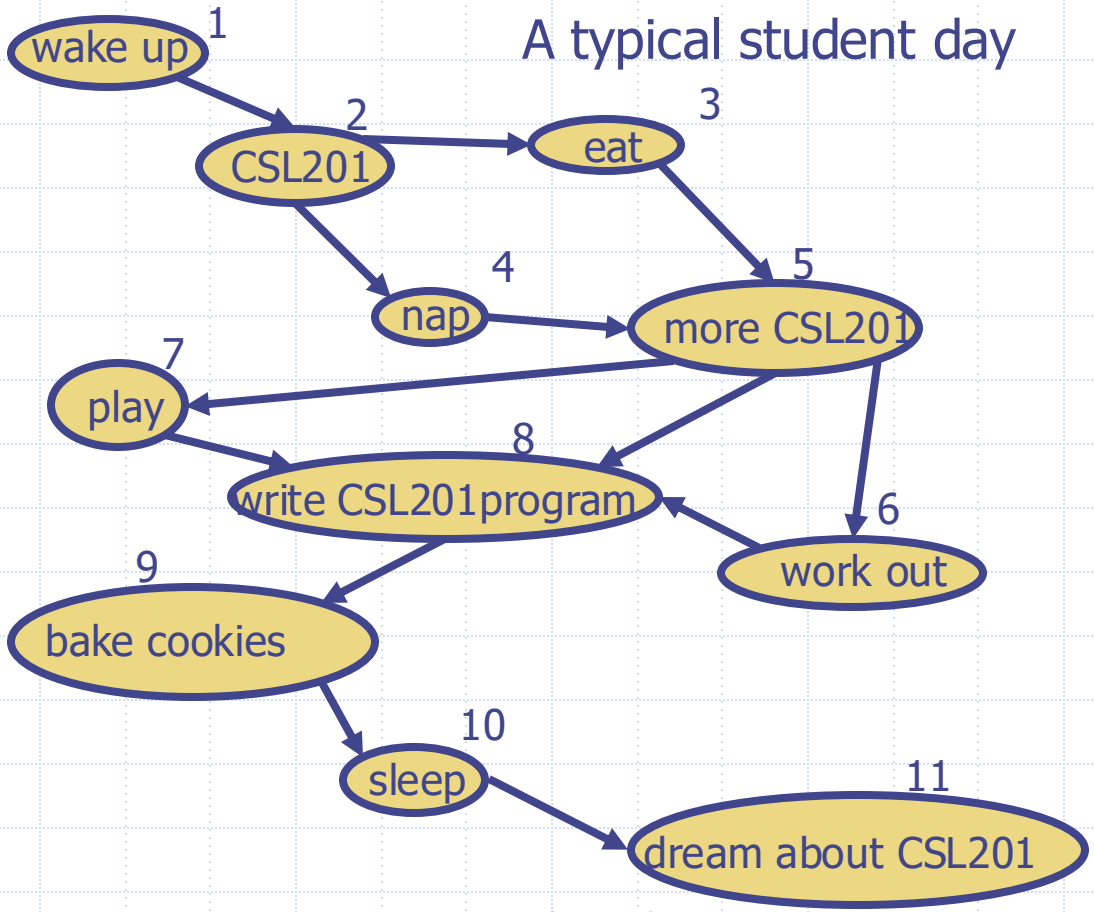
DAG G



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 the book

```
Algorithm 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)

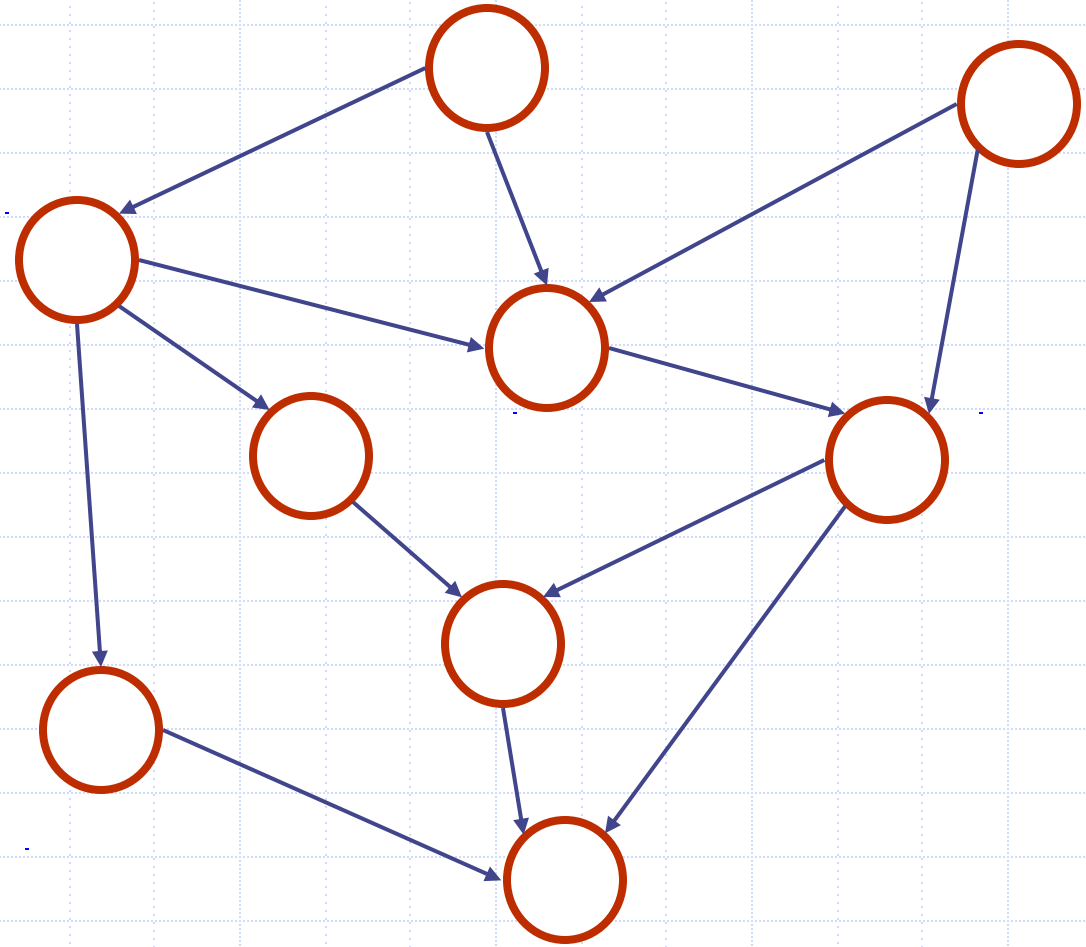
Implementation with DFS

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

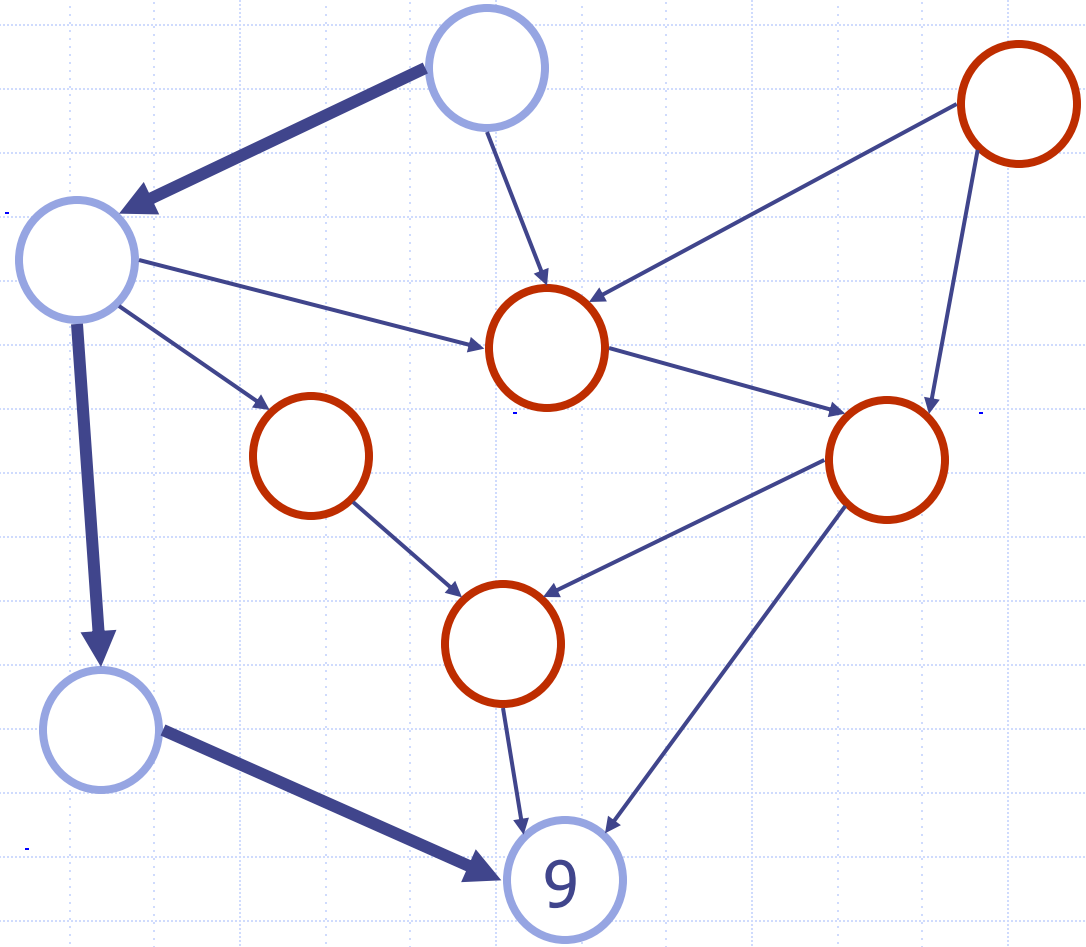
```
Algorithm topologicalDFS(G)
    Input dag G
    Output topological ordering of G
  n ← G.numVertices()
  for all u ∈ G.vertices()
      setLabel(u, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
        topologicalDFS(G, v)
```

```
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
  setLabel(v, VISITED)
  for all e ∈ G.outEdges(v)
      { outgoing edges }
    w ← opposite(v,e)
    if getLabel(w) = UNEXPLORED
        { e is a discovery edge }
        topologicalDFS(G, w)
    else
        { e is a forward or cross edge }
  Label v with topological number n
  n ← n - 1
```

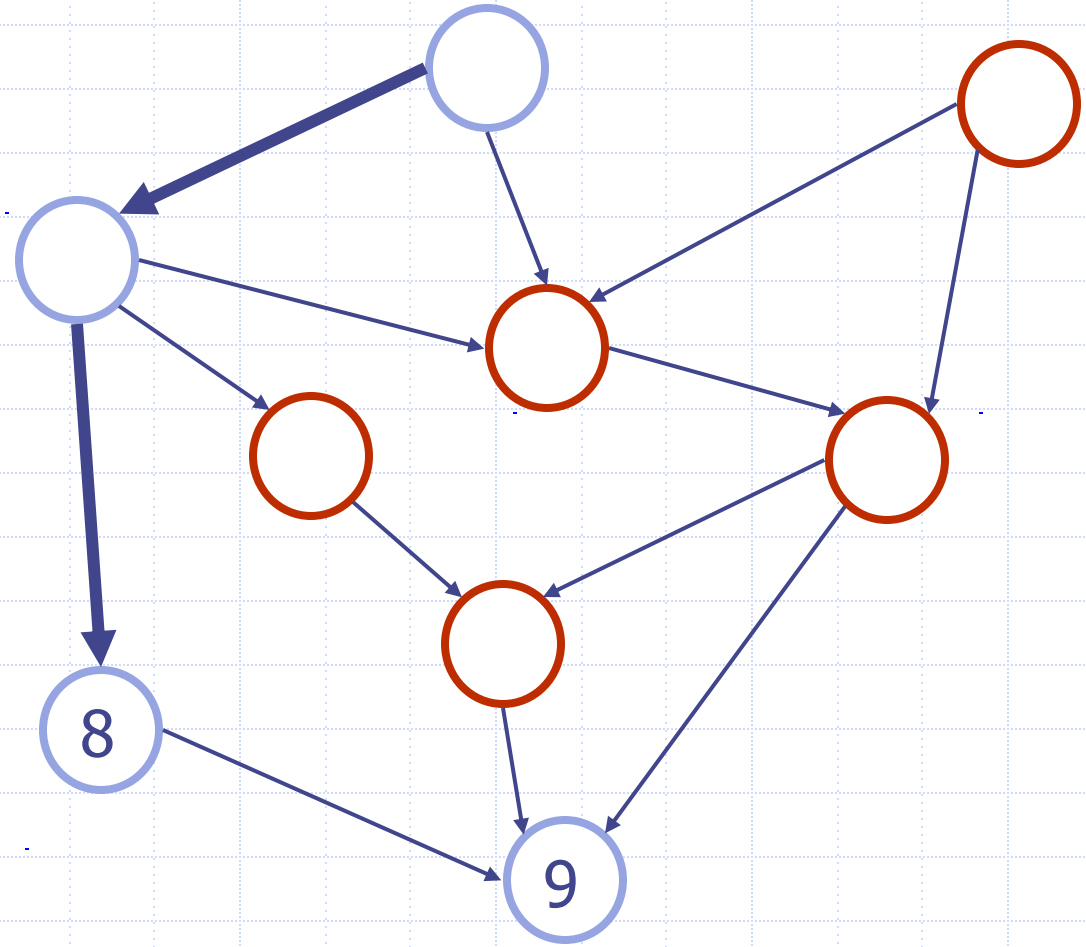
Topological Sorting Example



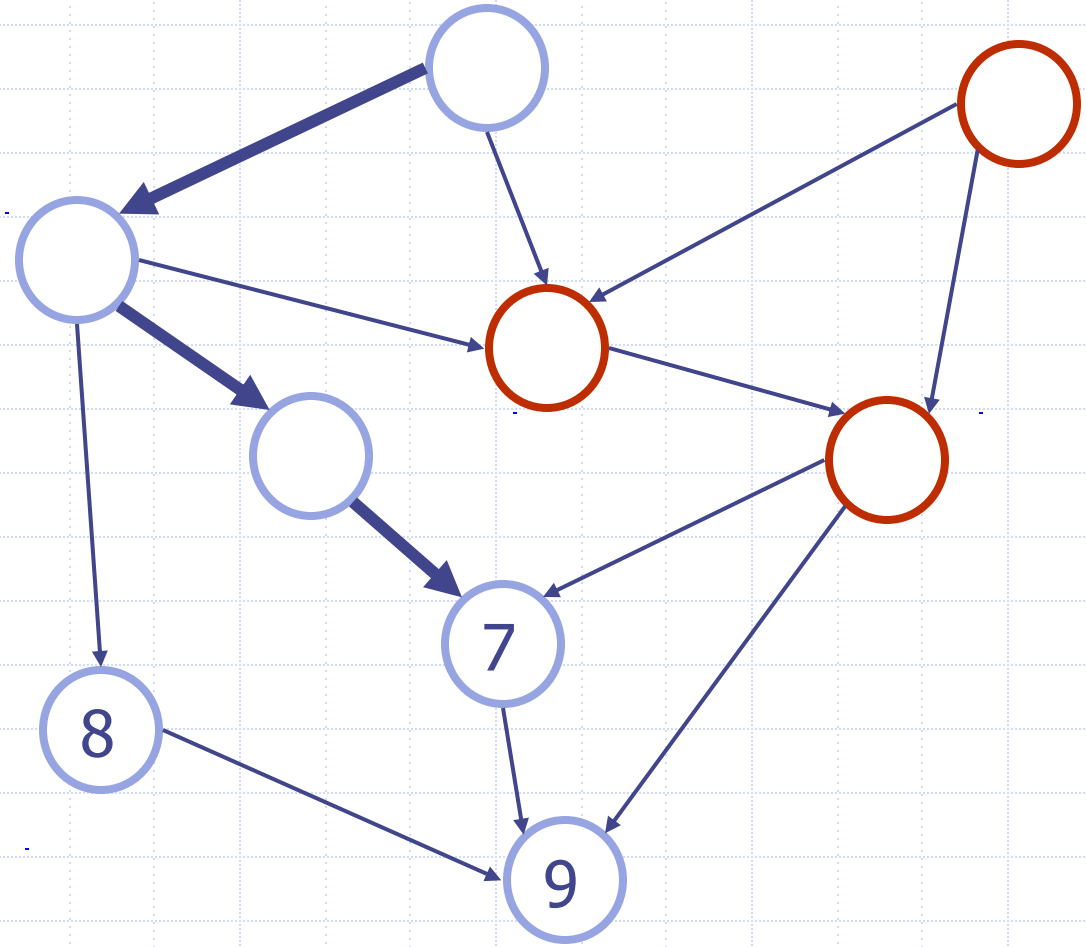
Topological Sorting Example



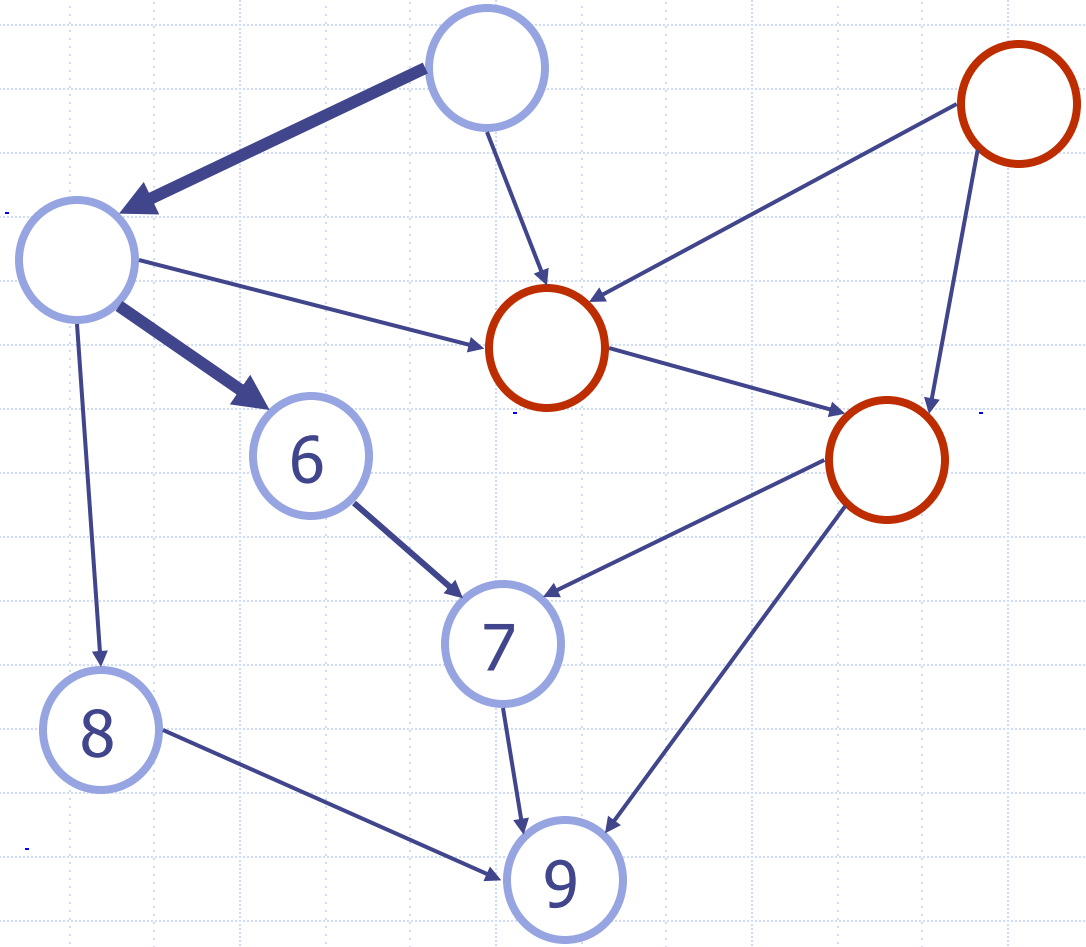
Topological Sorting Example



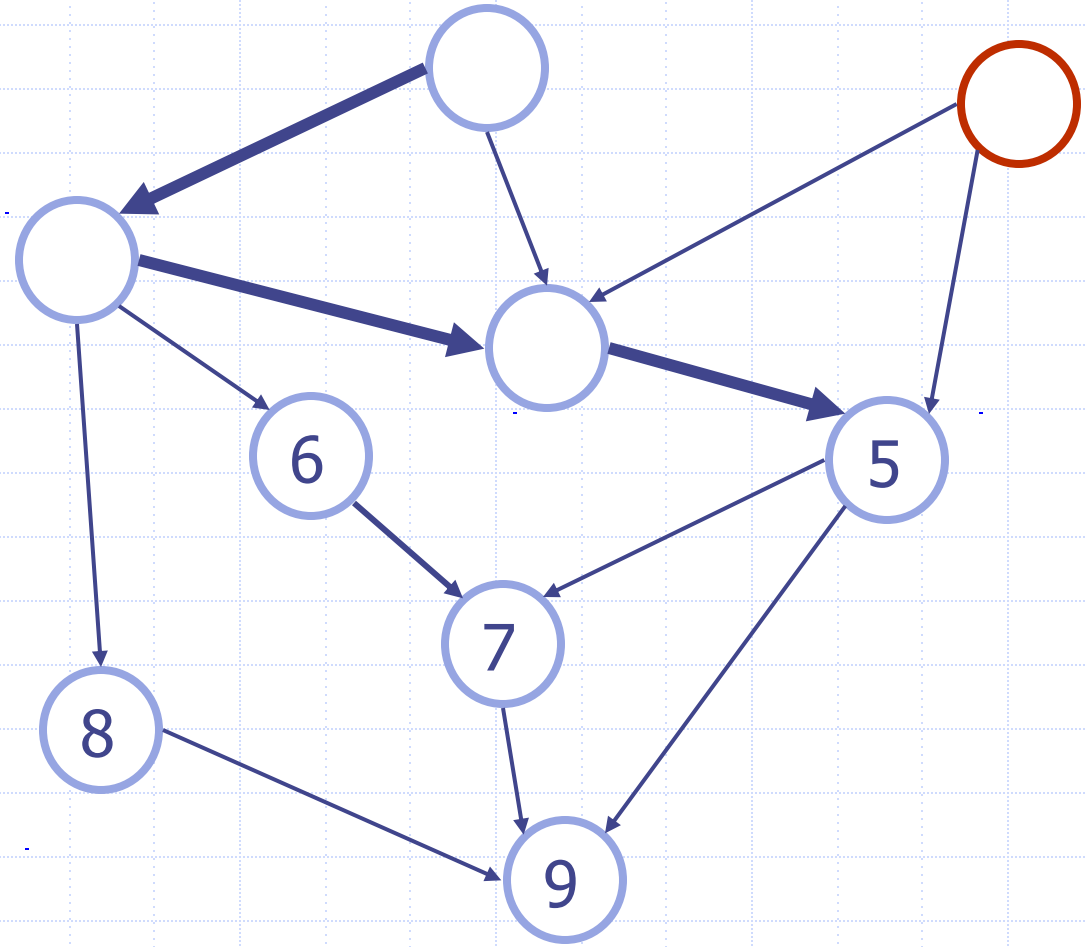
Topological Sorting Example



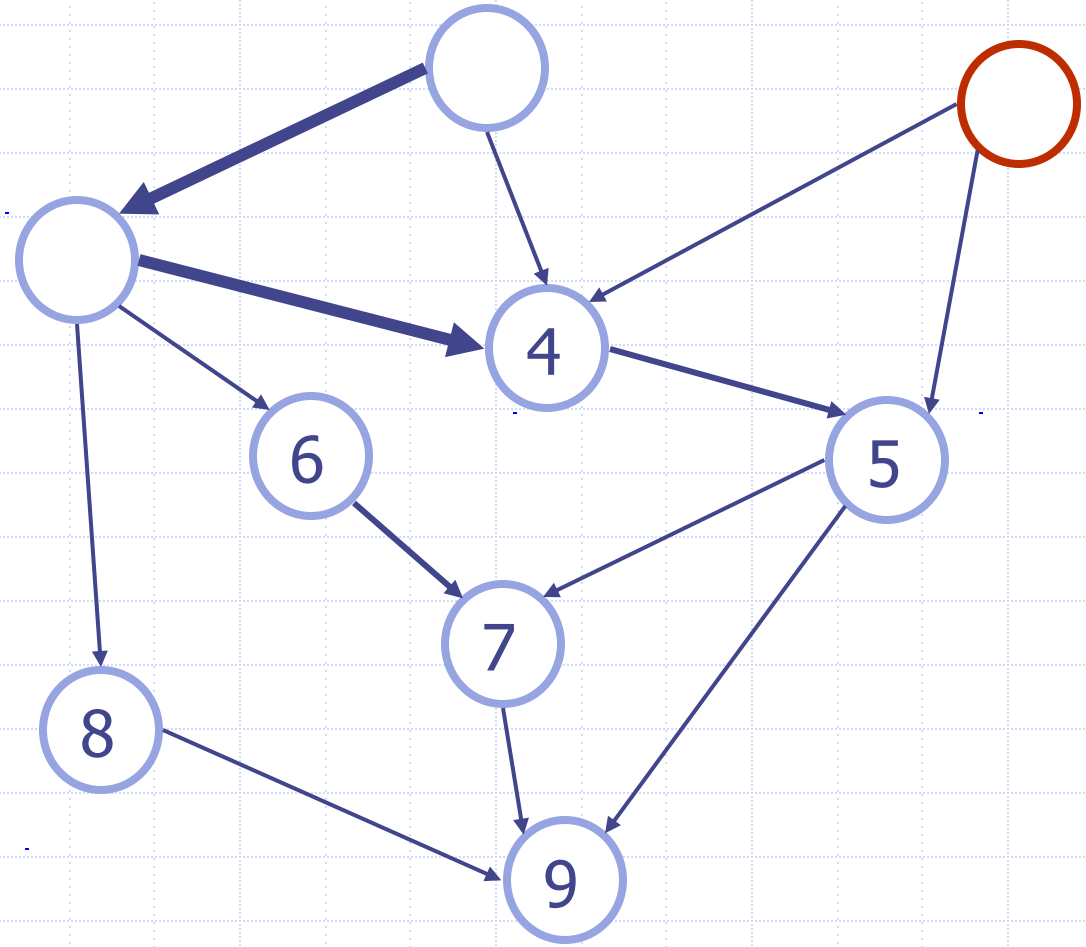
Topological Sorting Example



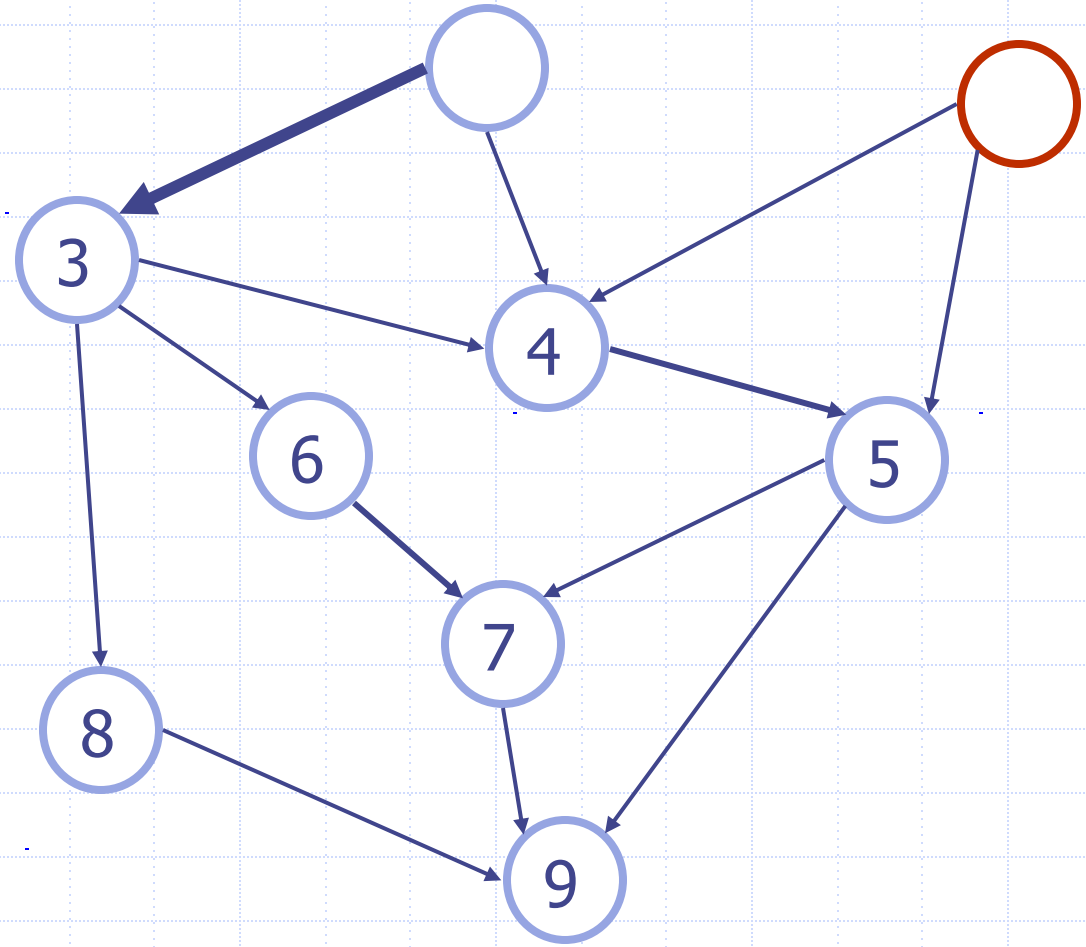
Topological Sorting Example



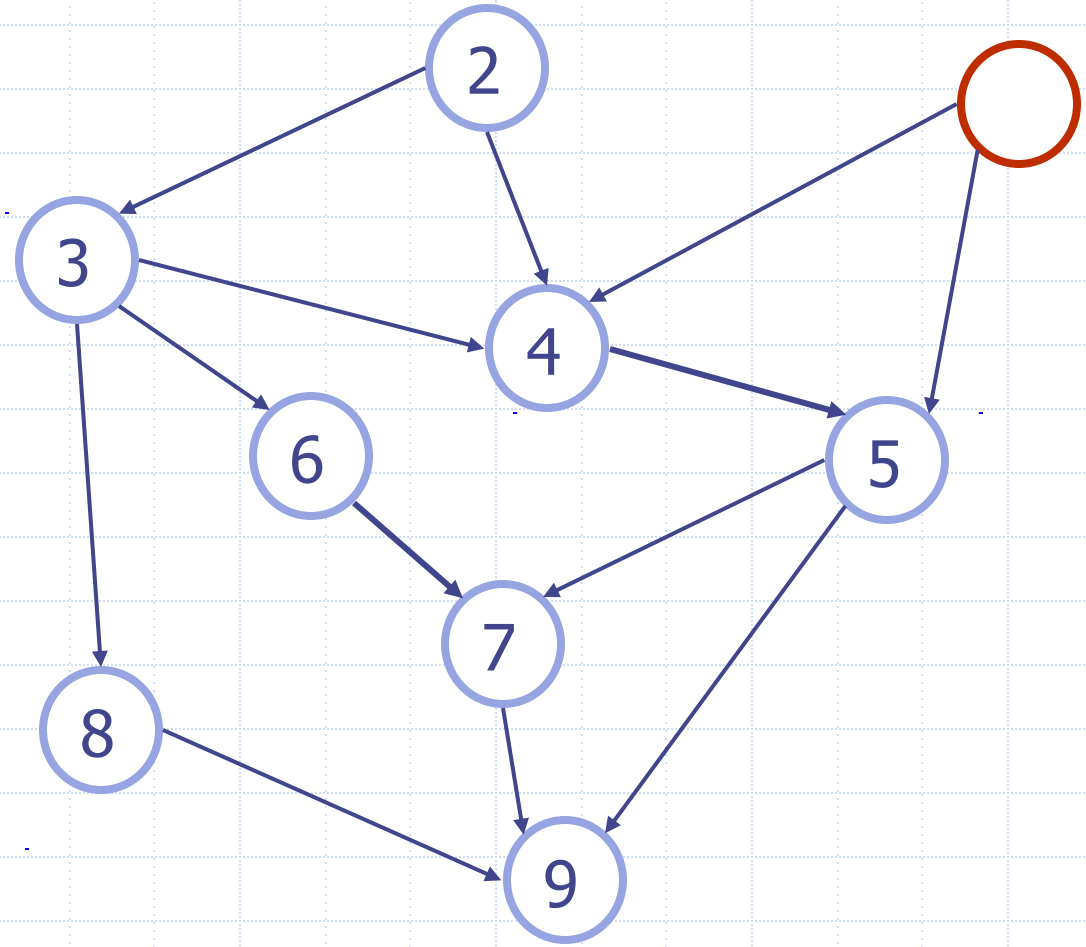
Topological Sorting Example



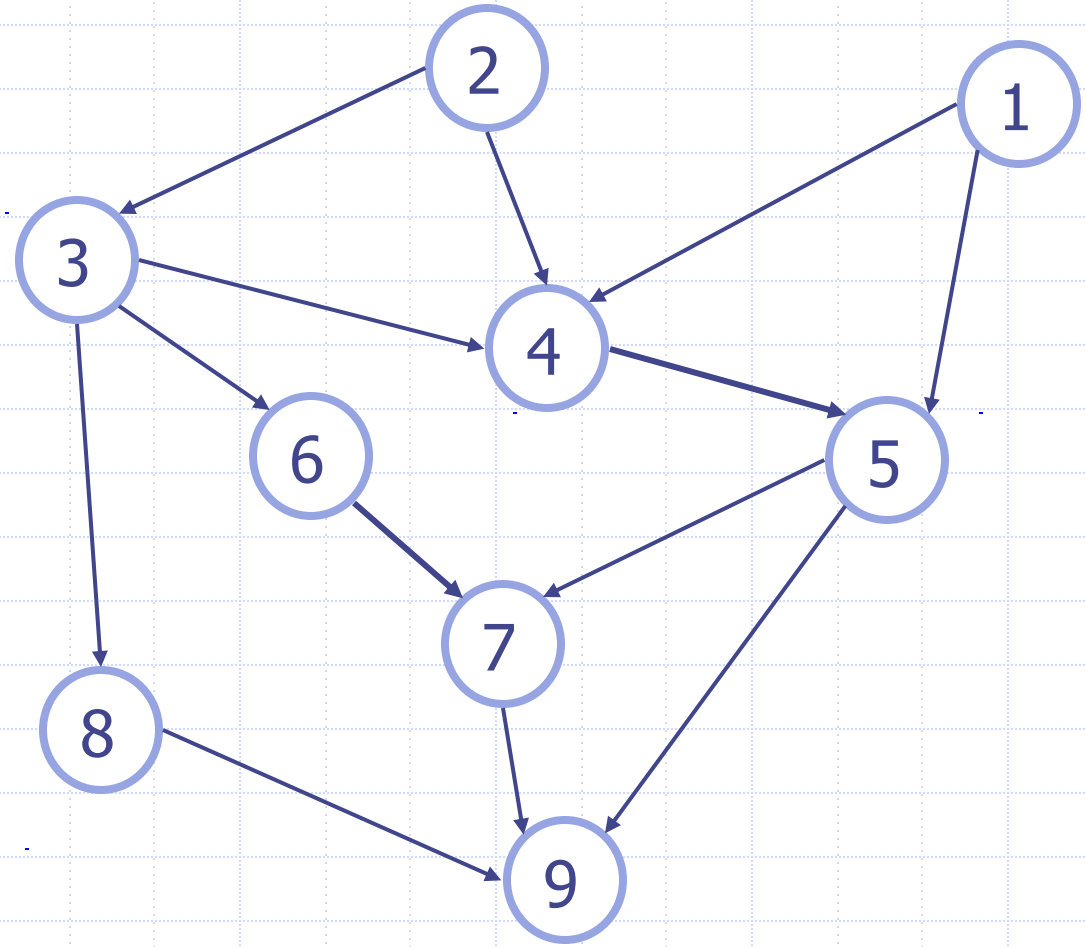
Topological Sorting Example



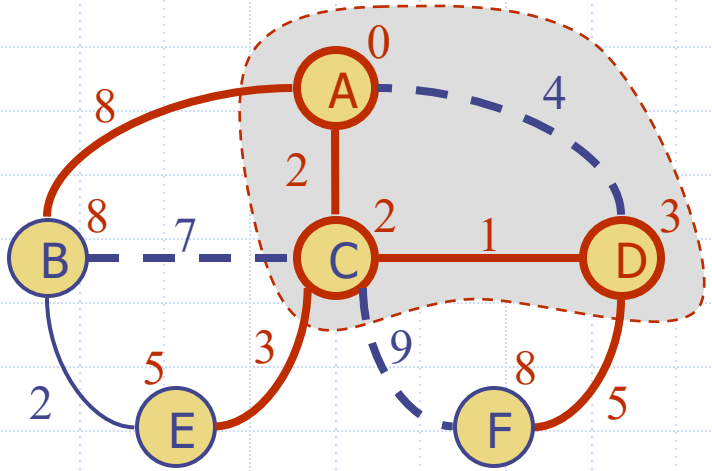
Topological Sorting Example



Topological Sorting Example

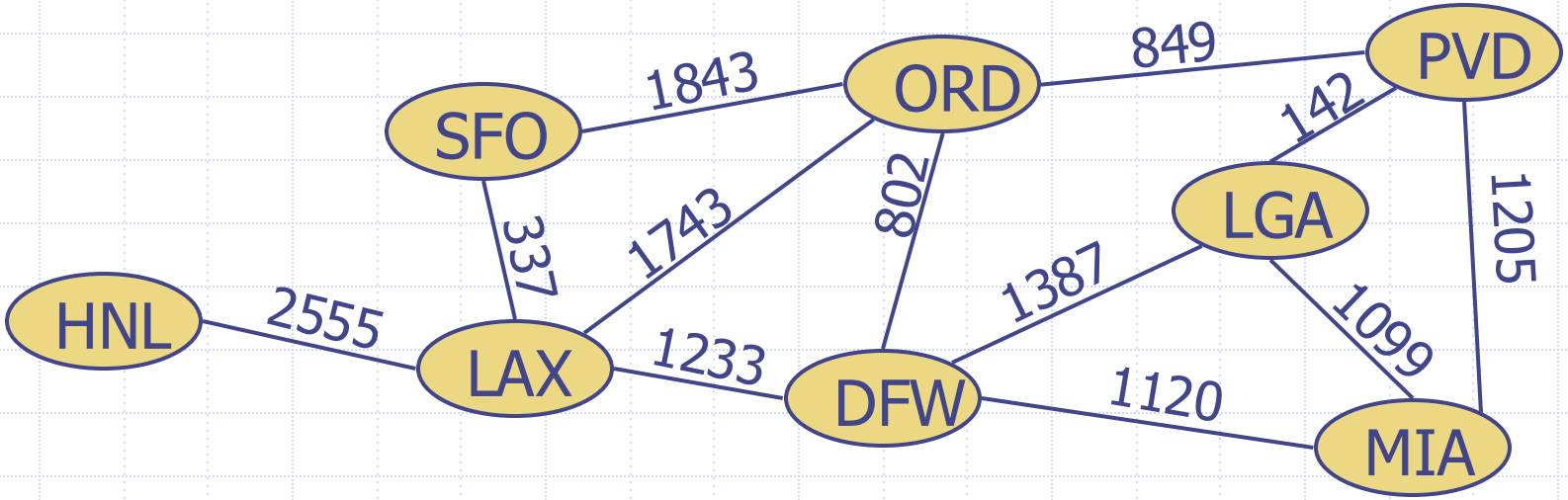


Shortest Paths



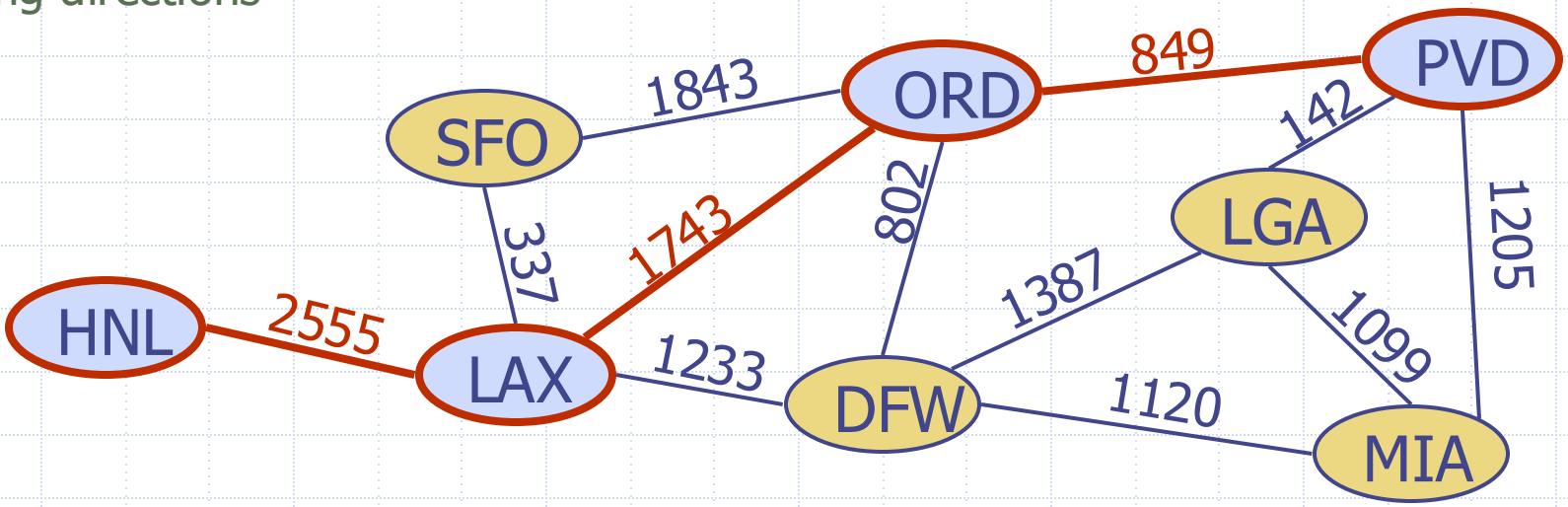
Weighted Graphs

- 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:

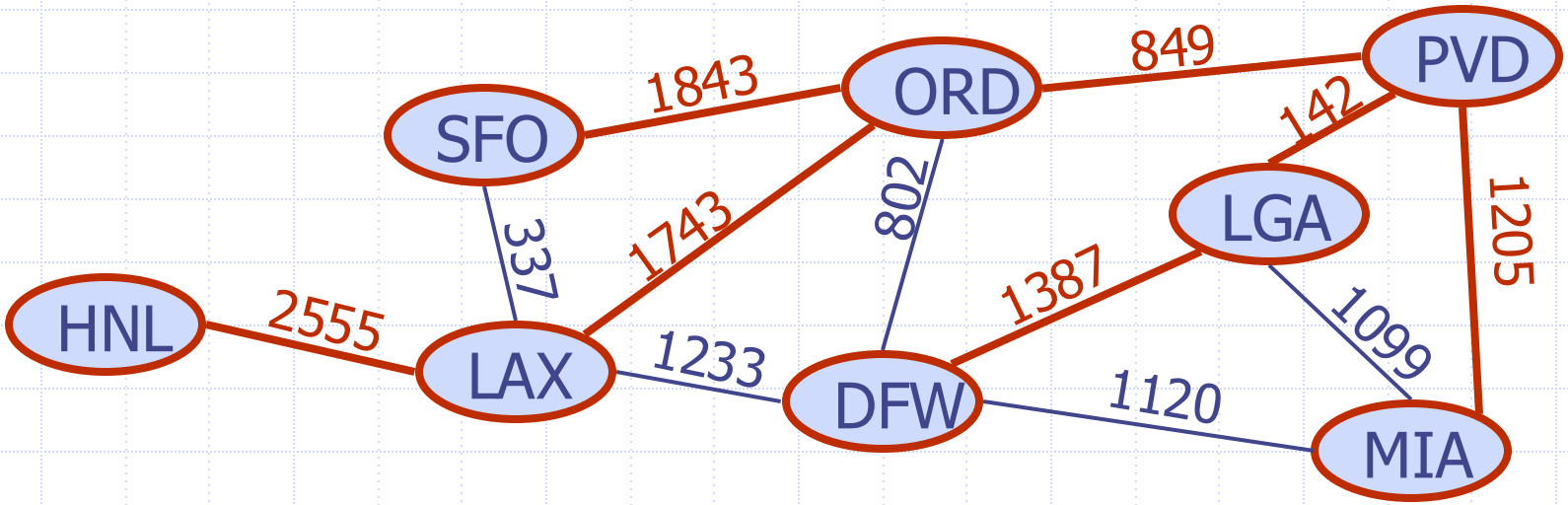
A subpath of a shortest path is itself a shortest path

Property 2:

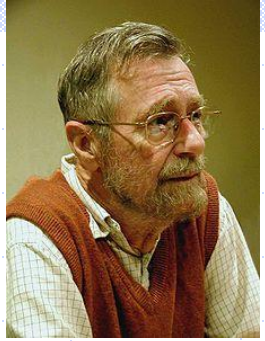
There is a tree of shortest paths from a start vertex to all the other vertices

Example:

Tree of shortest paths from Providence

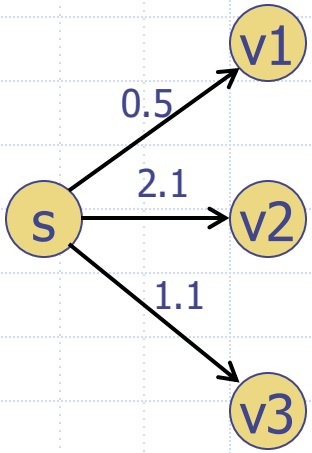


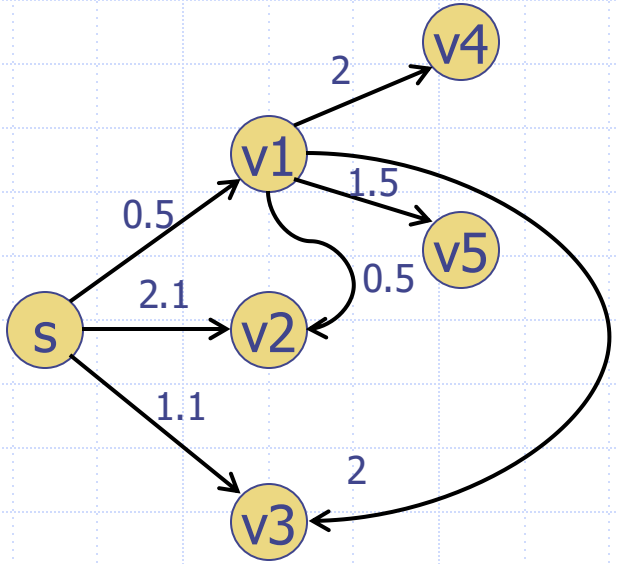
Dijkstra's Algorithm



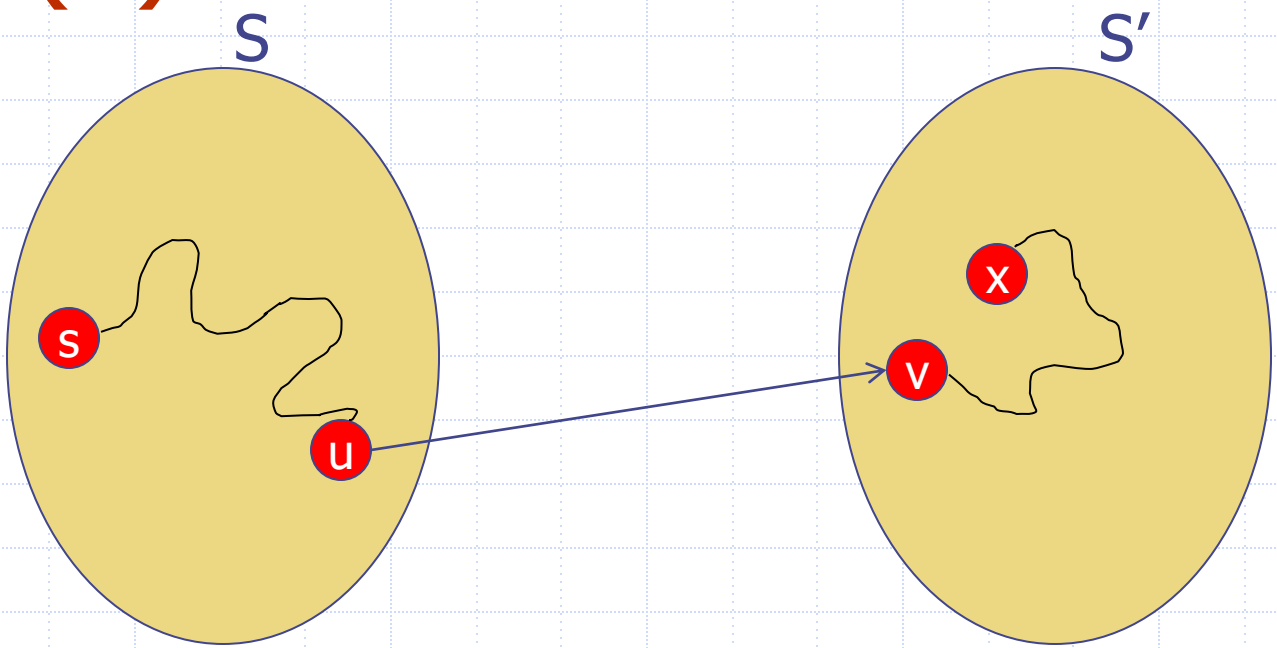
- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative
- We 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

Principle





Principle (2)

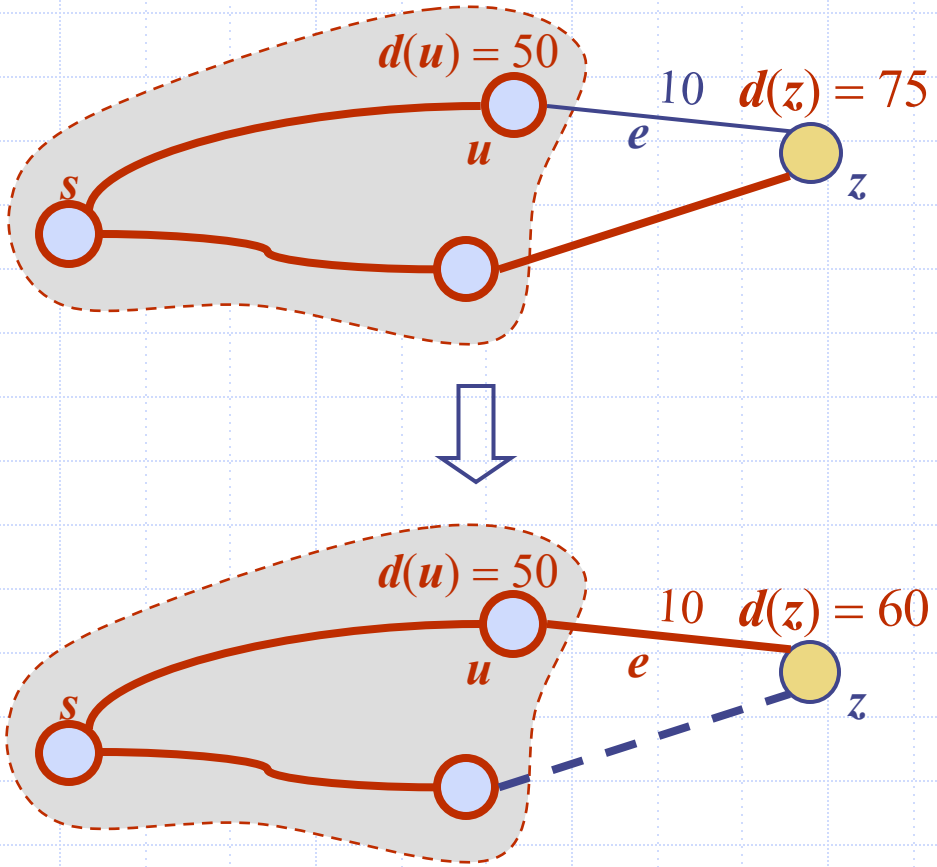


- For all vertices u in S, d[u] is the length of the shortest path from s to u.
- For all vertices in S′, d[x] is the upper bound on the length of the shortest path from s to x.
- Move from S′ to S vertex with minimum d[] value.

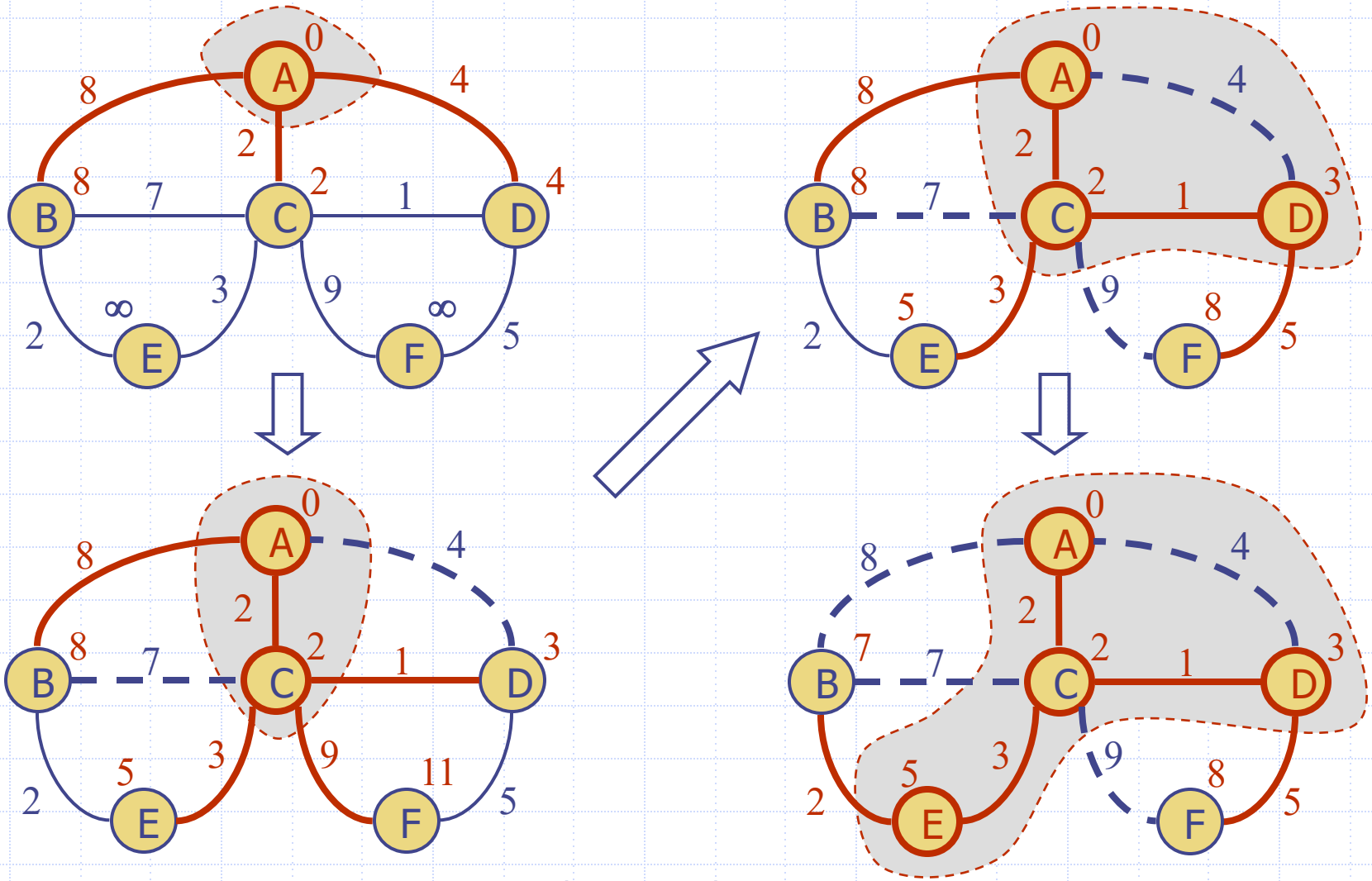
Edge Relaxation

- 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:

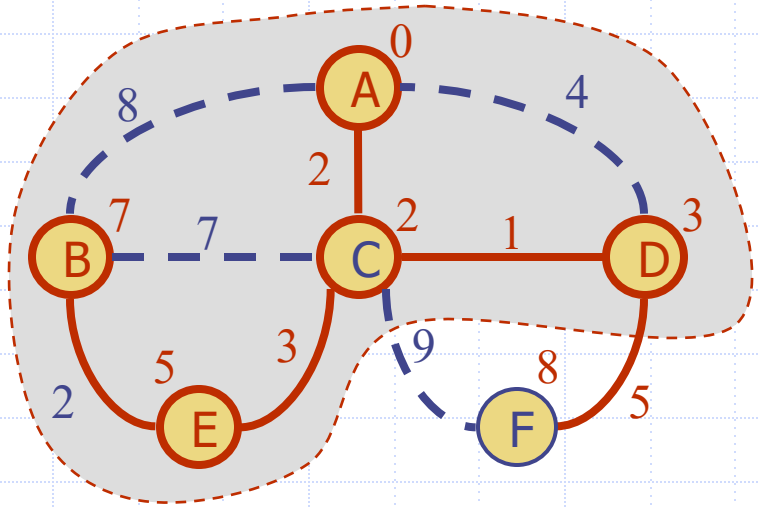
$$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$$

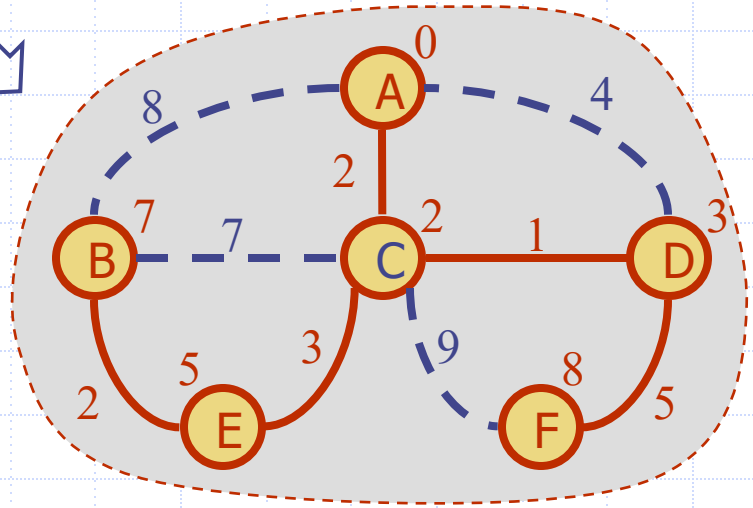


Example

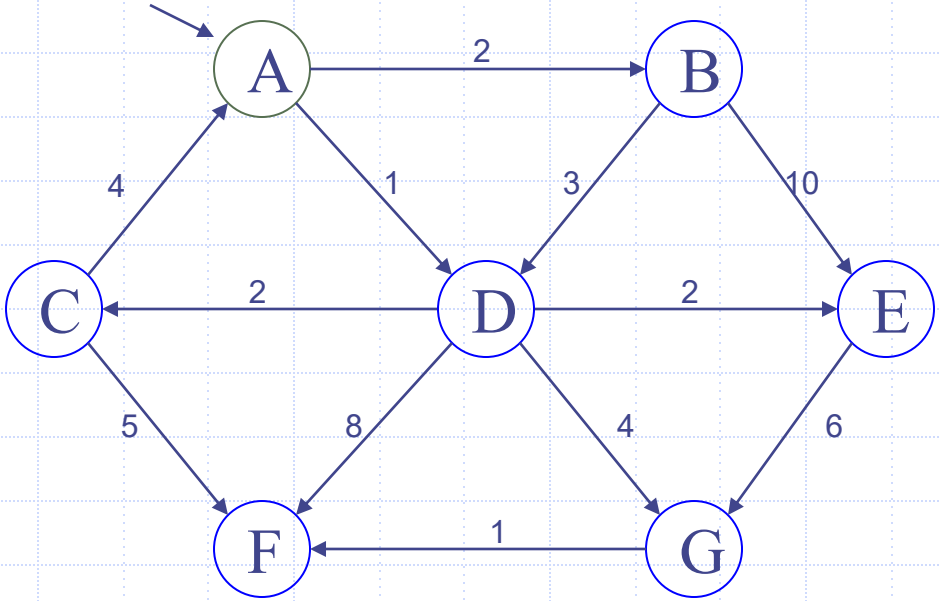

A
B
C
D
E
F
0
8
4
2
8
7
2
1
4
2
∞
3
9
∞
5
A
B
C
D
E
F
0
8
4
2
8
7
2
1
3
2
5
3
9
8
5
A
B
C
D
E
F
0
8
4
2
8
7
2
1
3
2
5
3
9
11
5
A
B
C
D
E
F
0
8
4
2
7
7
2
1
3
2
5
3
9
8
5

Example (cont.)



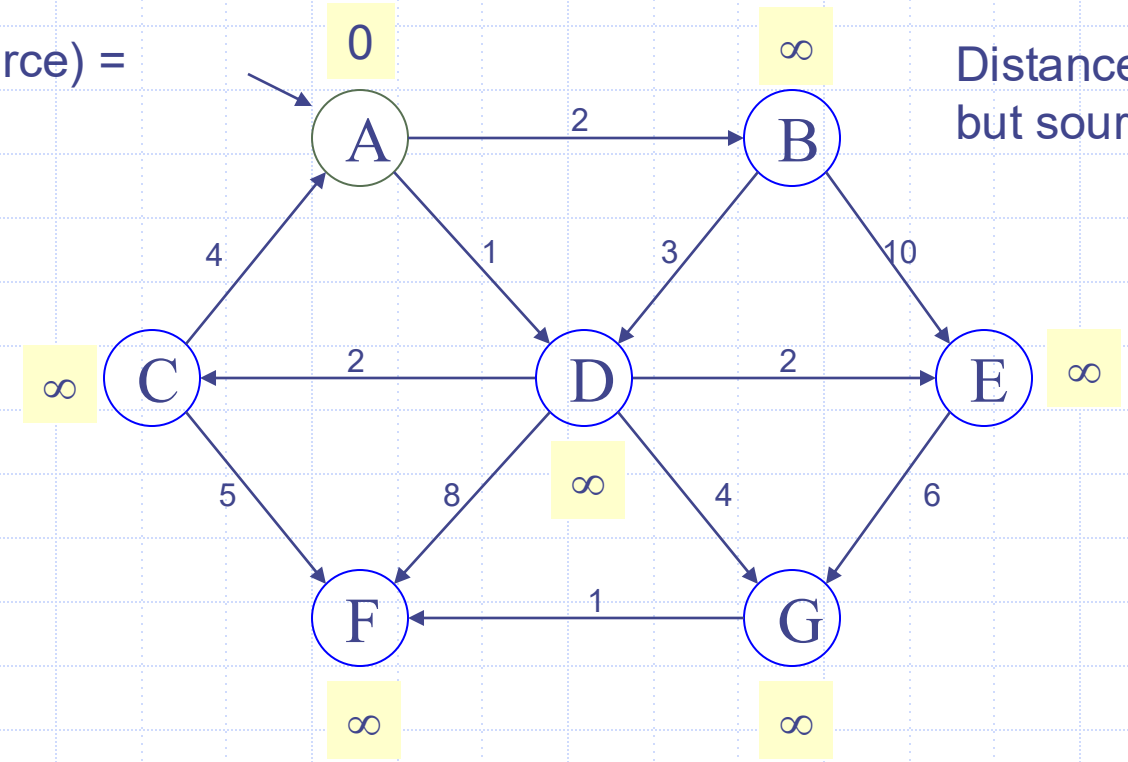


Example (2)



Example (2)

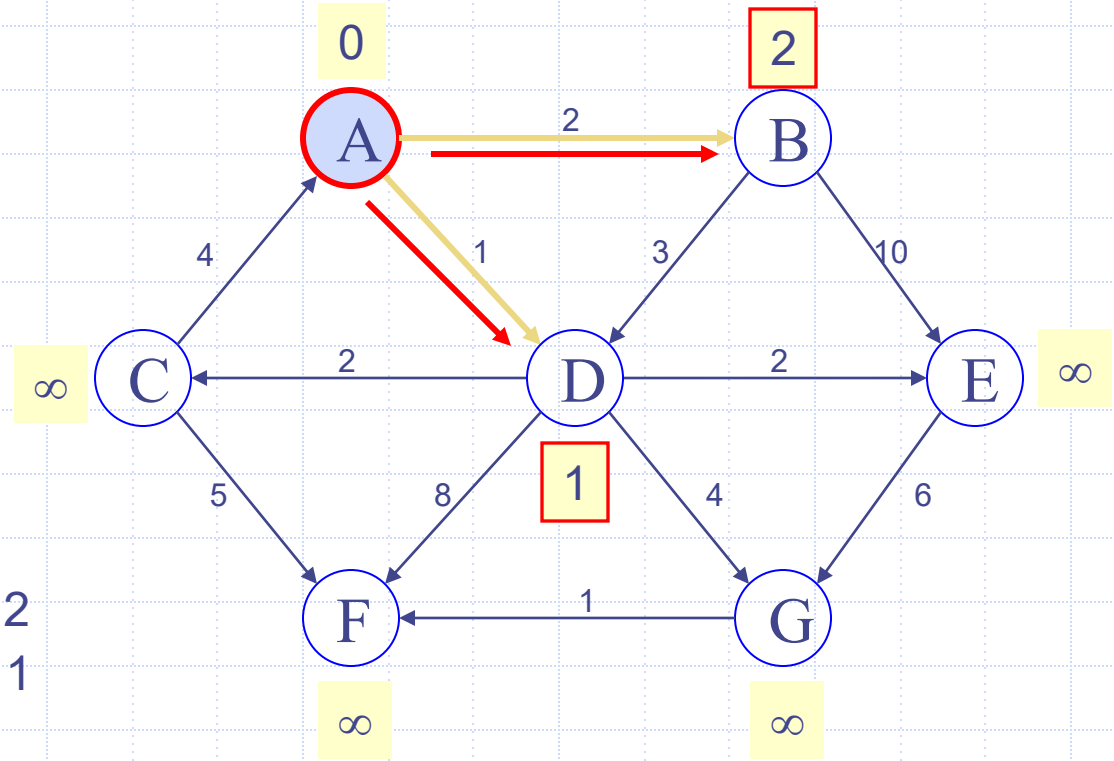
Distance(source) = 0



Distance (all vertices but source) = ∞

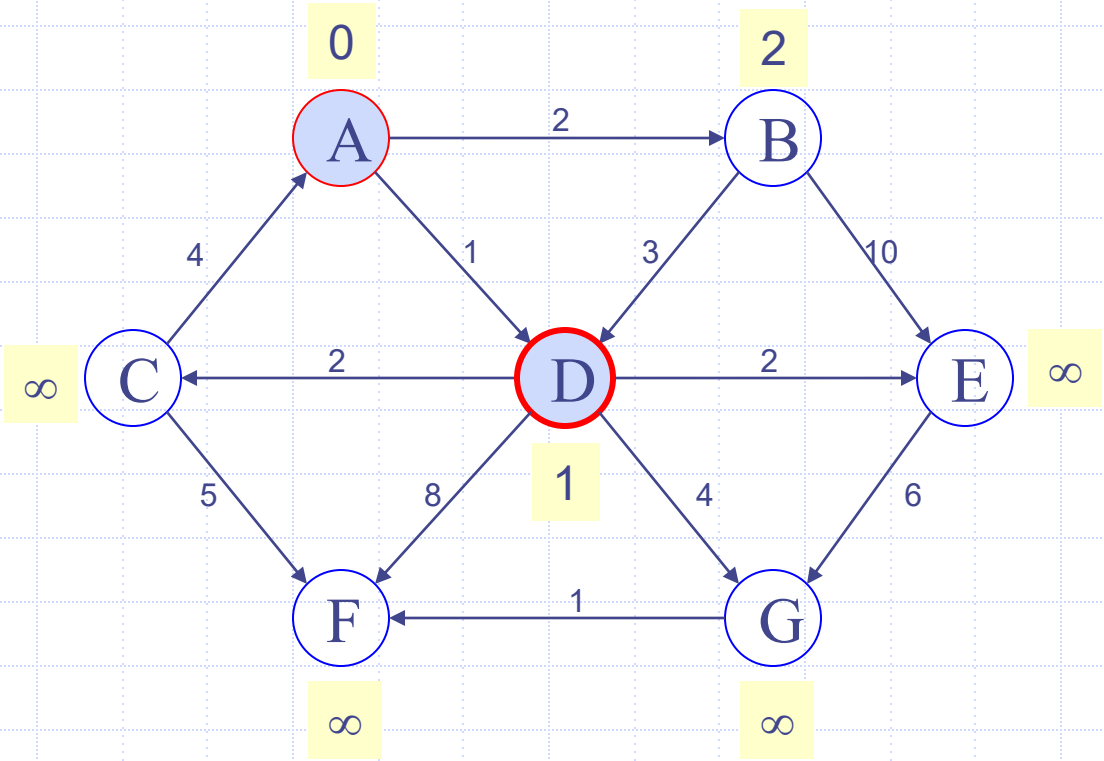
Pick vertex in List with minimum distance.

Example (2)



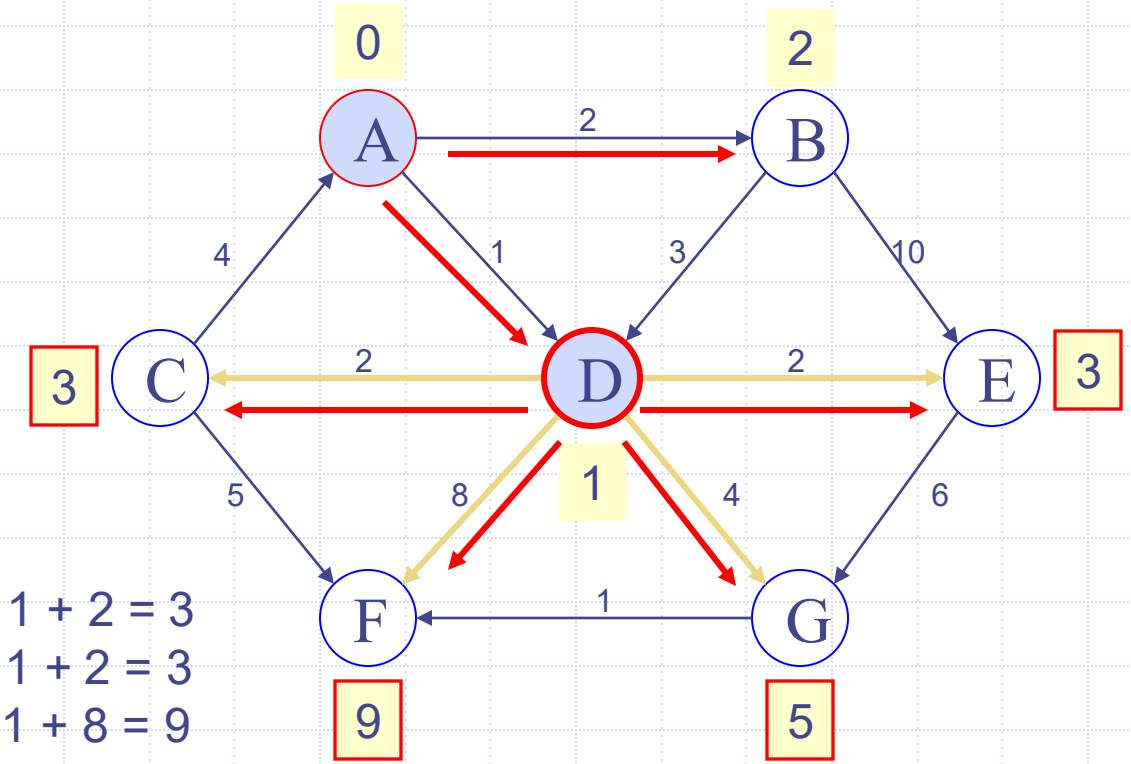
Distance(B) = 2
Distance(D) = 1

Example (2)



Pick vertex in List with minimum distance, i.e., D

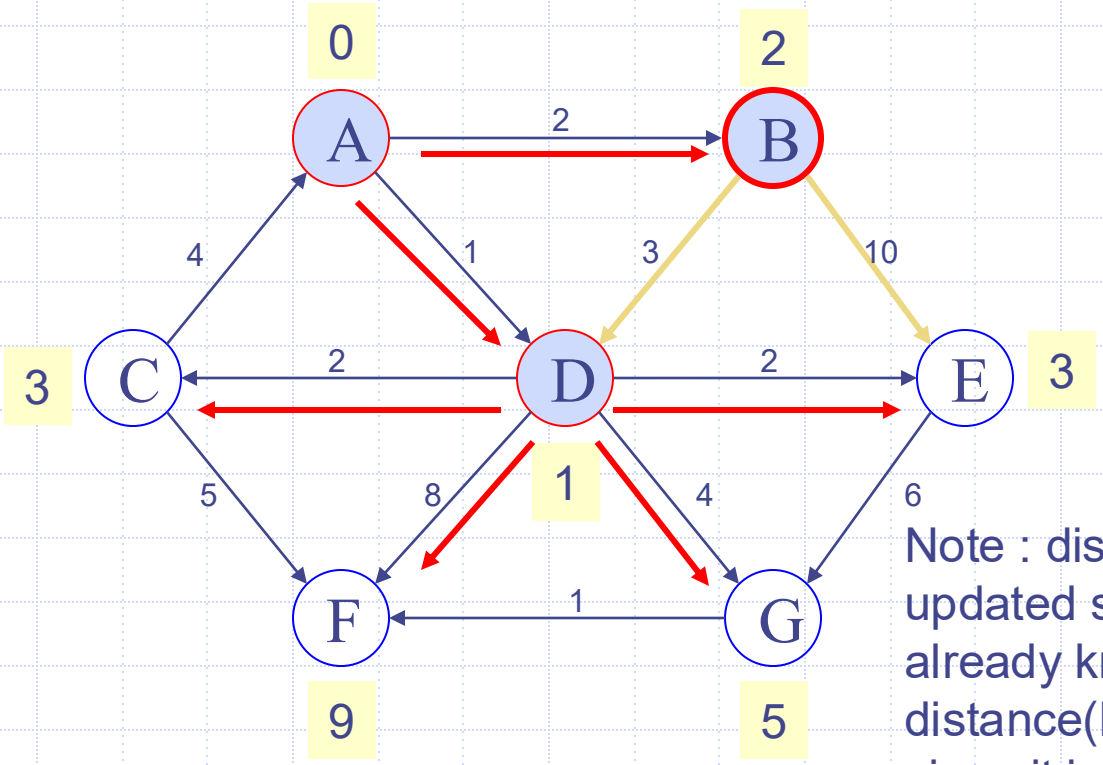
Example (2)



Distance(C) = 1 + 2 = 3
Distance(E) = 1 + 2 = 3
Distance(F) = 1 + 8 = 9
Distance(G) = 1 + 4 = 5

Example (2)

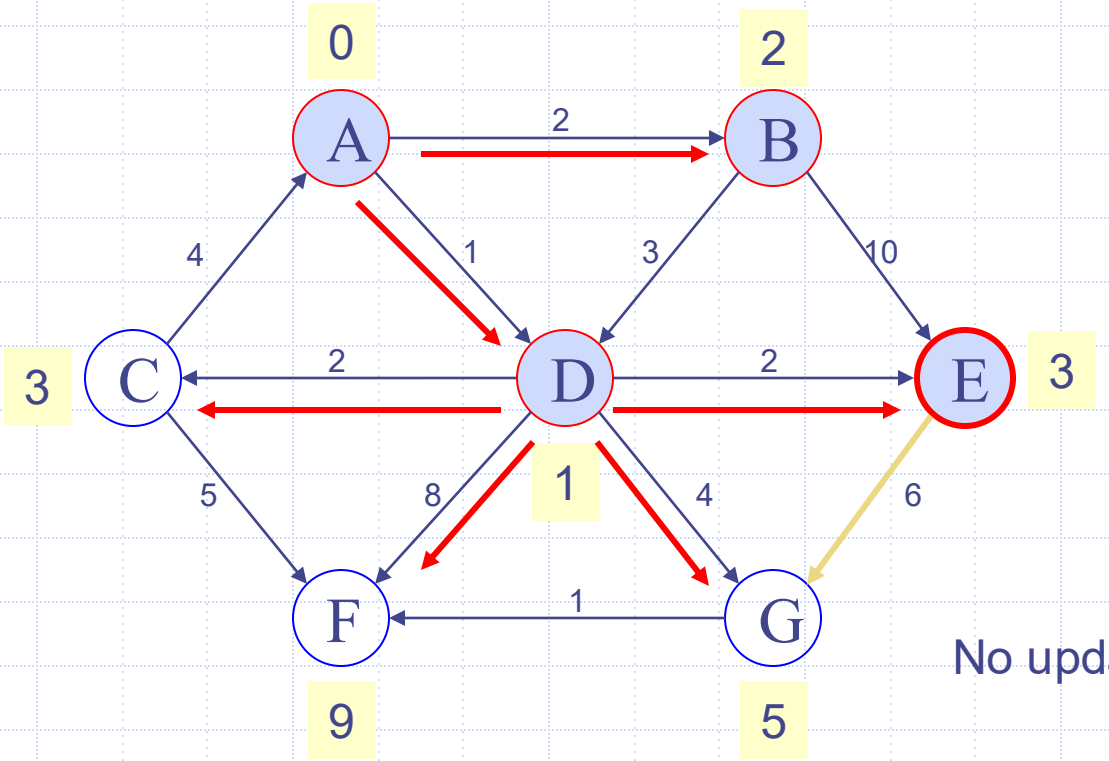
Pick vertex in List with minimum distance (B) and update neighbors



Note : distance(D) not updated since D is already known and distance(E) not updated since it is larger than previously computed

Example (2)

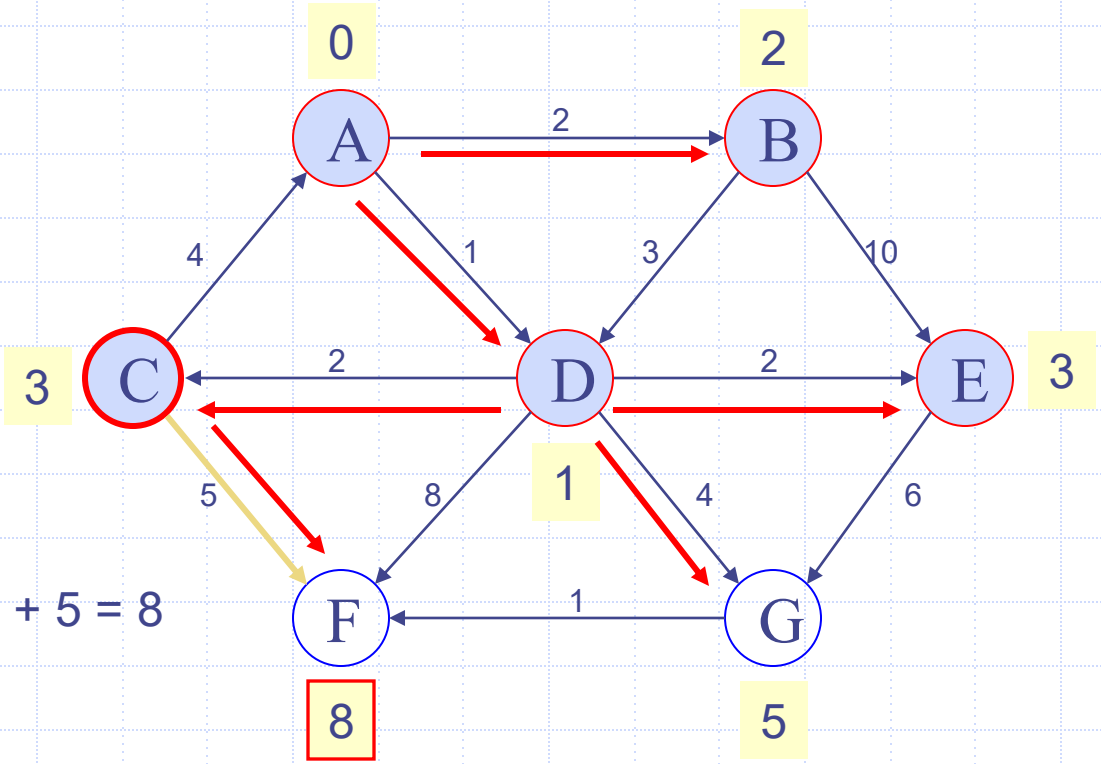
Pick vertex List with minimum distance (E) and update neighbors



No updating

Example (2)

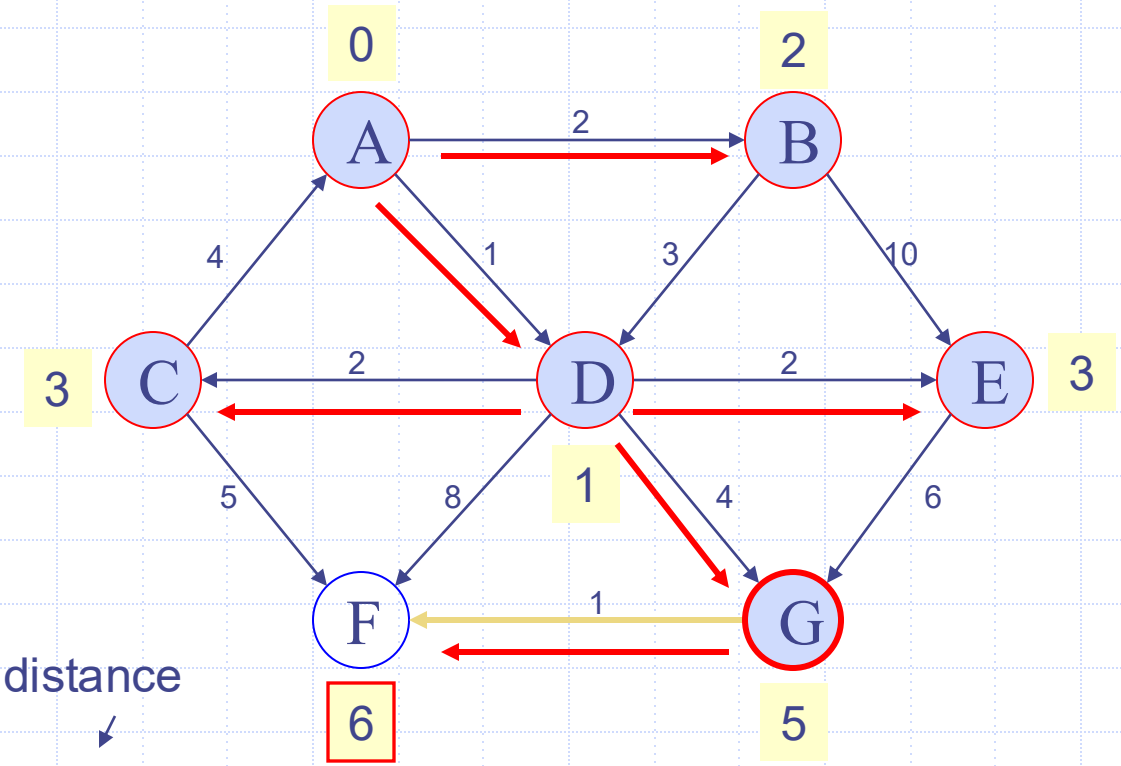
Pick vertex List with minimum distance (C) and update neighbors



Distance(F) = 3 + 5 = 8

Example (2)

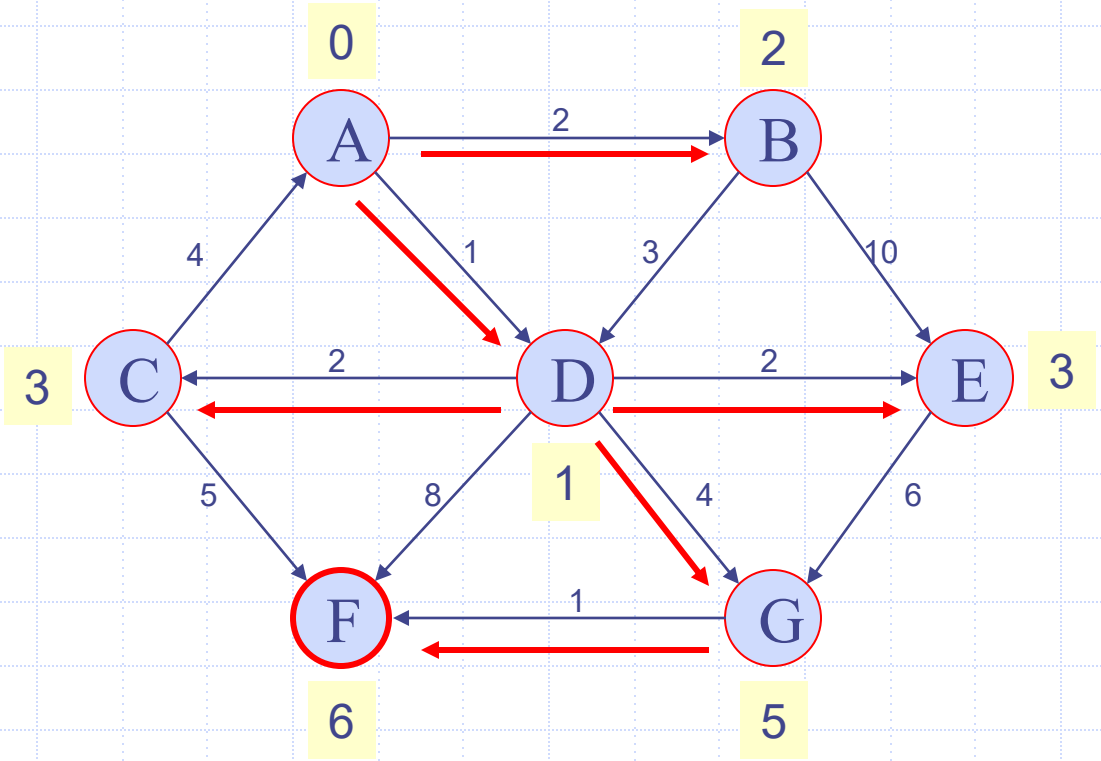
Pick vertex List with minimum distance (G) and update neighbors



Previous distance

Distance(F) = min (8, 5+1) = 6

Example (2)



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.

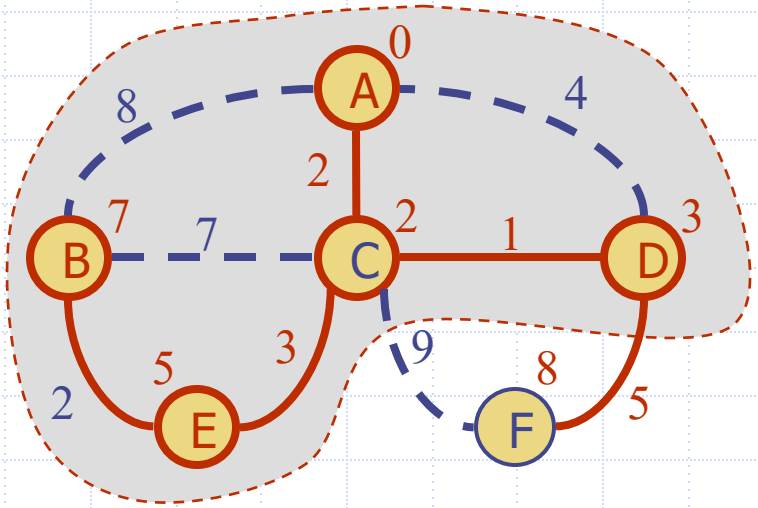
Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
Let a priority queue Q contain all the vertices of G using the D labels as keys.
while Q is not empty **do**
  {pull a new vertex u into the cloud}
  u = value returned by Q.remove_min()
  for each vertex v adjacent to u such that 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 $O(m \log n)$ if the graph is dense

Why Dijkstra's Algorithm Works

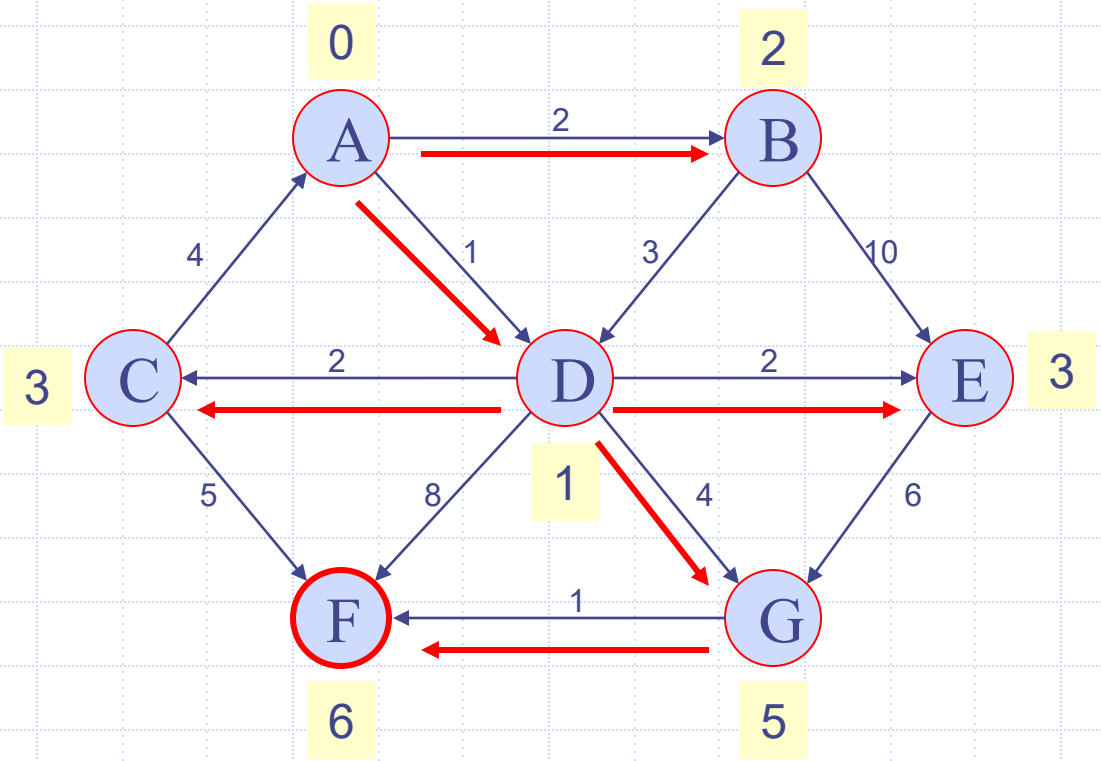
- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - Suppose it didn't find all shortest distances. Let F be the first wrong vertex the algorithm processed.
  - When the previous node, D, on the true shortest path was considered, its distance was correct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



Formal Proof of Correctness

- Induction Statement
  - For all vertices x in S, d[x] is the length of the shortest path from s to x.
  - For all vertices in S', d[x] is the length of the shortest path from s to x that traverses only through vertices contained in S.
- Base Step
  - s in S, V-s in S'
- Induction Hypothesis
  - induction statement hold good at a certain iteration

Shortest Path Computation

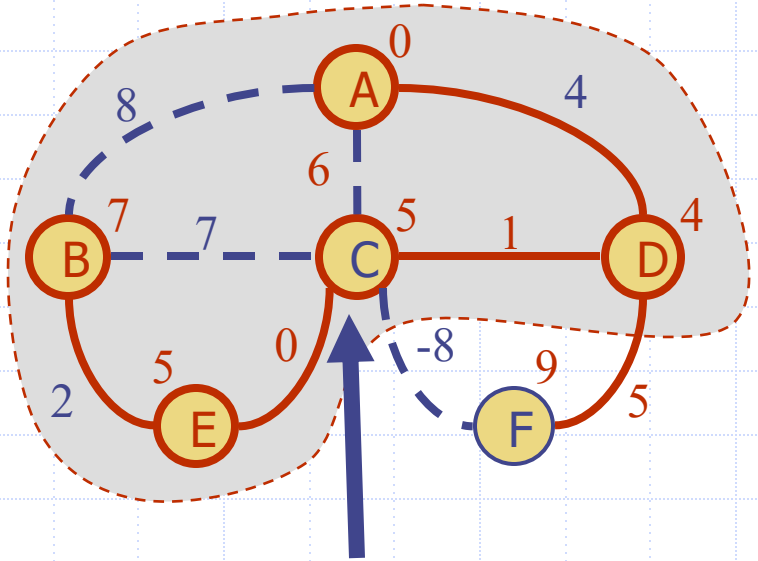


Shortest Path Computation

- Keep track of predecessor vertex.
  - pred[v]=u
  - d[v]=d[u]+w(u, v)
- FindShortestPath
  - x=v
  - while x != null
    - print x
    - x = pred[x]
  - end

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).

```
Algorithm BellmanFord(G, s)
    for all v ∈ G.vertices()
        if v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
    for i ← 1 to n − 1 do
        for each e ∈ G.edges()
            { relax edge e }
            u ← G.origin(e)
            z ← G.opposite(u,e)
            r ← getDistance(u) + weight(e)
            if r < getDistance(z)
                setDistance(z,r)
```

Bellman-Ford Example

Nodes are labeled with their d(v) values