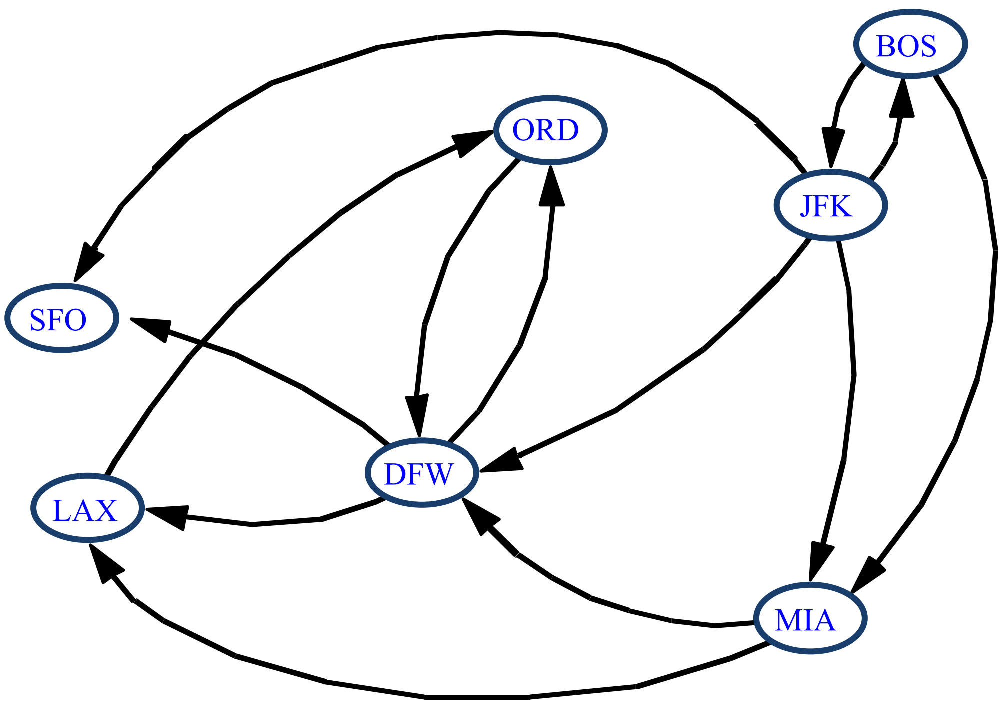
Directed Graphs (Digraphs)



Outline and Reading

Reachability (6.4.1)

- Directed DFS
- Strong connectivity

Transitive closure (6.4.2)

- The Floyd-Warshall Algorithm

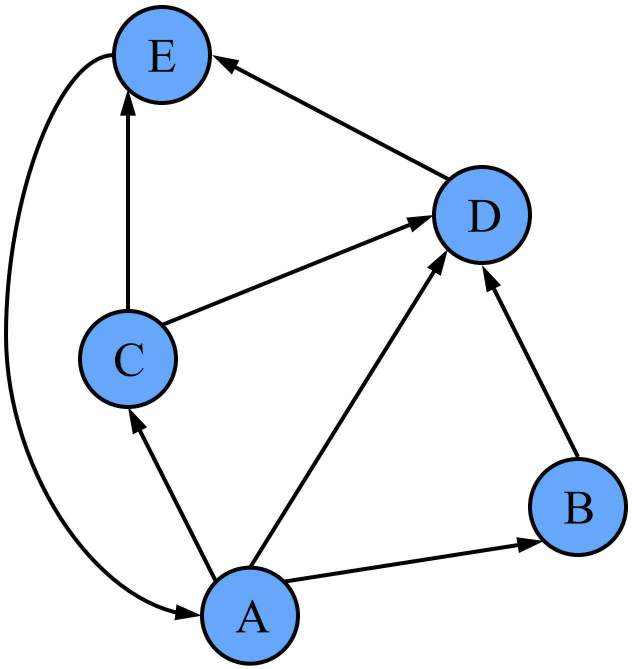
Directed Acyclic Graphs (DAGs) (6.4.4)

- Topological Sorting

Digraphs

A **digraph** (short for “directed graph”) is a graph whose edges are all directed

- Ex: Edge (a,b) goes from a to b, but not b to a.



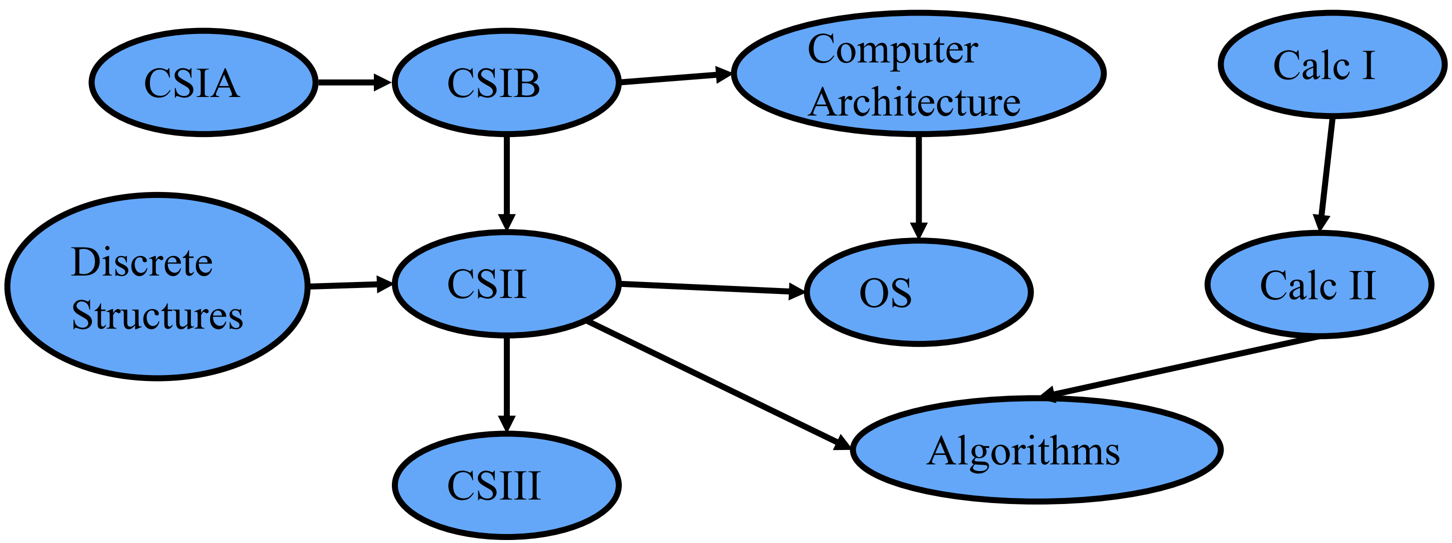
Properties:

- 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 the sets of in-edges and out-edges in time proportional to their size.

Applications include one-way streets, flights, and task scheduling.

Digraph Application

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

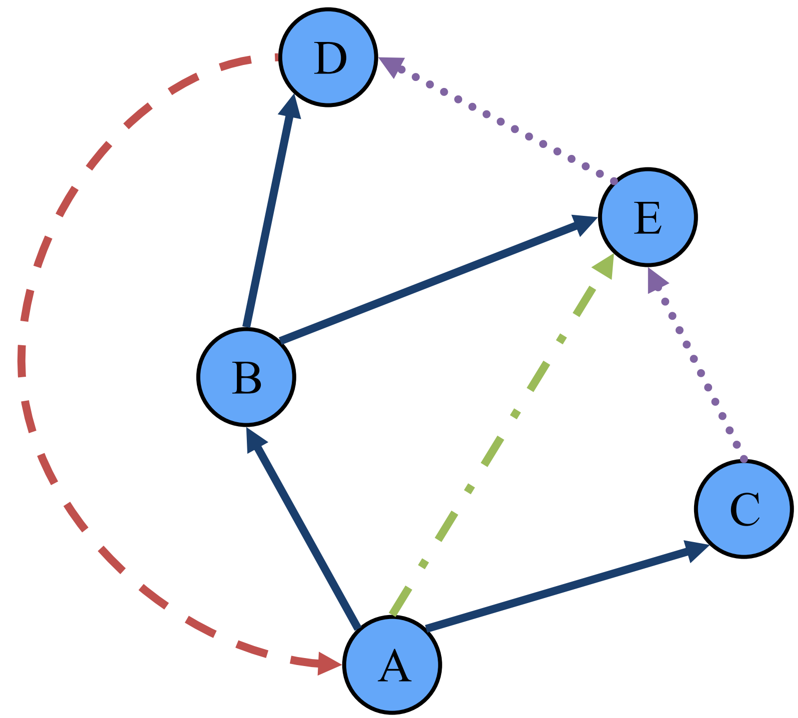


Directed DFS

- 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***

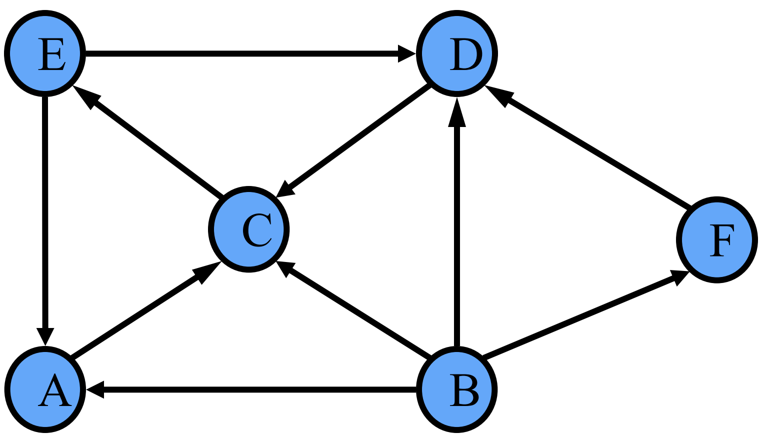


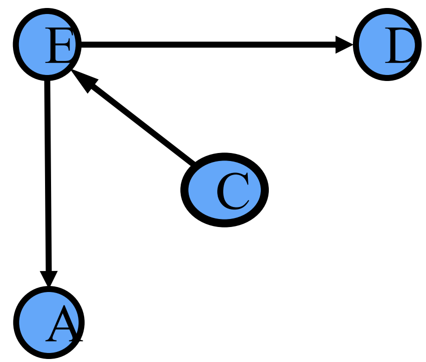
Reachability

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

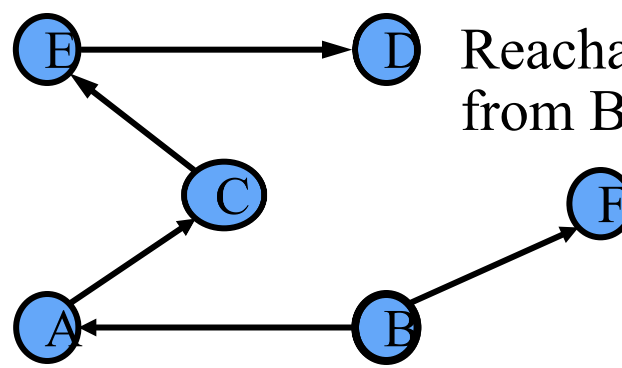
Applications:

- Dead code detection/elimination
- Garbage collection



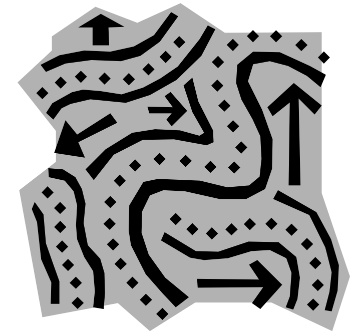


Reachable from C



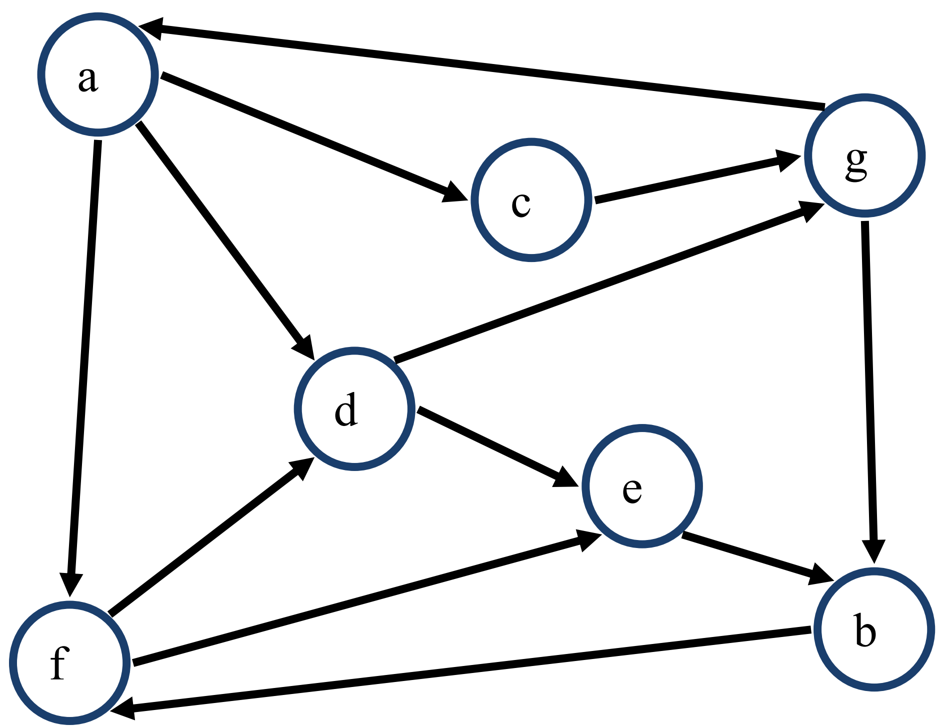
Reachable from B

Strong Connectivity

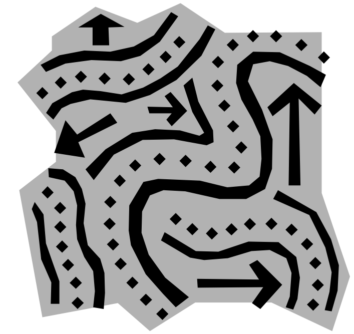


Each vertex can reach all other vertices

- How can we test if G is strongly connected?



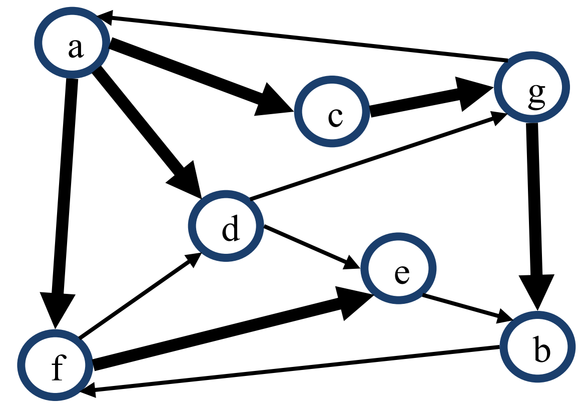
Strong Connectivity Algorithm



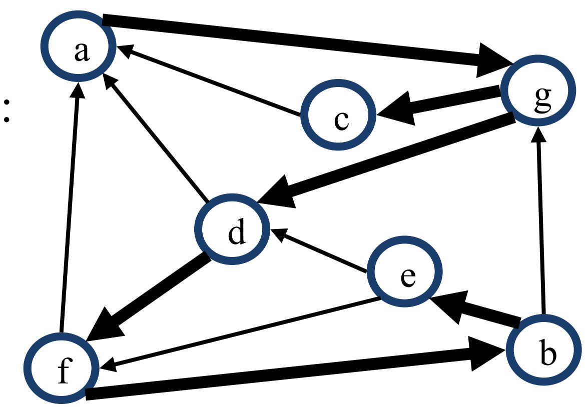
Determine if G is strongly connected

- 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"

G:



G' :

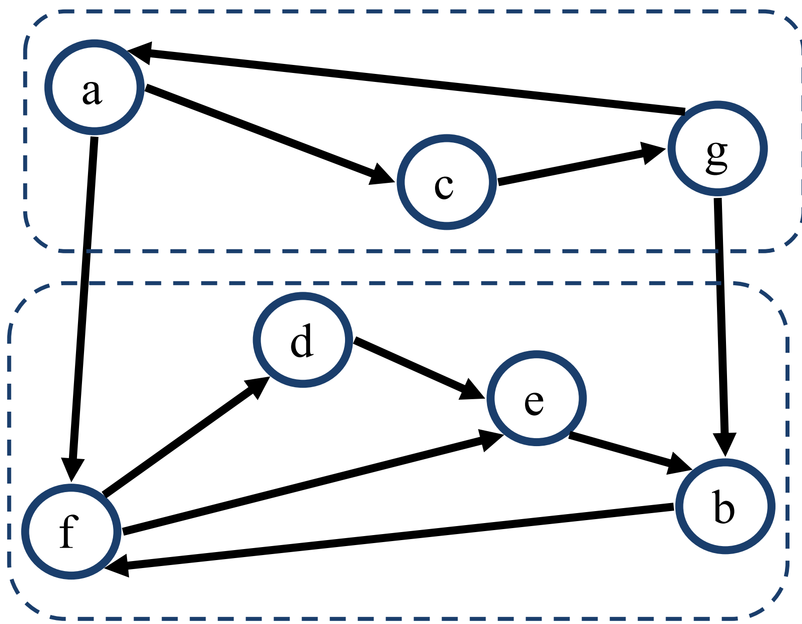


Running time: $O(n+m)$.

Strongly Connected Components

A strongly connected component is a maximal subgraph 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 }

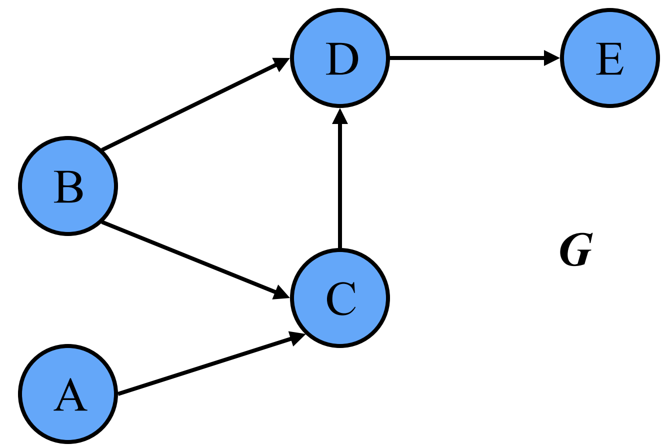
{ f , d , e , b }

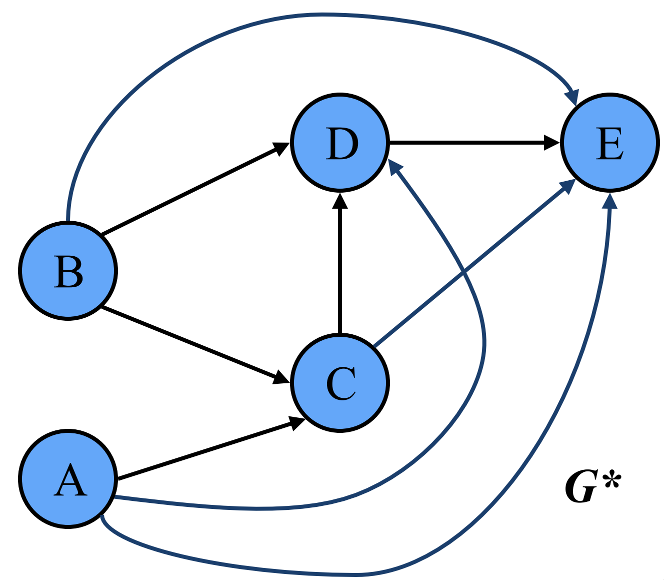
Transitive Closure

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.



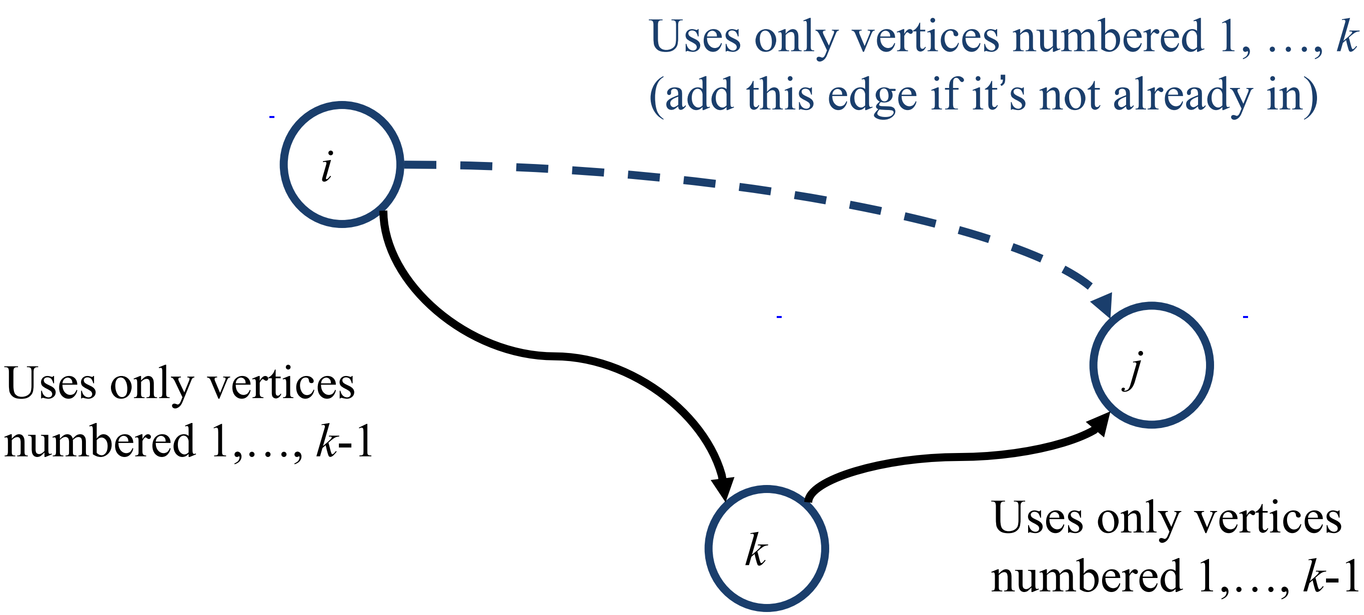


Computing the Transitive Closure

- One idea: perform DFS starting at each vertex
  - This is $O(n(n+m))$ time
  - Recall that m is $O(n^2)$
- Second idea: use dynamic programming
  - Observe that 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.
  - This becomes part of our subproblem characterization
  - This is known as **Floyd-Warshall's algorithm**, which runs in $O(n^3)$ time using an adjacency matrix

Floyd-Warshall Transitive Closure

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

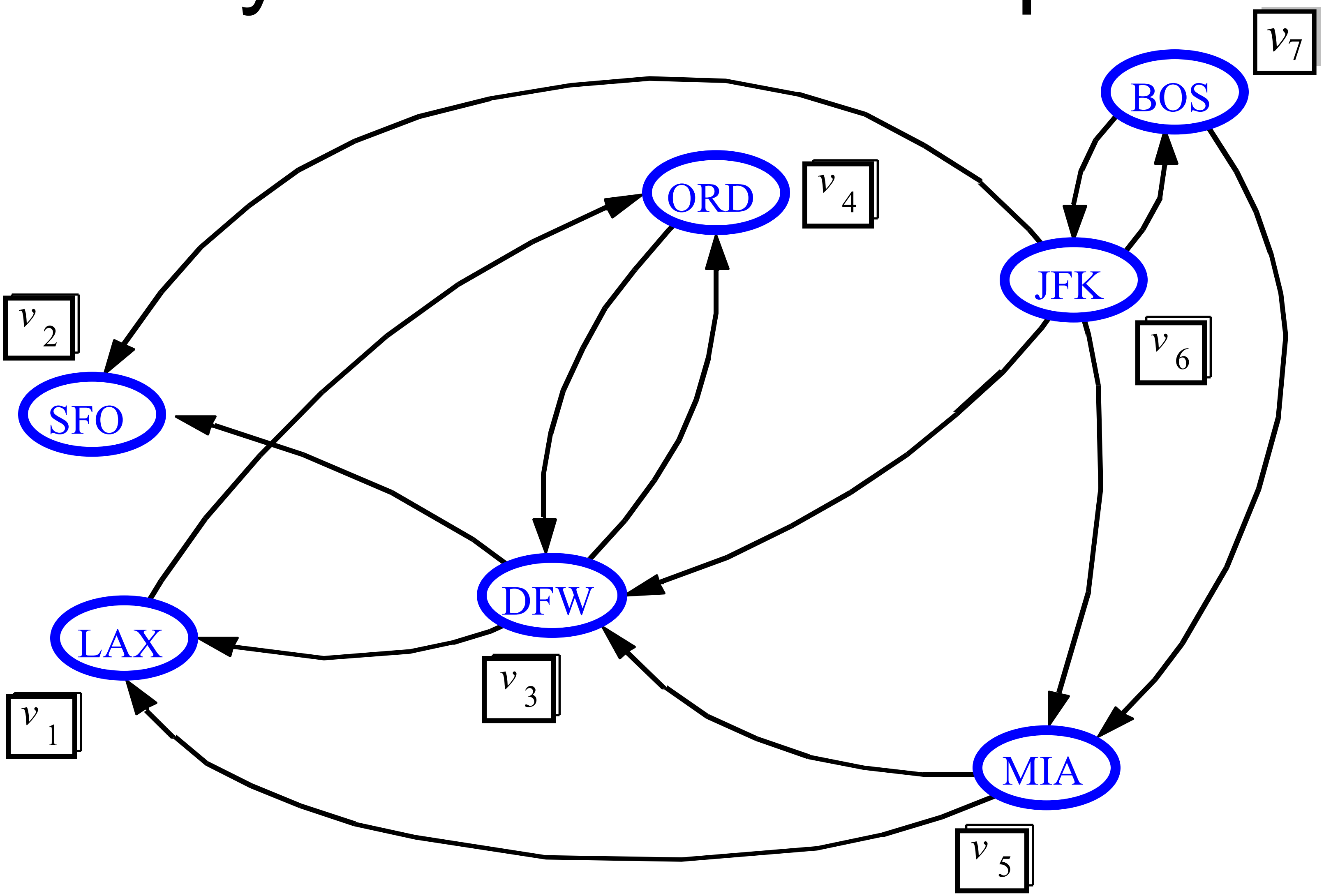


Floyd-Warshall's Algorithm

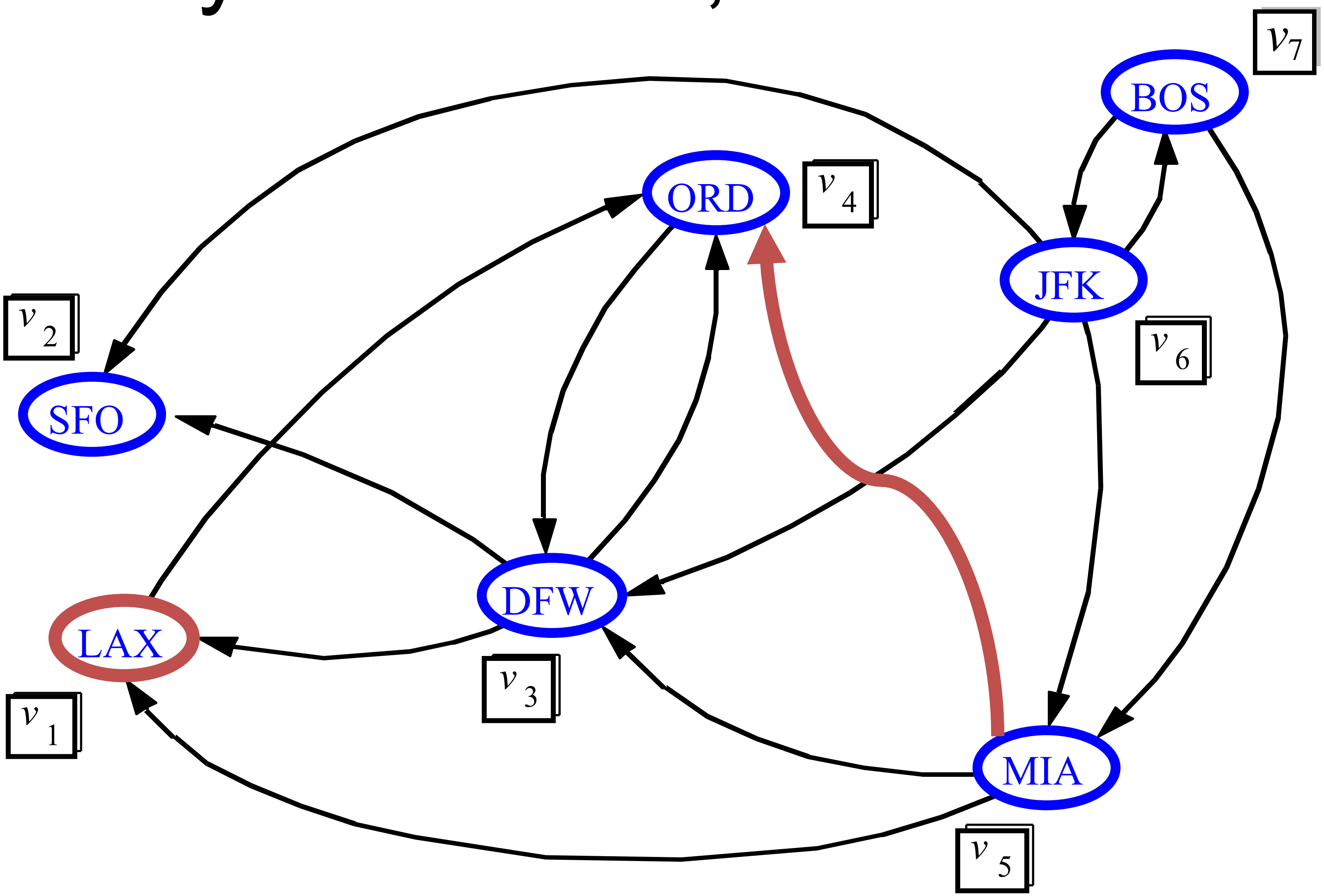
- Numbers the vertices of G as v_1 , …, v_n and computes a series of digraphs G_0, …, G_n
  - $G_0 = G$
  - G_k has a directed edge (v_i, v_j) if G has a directed path from v_i to v_j with intermediate vertices in the set $\{v_1, \ldots, v_k\}$
- We have that $G_n = G^*$
- In phase k, digraph G_k is 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 ← 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
```

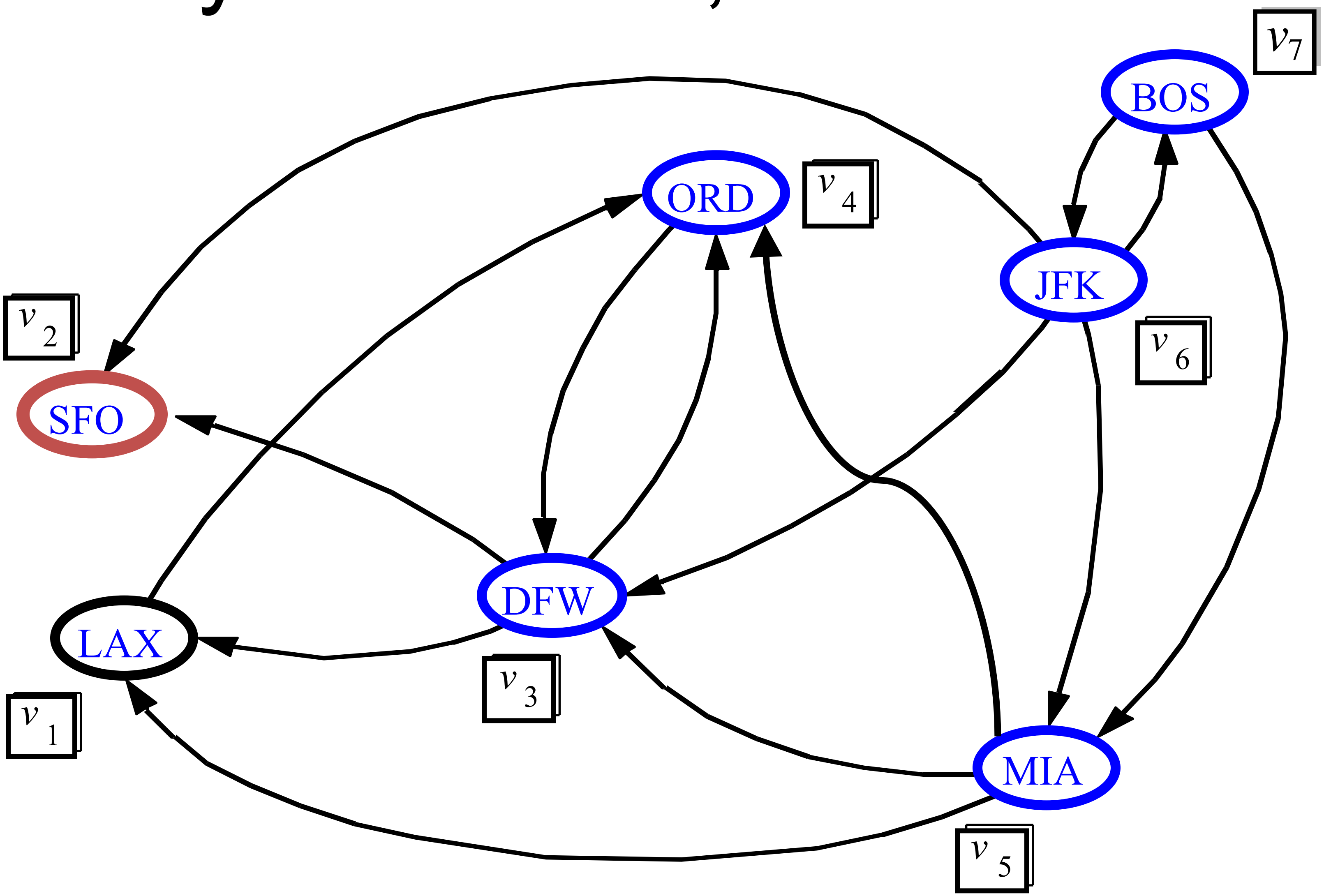
Floyd-Warshall Example



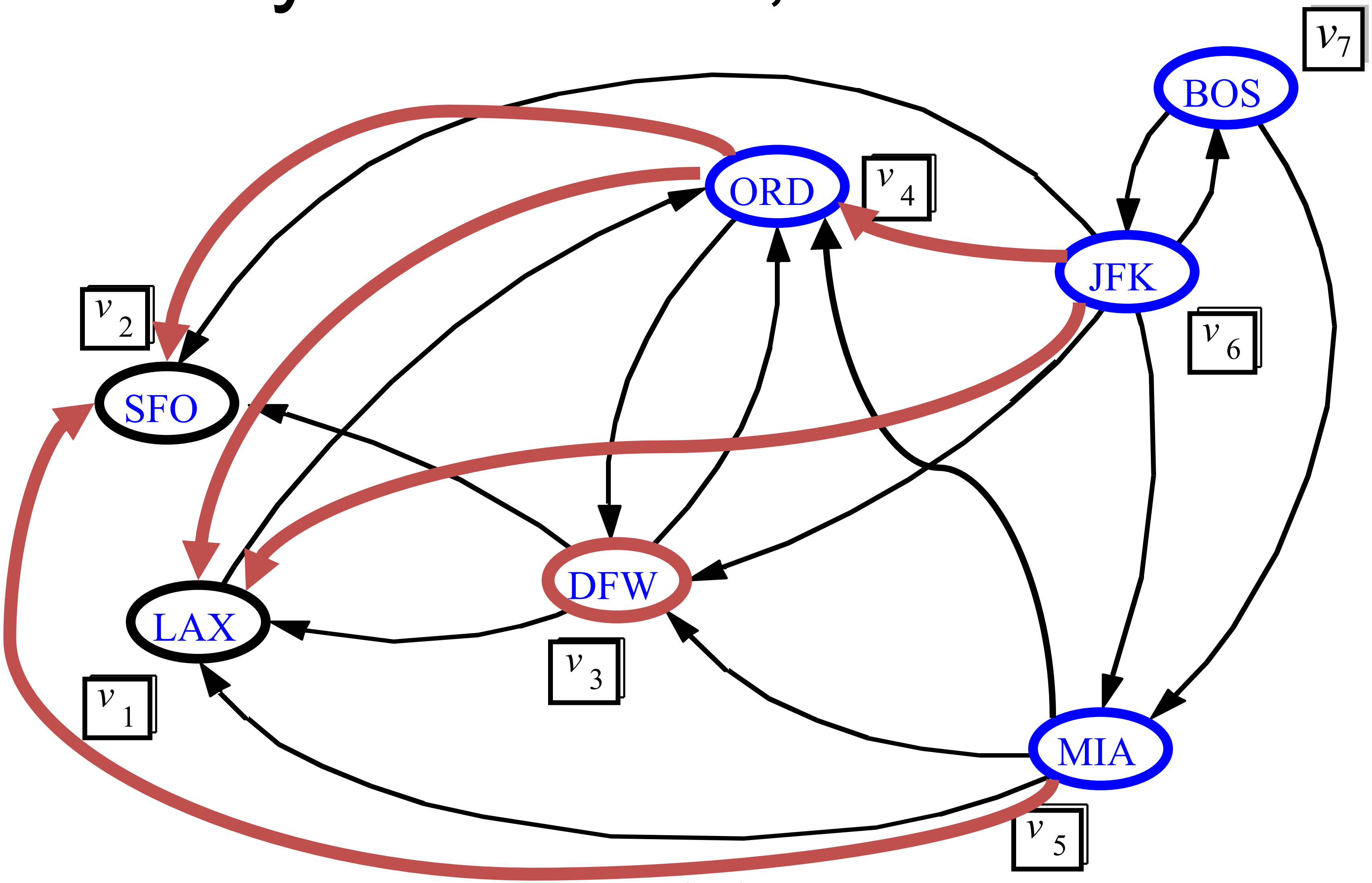
Floyd-Warshall, Iteration 1



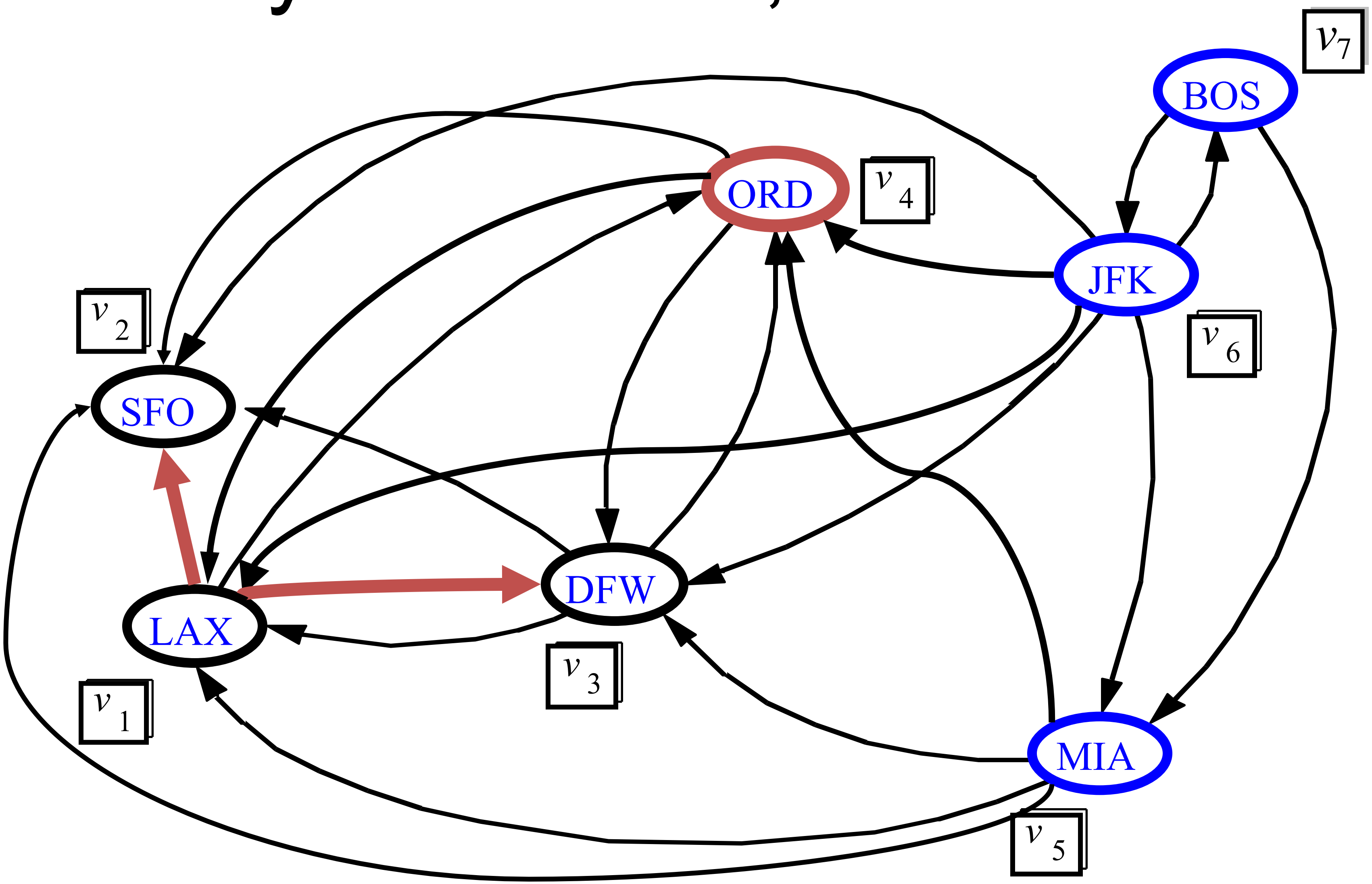
Floyd-Warshall, Iteration 2



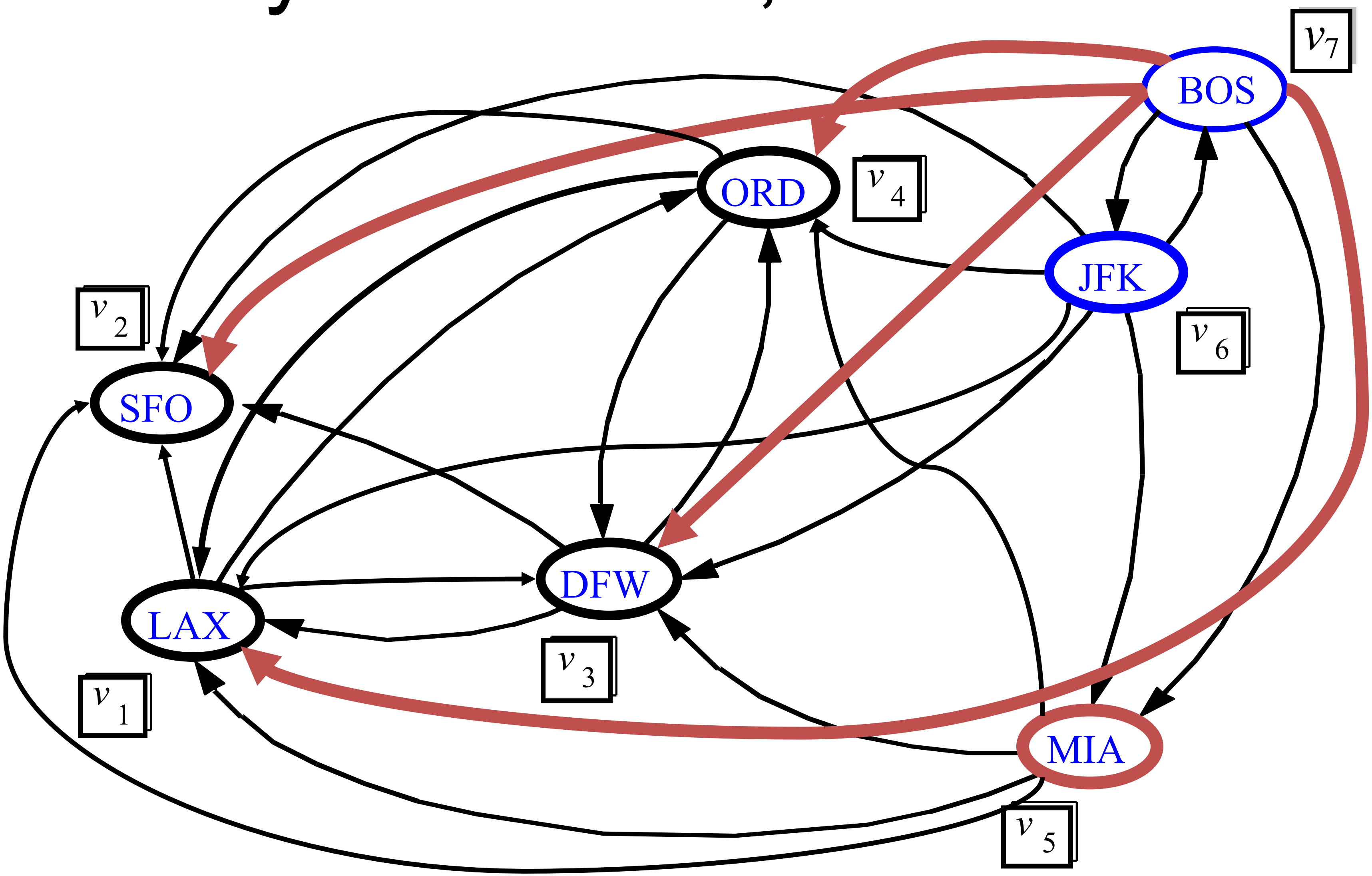
Floyd-Warshall, Iteration 3



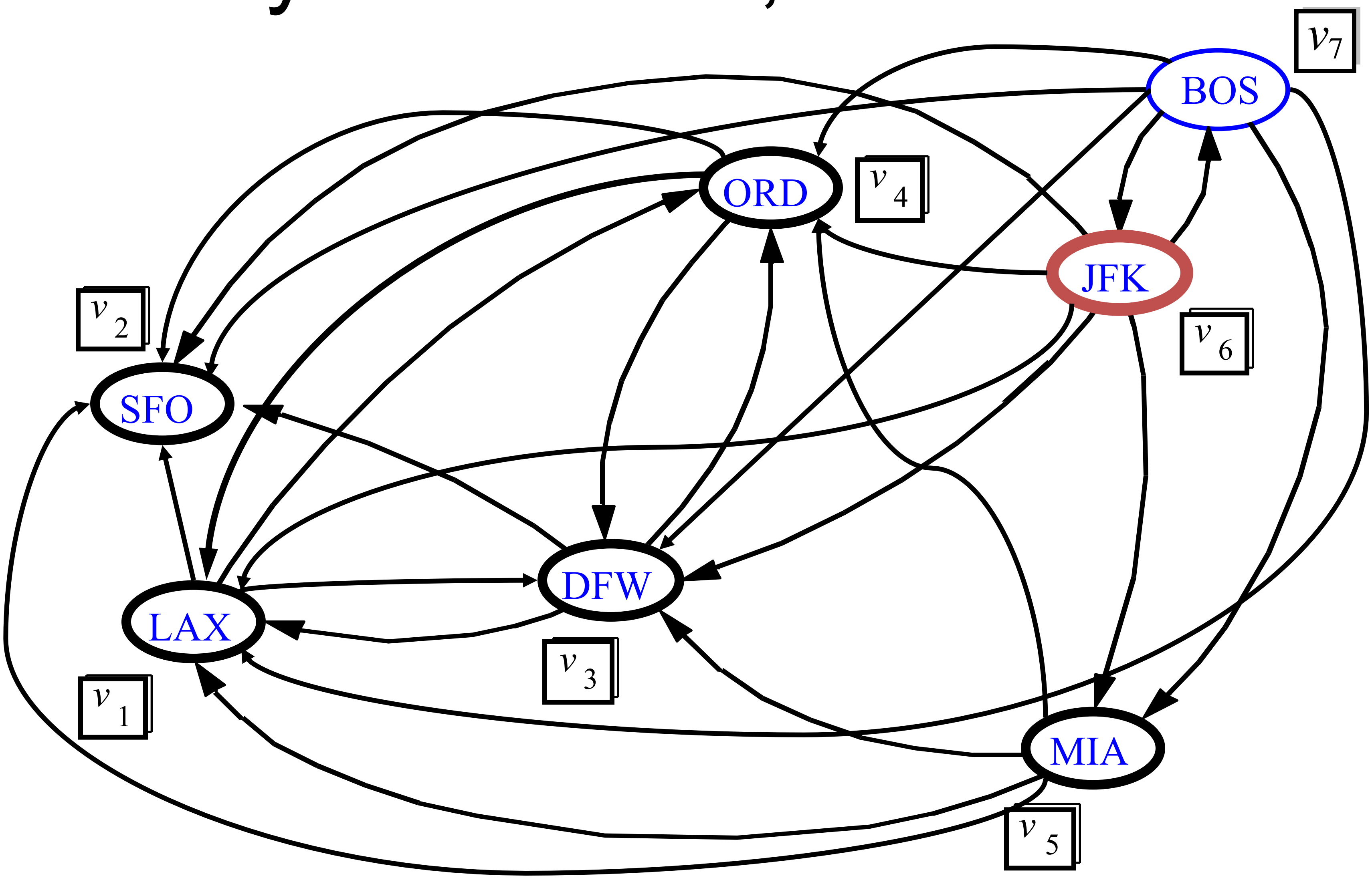
Floyd-Warshall, Iteration 4



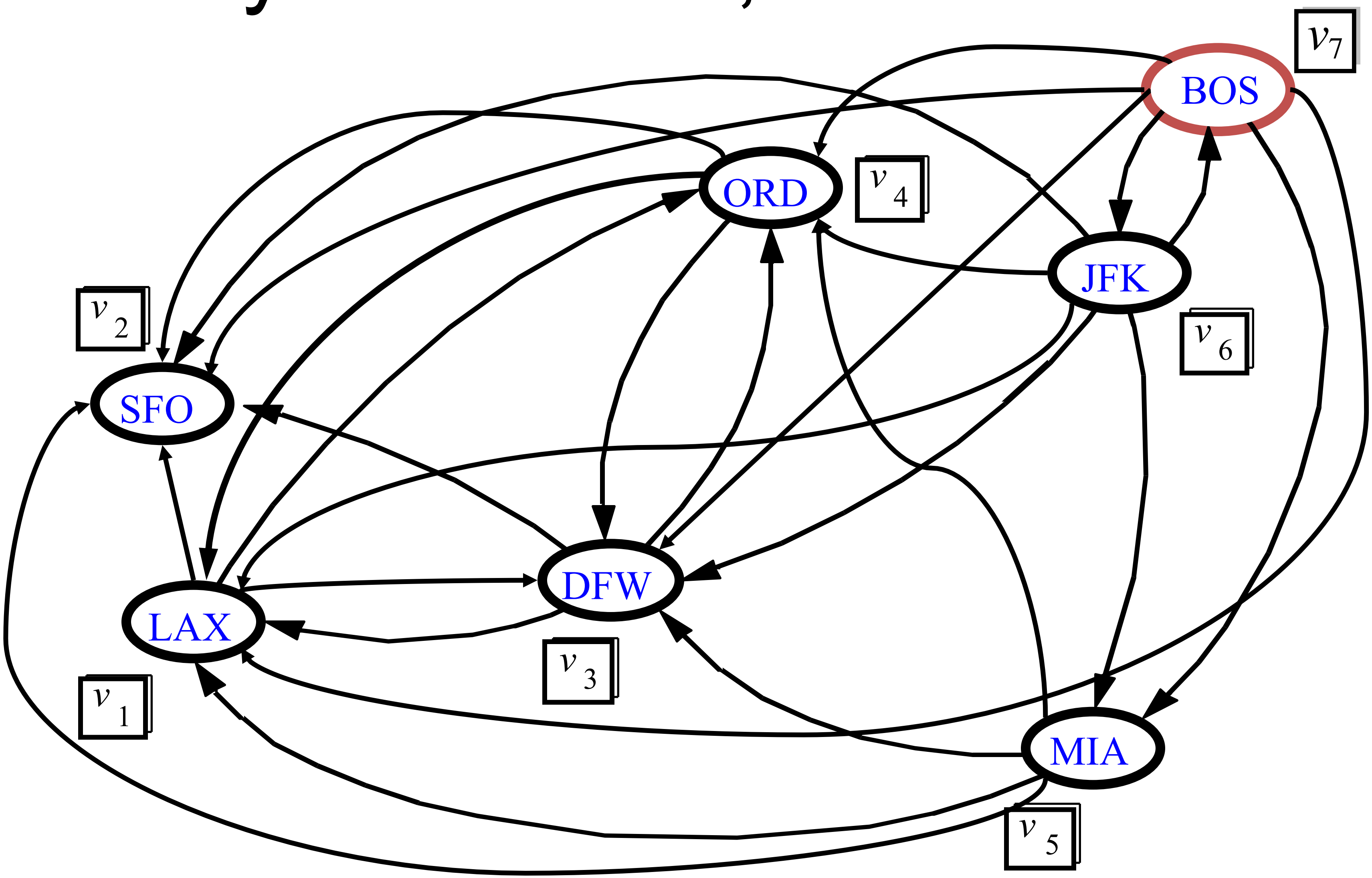
Floyd-Warshall, Iteration 5



Floyd-Warshall, Iteration 6

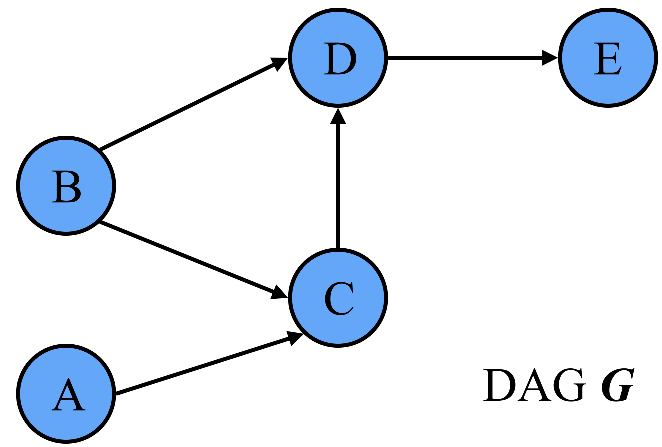


Floyd-Warshall, Conclusion



DAGs and Topological Ordering

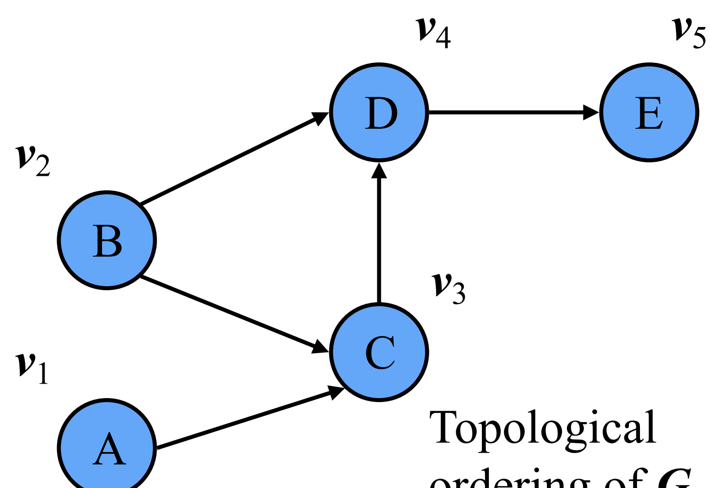
- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering $v_1, \ldots, v_n$ of the vertices such that for every edge (v_i, v_j), we have $i < j$
- Ex: in a task scheduling digraph, a topological order is a task sequence that satisfies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



DAG G

Theorem

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

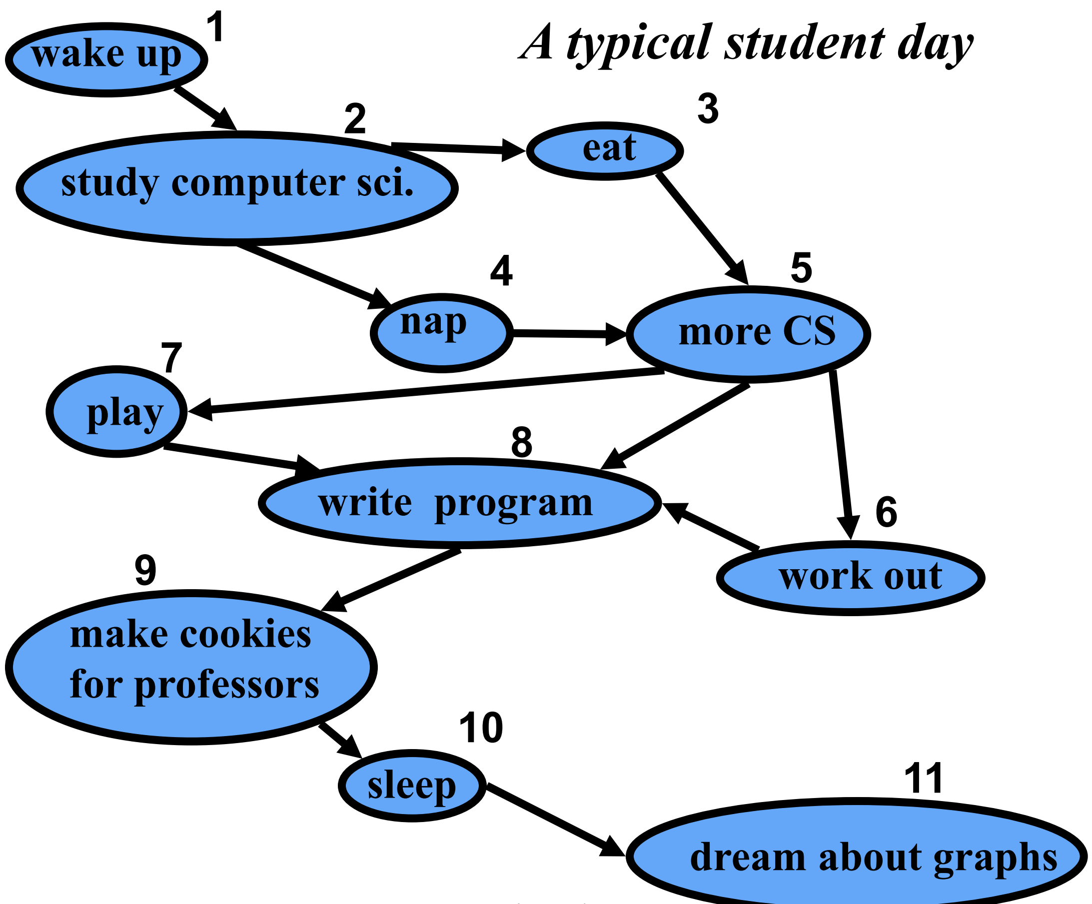


Topological ordering of G

xkcd #754

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***)
    $H \leftarrow G$ // 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). How…?

Topological Sorting Algorithm using DFS

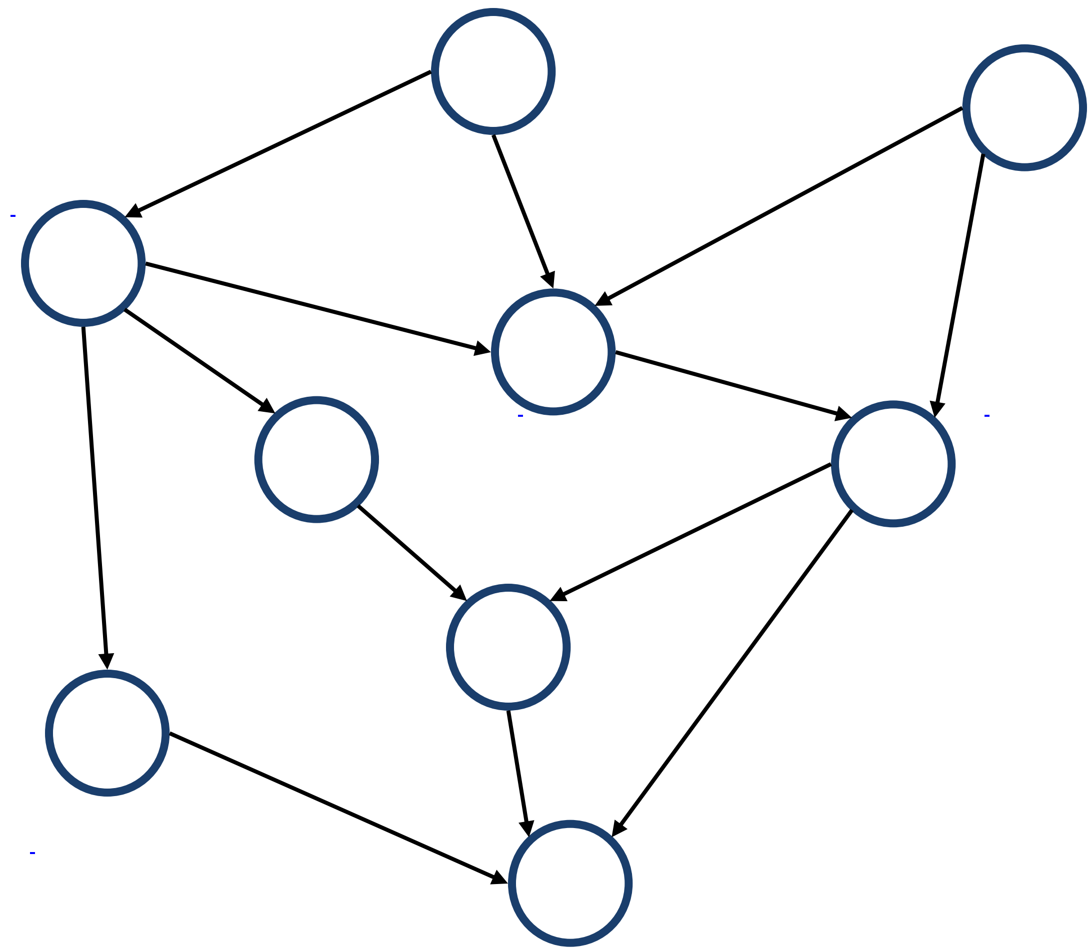
Simulate the algorithm by using DFS

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*)

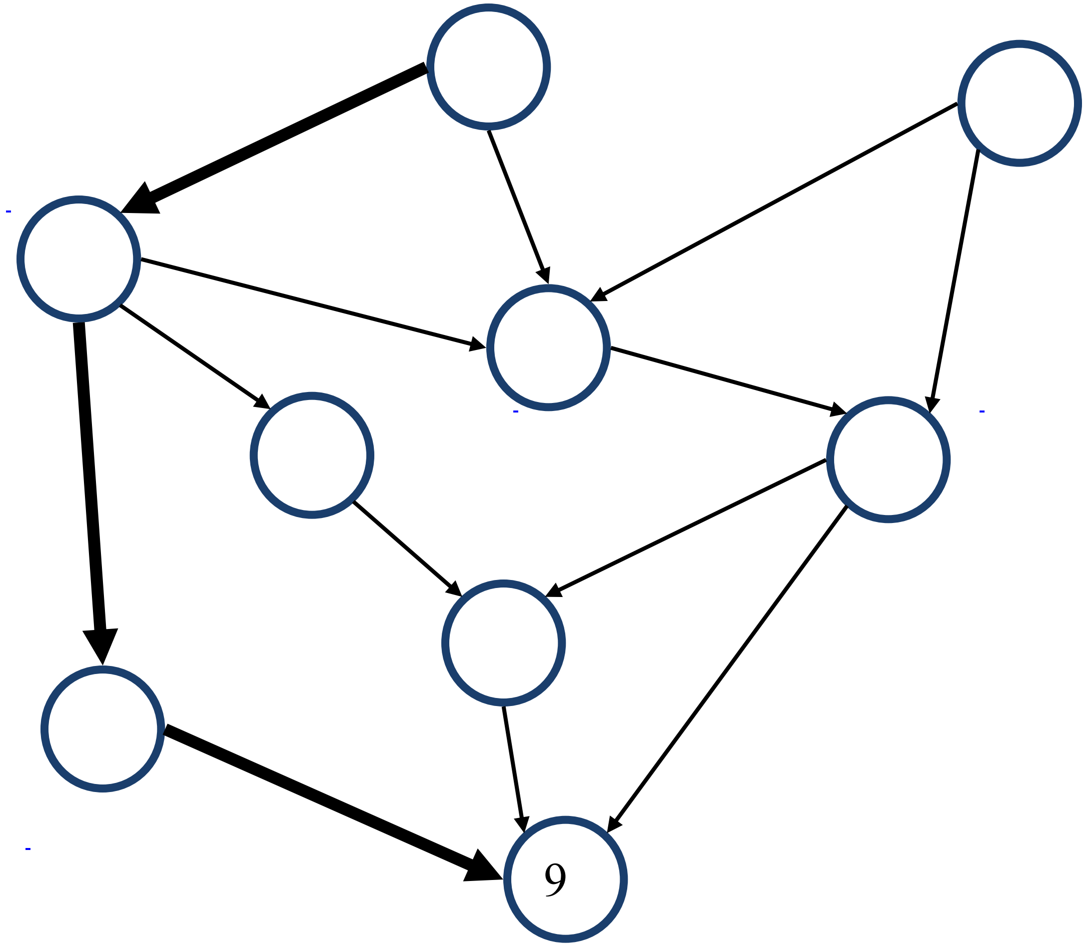
Algorithm *topologicalDFS*(*G, v*)
  Input graph *G* and a start vertex *v* of *G*
  Output labeling of the vertices of *G*
    in the connected component of *v*
  setLabel(*v, VISITED*)
  for all $e \in G.outgoingIncidentEdges(v)$
    if $getLabel(e) = UNEXPLORED$
      $w \leftarrow opposite(v,e)$
      if $getLabel(w) = UNEXPLORED$
        setLabel(*e, DISCOVERY*)
        topologicalDFS(*G, w*)
      else
        {*e* is a forward or cross edge}
  Label *v* with topological number *n*
  $n \leftarrow n - 1$

- O(n+m) time.

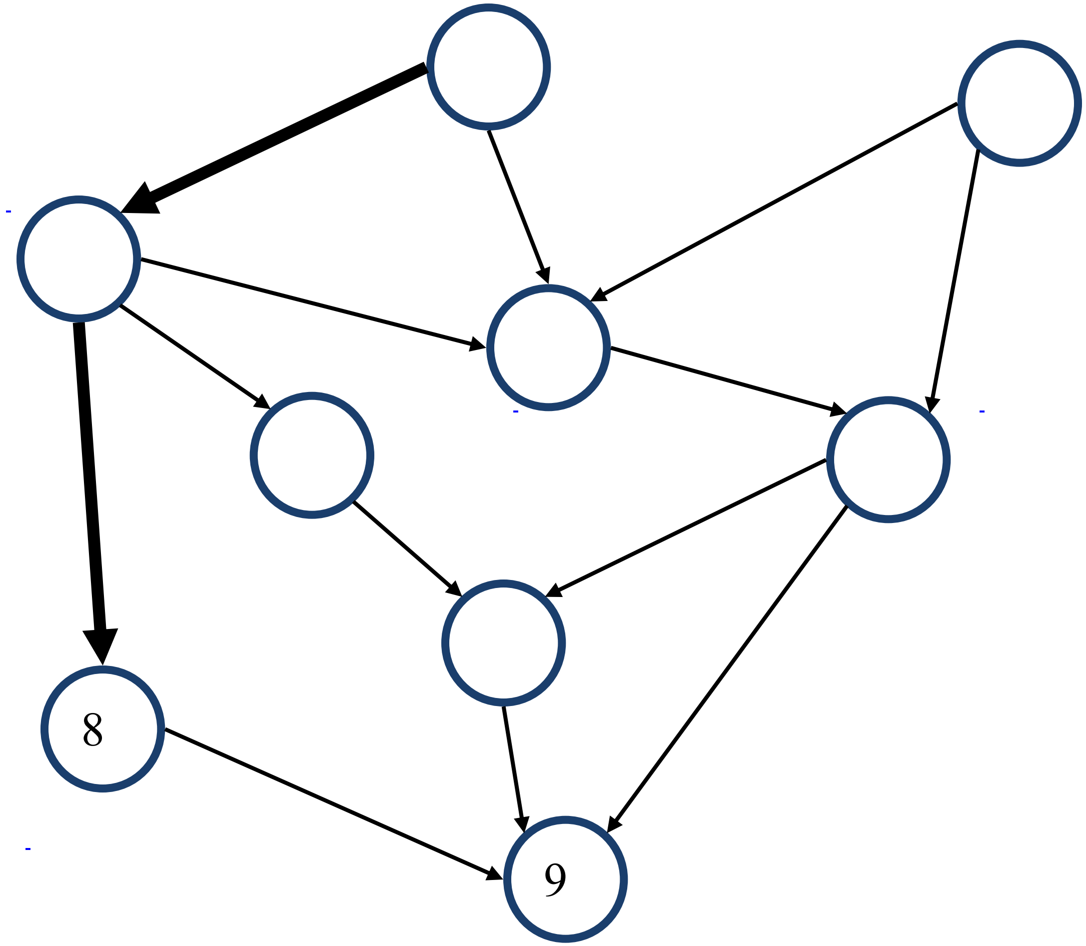
Topological Sorting Example



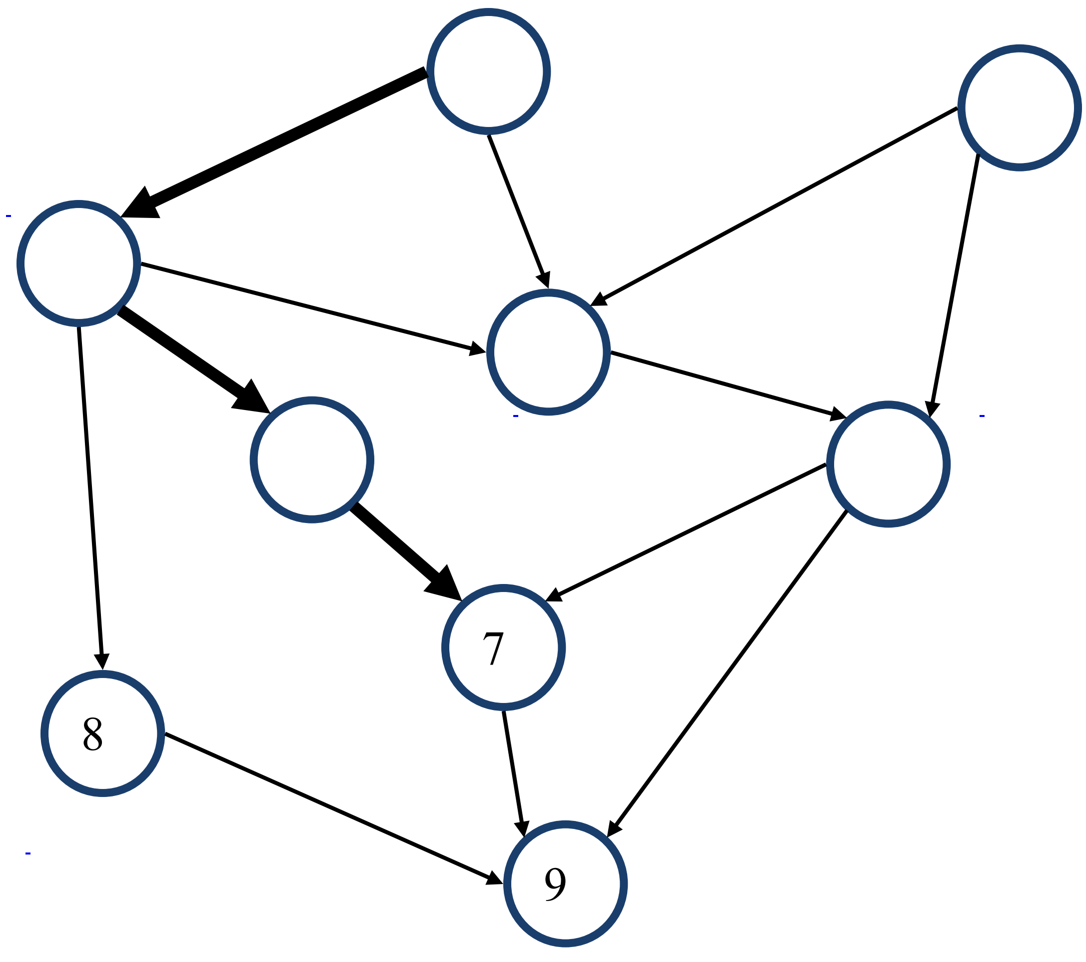
Topological Sorting Example



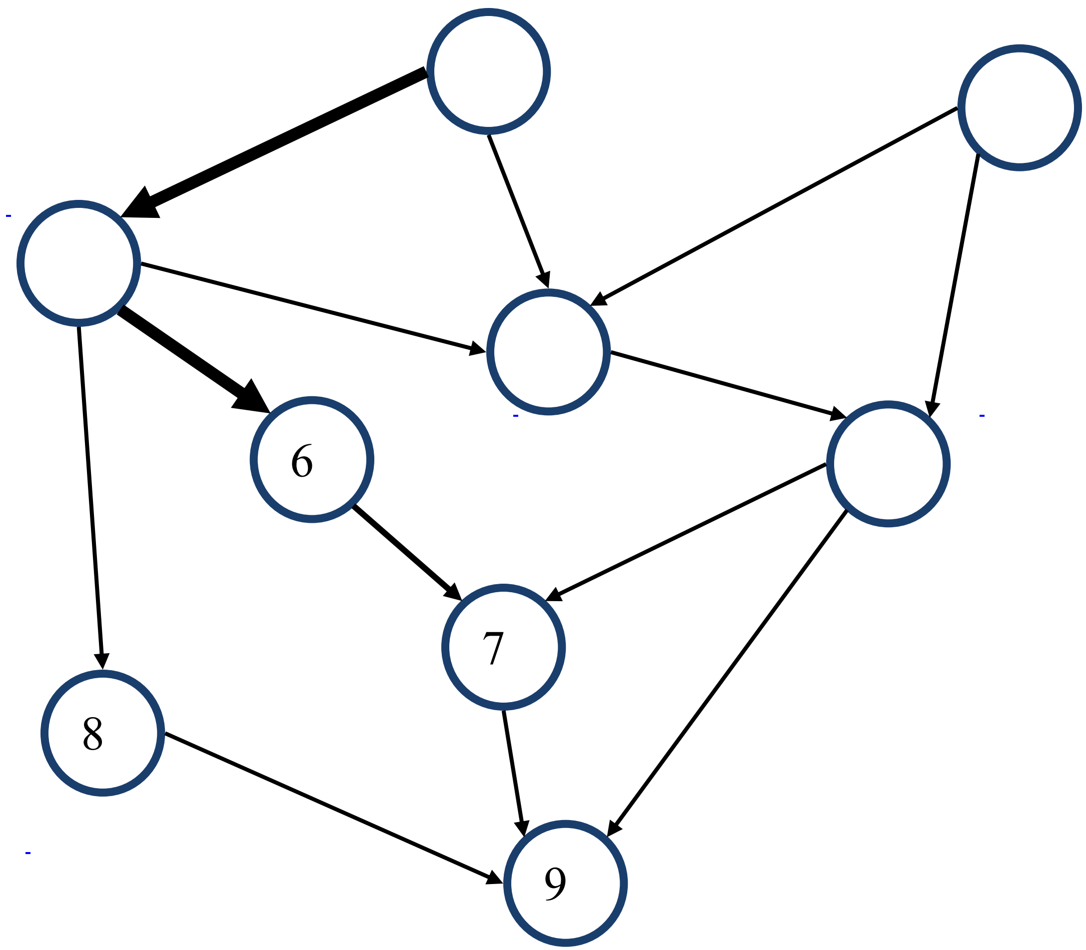
Topological Sorting Example



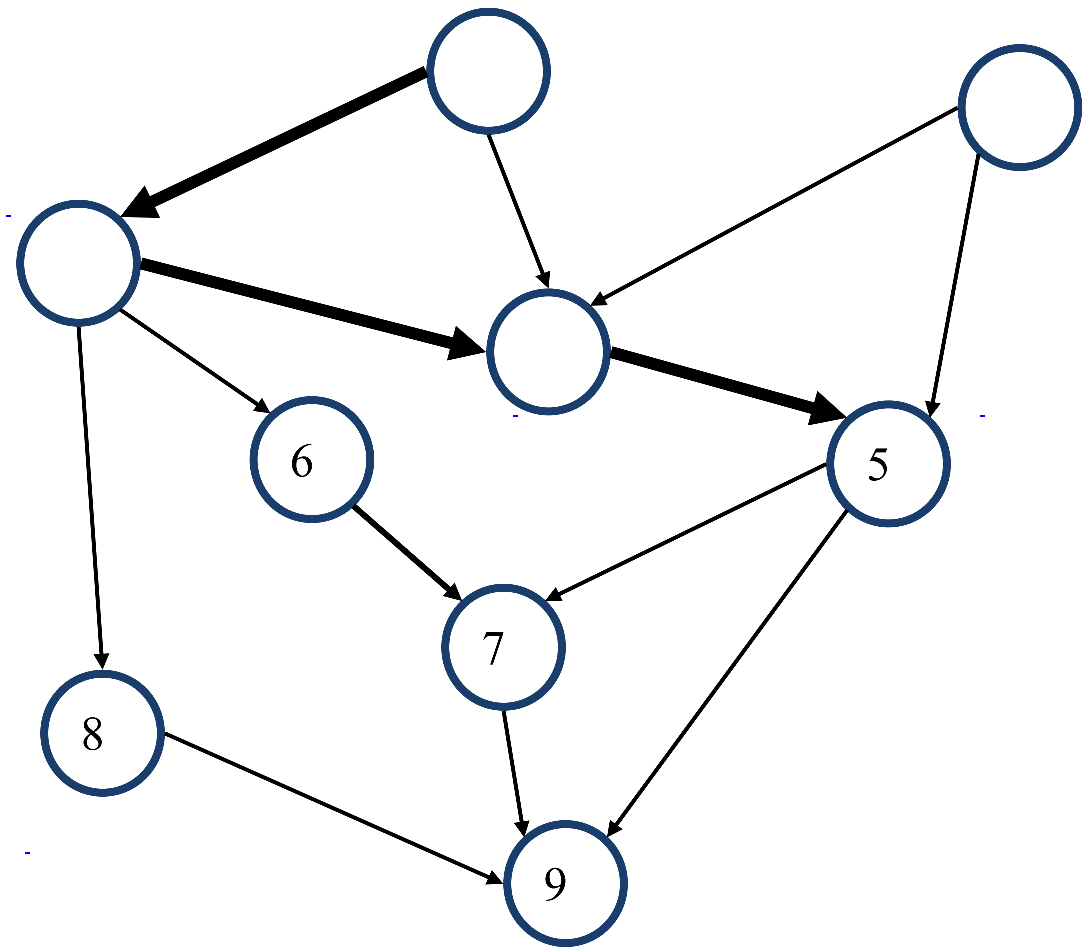
Topological Sorting Example



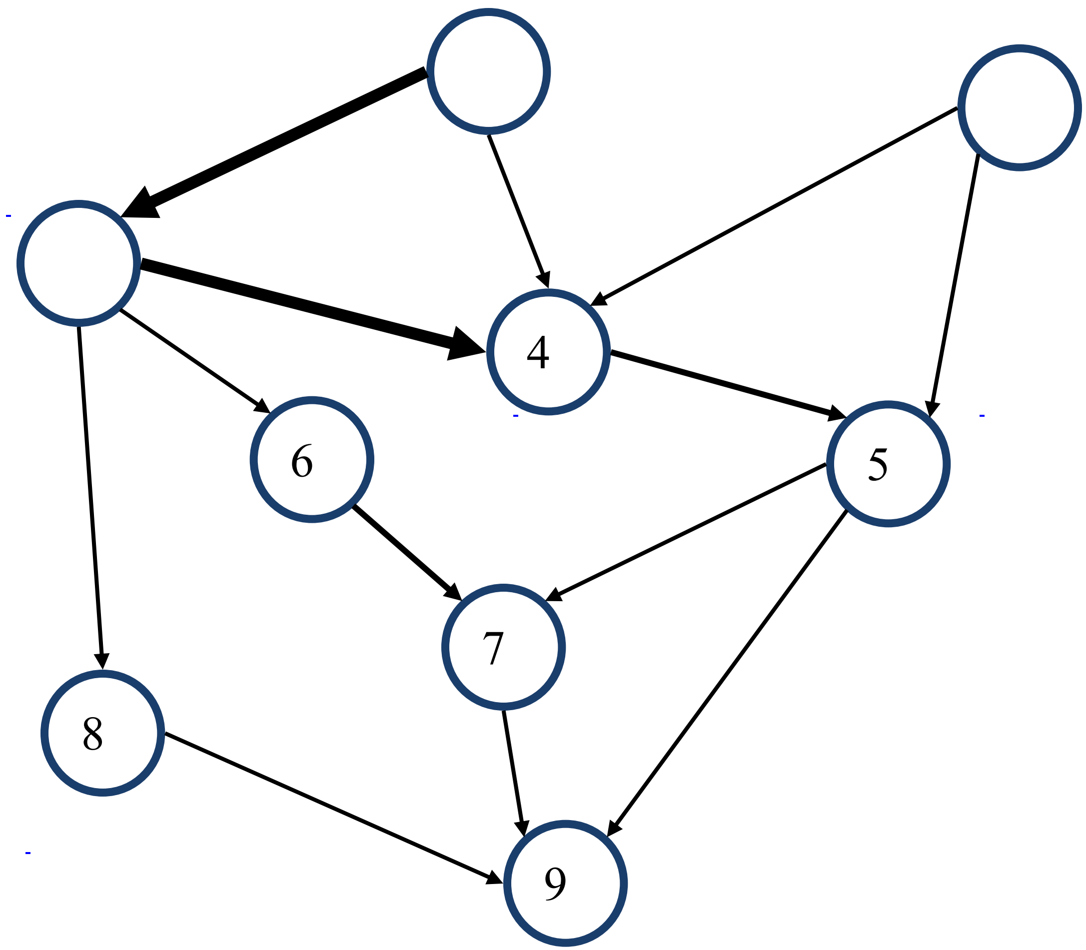
Topological Sorting Example



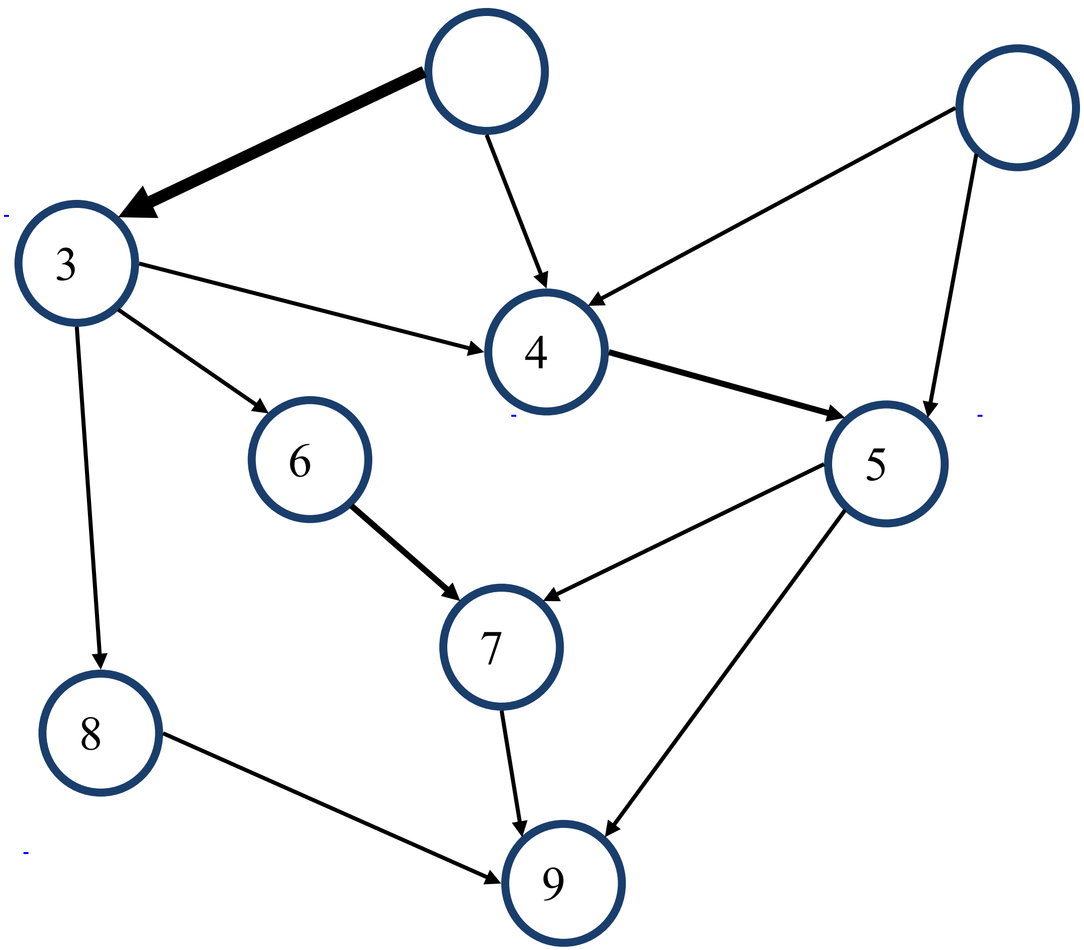
Topological Sorting Example



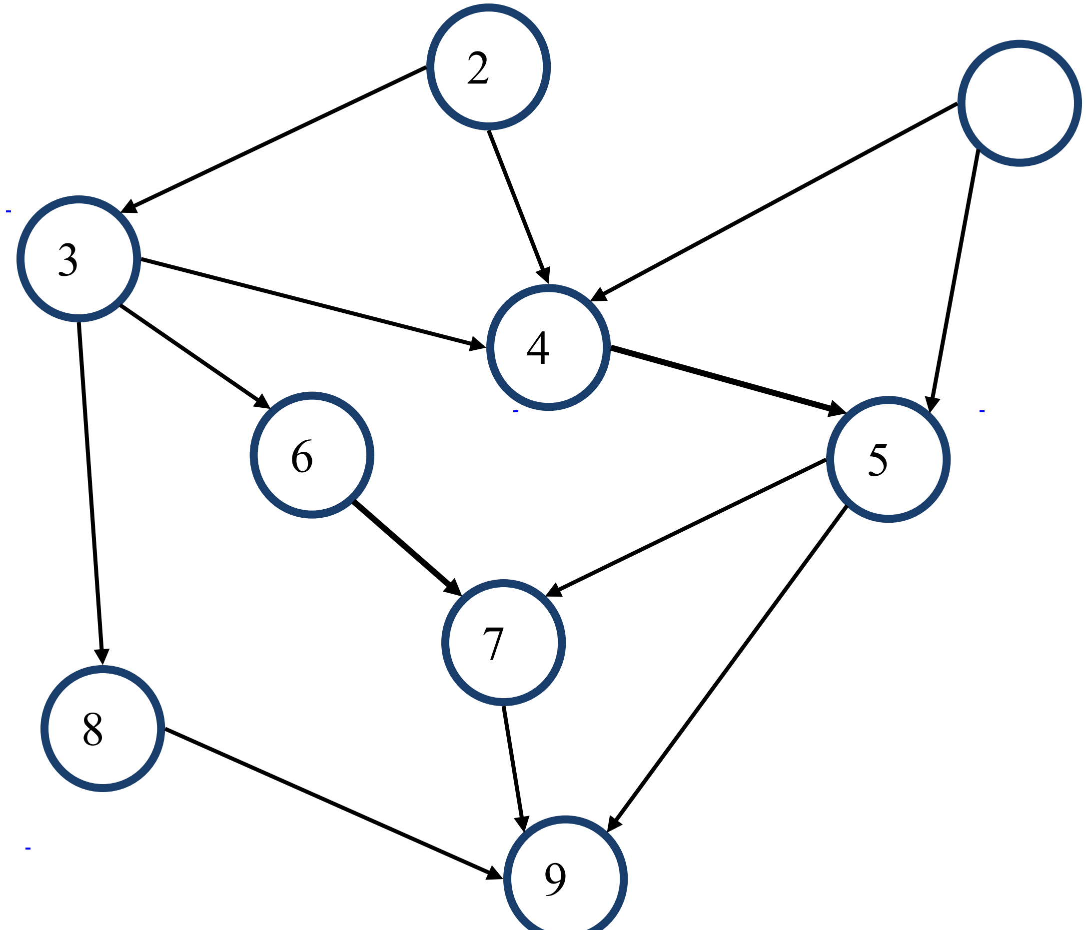
Topological Sorting Example



Topological Sorting Example



Topological Sorting Example



Topological Sorting Example