Data Structures & Algorithms 2

Directed Graphs

Lecturer: Dr. Hadi Tabatabaee
Materials: Dr. Hadi Tabatabaee
Maynooth University
Online at http://moodle.maynoothuniversity.ie

Overview

Aims

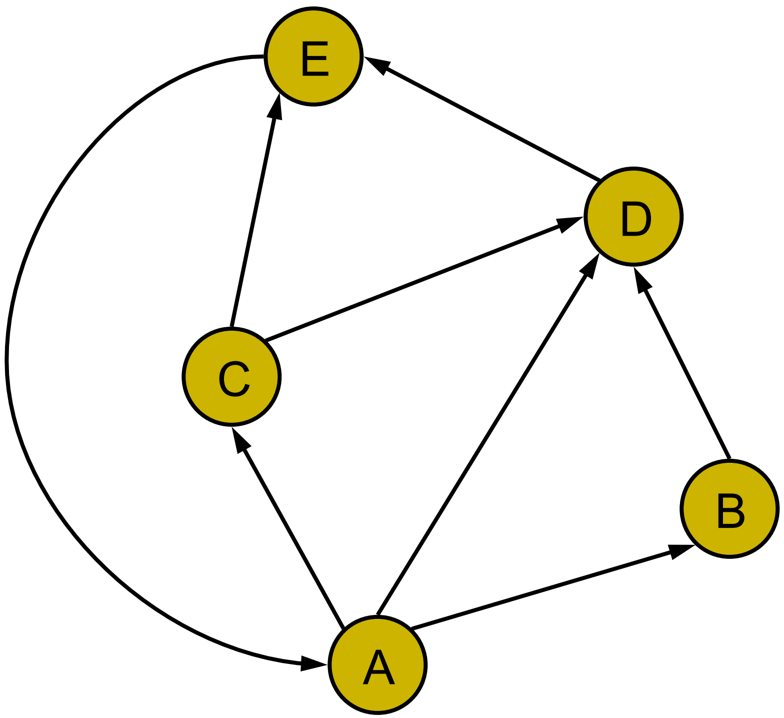
- Introduce Transitive Closure in a directed graph.
- Introduce Directed Acyclic Graphs.

Learning outcomes: You should be able to…

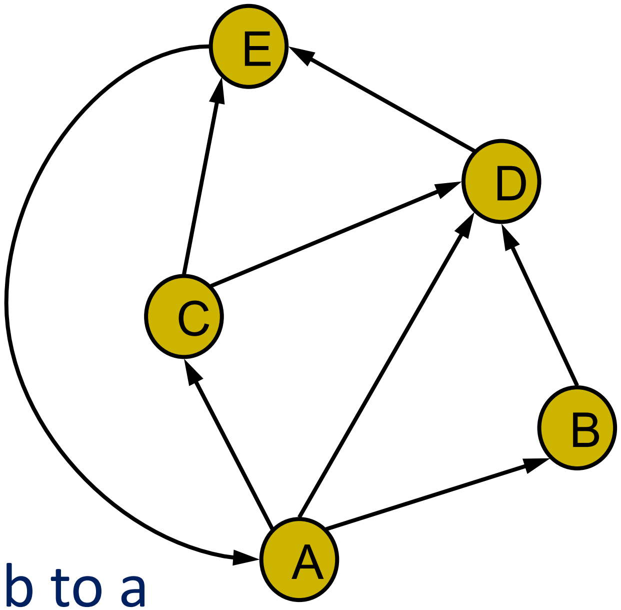
- Use Floyd-Warshall Algorithm to investigate the reachability of vertices in a directed graph.
- Use topological ordering in a directed acyclic graph.

Digraphs

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



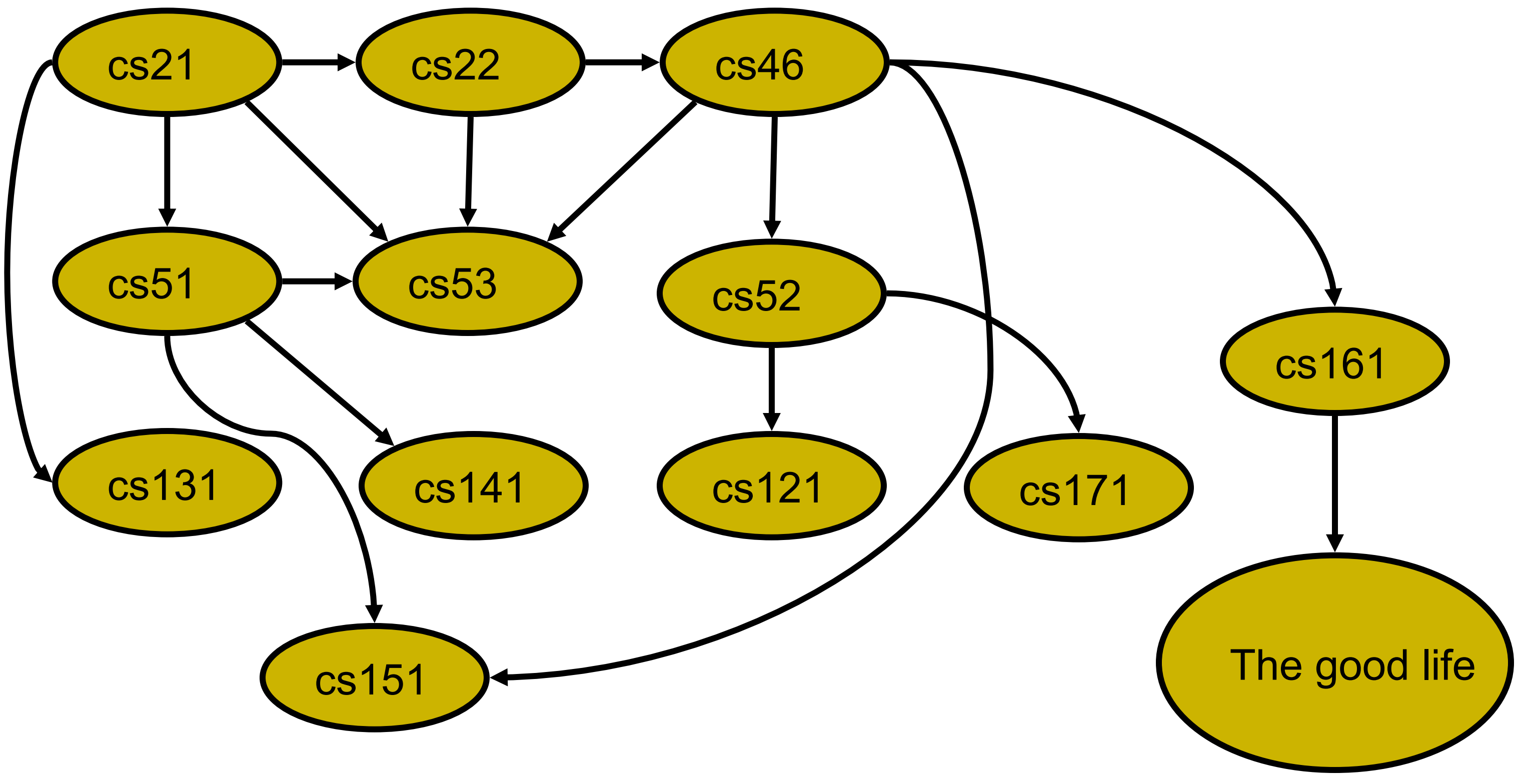
Digraph Properties



- 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 a listing of incoming edges and outgoing edges in time proportional to their size

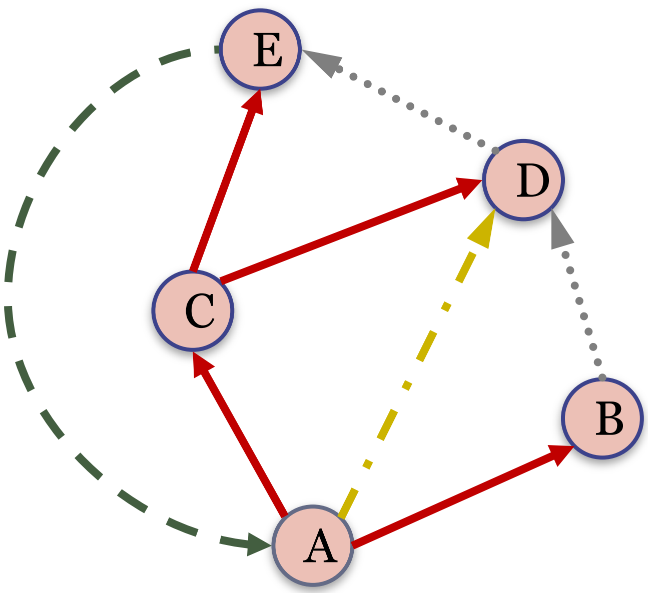
Digraph Application

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



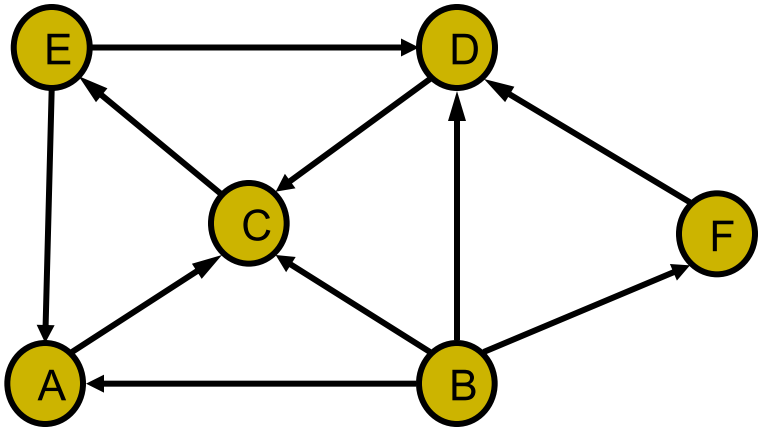
Directed DFS

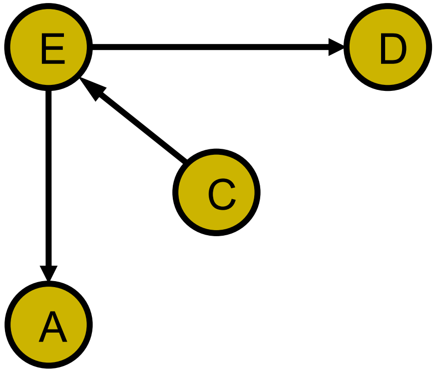
- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction.
- In the directed DFS algorithm, we have four types of edges
  - discovery edges: discovers a new vertex (unvisited)
  - back edges: connect a vertex to an ancestor in the DFS tree
  - forward edges: which connect a vertex to a descendant in the DFS tree
  - cross edges: connect a vertex to a vertex that is neither its ancestor nor its descendant
- A directed DFS starting at a vertex s determines the vertices reachable from s.

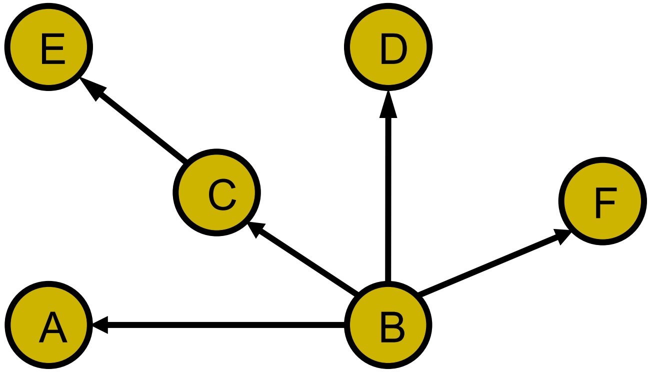


Reachability

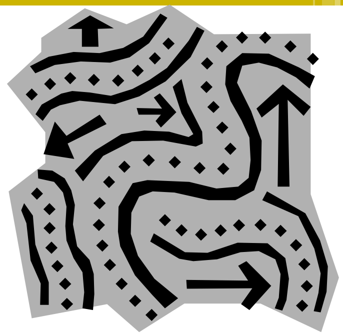
DFS tree rooted at v: vertices reachable from v via directed paths.





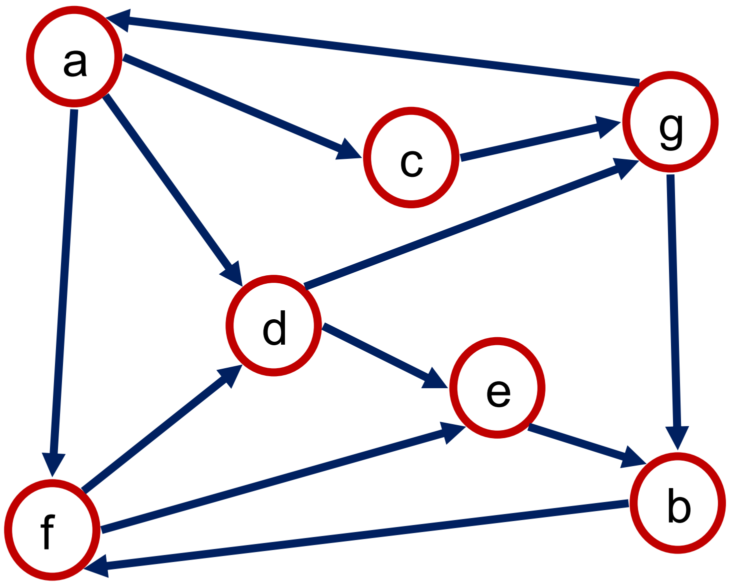


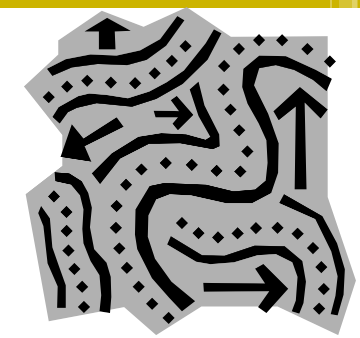
Strong Connectivity



Each vertex can reach all other vertices

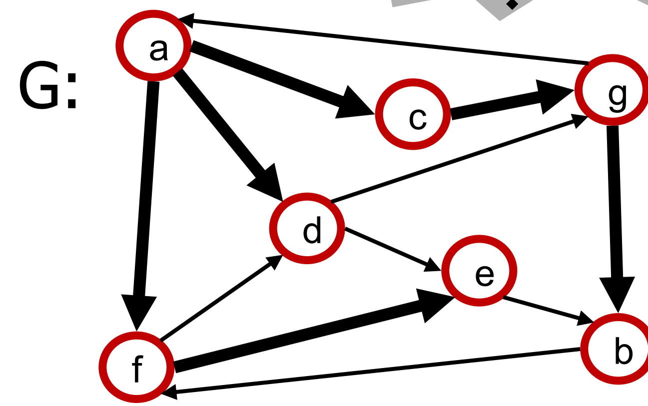
If we start an independent call to DFS from each vertex, we could determine whether this was the case, but those n calls when combined would run in O(n(n+m)). However, we can determine if G is strongly connected much faster than this, requiring only two depth-first searches.

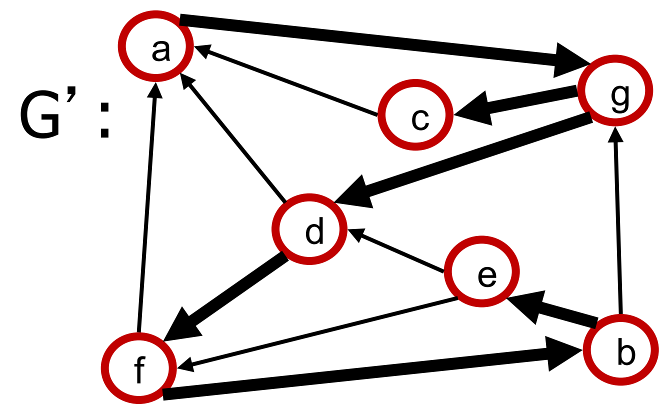




Strong Connectivity Algorithm

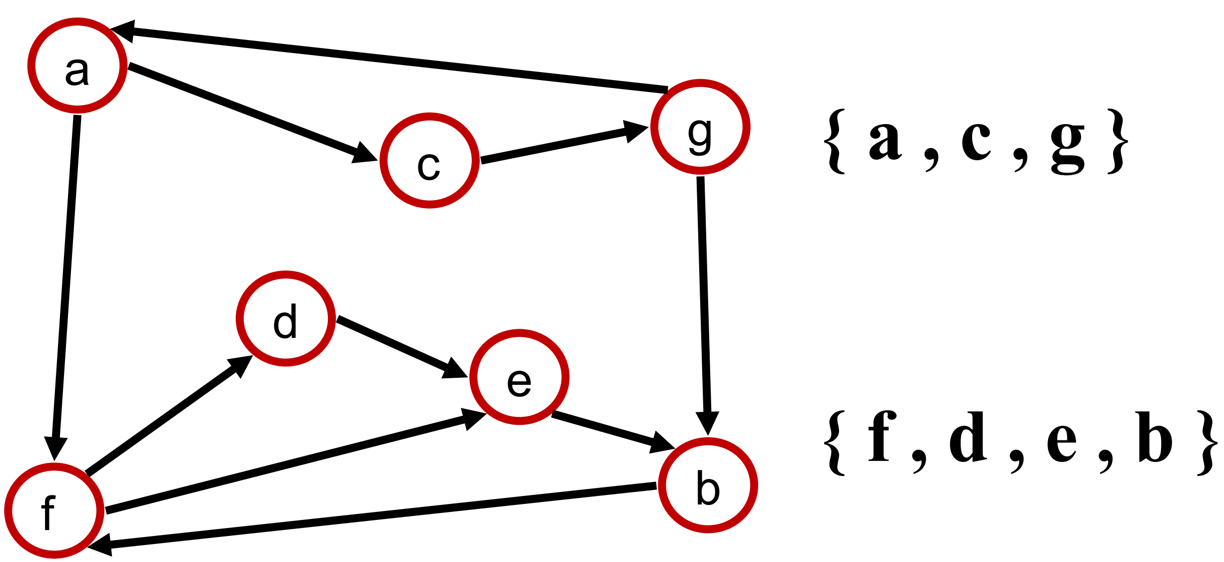
- Pick a vertex v in G
- Perform a DFS from v in G
  - If there's a w not visited, print "no"
- Let G' be G with edges reversed
- Perform a DFS from v in G'
  - If there's a w not visited, print "no"
  - Else, print "yes"
- Running time: O(n+m)





Strongly Connected Components

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



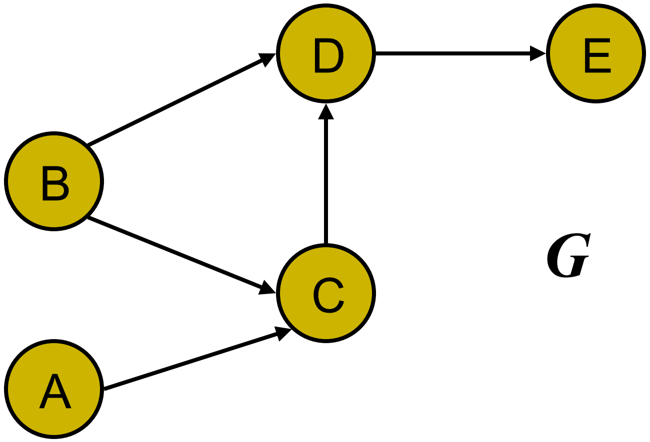
Transitive Closure

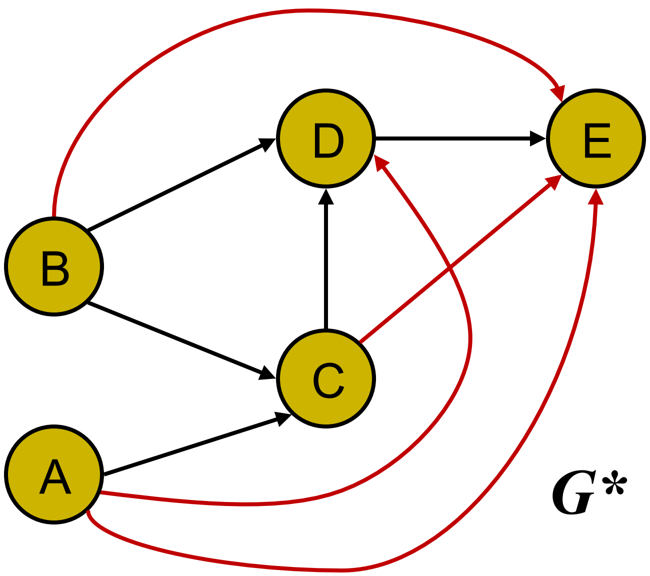
Transitive Closure

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

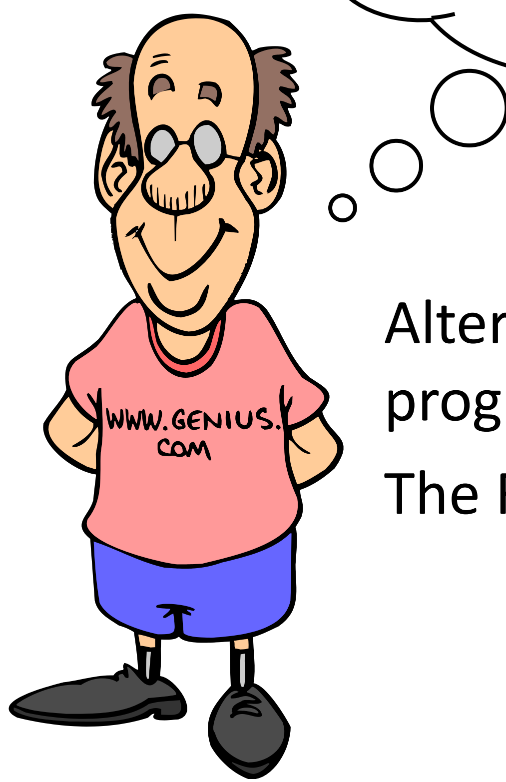




Computing the Transitive Closure

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

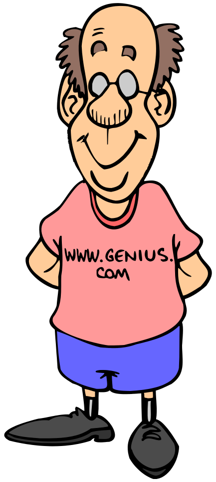
If there's a way to get from A to B and from B to C, then there's a way to get from A to C.



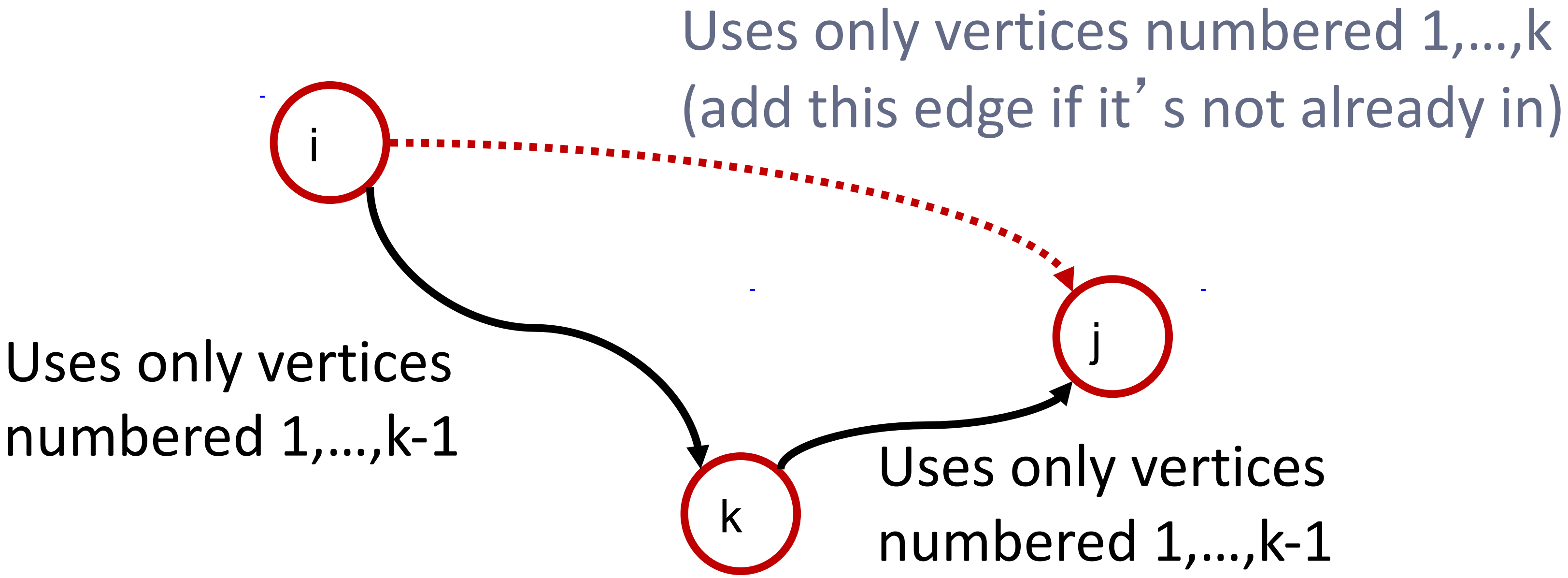
Alternatively ... Use dynamic programming:

The Floyd-Warshall Algorithm

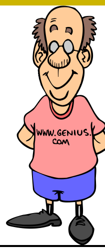
Floyd-Warshall Transitive Closure



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



Floyd-Warshall's Algorithm



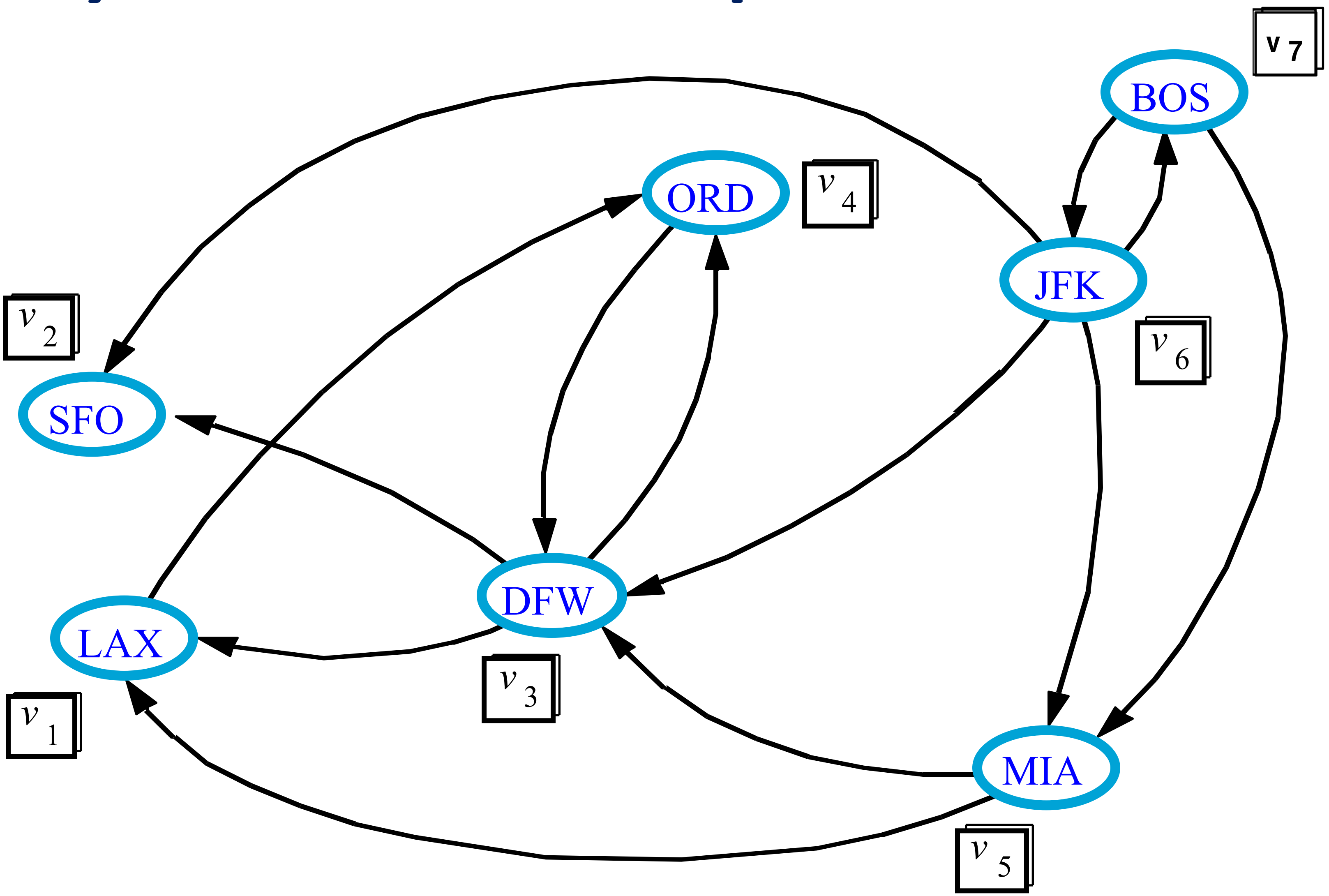
- Number vertices $v_1, \ldots, v_n$
- Compute digraphs $G_0, \ldots, G_n$
  - $G_0 = G$
  - G_k has directed edge (v_i, v_j) if G has a directed path from v_i to v_j with intermediate vertices in $\{v_1, \ldots, v_k\}$
- We have that $G_n = G^*$
- In phase k, digraph G_k is computed from G_{k-1}
- Running 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
```

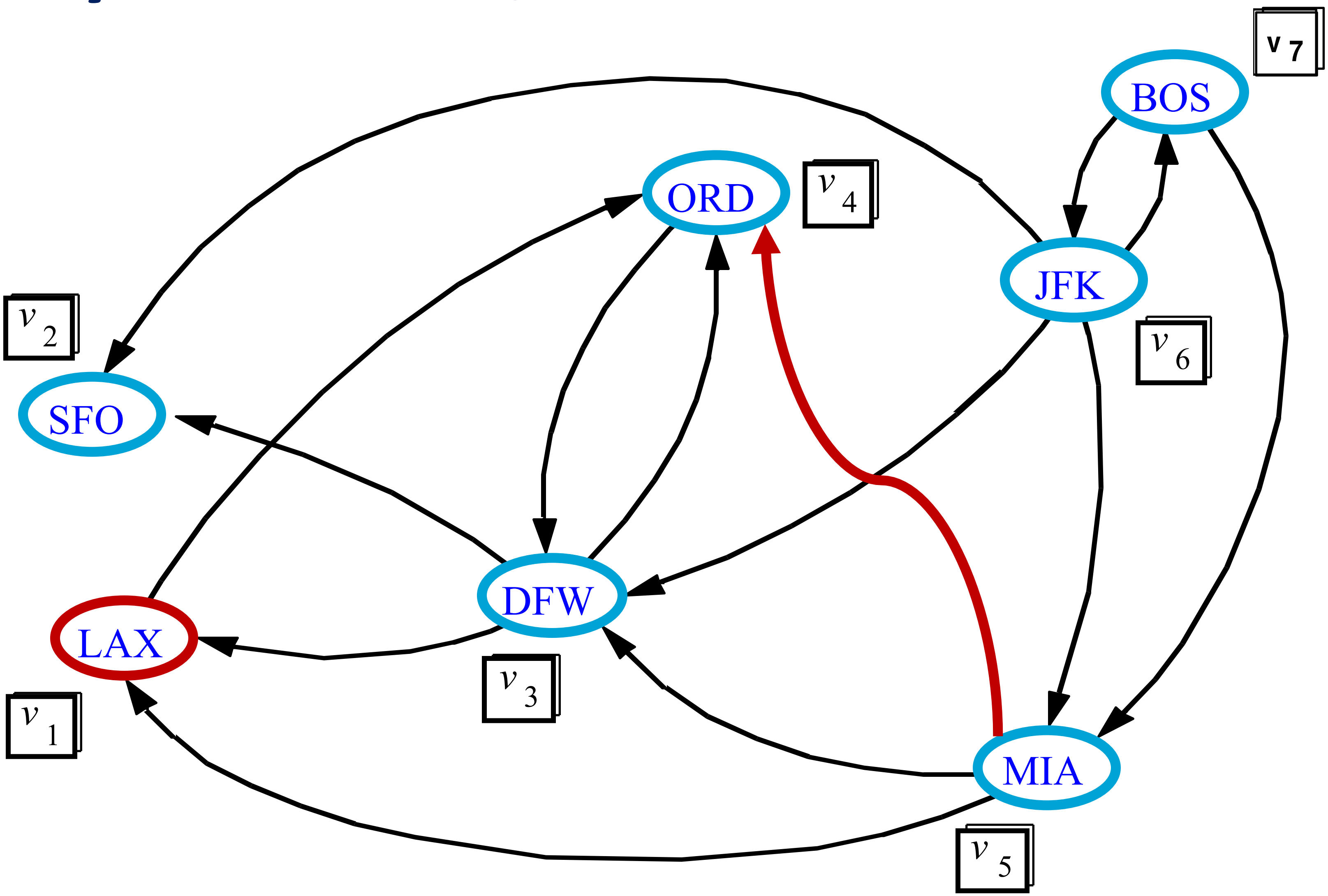
Java Implementation

```java
/** Converts graph g into its transitive closure. */
public static <V,E> void transitiveClosure(Graph<V,E> g) {
  for (Vertex<V> k : g.vertices())
    for (Vertex<V> i : g.vertices())
      // verify that edge (i,k) exists in the partial closure
      if (i != k && g.getEdge(i,k) != null)
        for (Vertex<V> j : g.vertices())
          // verify that edge (k,j) exists in the partial closure
          if (i != j && j != k && g.getEdge(k,j) != null)
            // if (i,j) not yet included, add it to the closure
            if (g.getEdge(i,j) == null)
              g.insertEdge(i, j, null);
}
```

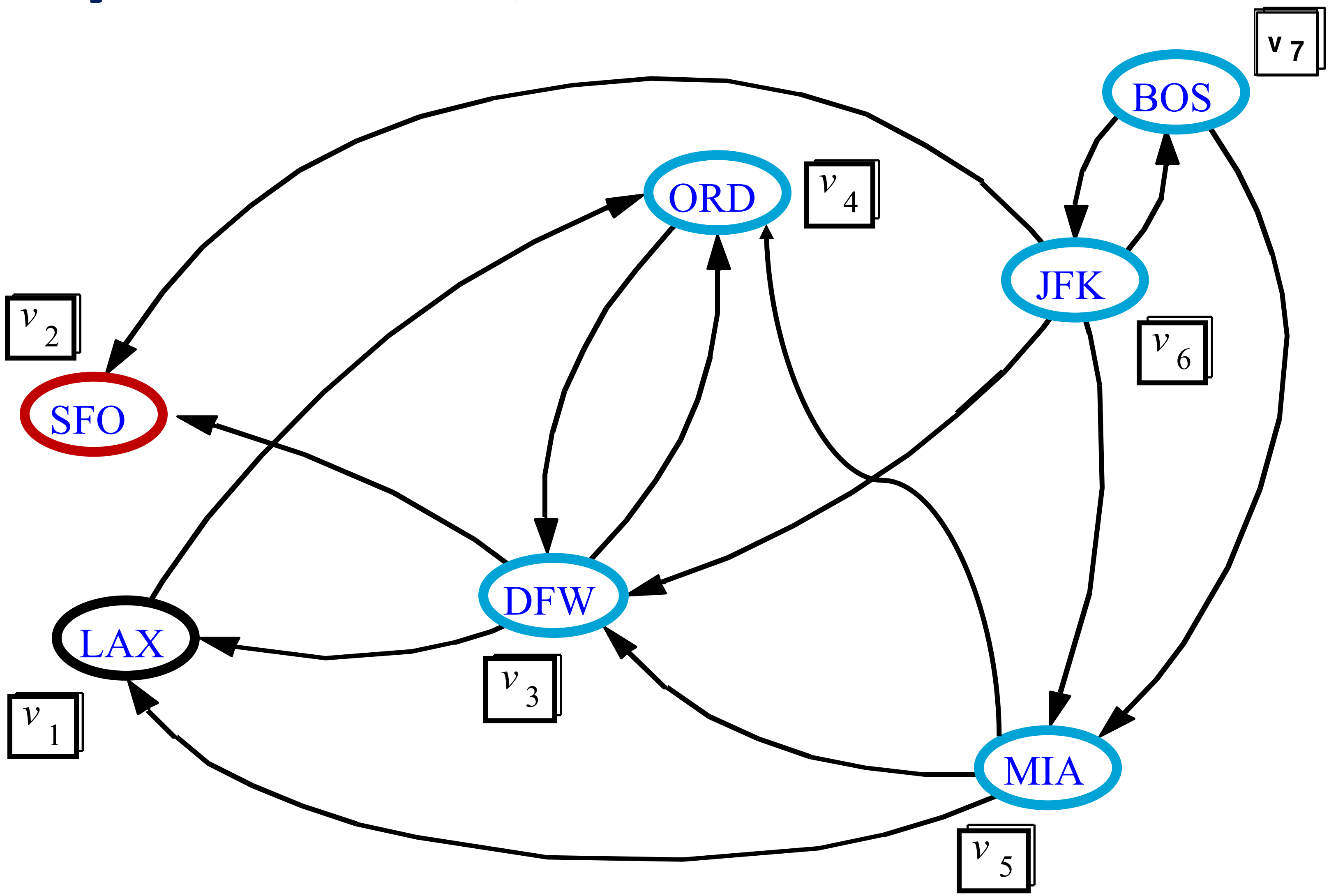
Floyd-Warshall Example



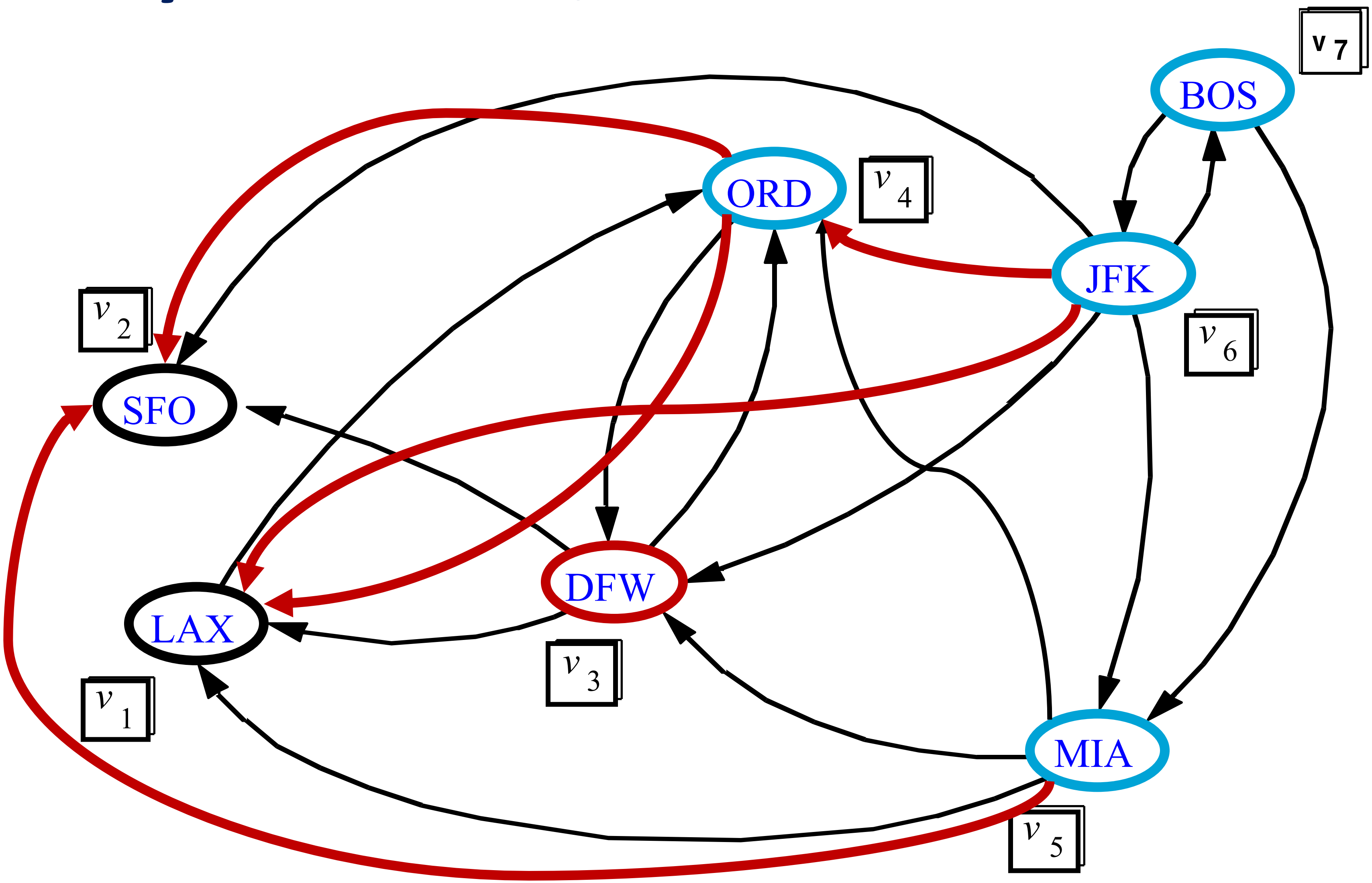
Floyd-Warshall, Iteration 1



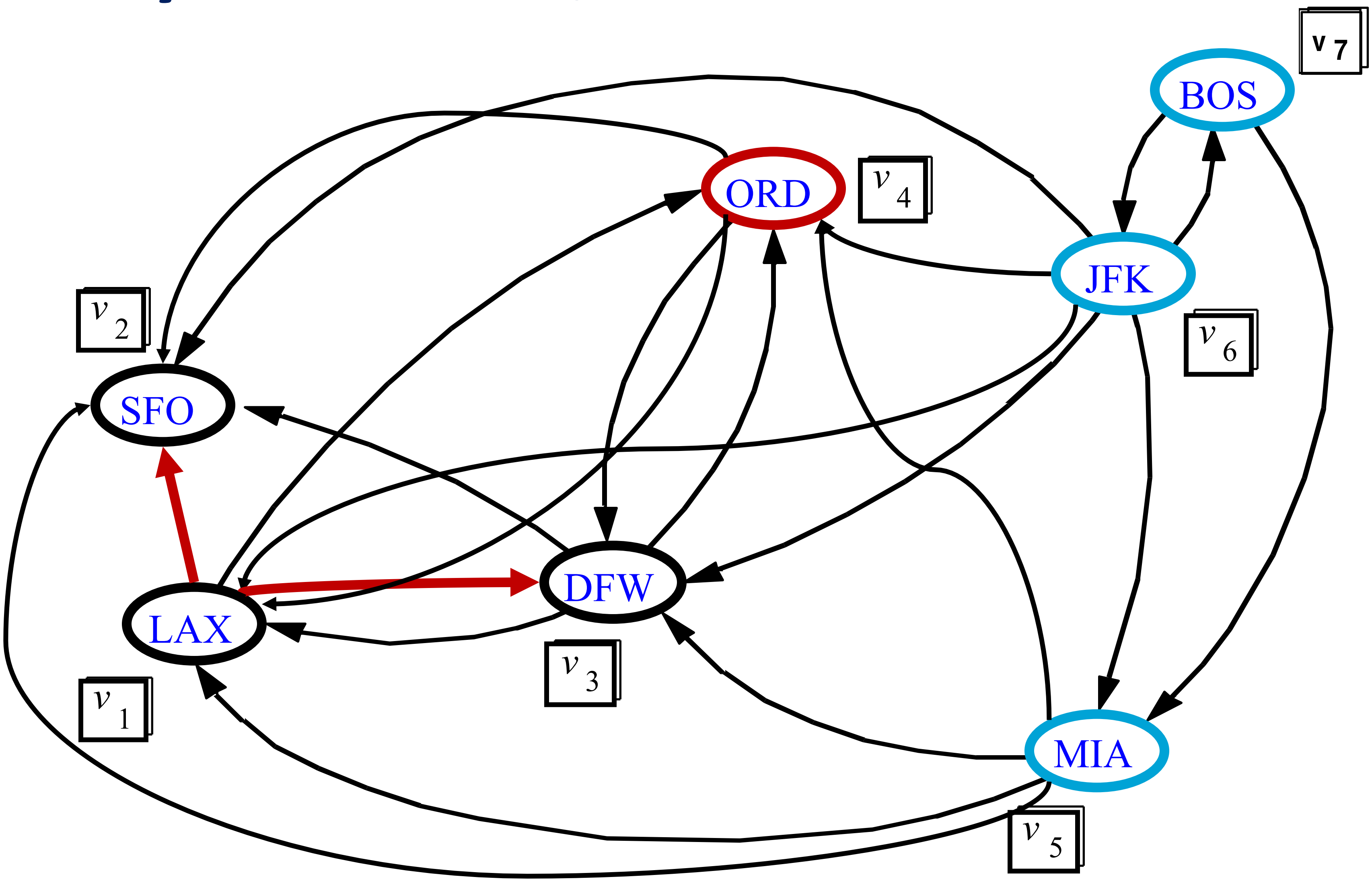
Floyd-Warshall, Iteration 2



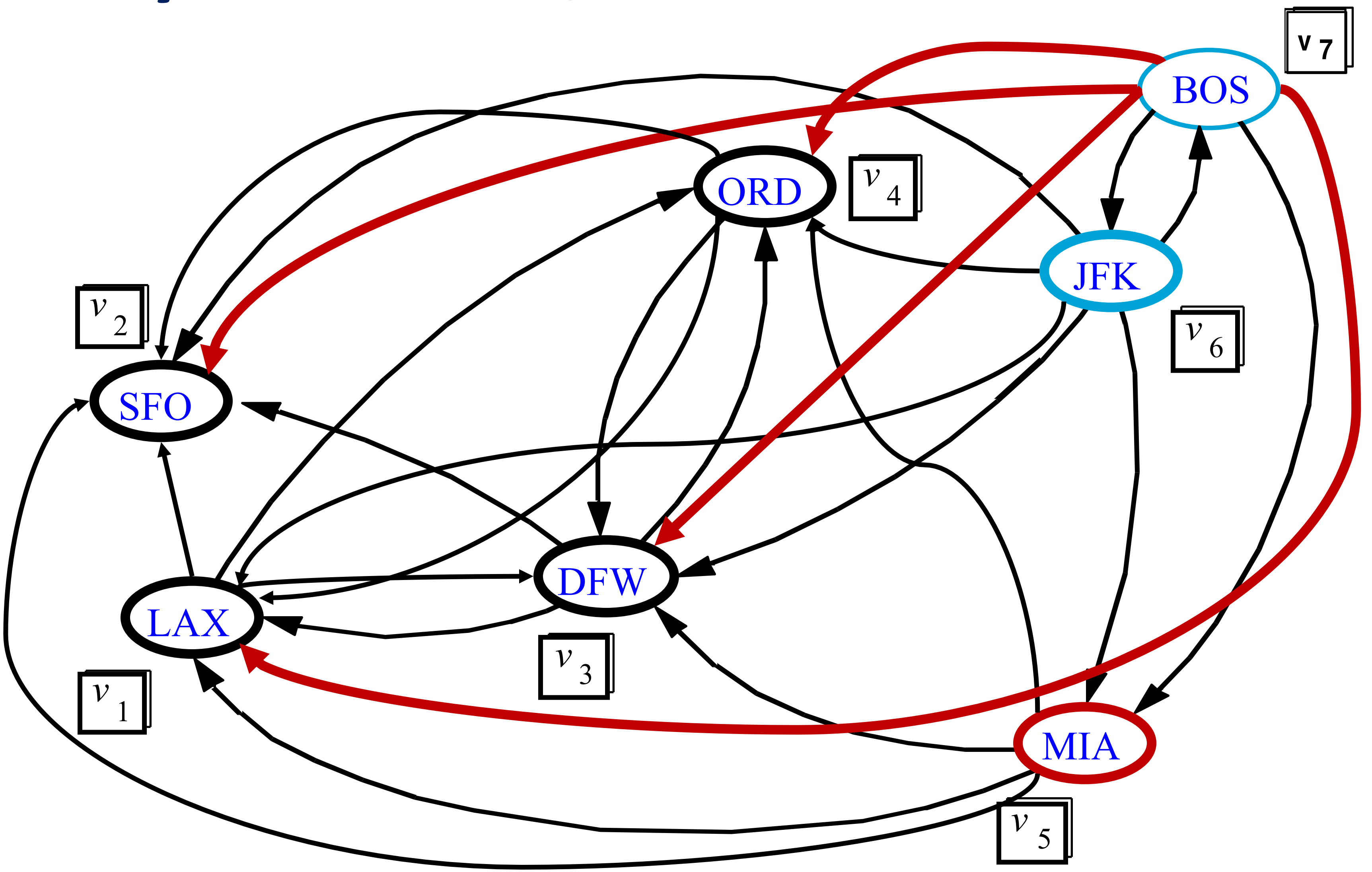
Floyd-Warshall, Iteration 3



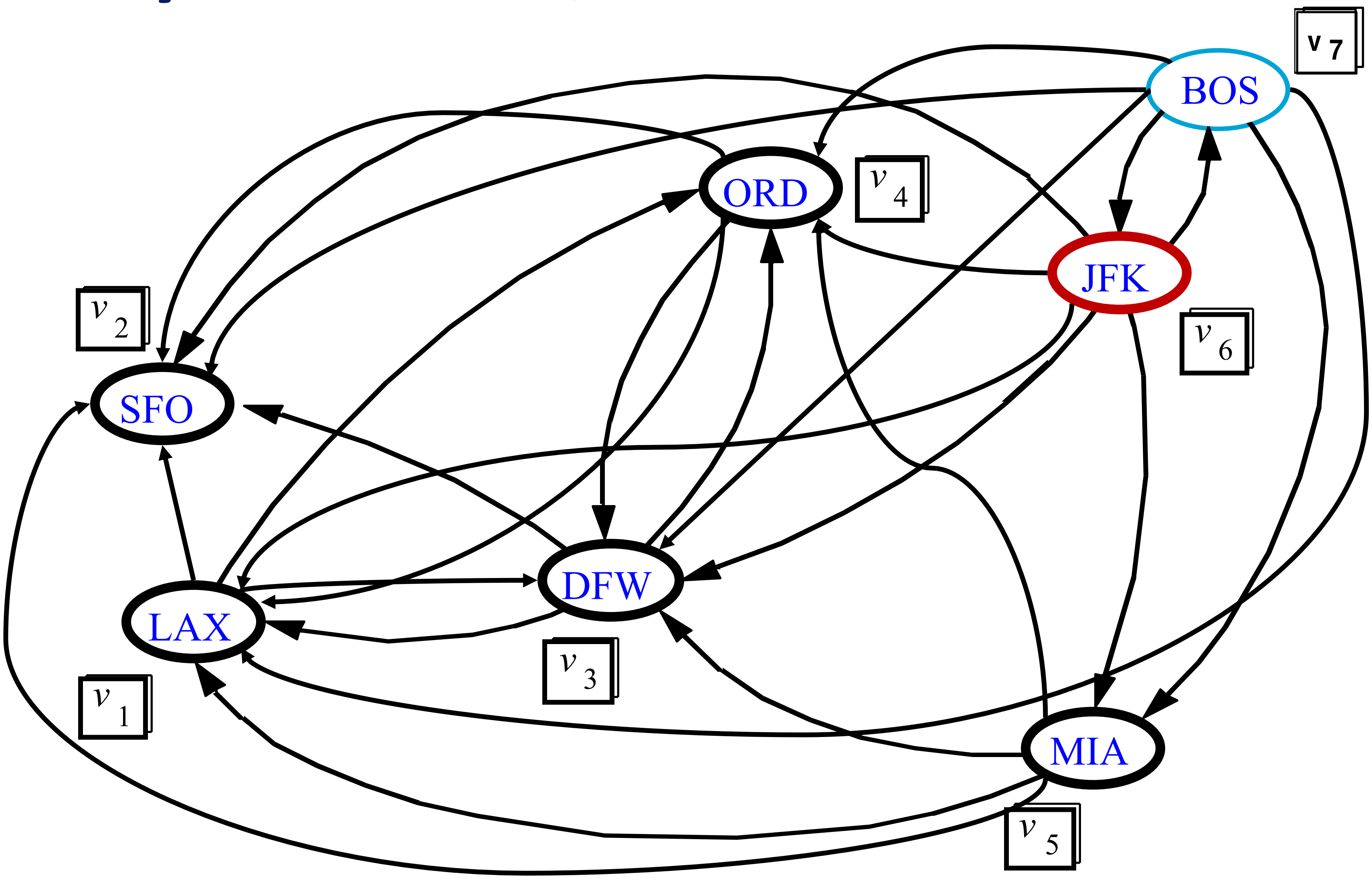
Floyd-Warshall, Iteration 4



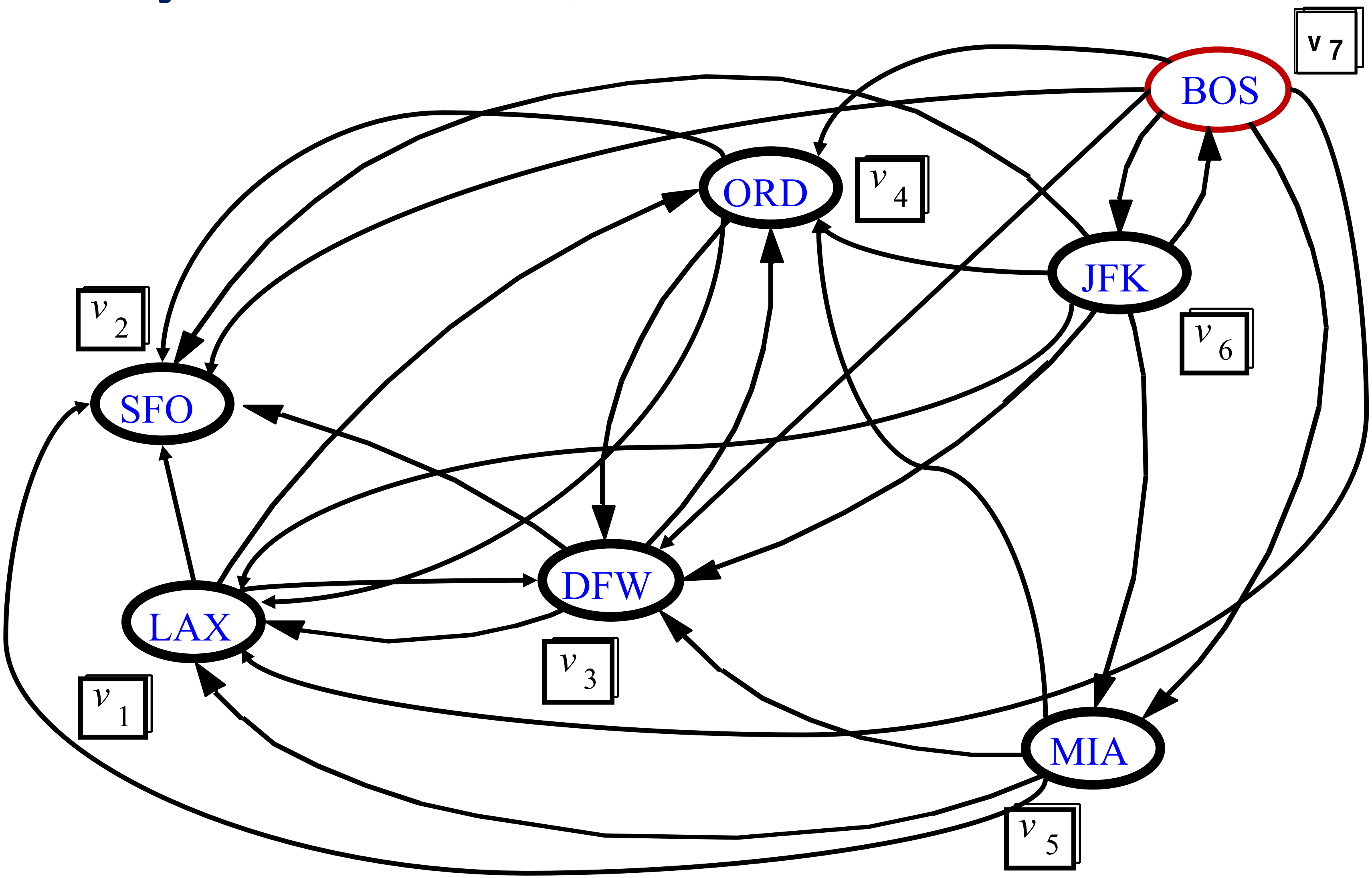
Floyd-Warshall, Iteration 5



Floyd-Warshall, Iteration 6



Floyd-Warshall, Conclusion



Directed Acyclic Graphs (DAG)

DAGs and Topological Ordering

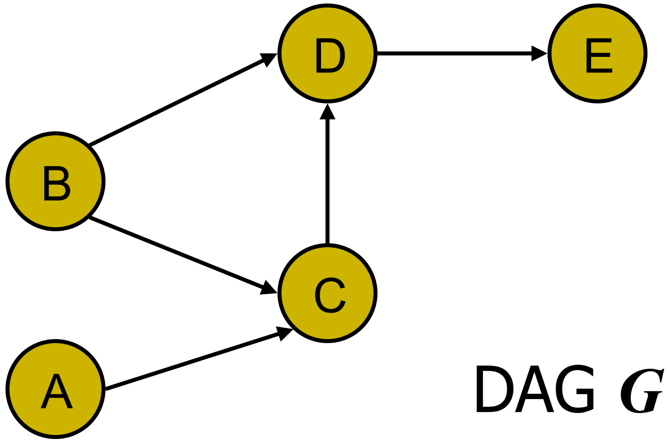
- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

  $$v_1, \ldots, v_n$$

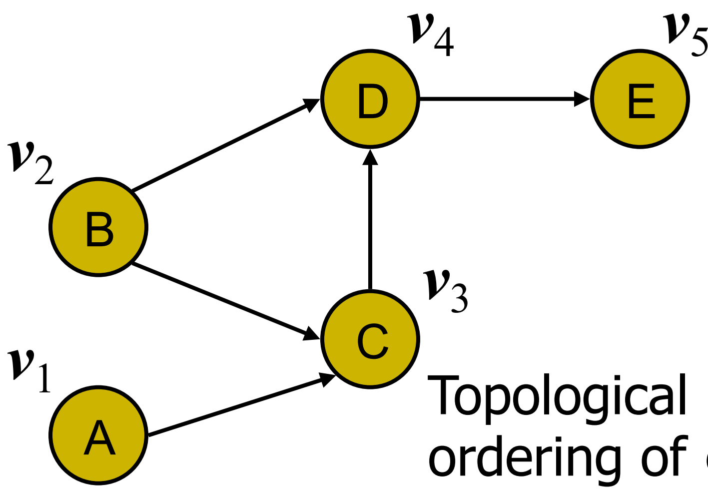
  of the vertices such that for every edge (v_i, v_j), we have $i < j$
- For example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints

Theorem

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

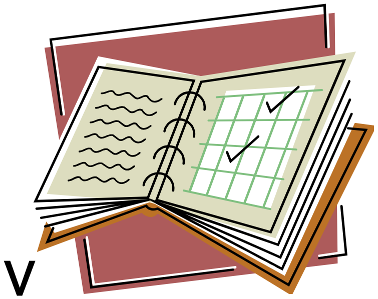


DAG G

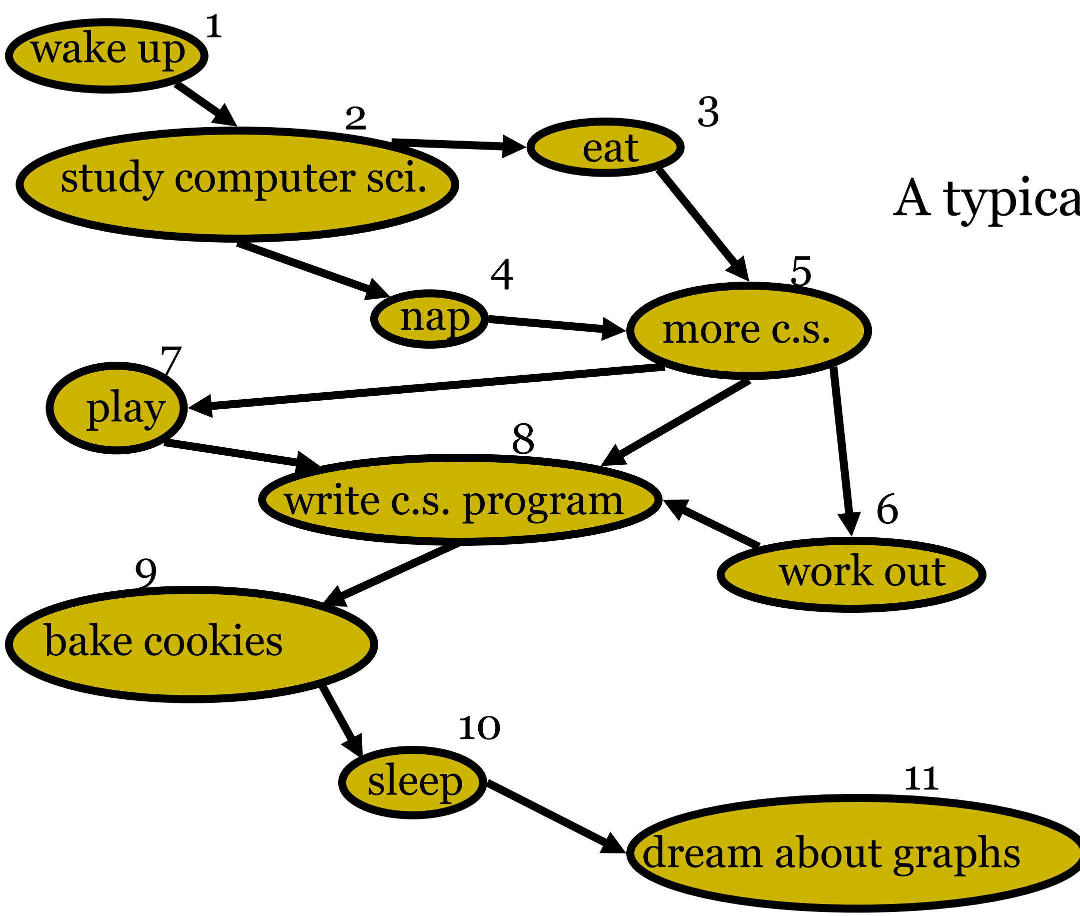


Topological ordering of G

Topological Sorting



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



A typical student day

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)

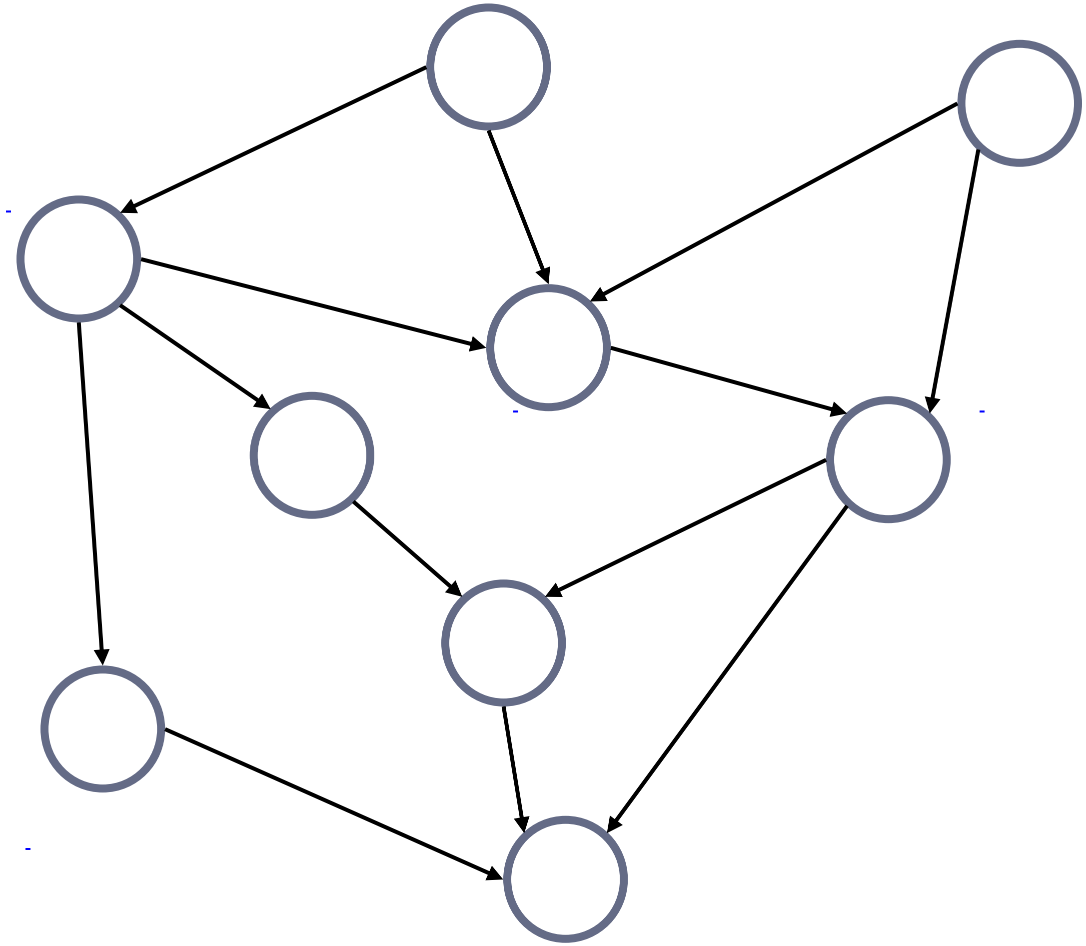
Implementation with DFS

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

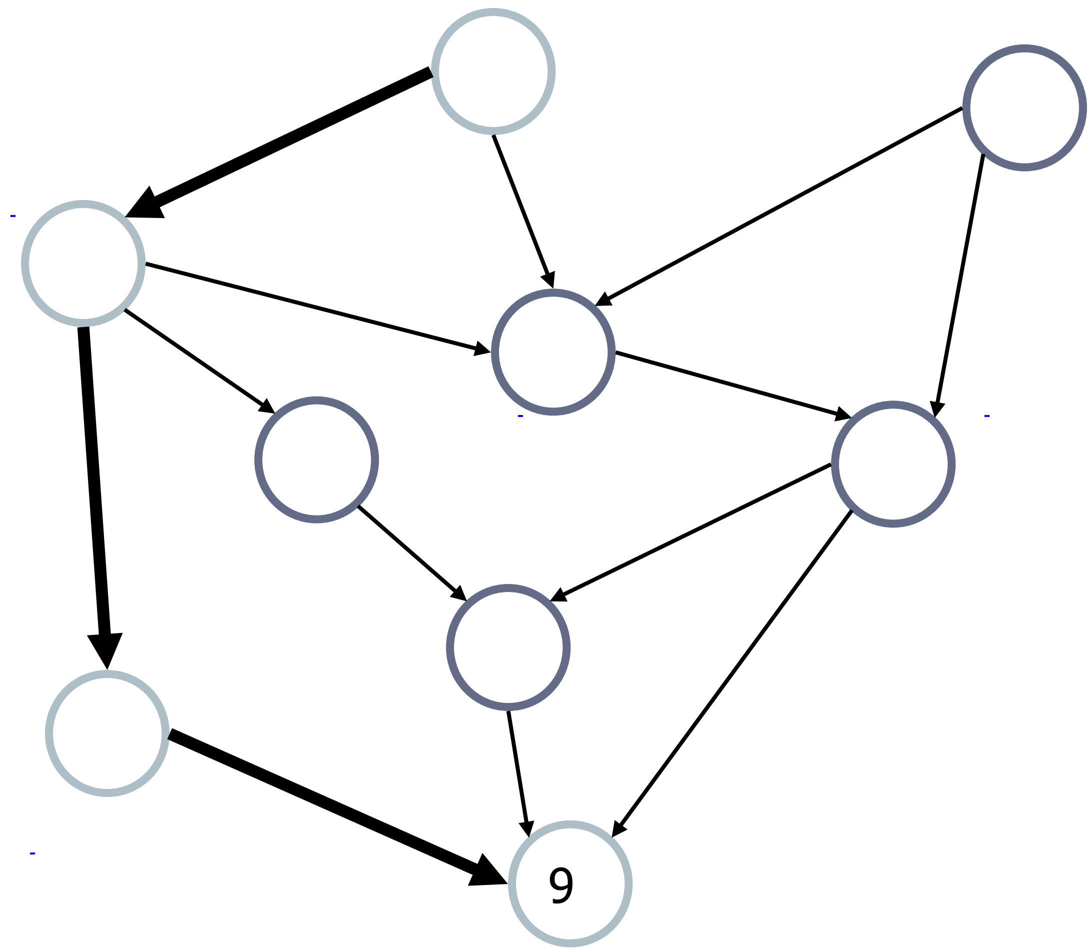
```
Algorithm topologicalDFS(G)
   Input dag G
   Output topological ordering of G
  n ← G.numVertices()
  for all  u ∈ G.vertices()
     setLabel(u, UNEXPLORED)
  for all  v ∈ G.vertices()
     if  getLabel(v) = UNEXPLORED
        topologicalDFS(G, v)
```

```
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
```

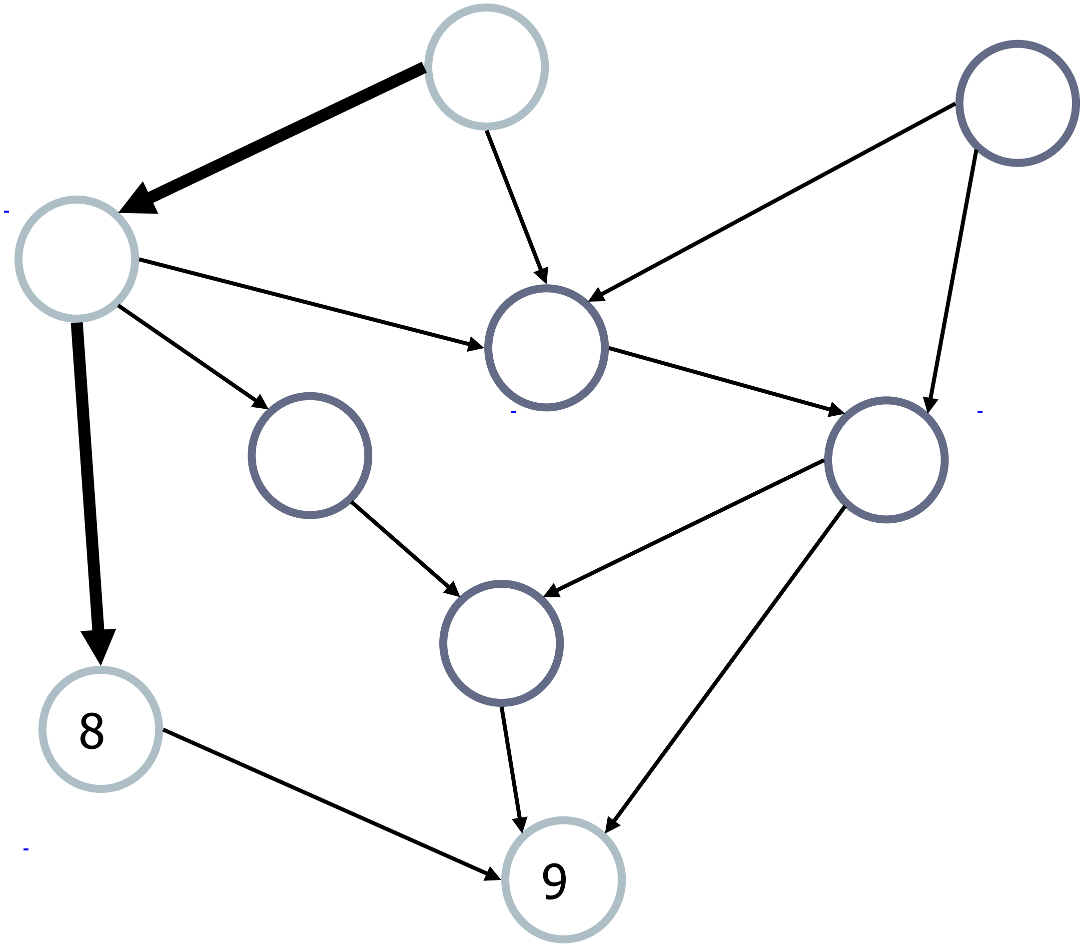
Topological Sorting Example



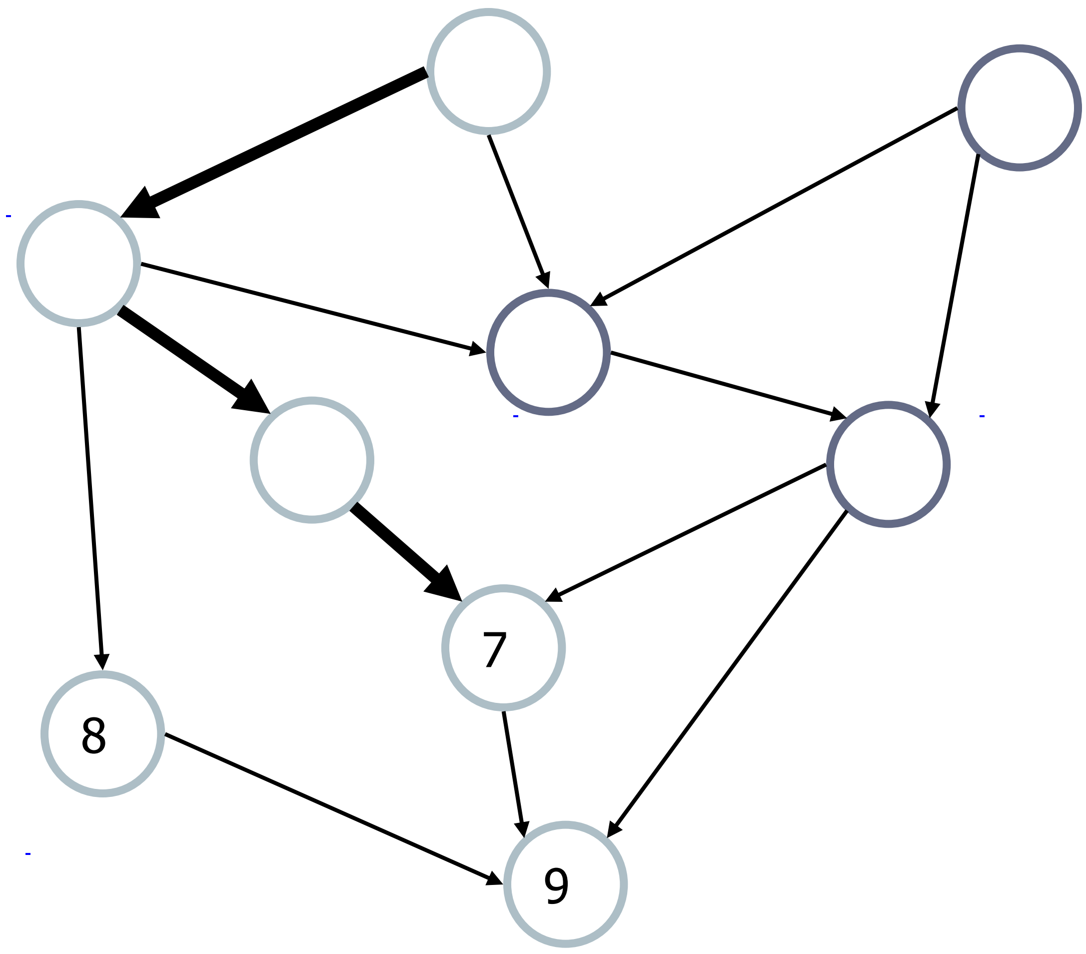
Topological Sorting Example



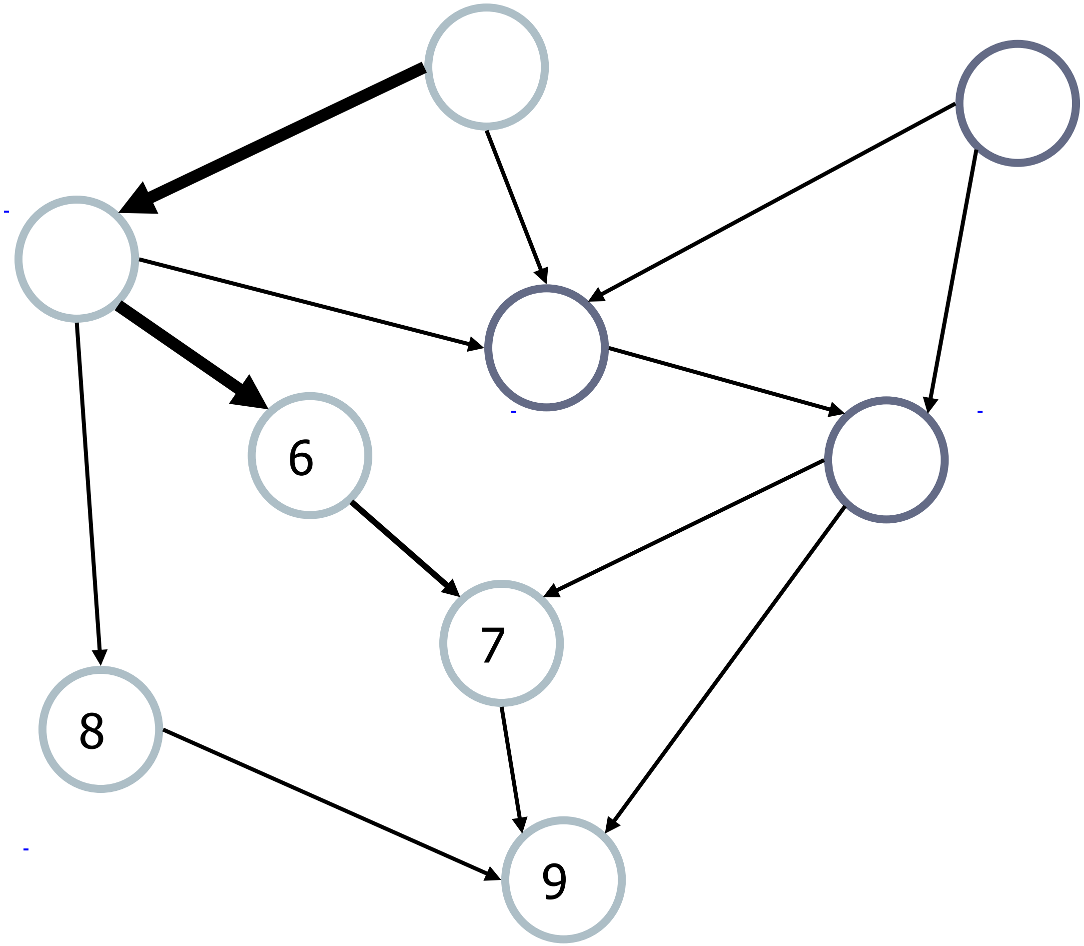
Topological Sorting Example



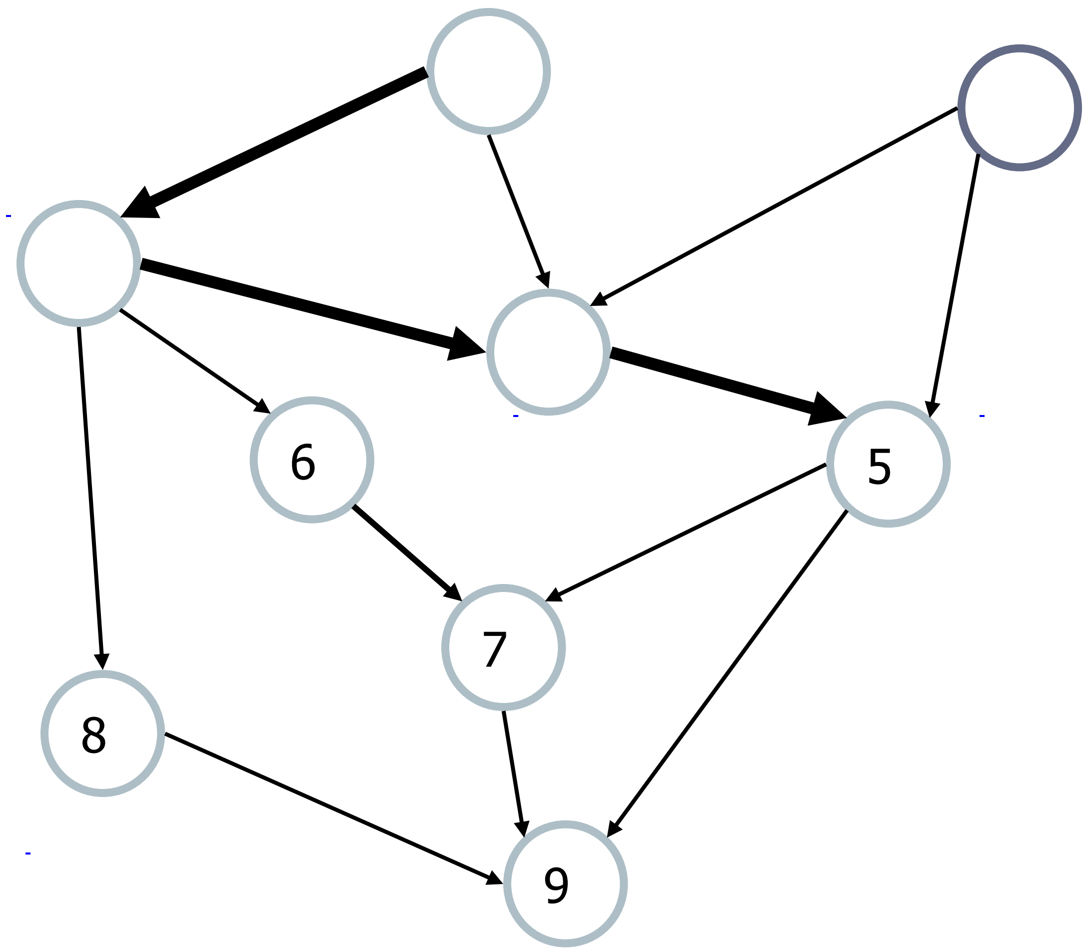
Topological Sorting Example



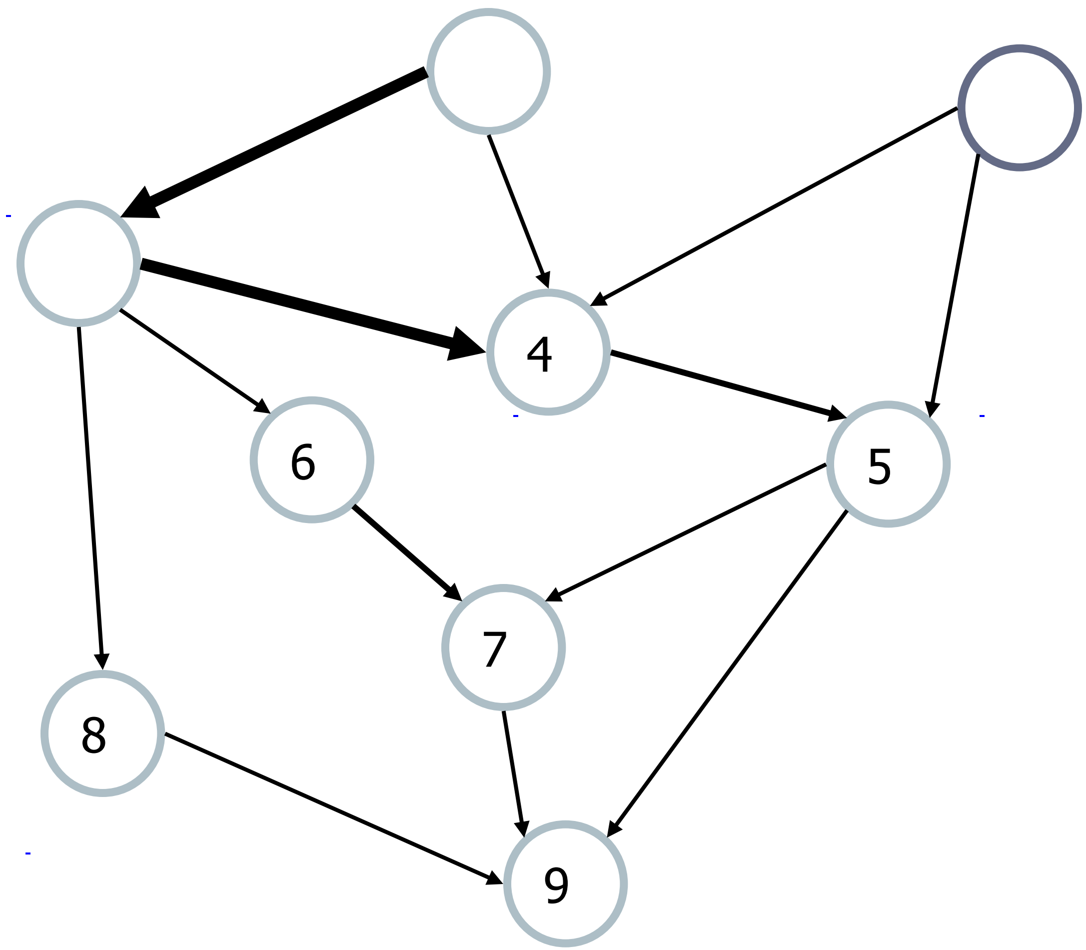
Topological Sorting Example



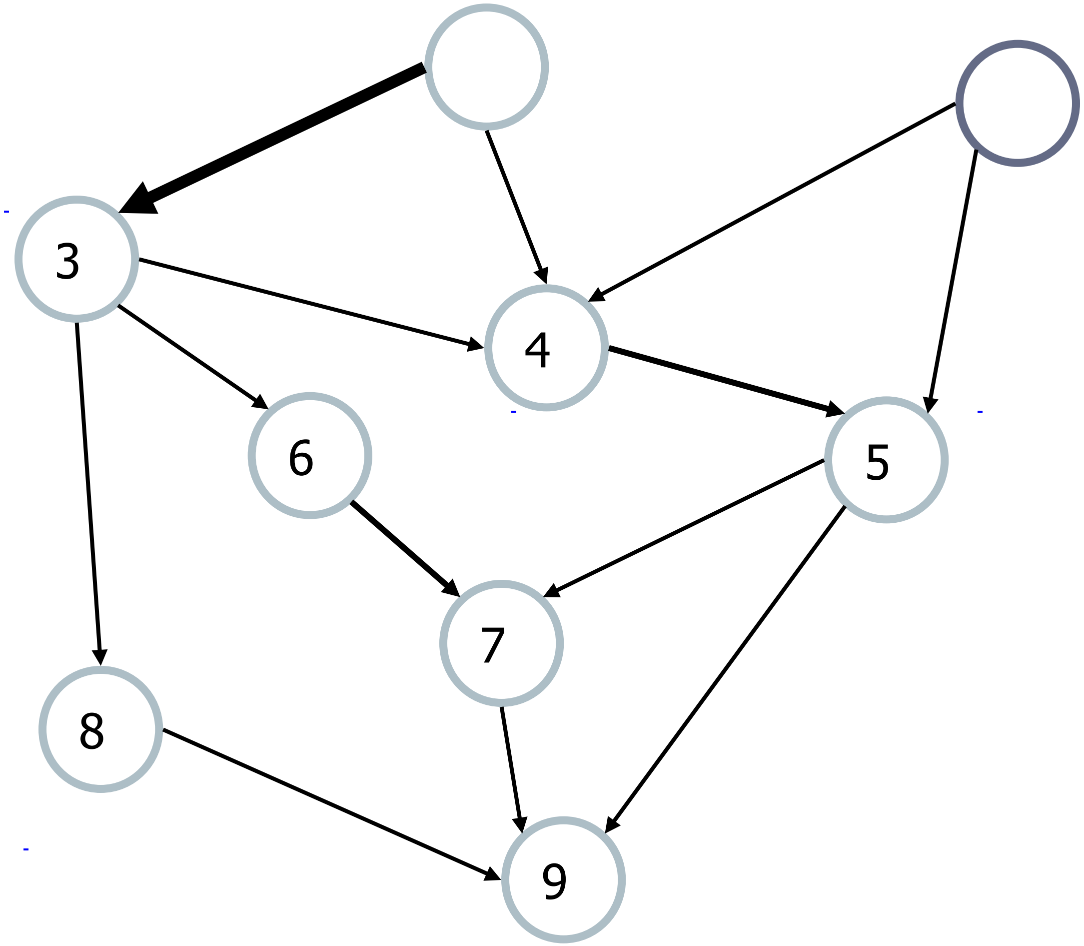
Topological Sorting Example



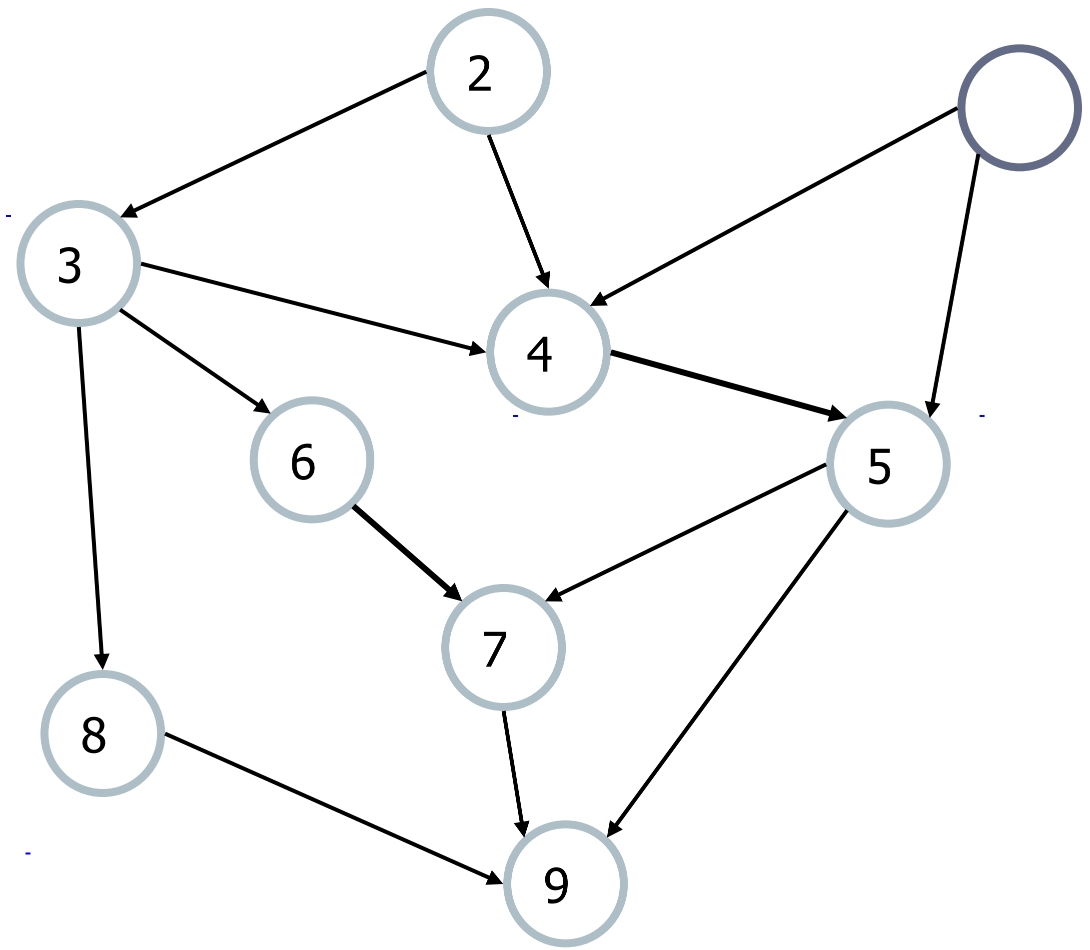
Topological Sorting Example



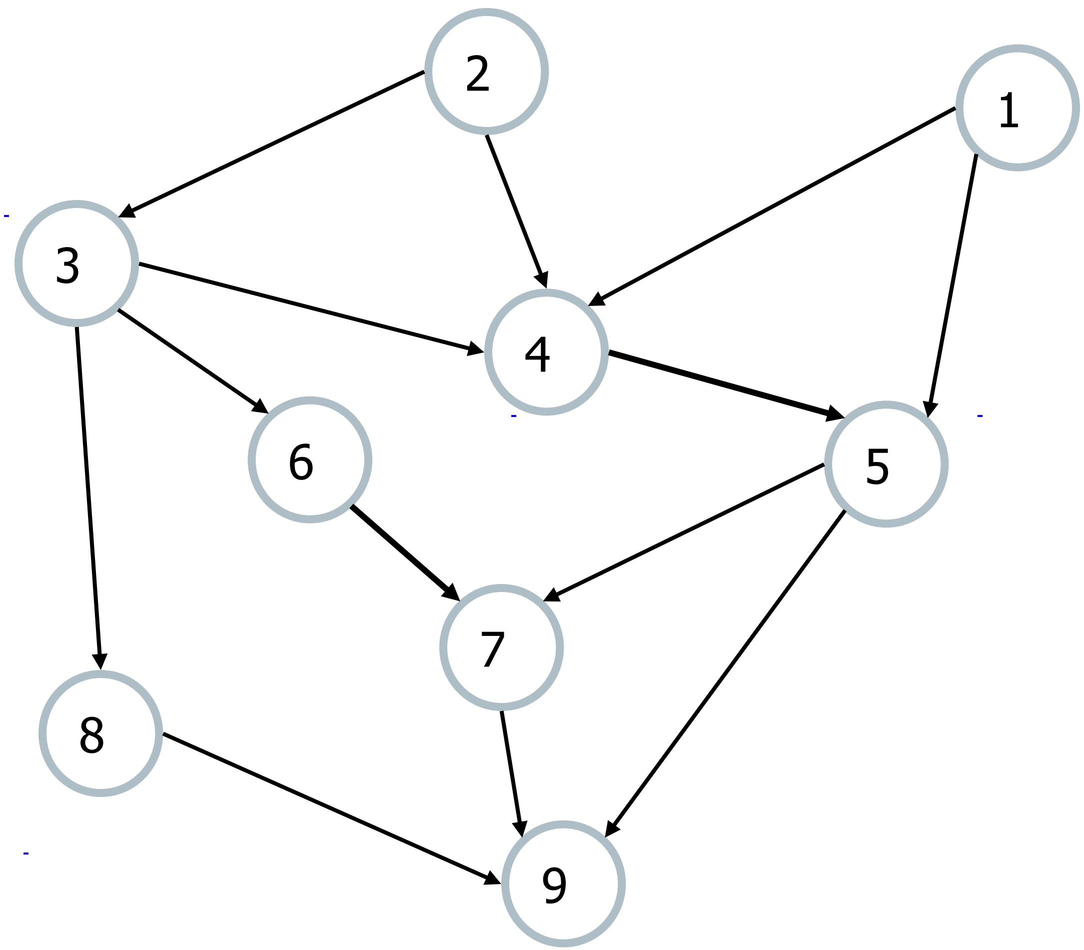
Topological Sorting Example



Topological Sorting Example



Topological Sorting Example



Java Implementation

```java
/** Returns a list of verticies of directed acyclic graph g in topological order. */
public static <V,E> PositionalList<Vertex<V>> topologicalSort(Graph<V,E> g) {
  // list of vertices placed in topological order
  PositionalList<Vertex<V>> topo = new LinkedPositionalList<>();
  // container of vertices that have no remaining constraints
  Stack<Vertex<V>> ready = new LinkedStack<>();
  // map keeping track of remaining in-degree for each vertex
  Map<Vertex<V>, Integer> inCount = new ProbeHashMap<>();
  for (Vertex<V> u : g.vertices()) {
    inCount.put(u, g.inDegree(u));                    // initialize with actual in-degree
    if (inCount.get(u) == 0)                          // if u has no incoming edges,
      ready.push(u);                                  // it is free of constraints
  }
  while (!ready.isEmpty()) {
    Vertex<V> u = ready.pop();
    topo.addLast(u);
    for (Edge<E> e : g.outgoingEdges(u)) {            // consider all outgoing neighbors of u
      Vertex<V> v = g.opposite(u, e);
      inCount.put(v, inCount.get(v) - 1);             // v has one less constraint without u
      if (inCount.get(v) == 0)
        ready.push(v);
    }
  }
  return topo;
}
```

Questions