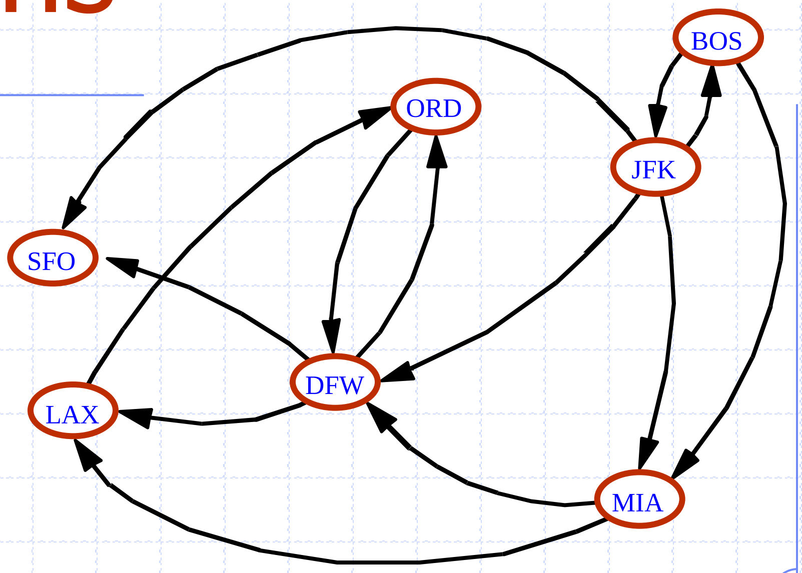
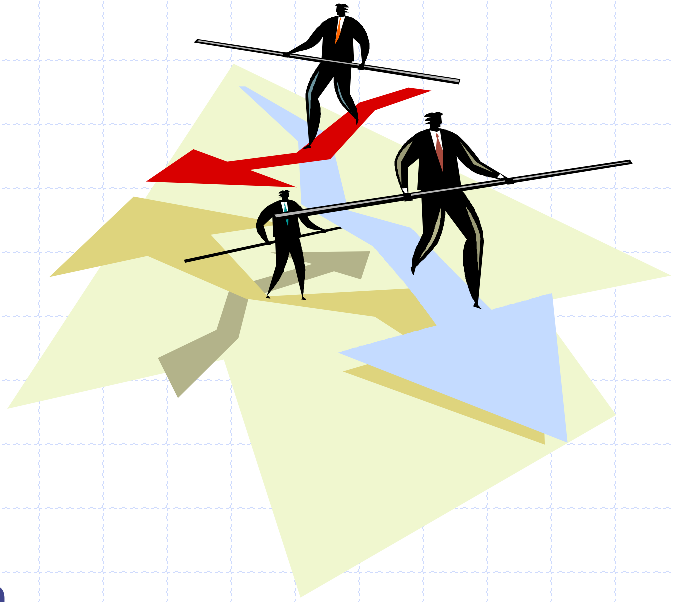
Directed Graphs



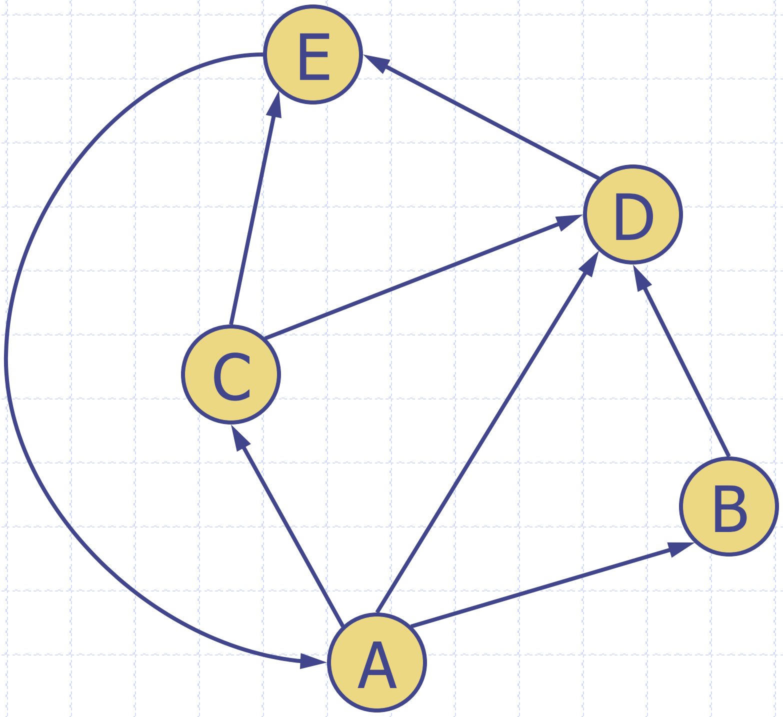
Outline and Reading

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

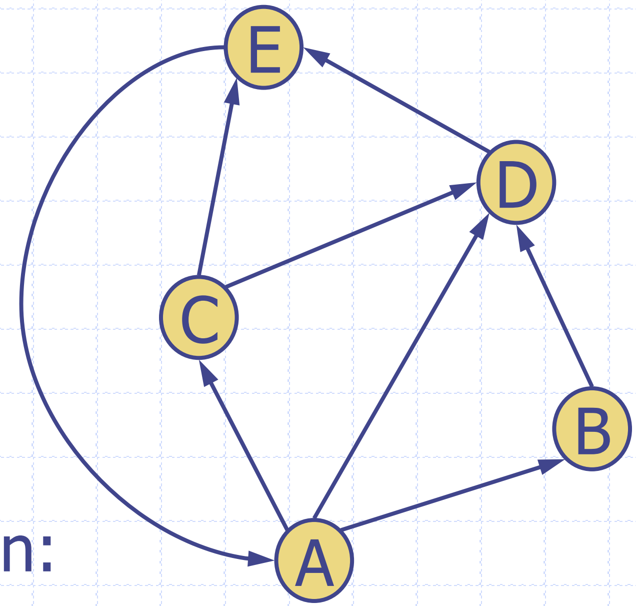


Digraphs

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



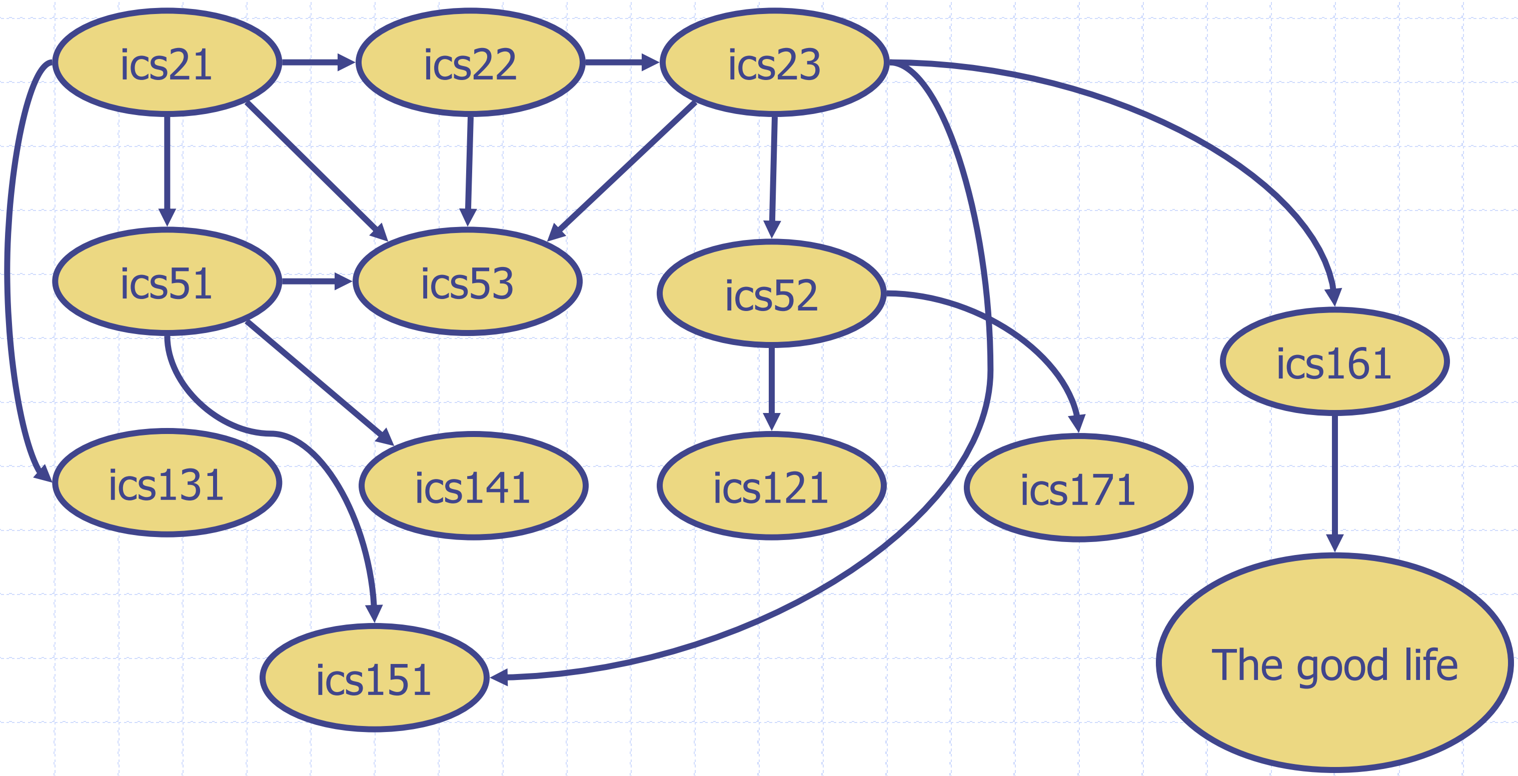
Digraph Properties



- 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, $m \le n*(n-1)$.
- If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of in-edges and out-edges in time proportional to their size.

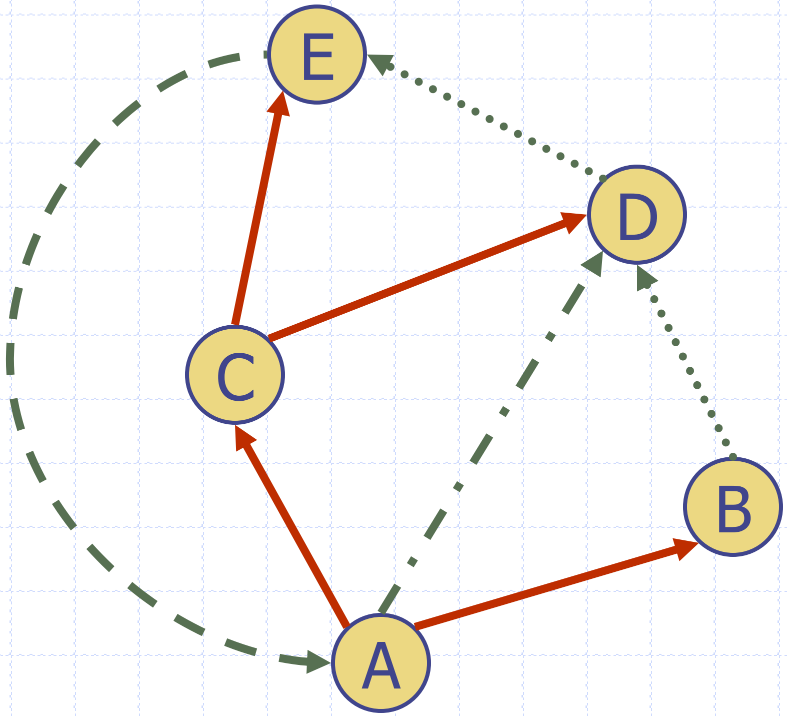
Digraph Application

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



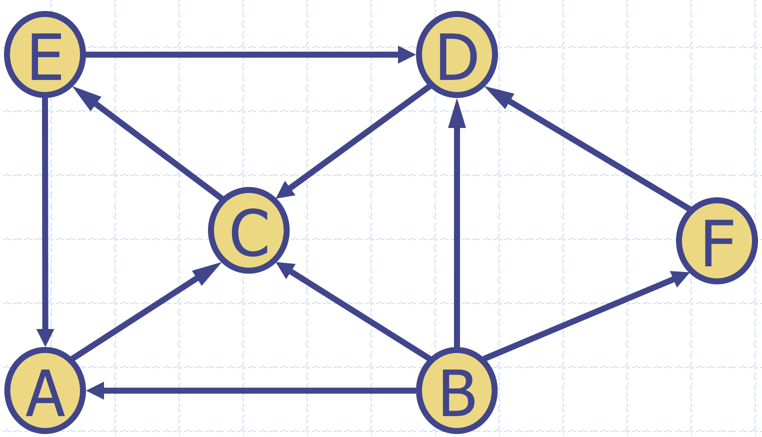
Directed DFS

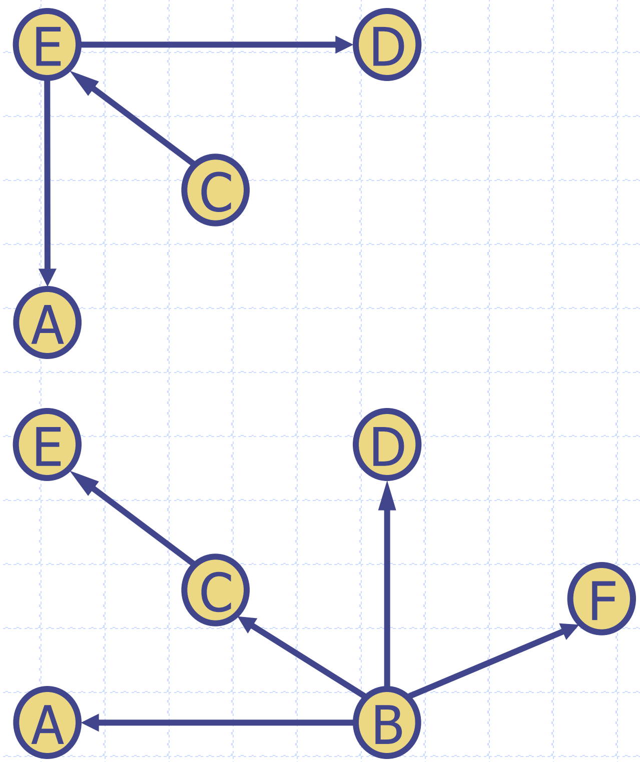
- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction
- In the directed DFS algorithm, we have four types of edges
  - discovery edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting a a vertex *s* determines the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



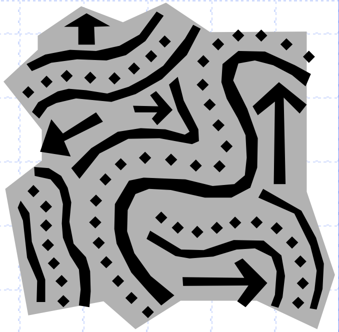
Reachability

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

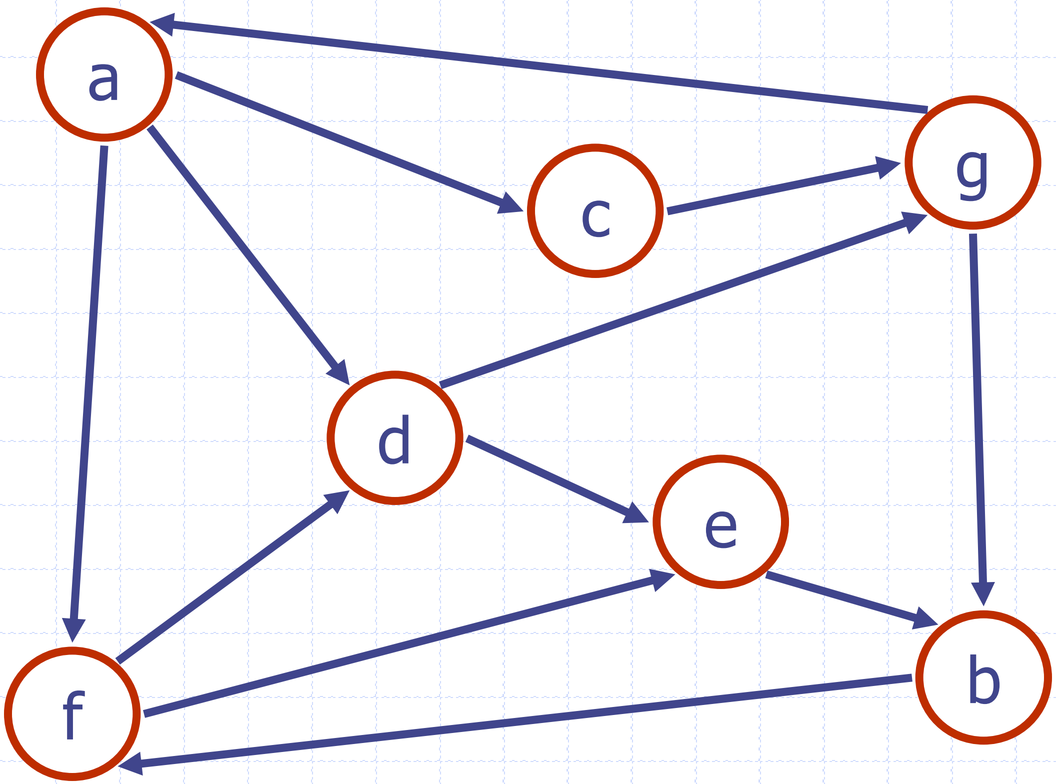




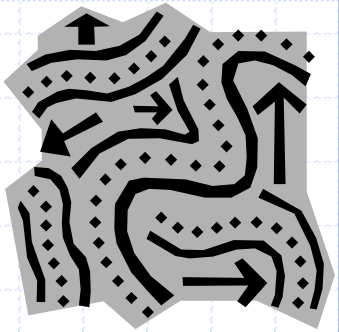
Strong Connectivity



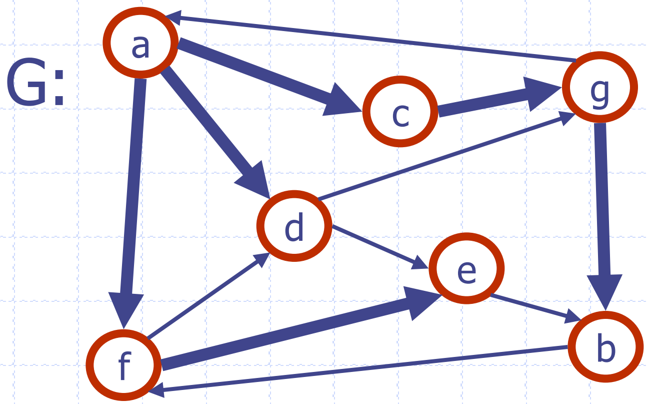
- Each vertex can reach all other vertices

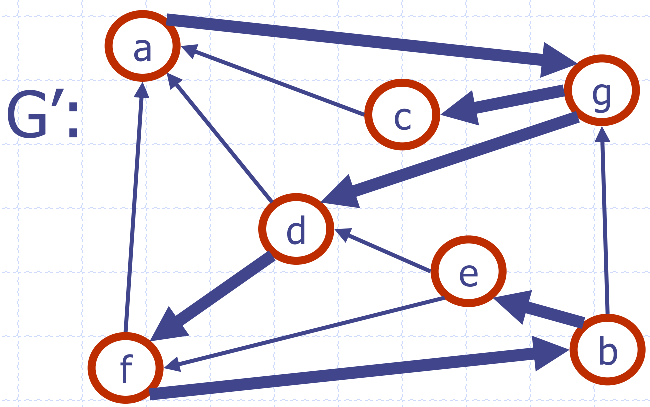


Strong Connectivity Algorithm

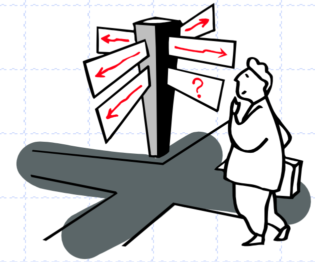


- Pick a vertex v in G.
- Perform a DFS from v in G.
  - If there's a w not visited, print "no".
- Let G' be G with edges reversed.
- Perform a DFS from v in G'.
  - If there's a w not visited, print "no".
  - Else, print "yes".
- Running time: O(n+m).

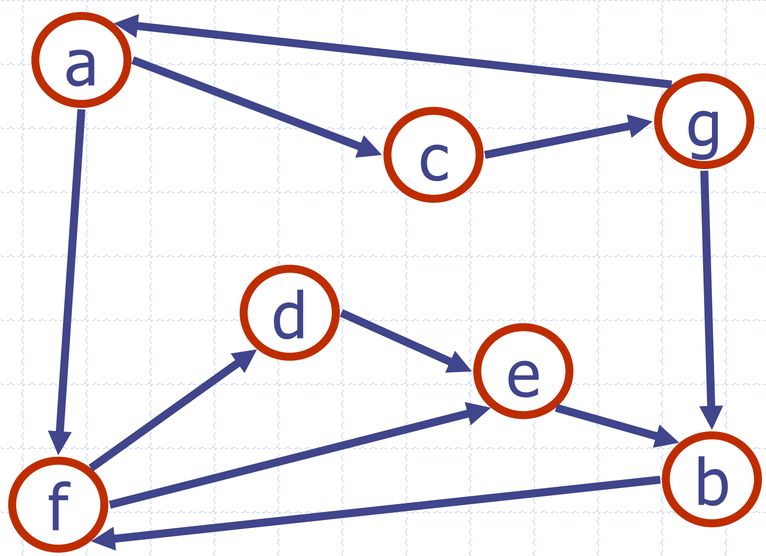




Strongly Connected Components



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

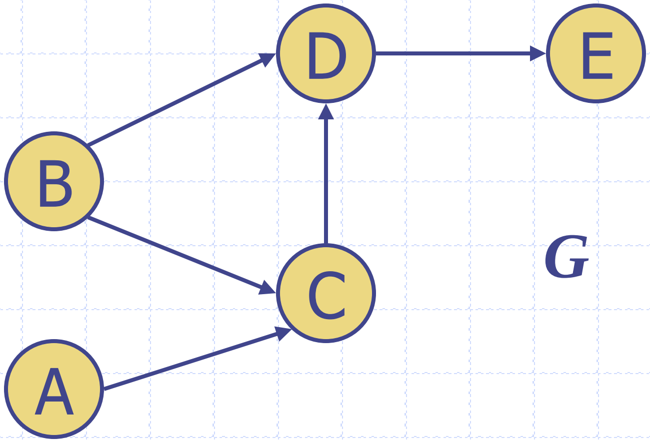


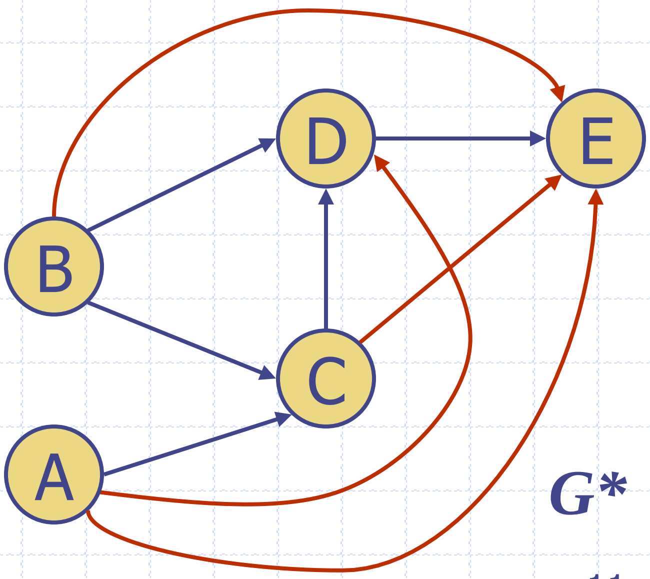
{ a , c , g }

{ 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

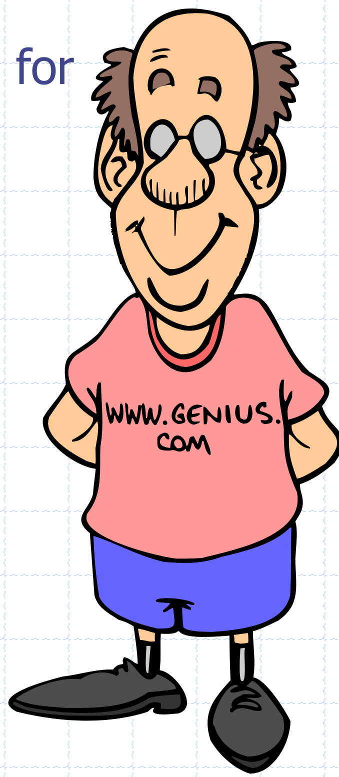




Computing the Transitive Closure

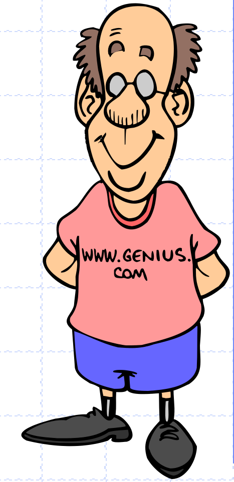
- We can perform DFS starting at each vertex
  - O(n(n+m))
    - Actually $O(n^3)$ for dense graphs
  - Complex

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.

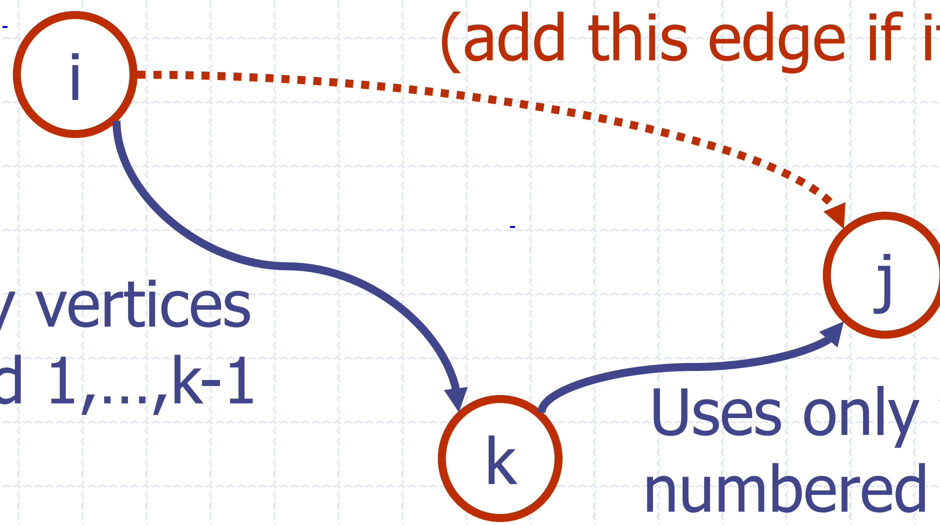


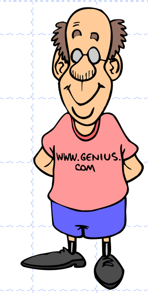
- Alternatively ... Use dynamic programming: The Floyd-Warshall Algorithm
  - $O(n^3)$
  - Simple

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:



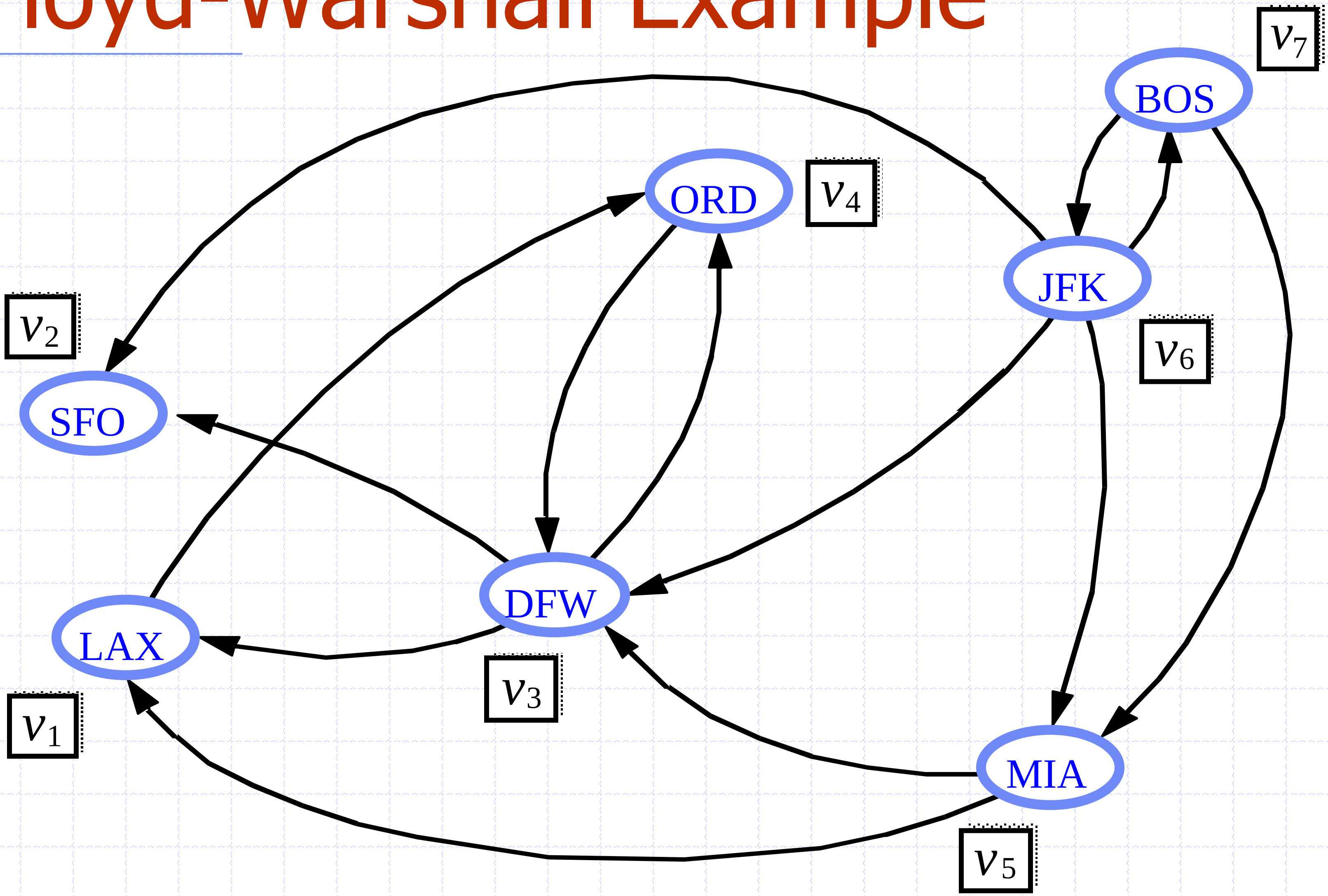


Floyd-Warshall's Algorithm

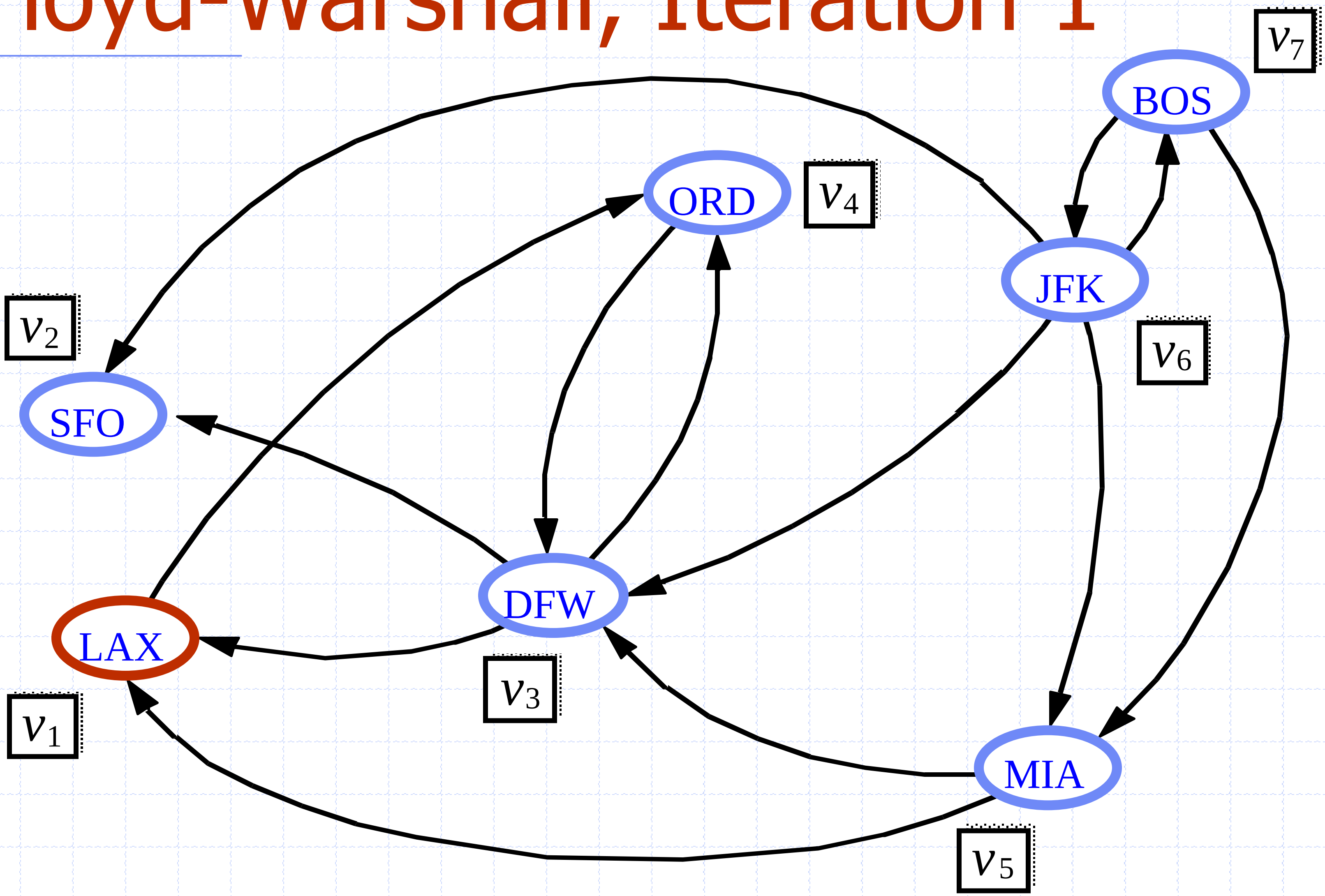
- Floyd-Warshall's algorithm numbers the vertices of G as $v_1, \dots, v_n$ and computes a series of digraphs $G_0, \dots, G_n$
  - $G_0 = G$
  - G_k has a directed edge (v_i, v_j) if G has a directed path from v_i to v_j with intermediate vertices in the set $\{v_1, \dots, v_k\}$
- We have that $G_n = G^*$
- In phase k, digraph G_k is computed from G_{k-1}
- Running time: O(n³), 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

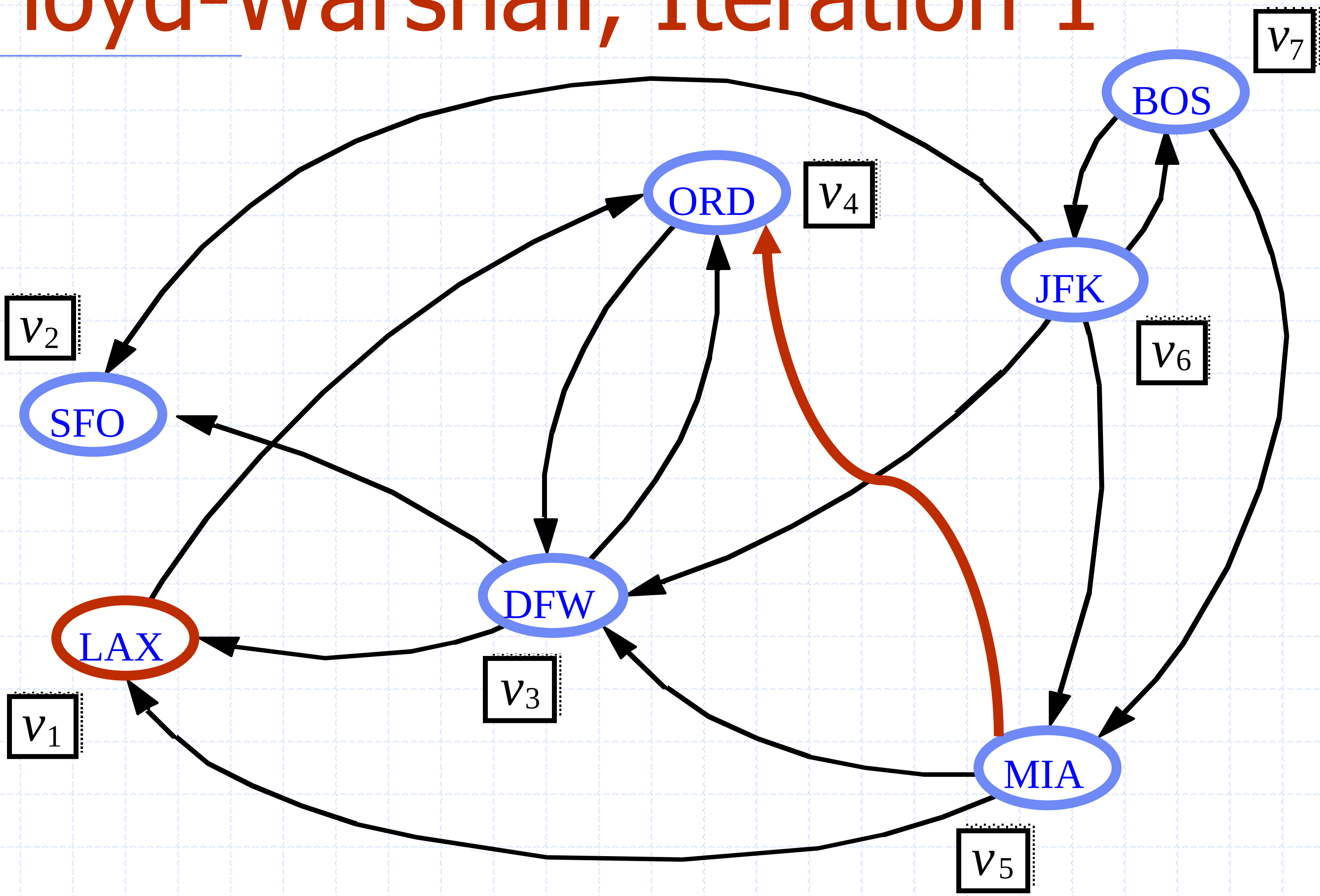
Floyd-Warshall Example



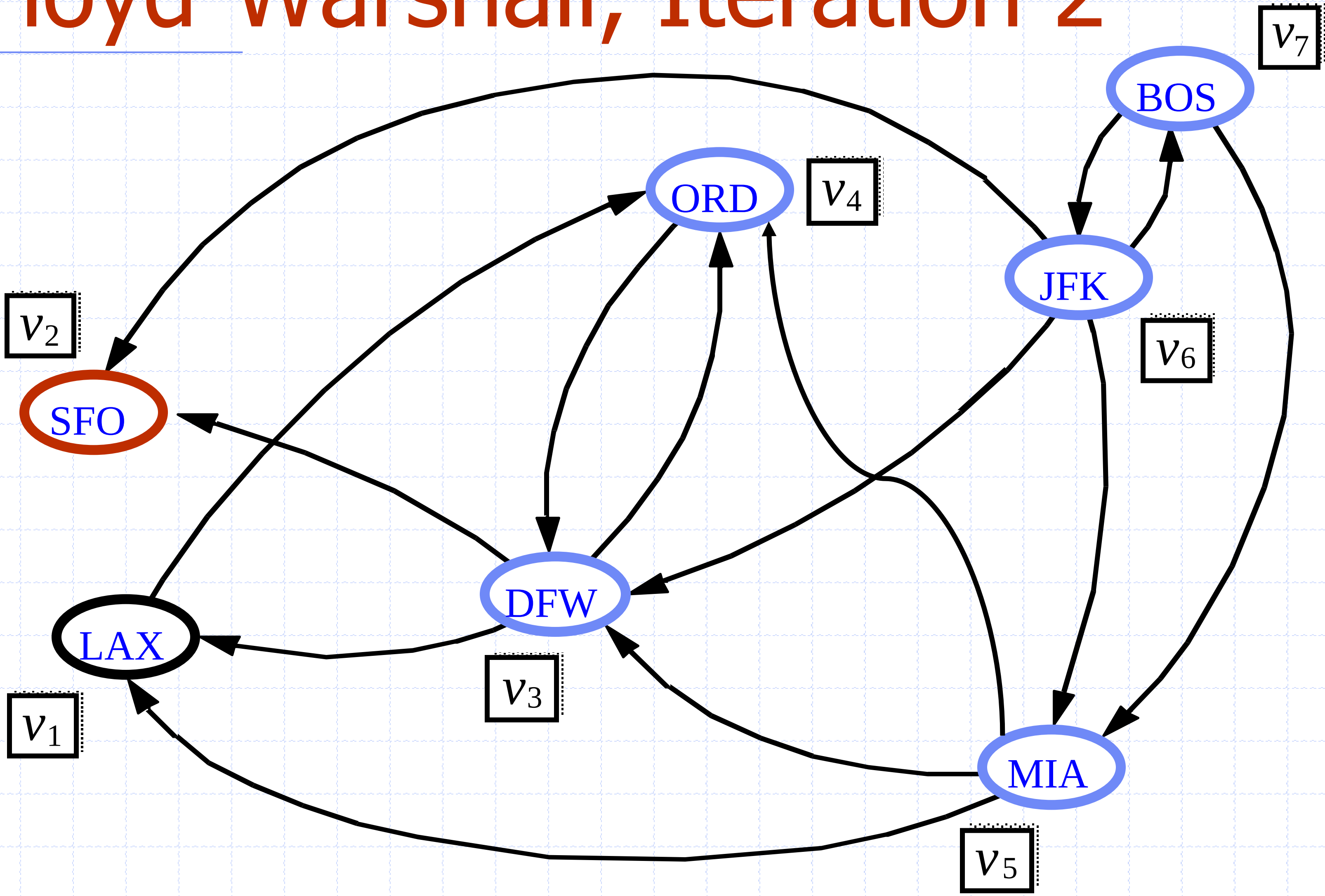
Floyd-Warshall, Iteration 1



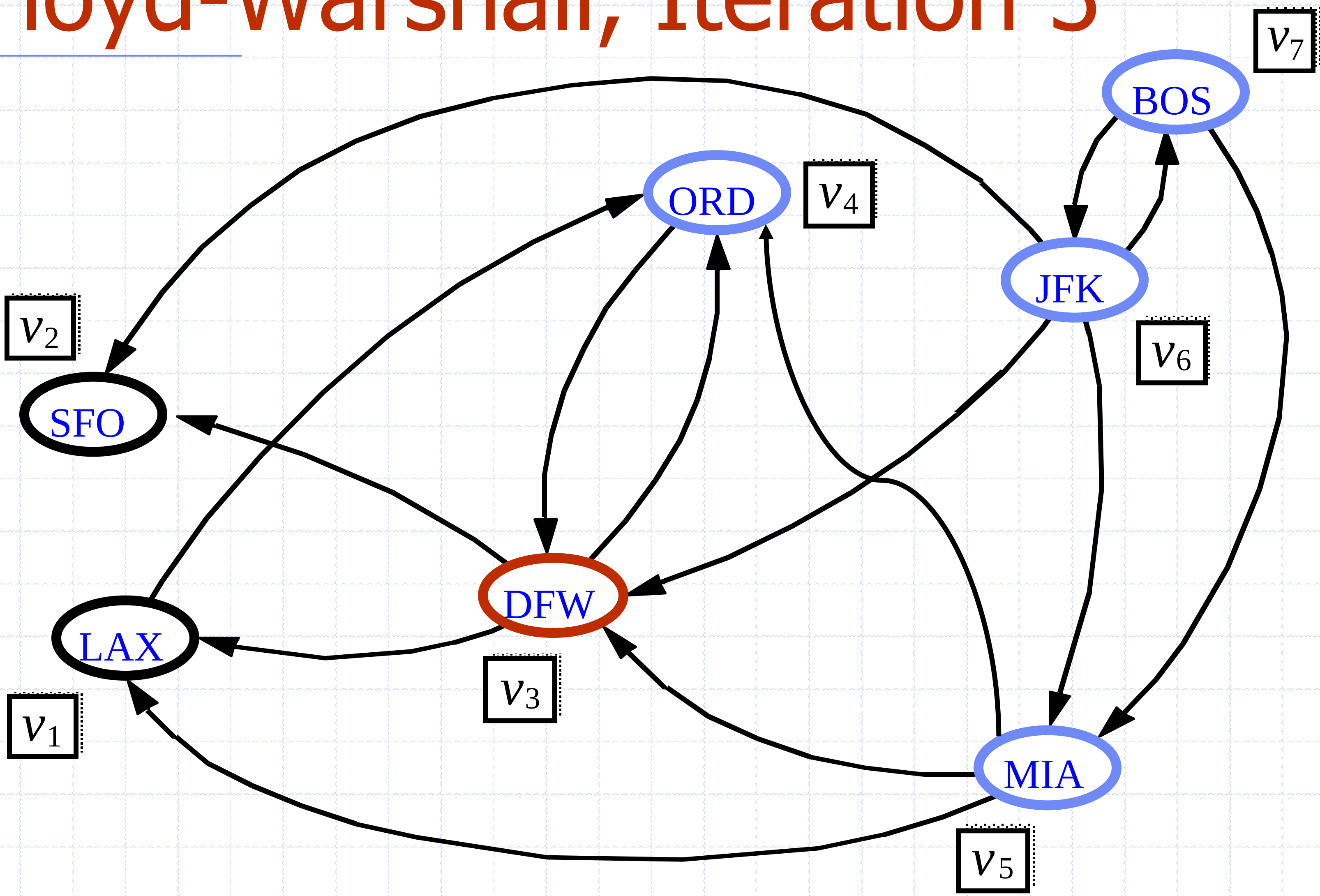
Floyd-Warshall, Iteration 1



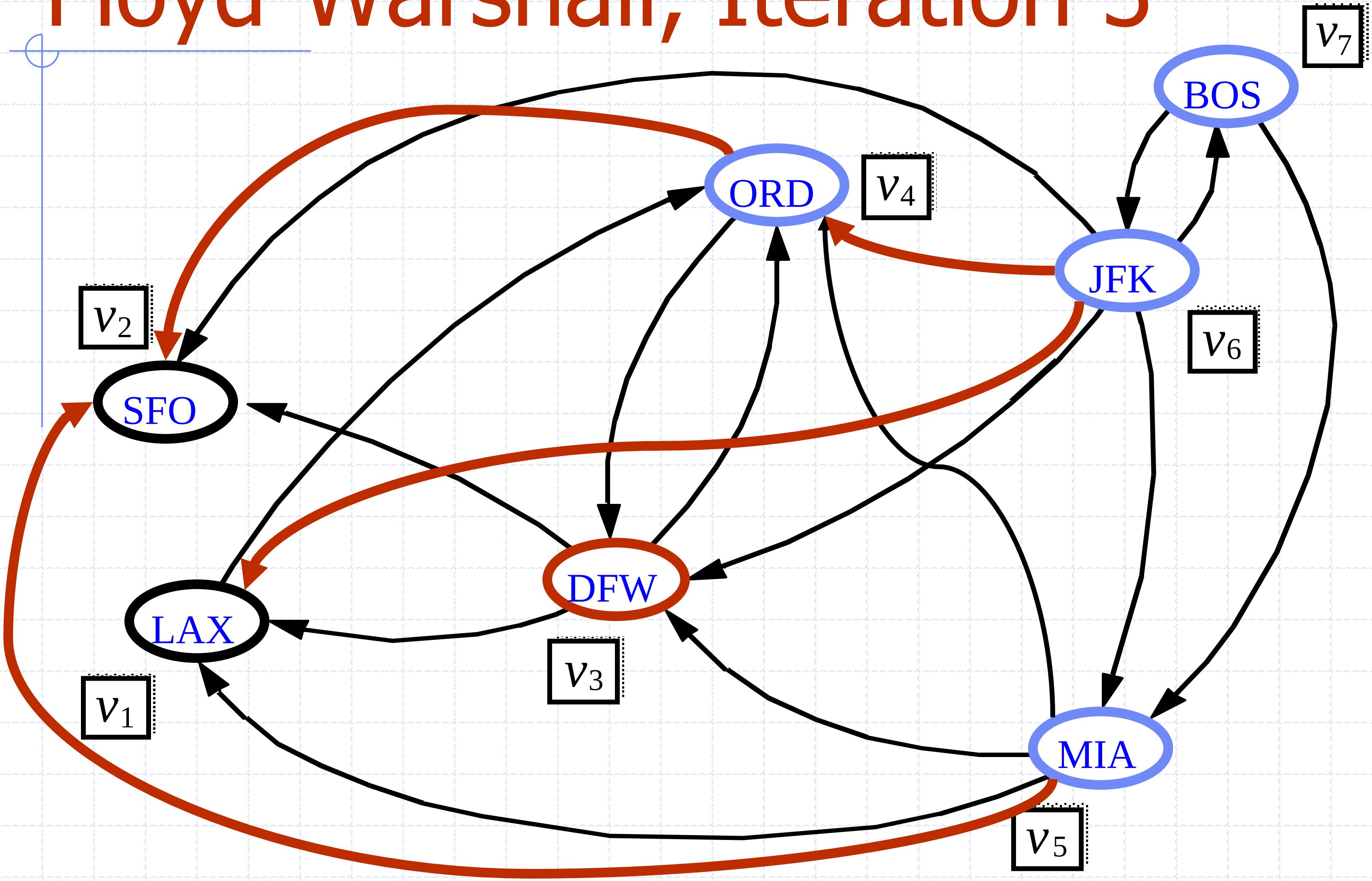
Floyd-Warshall, Iteration 2



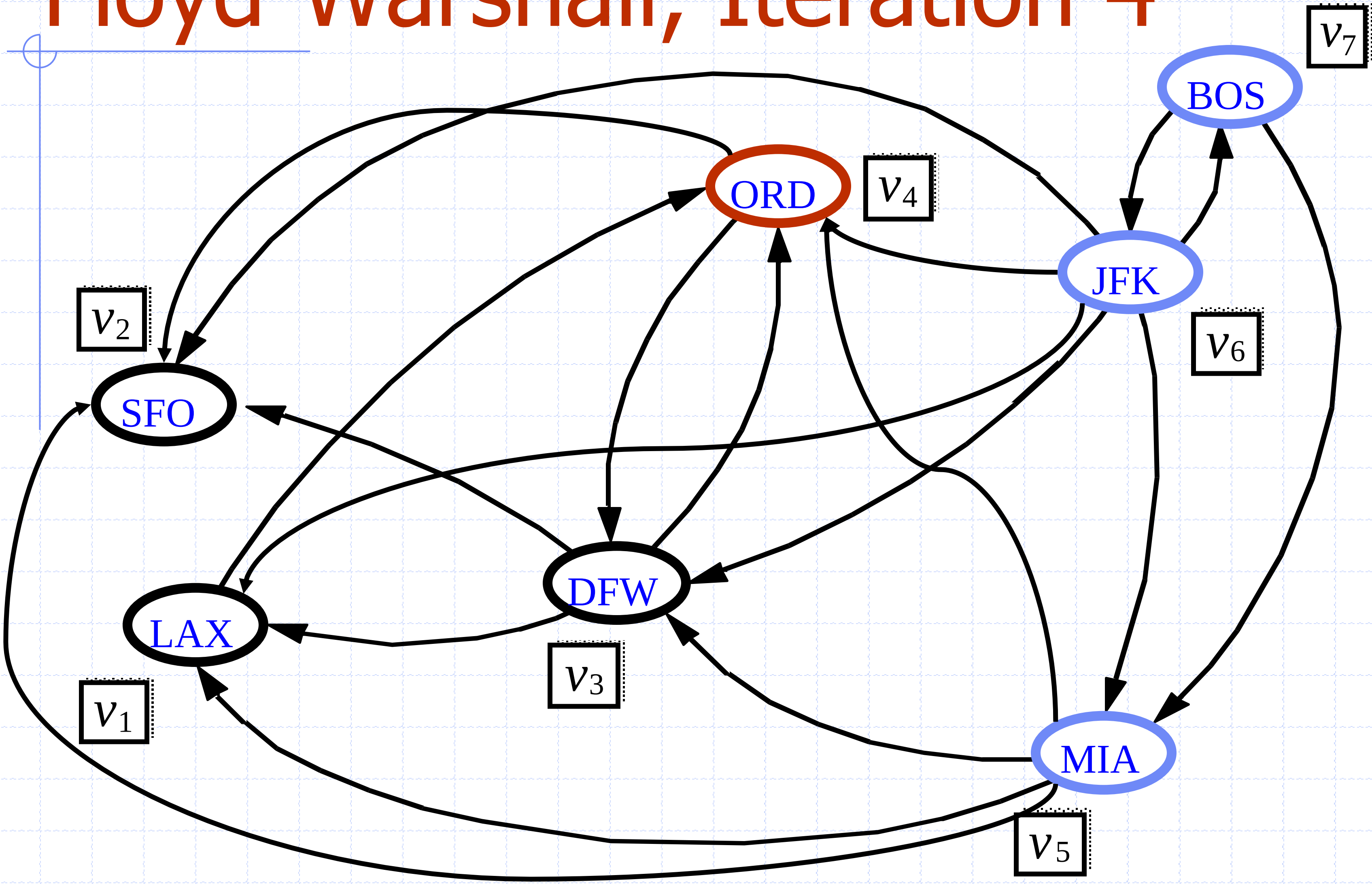
Floyd-Warshall, Iteration 3



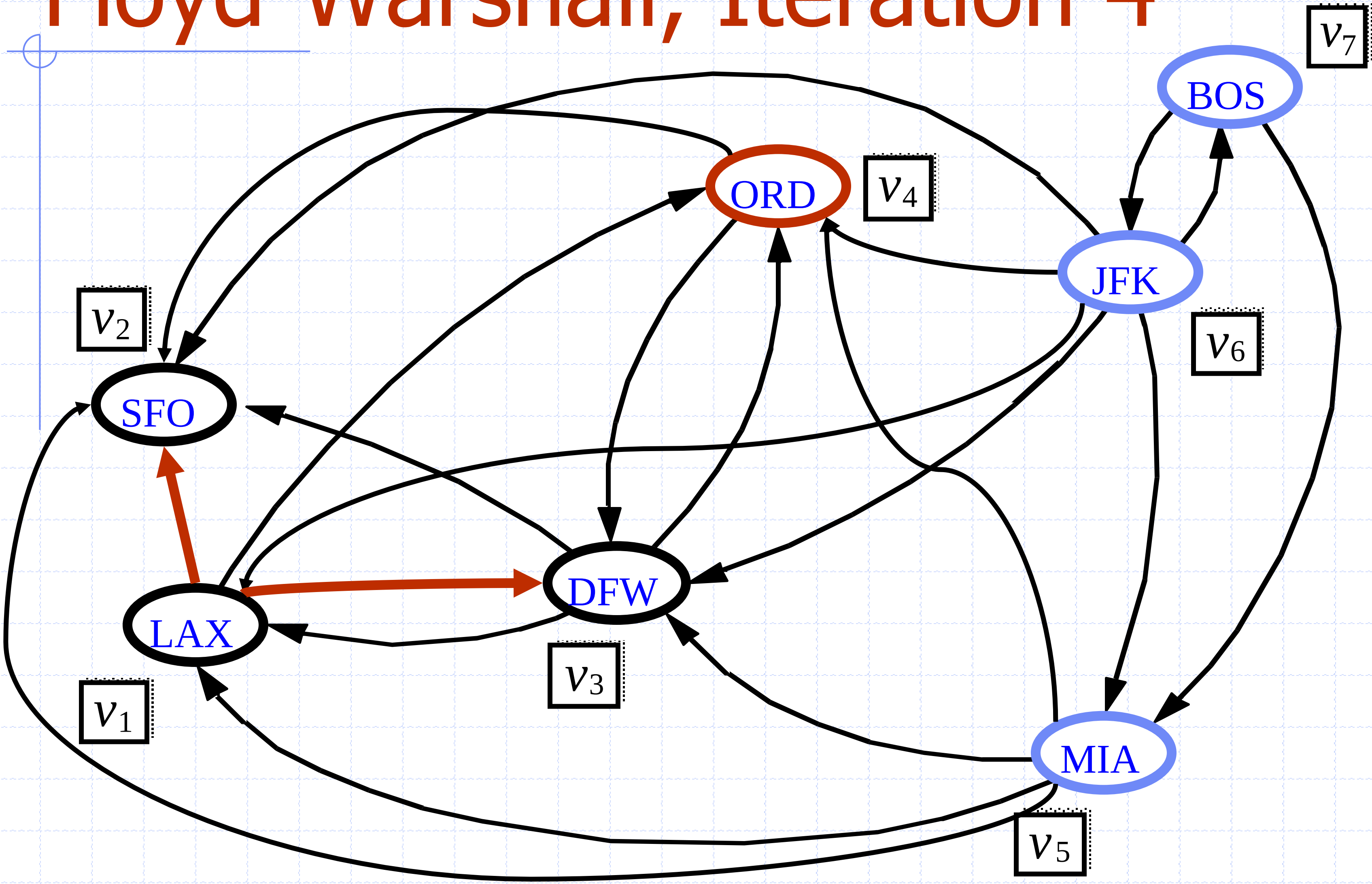
Floyd-Warshall, Iteration 3



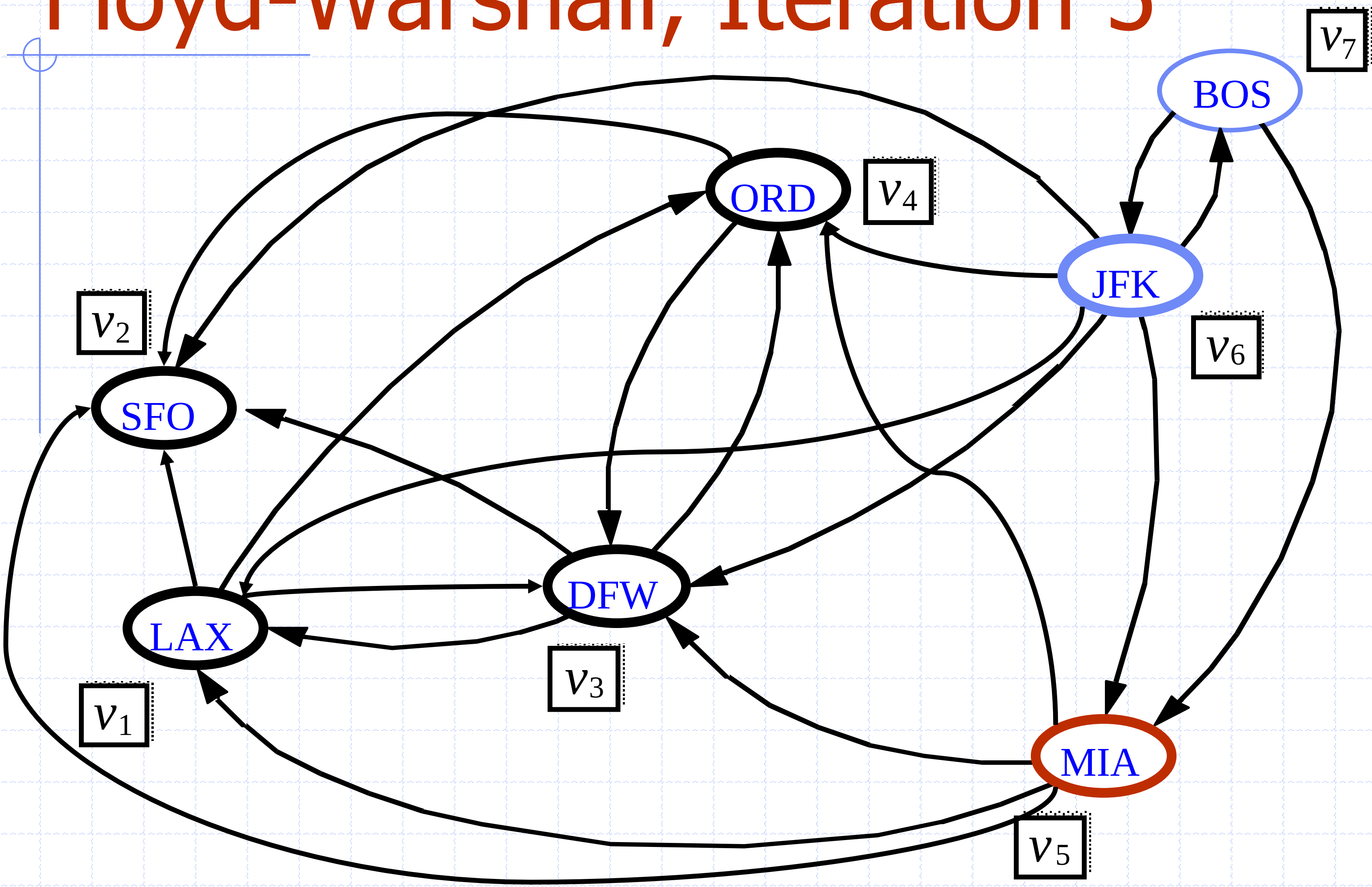
Floyd-Warshall, Iteration 4



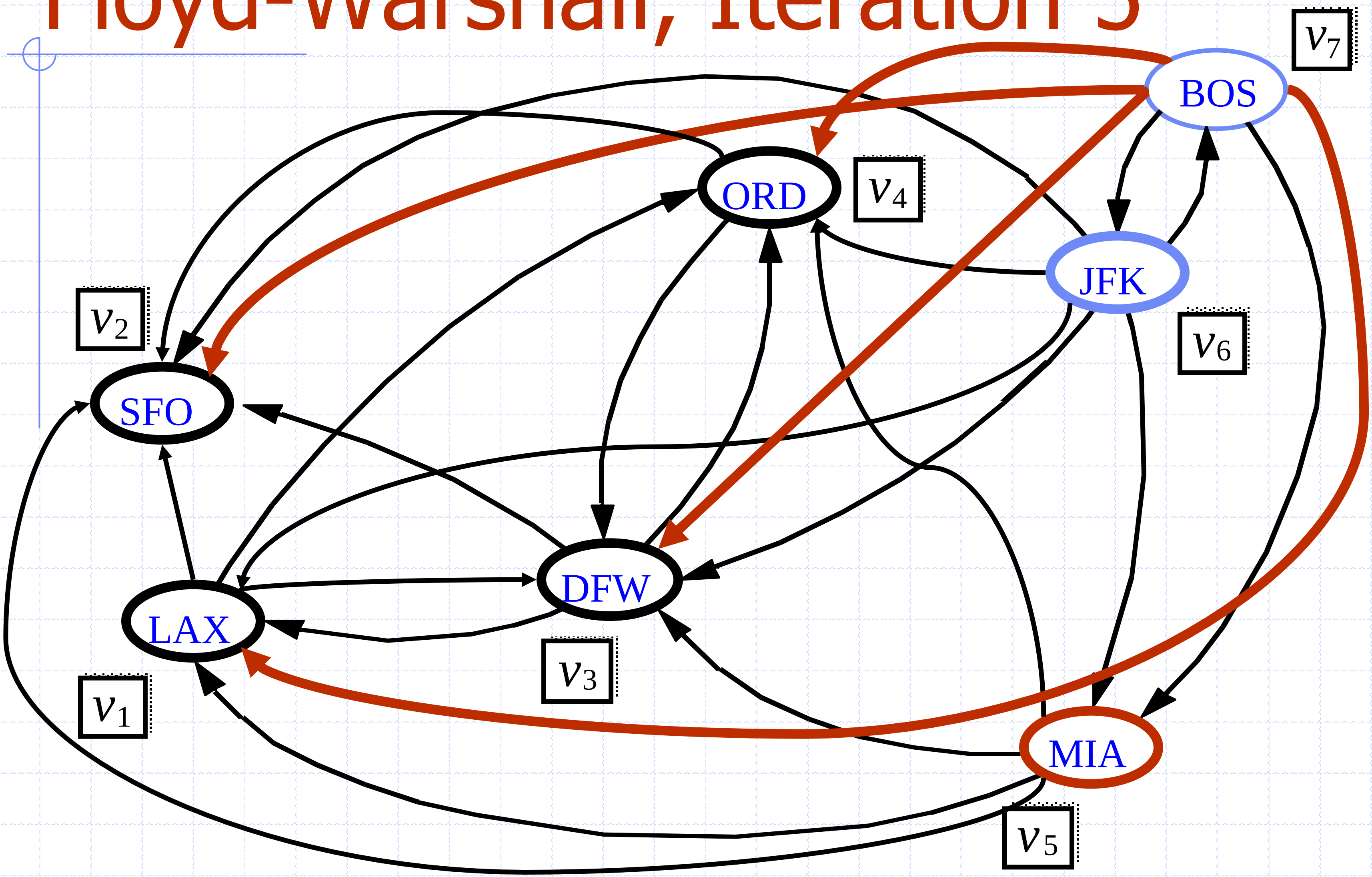
Floyd-Warshall, Iteration 4



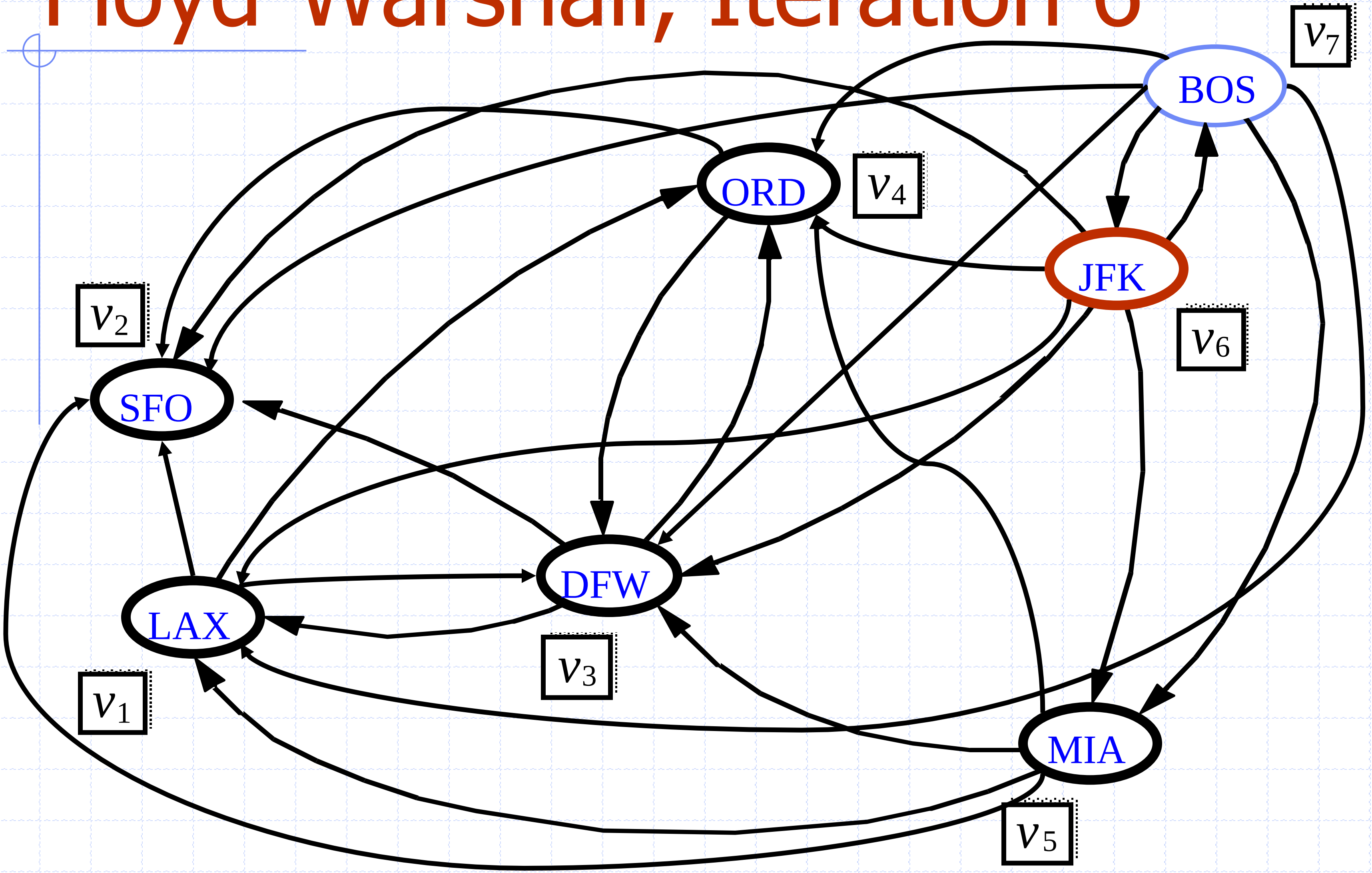
Floyd-Warshall, Iteration 5



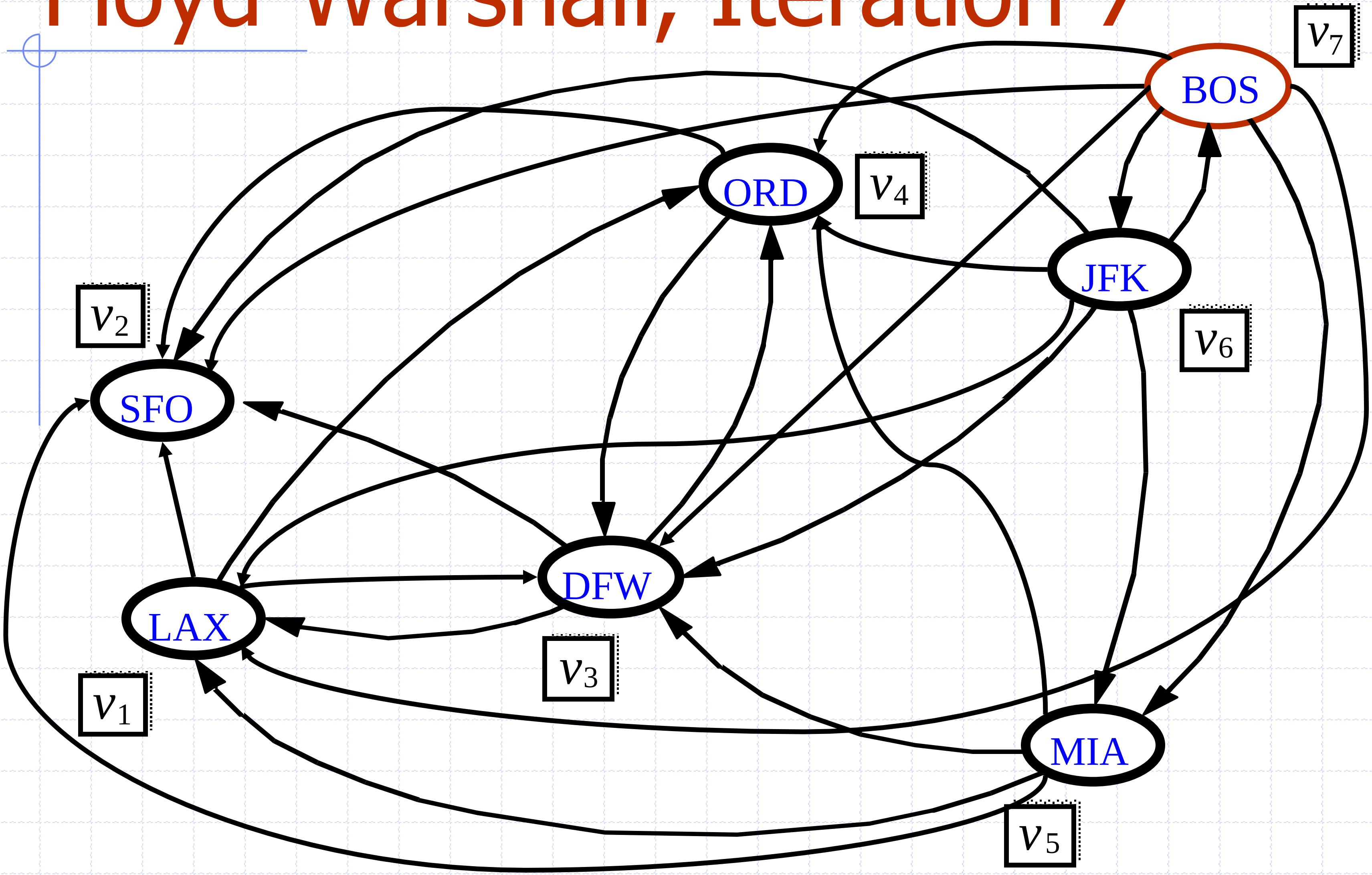
Floyd-Warshall, Iteration 5



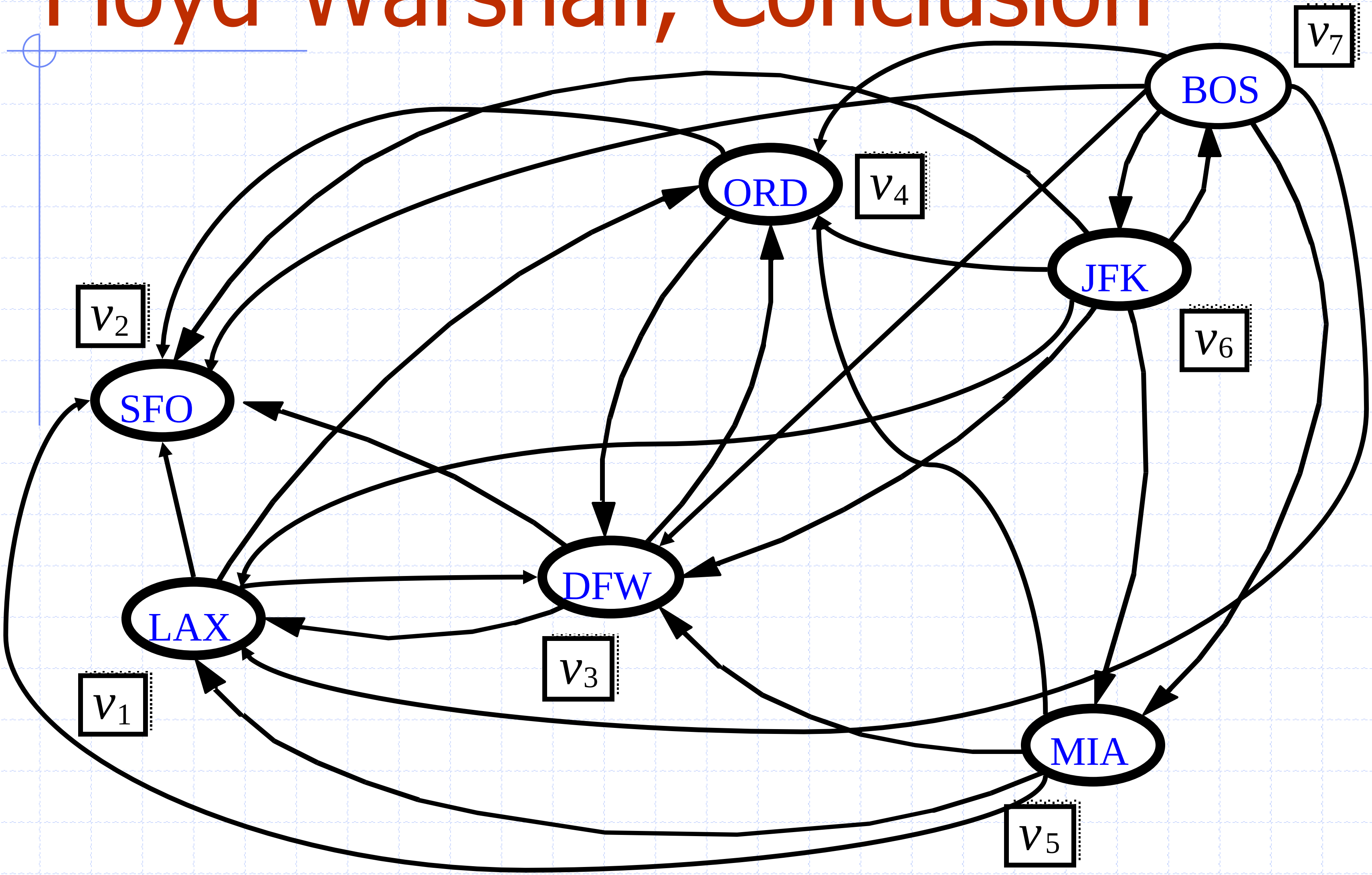
Floyd-Warshall, Iteration 6



Floyd-Warshall, Iteration 7



Floyd-Warshall, Conclusion



DAGs and Topological Ordering

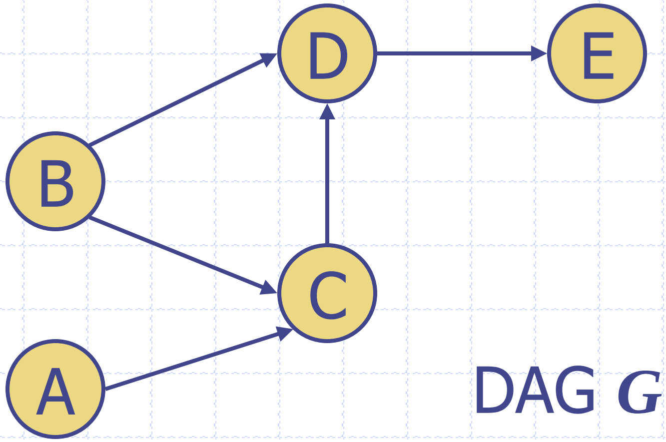
- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

  $$v_1, \ldots, v_n$$

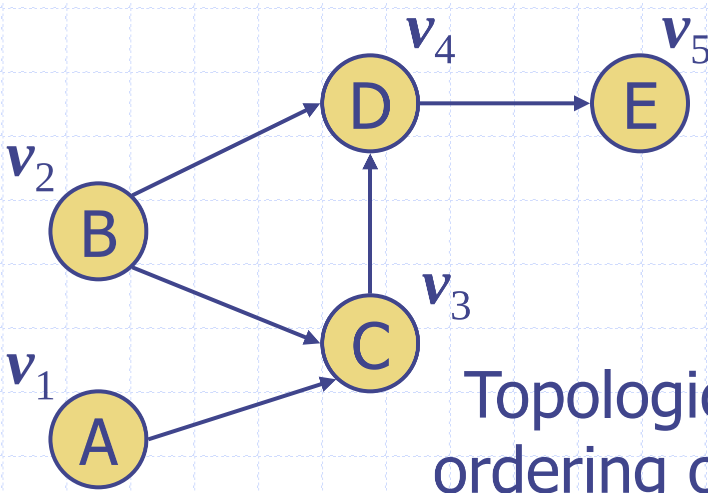
  of the vertices such that for every edge (v_i, v_j), we have $i < j$
- Example: in a task scheduling digraph, a topological ordering of a task sequence satisfies the precedence constraints

Theorem

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

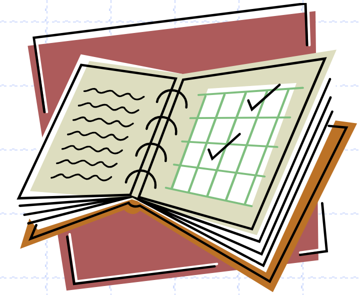


DAG G

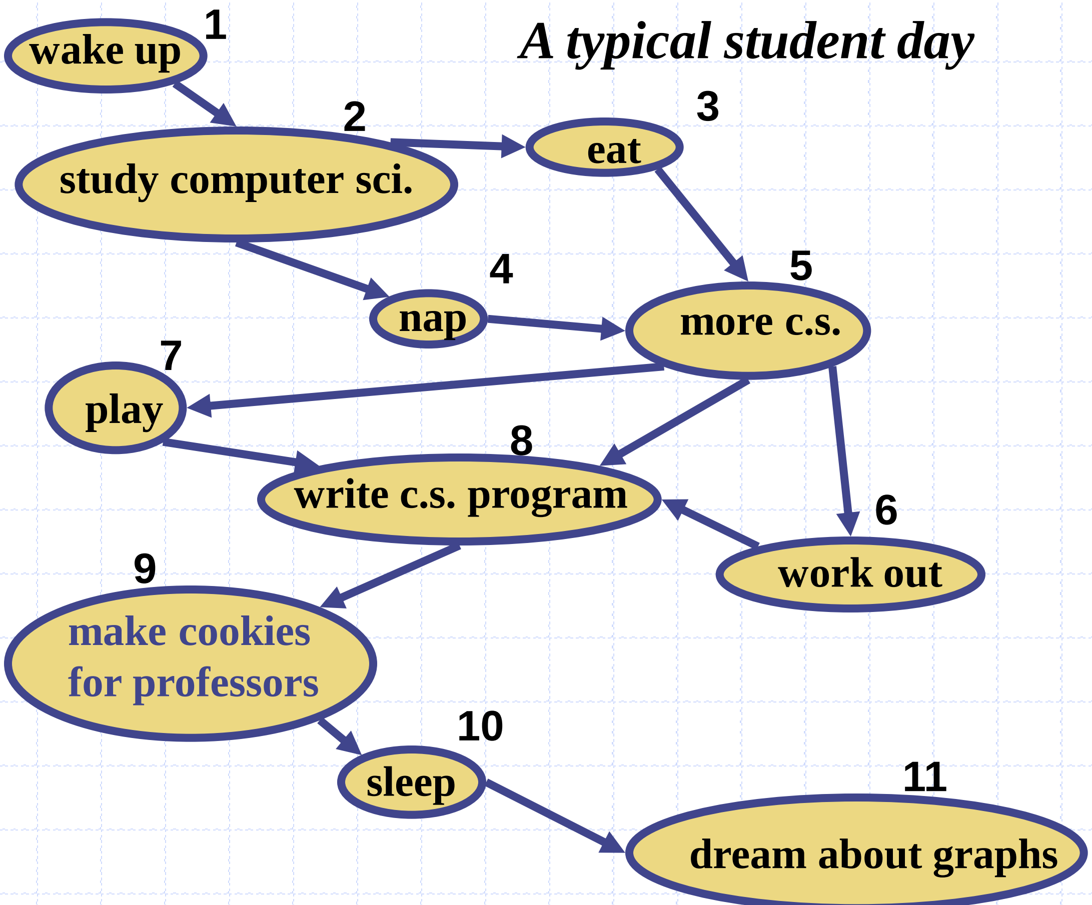


Topological ordering of G

Topological Sorting



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



Algorithm for Topological Sorting

- Note: This algorithm is different (more compact) than the one in Goodrich-Tamassia (yet of the same big-Oh)

```
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). Why…?

Topological Sorting Algorithm using DFS

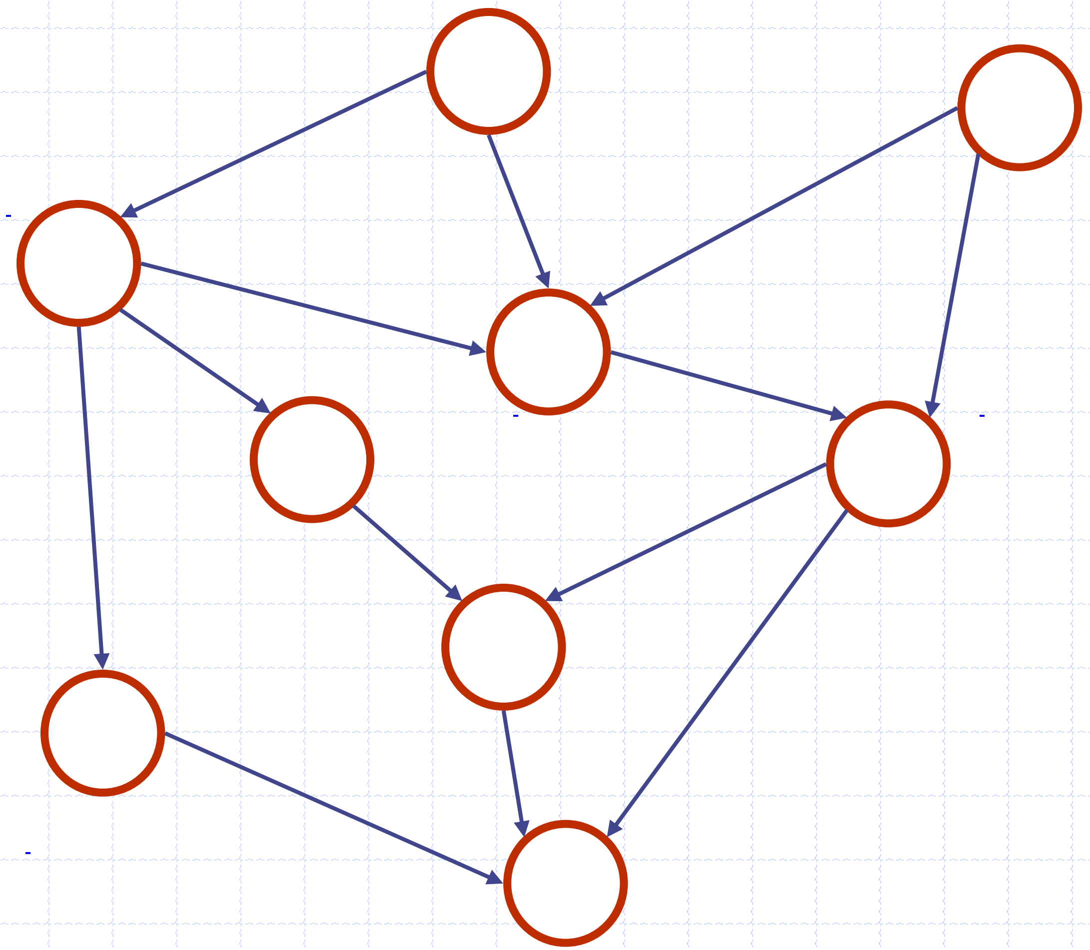
- Simulate the algorithm by using depth-first search

```
Algorithm topologicalDFS(G)
   Input dag G
   Output topological ordering of G
  n ← G.numVertices()
  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
      topologicalDFS(G, v)
```

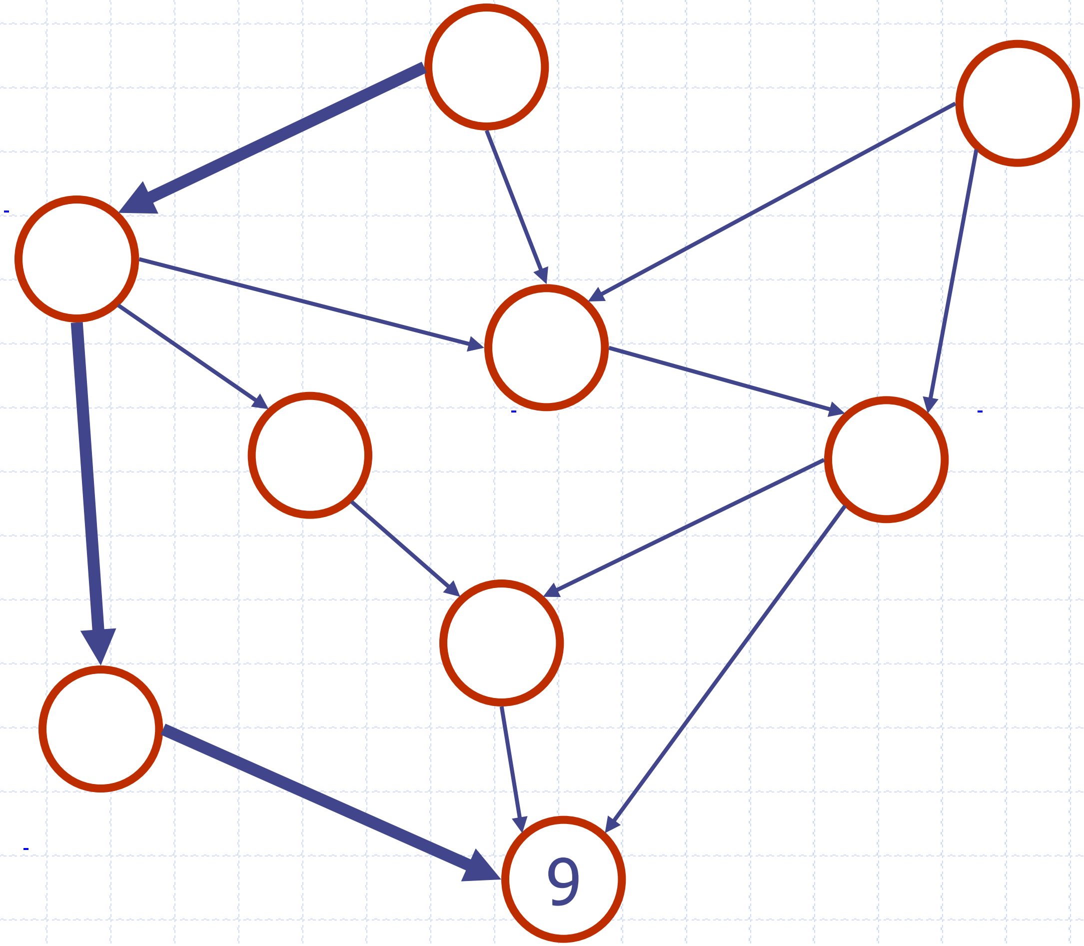
- O(n+m) time.

```
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 ∈ G.outgoingEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← 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 ← n - 1
```

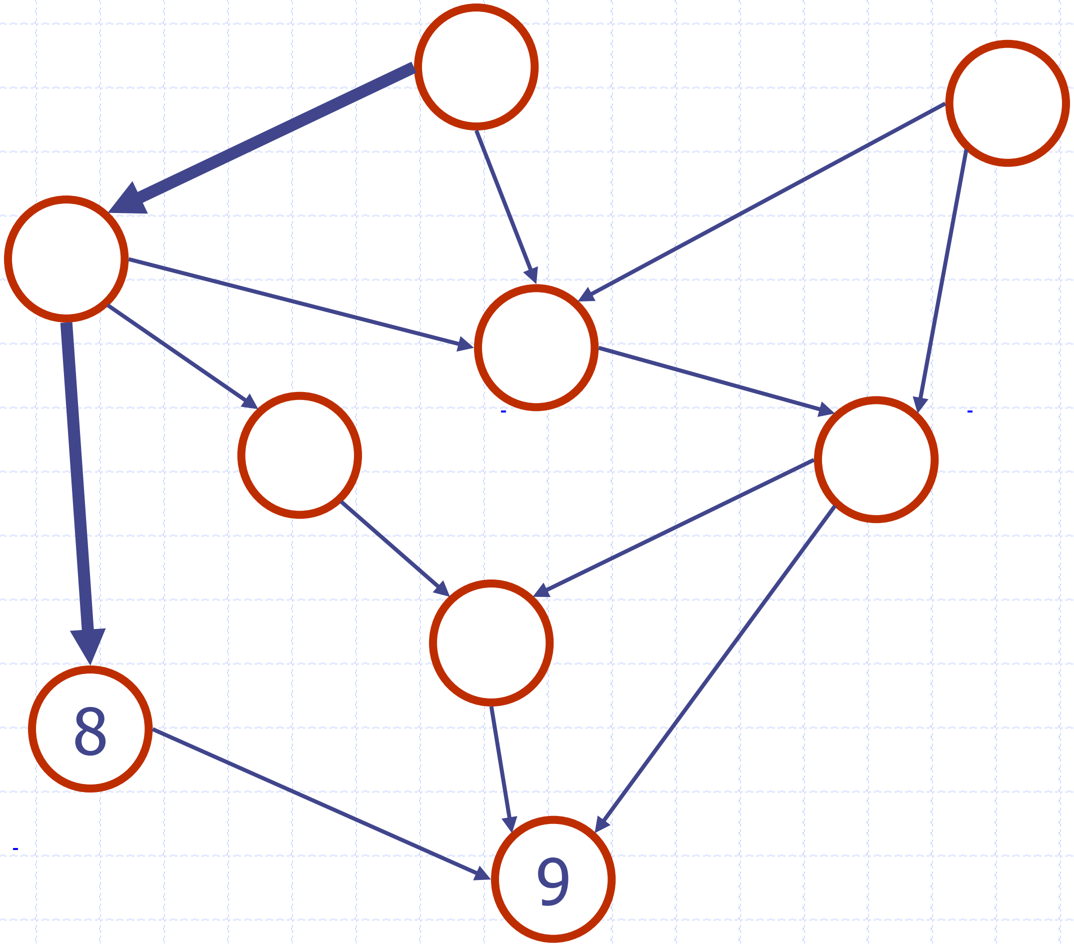
Topological Sorting Example



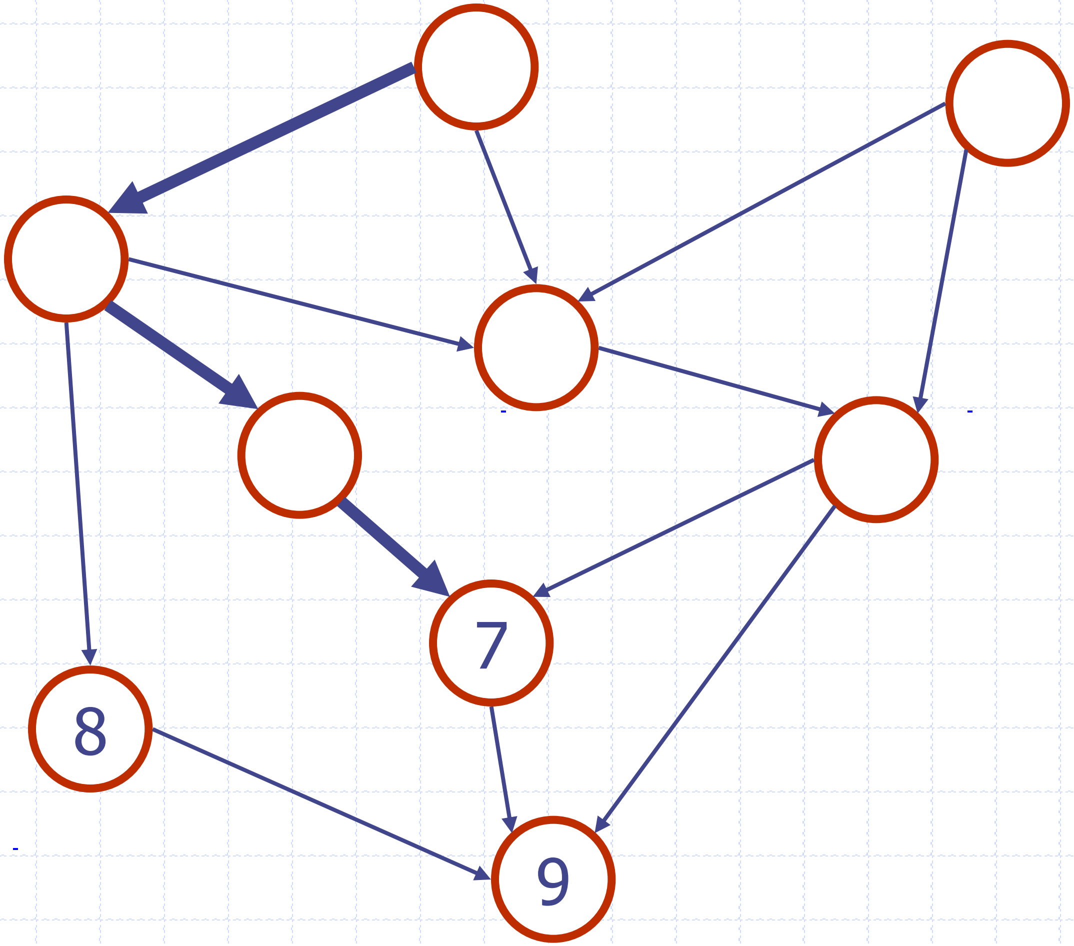
Topological Sorting Example



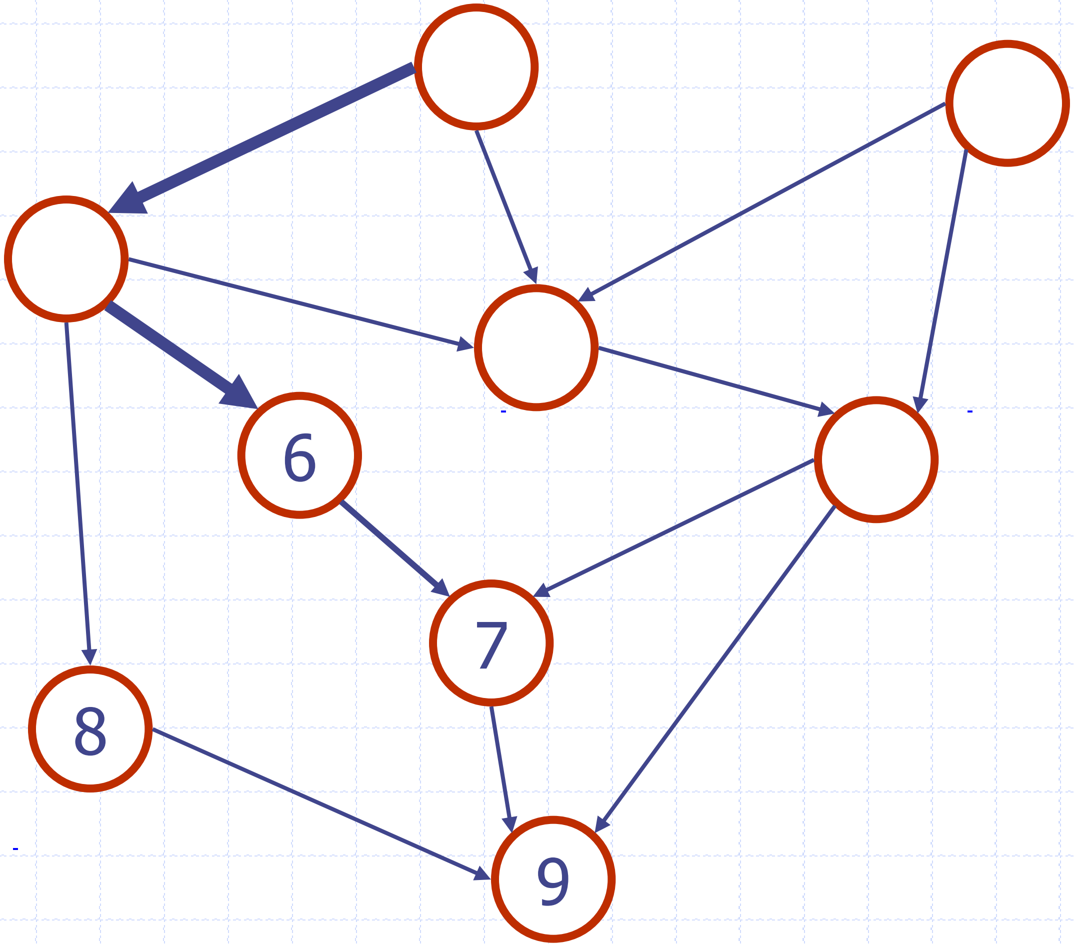
Topological Sorting Example



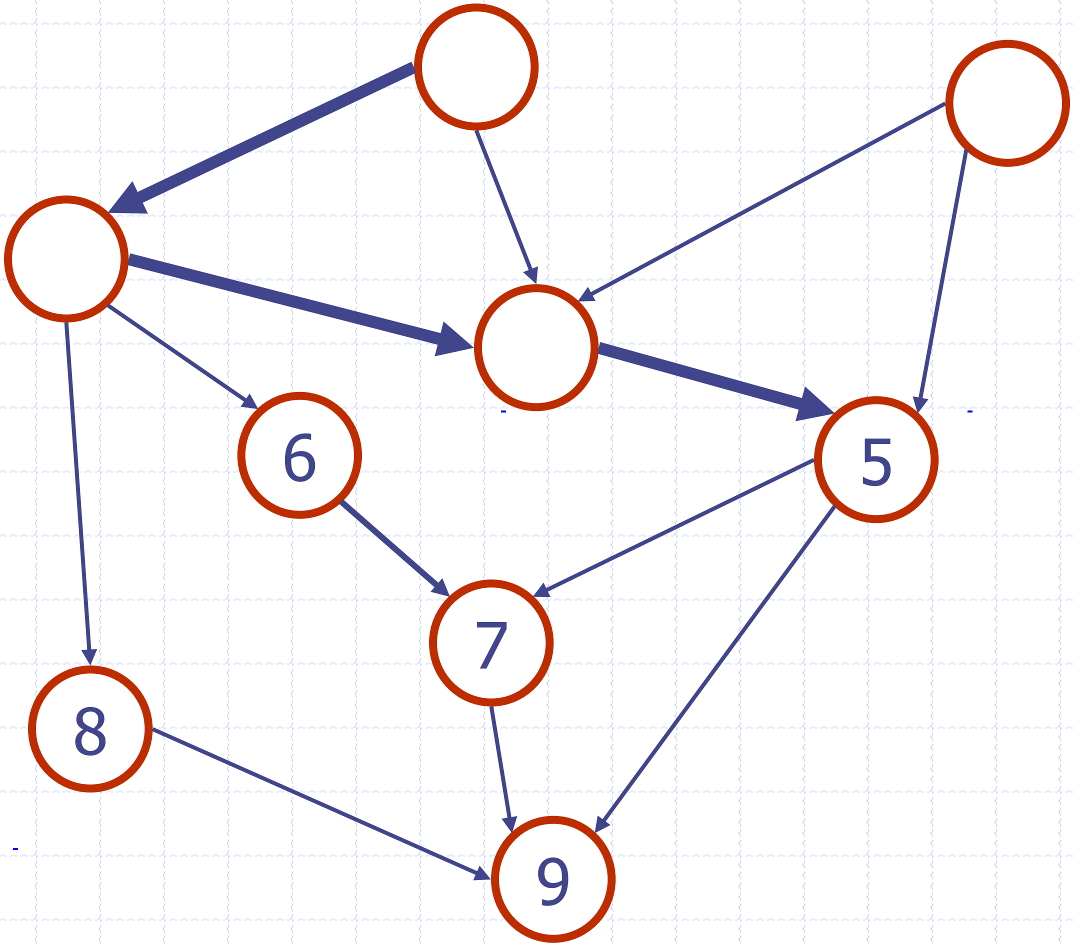
Topological Sorting Example



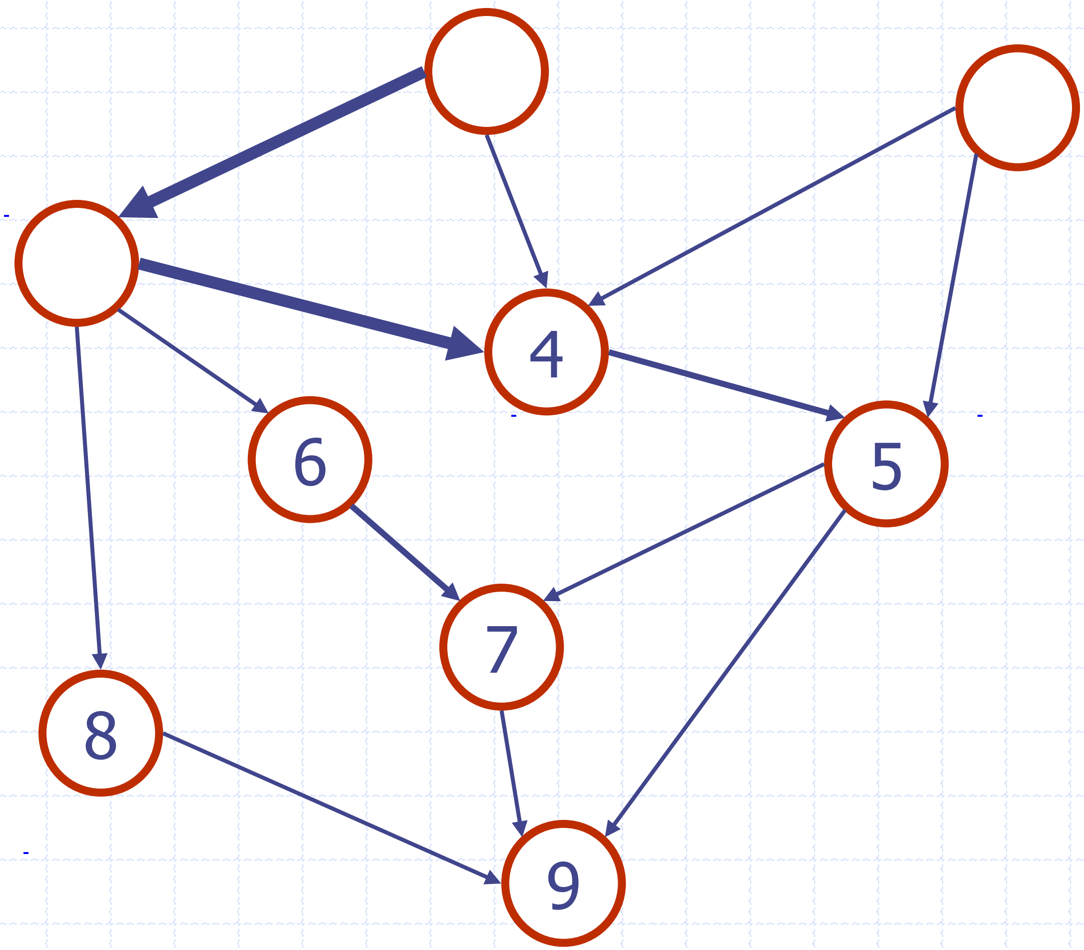
Topological Sorting Example



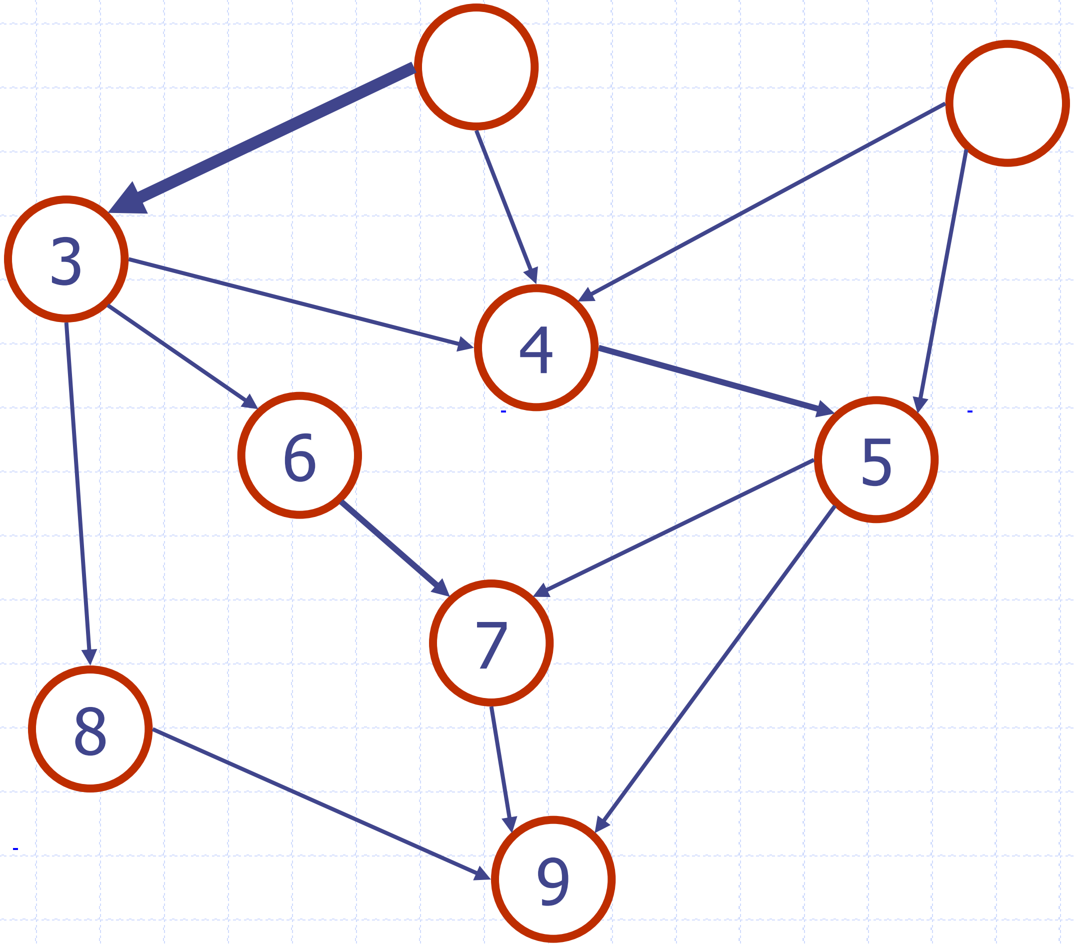
Topological Sorting Example



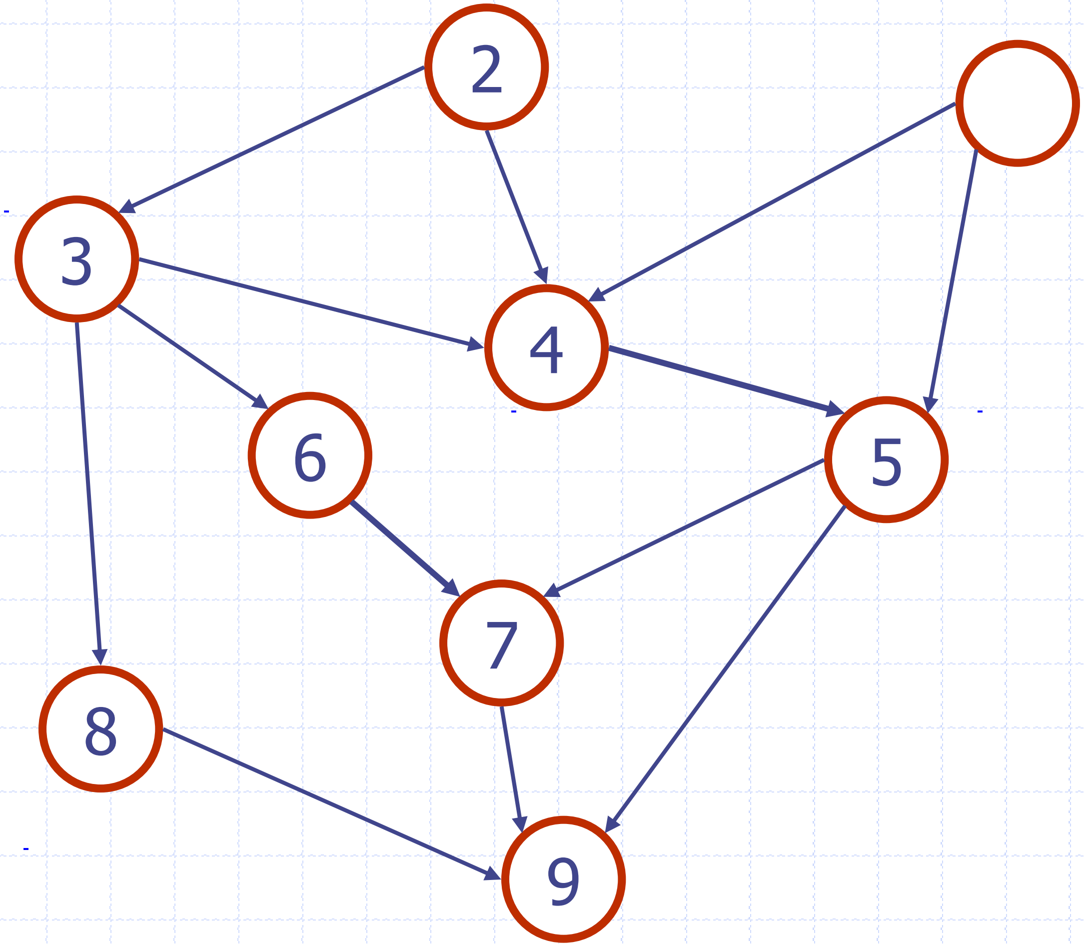
Topological Sorting Example



Topological Sorting Example



Topological Sorting Example



Topological Sorting Example