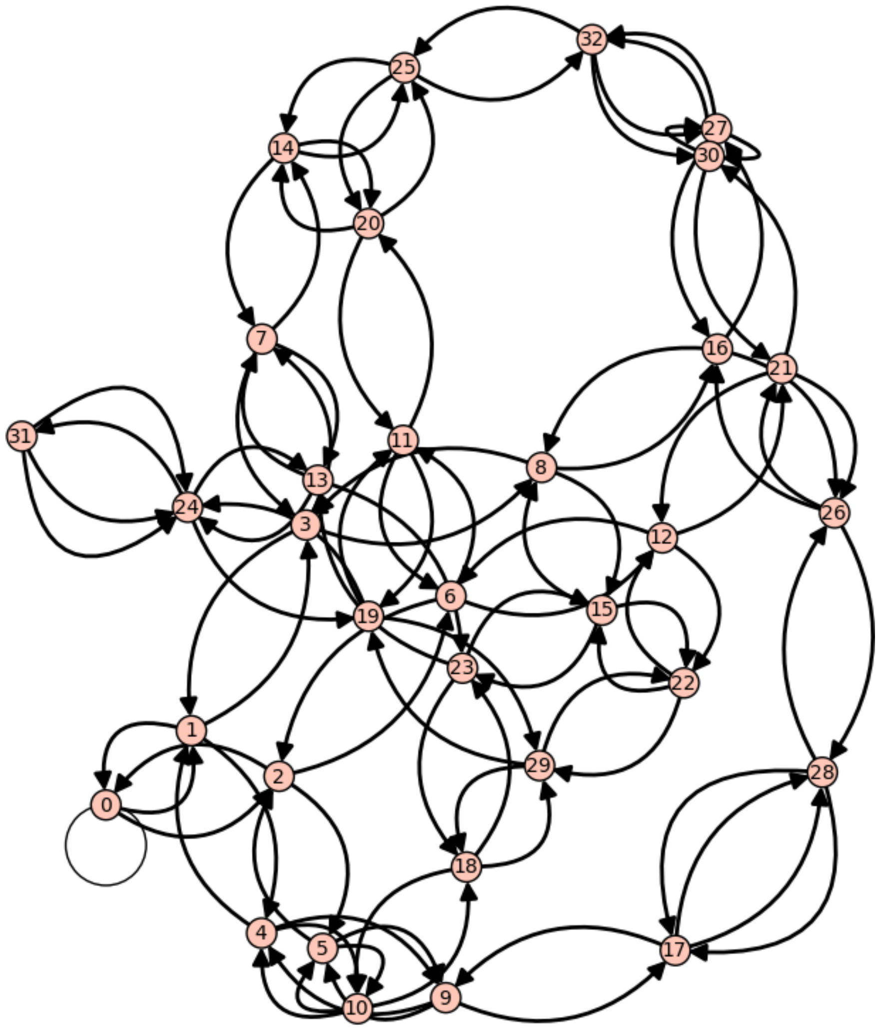
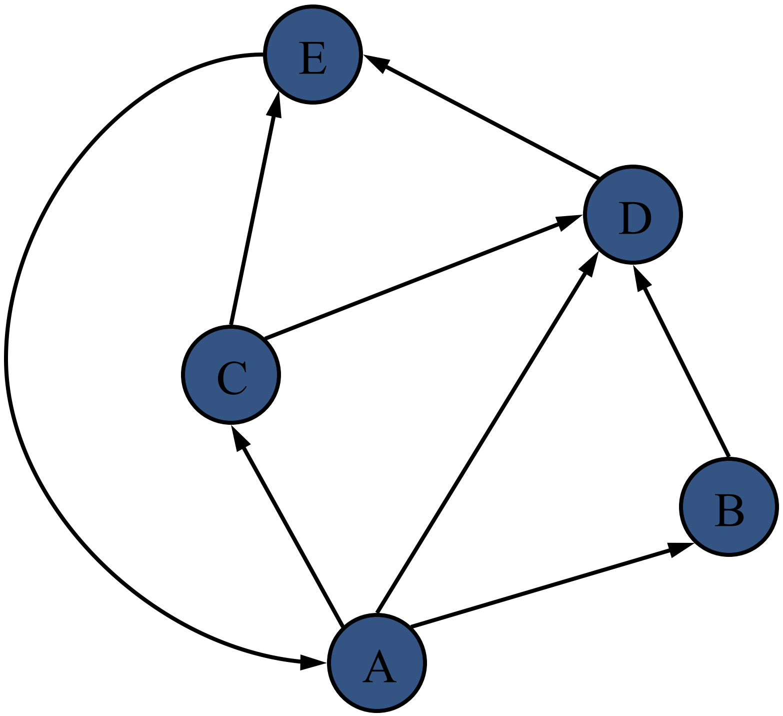


Graphs II

Digraphs, Strongly Connective Component, Topological Sorting, and Minimum Spanning Tree

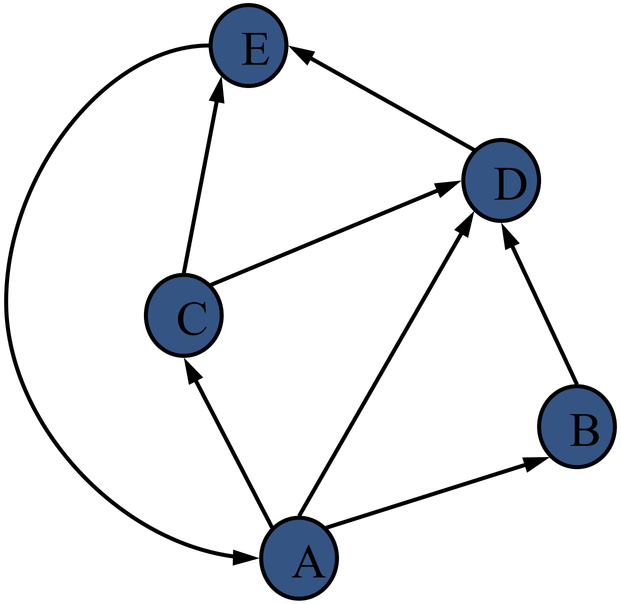
Digraphs

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



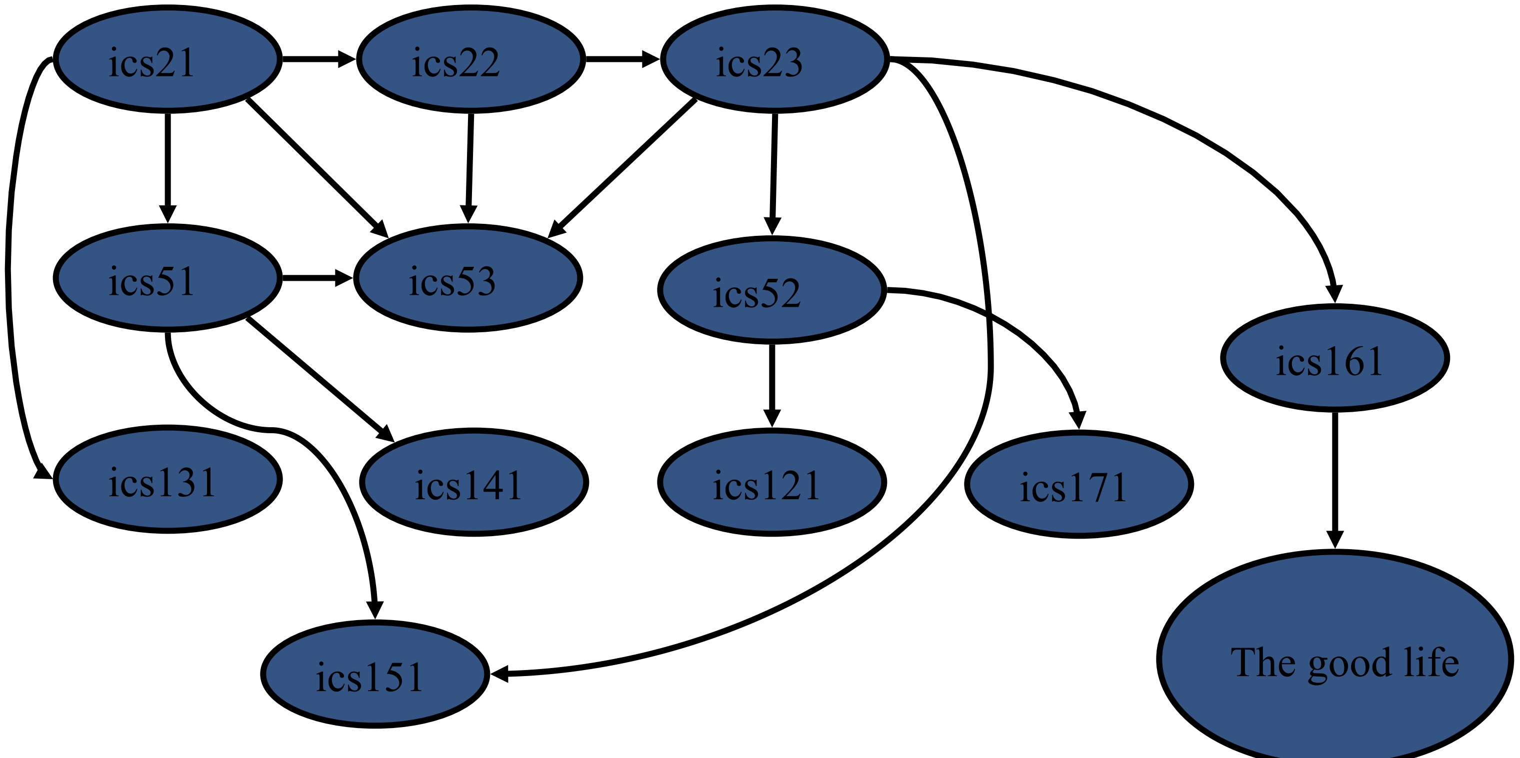
Digraph Properties

- 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, $\boldsymbol{m} \leq \boldsymbol{n} \cdot (\boldsymbol{n} - 1)$
- 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



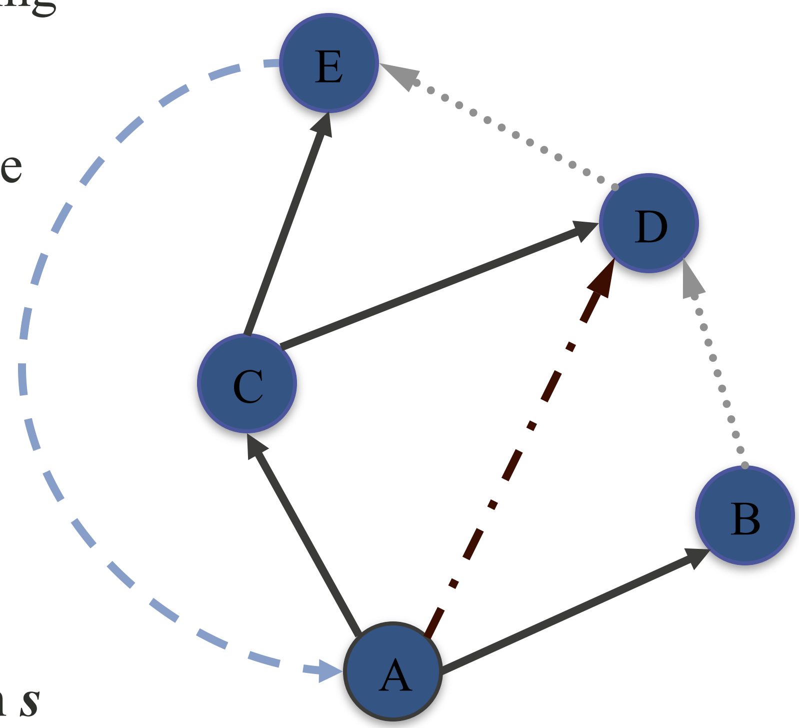
Digraph Application

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



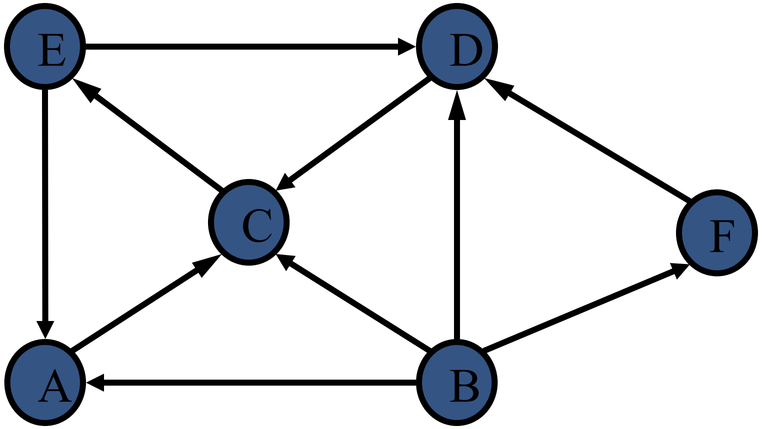
Directed DFS

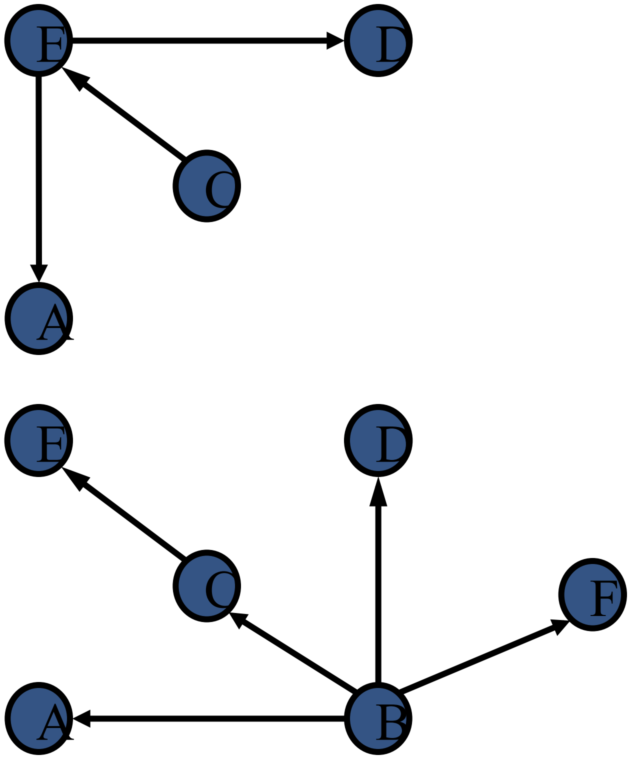
- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction
- In the directed DFS algorithm, we have four types of edges
  - discovery edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex ***s*** determines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



Reachability

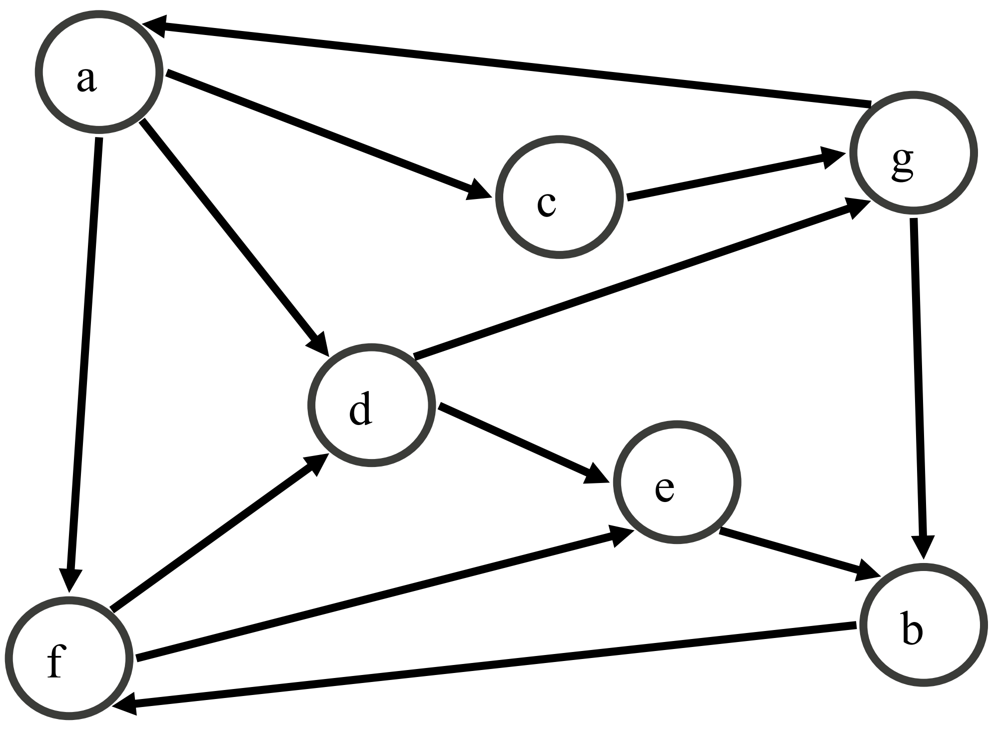
- DFS tree rooted at v: vertices reachable from v via directed paths





Strong Connectivity

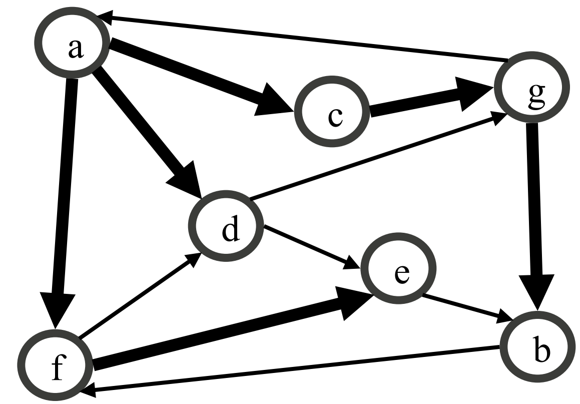
- Each vertex can reach all other vertices



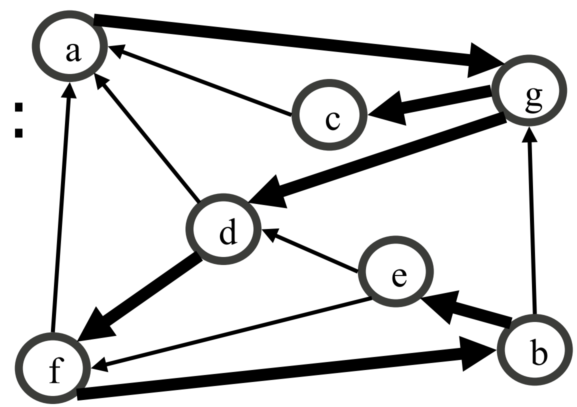
Strong Connectivity Algorithm

- 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)

G:

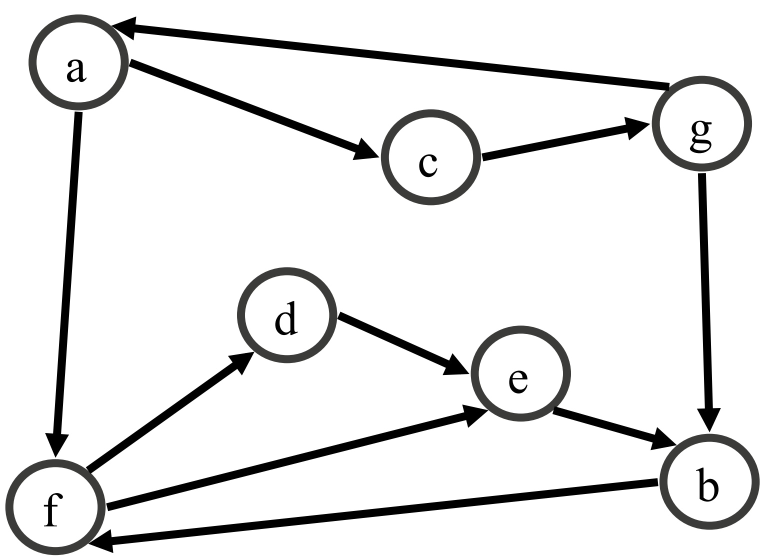


G' :



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).

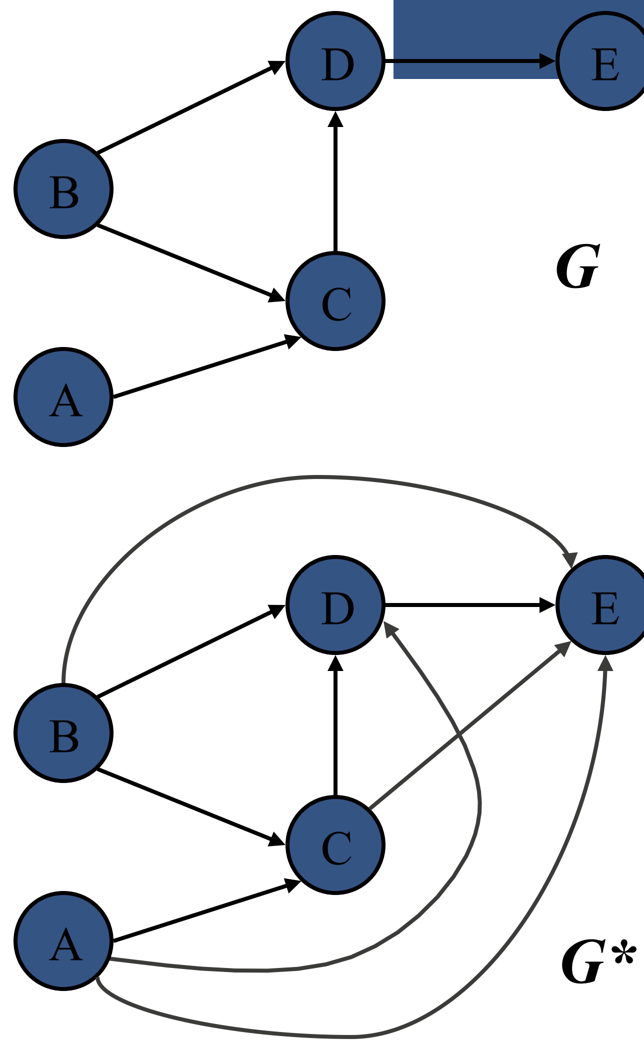


{ a , c , g }

{ f , d , e , b }

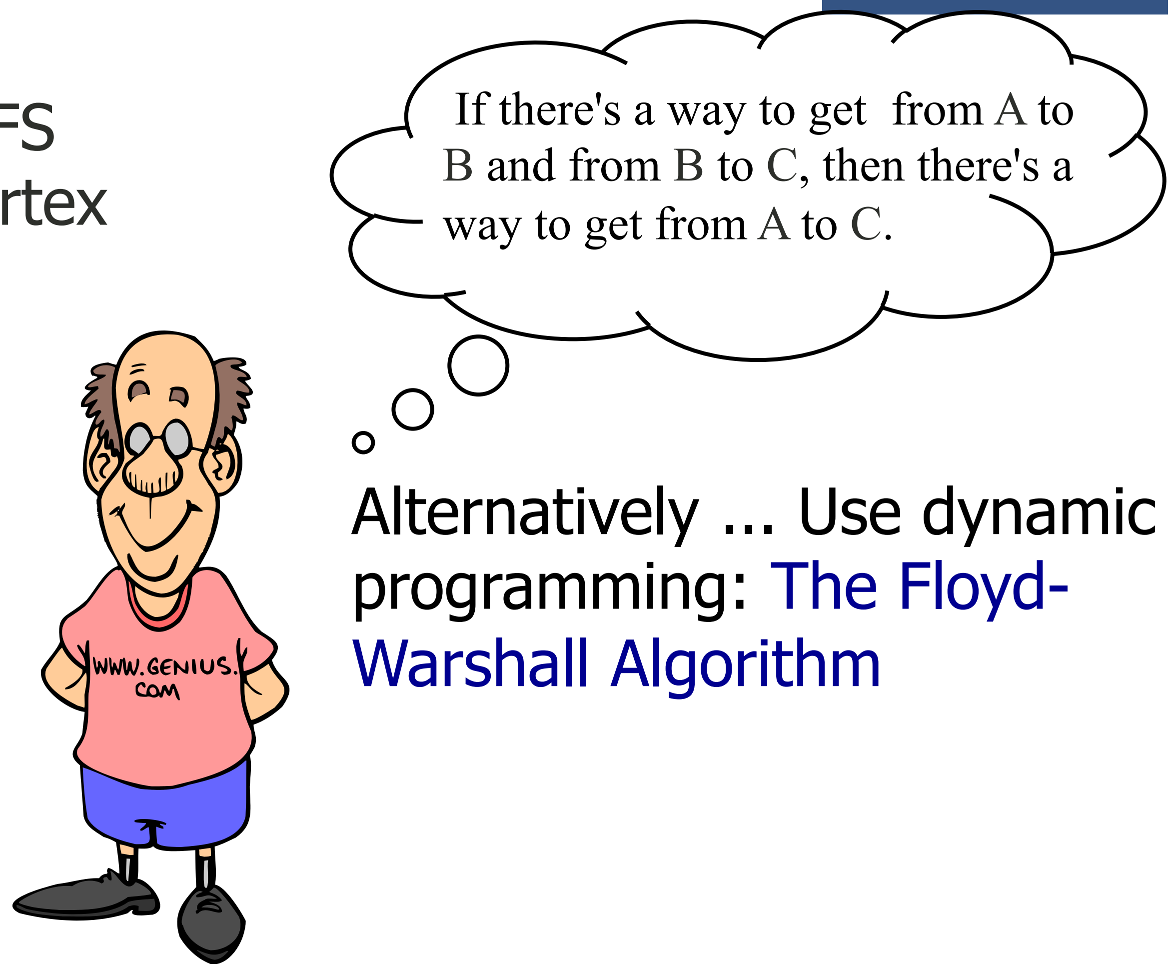
Transitive Closure

- 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

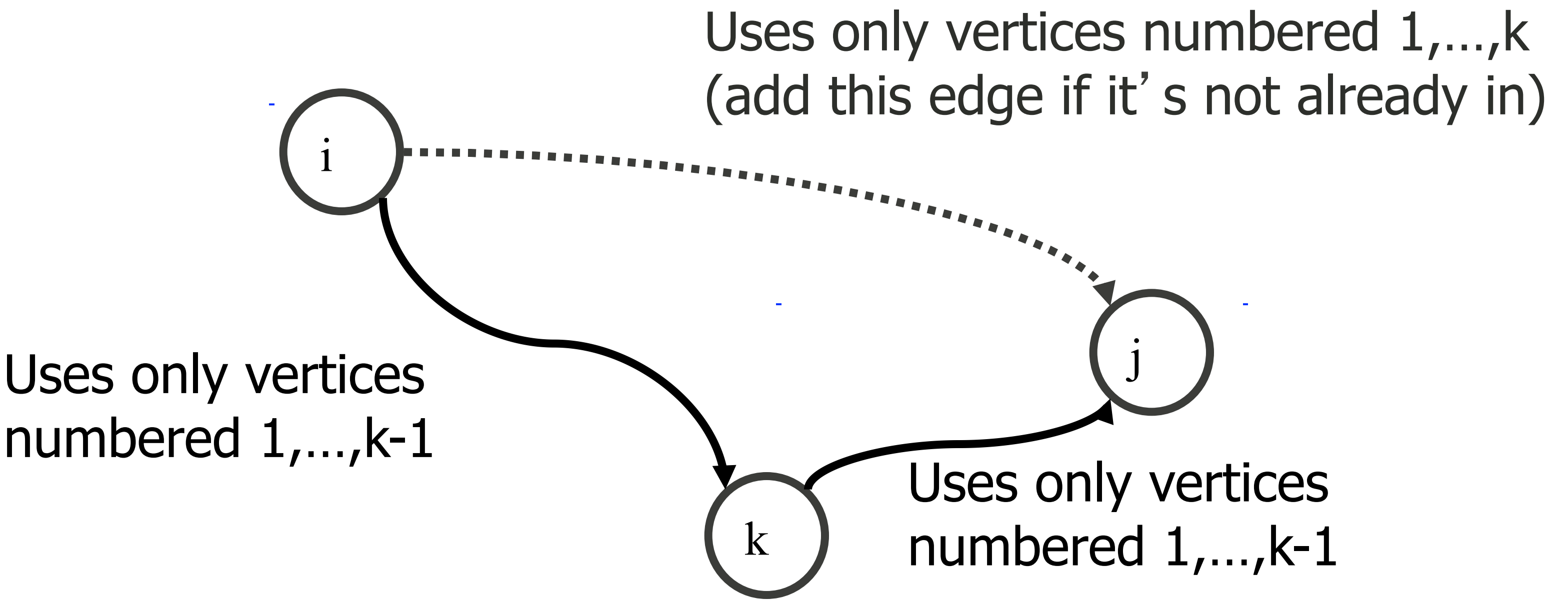
- We can perform DFS starting at each vertex
  - O(n(n+m))



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:

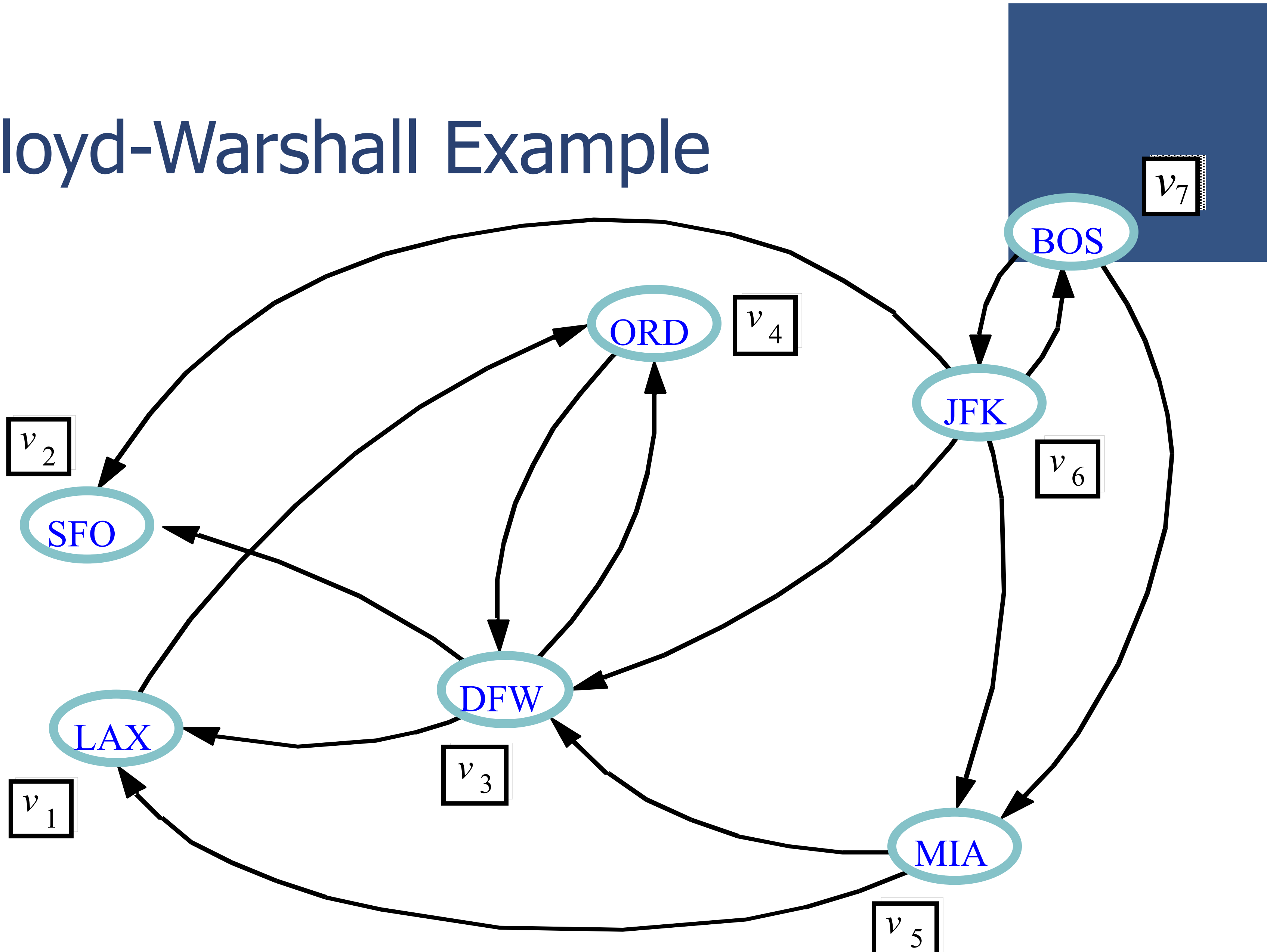


Floyd-Warshall's Algorithm

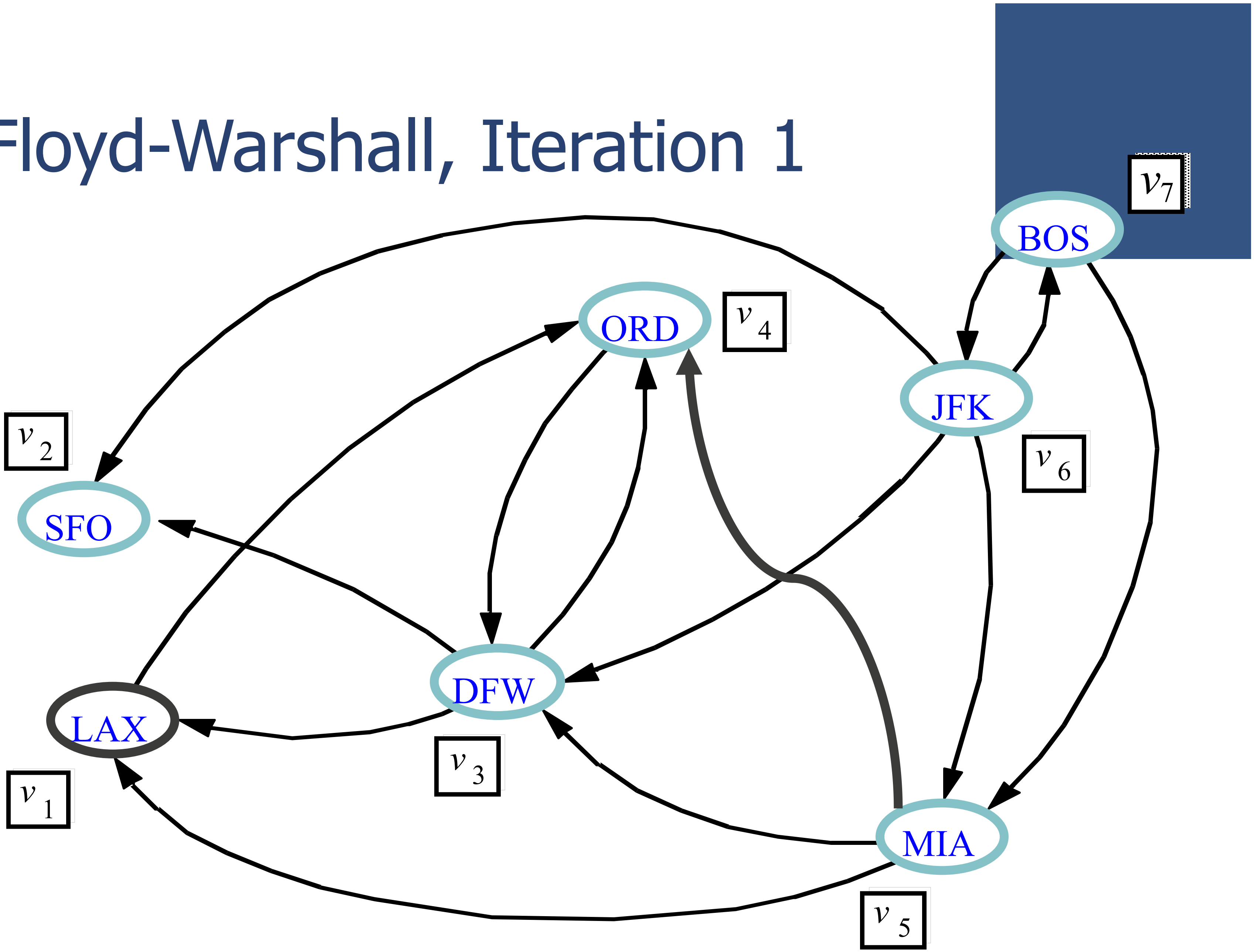
- Number vertices $v_1, \ldots, v_n$
- Compute digraphs $G_0, \ldots, G_n$
  - $G_0 = G$
  - G_k has directed edge (v_i, v_j) if G has a directed path from v_i to v_j with intermediate vertices in $\{v_1, \ldots, v_k\}$
- We have that $G_n = G^*$
- In phase k, digraph G_k is computed from G_{k-1}
- Running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

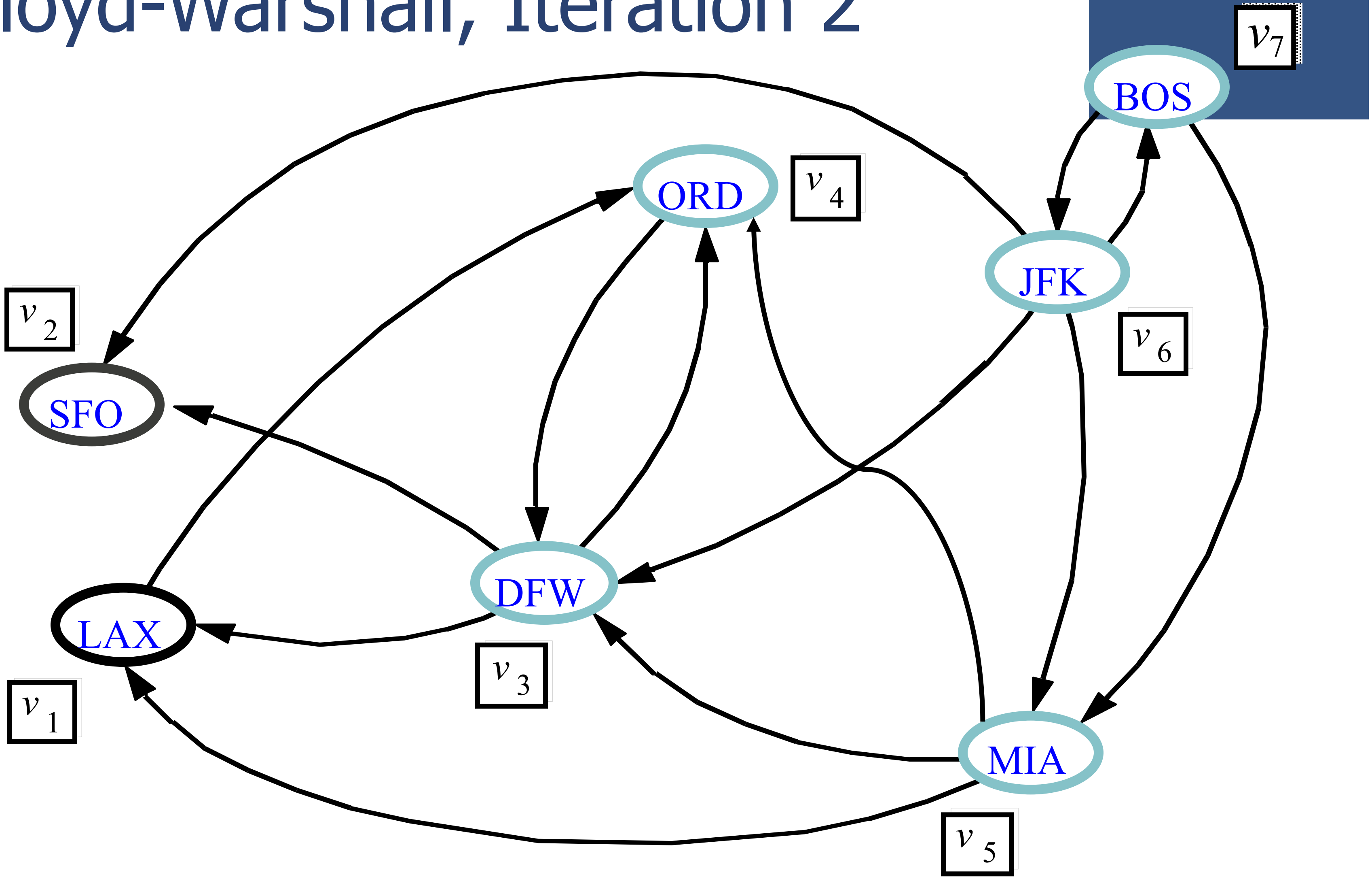
Floyd-Warshall Example



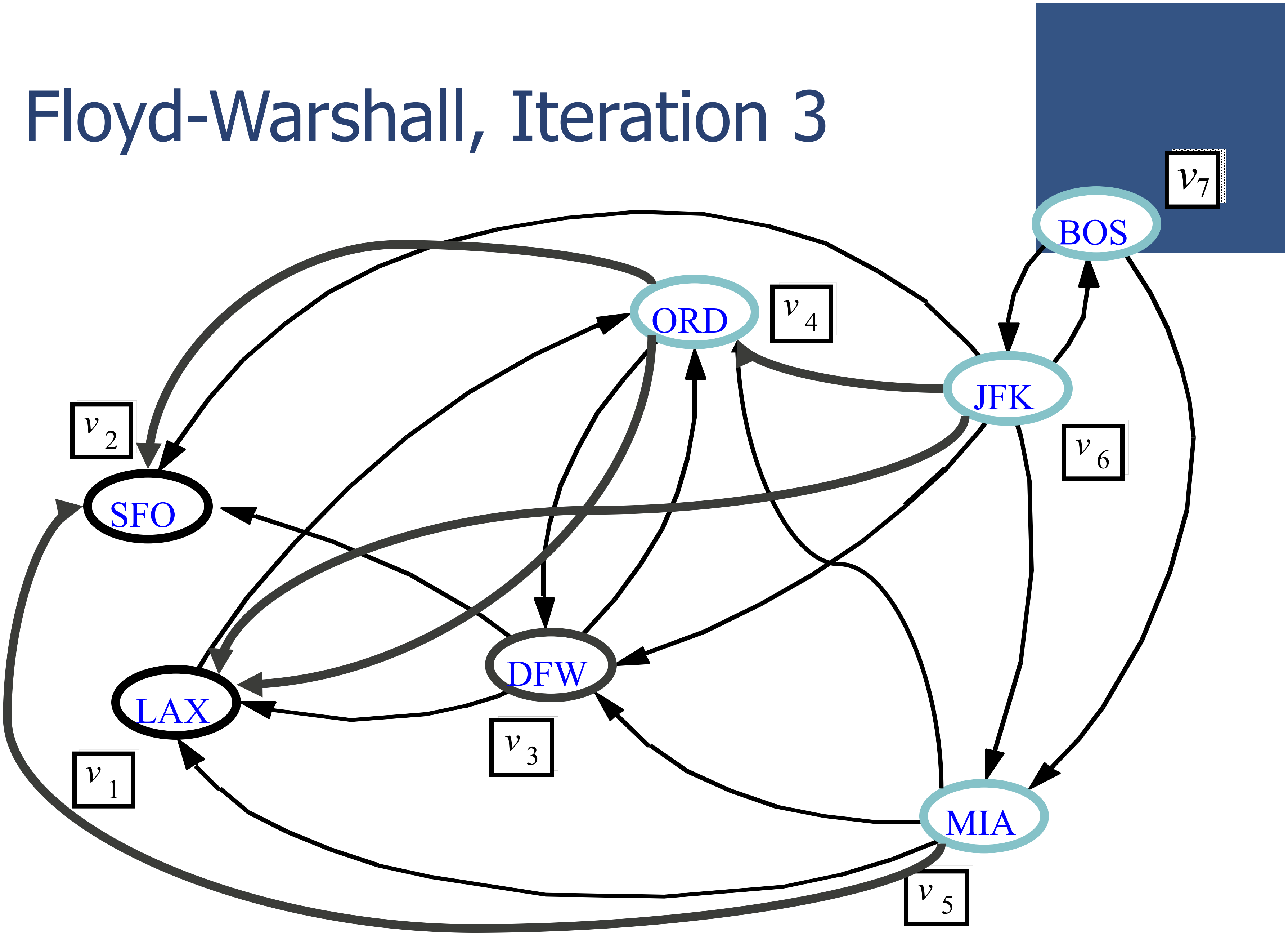
Floyd-Warshall, Iteration 1



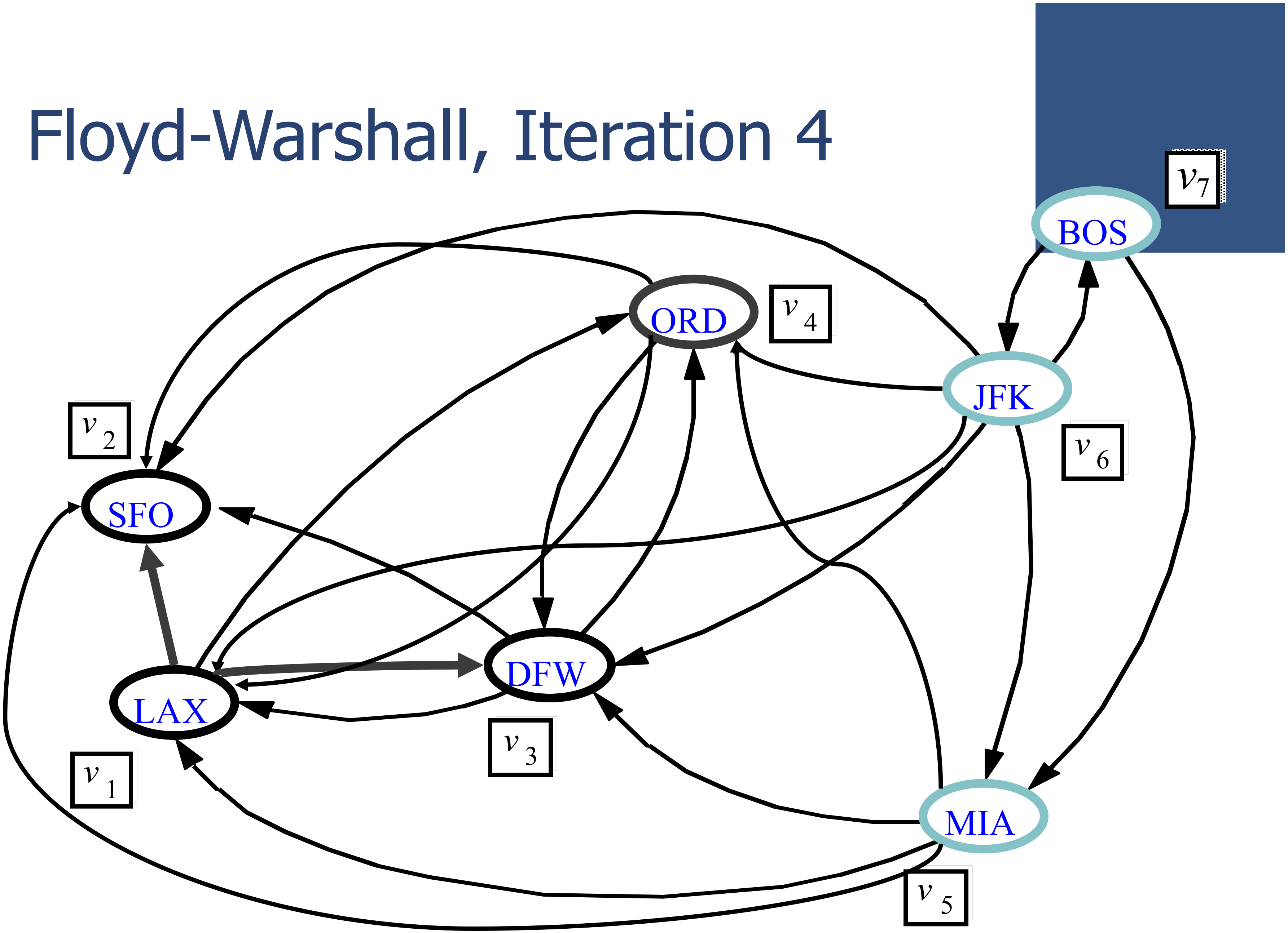
Floyd-Warshall, Iteration 2



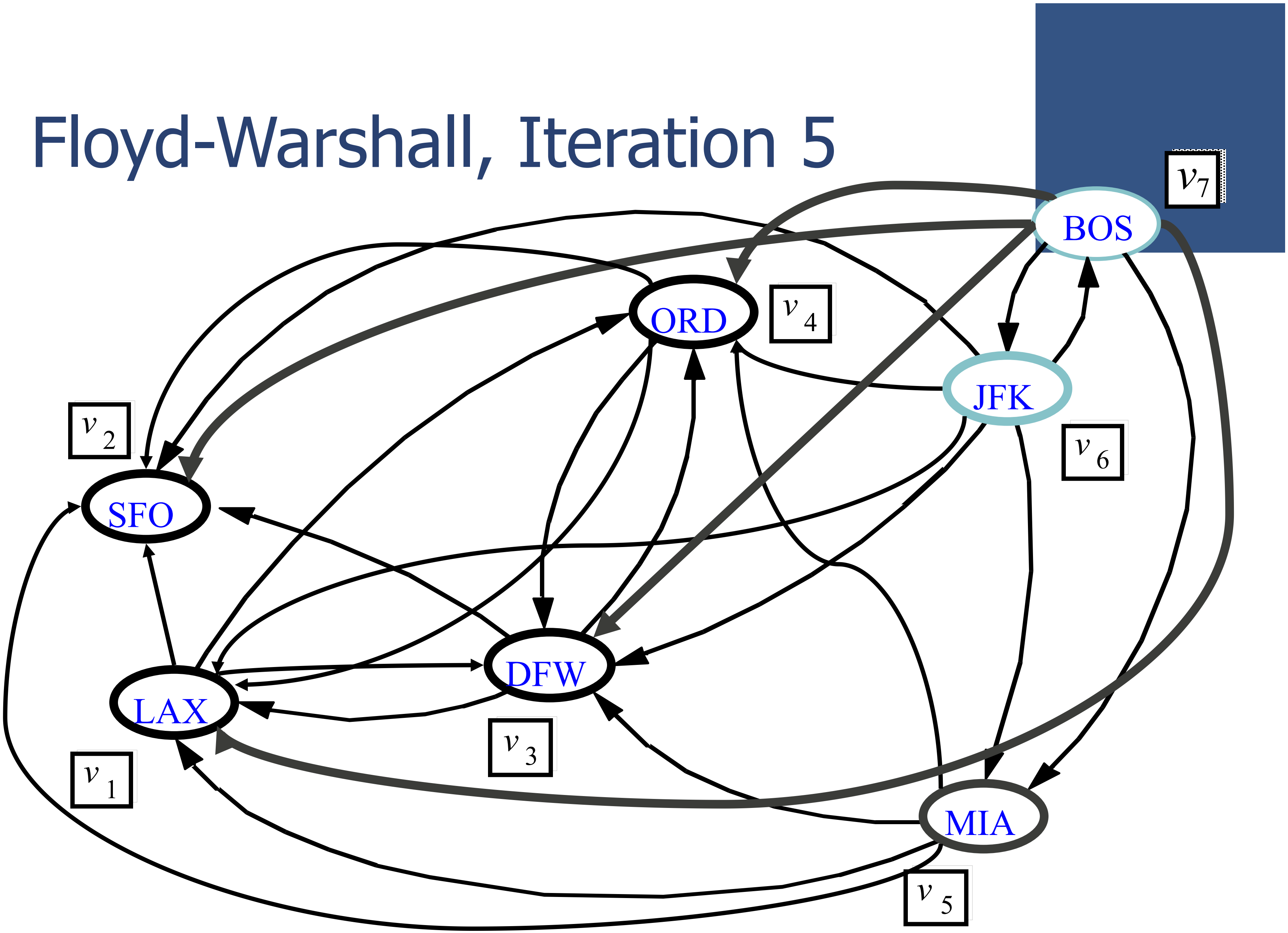
Floyd-Warshall, Iteration 3



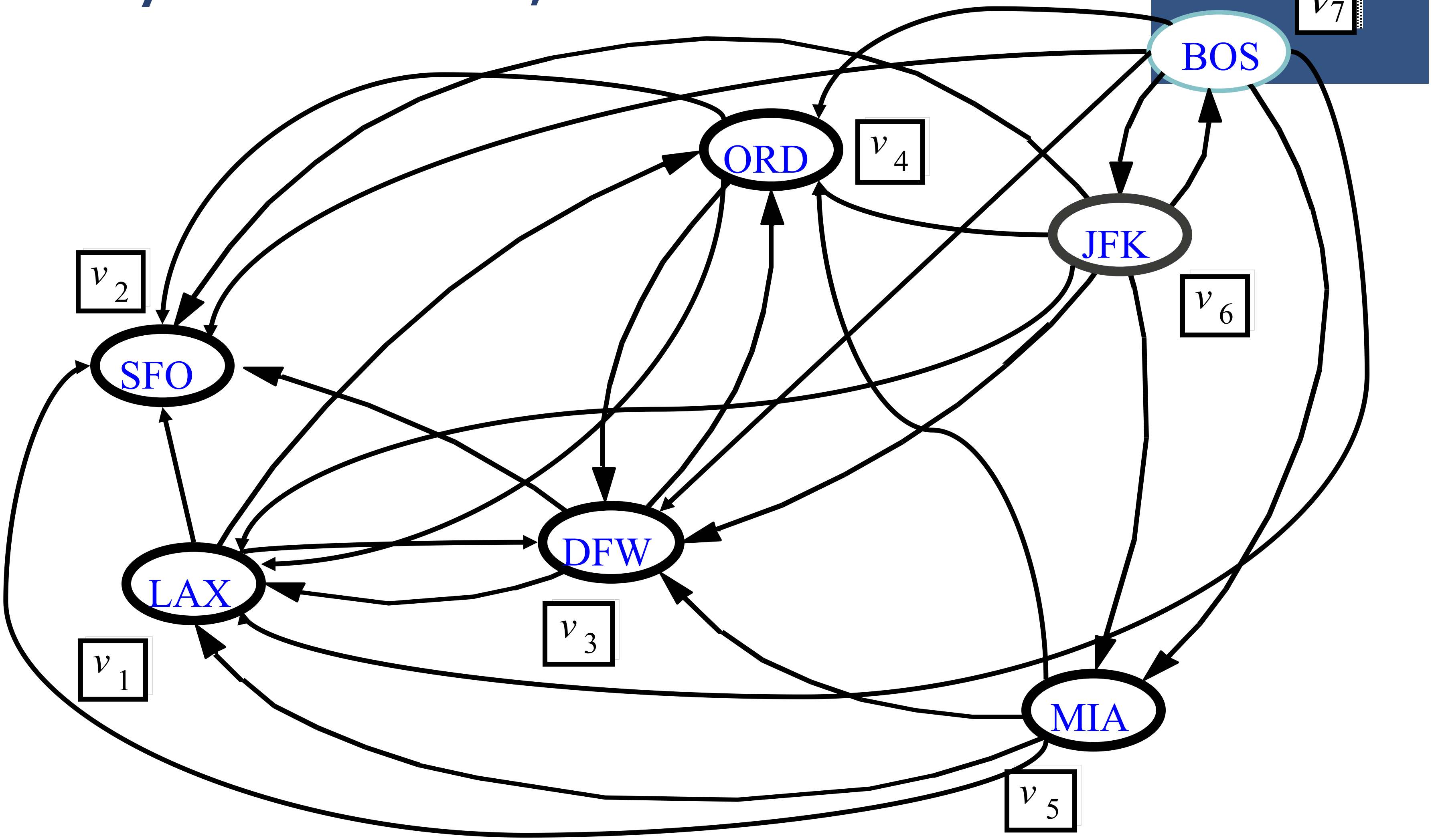
Floyd-Warshall, Iteration 4



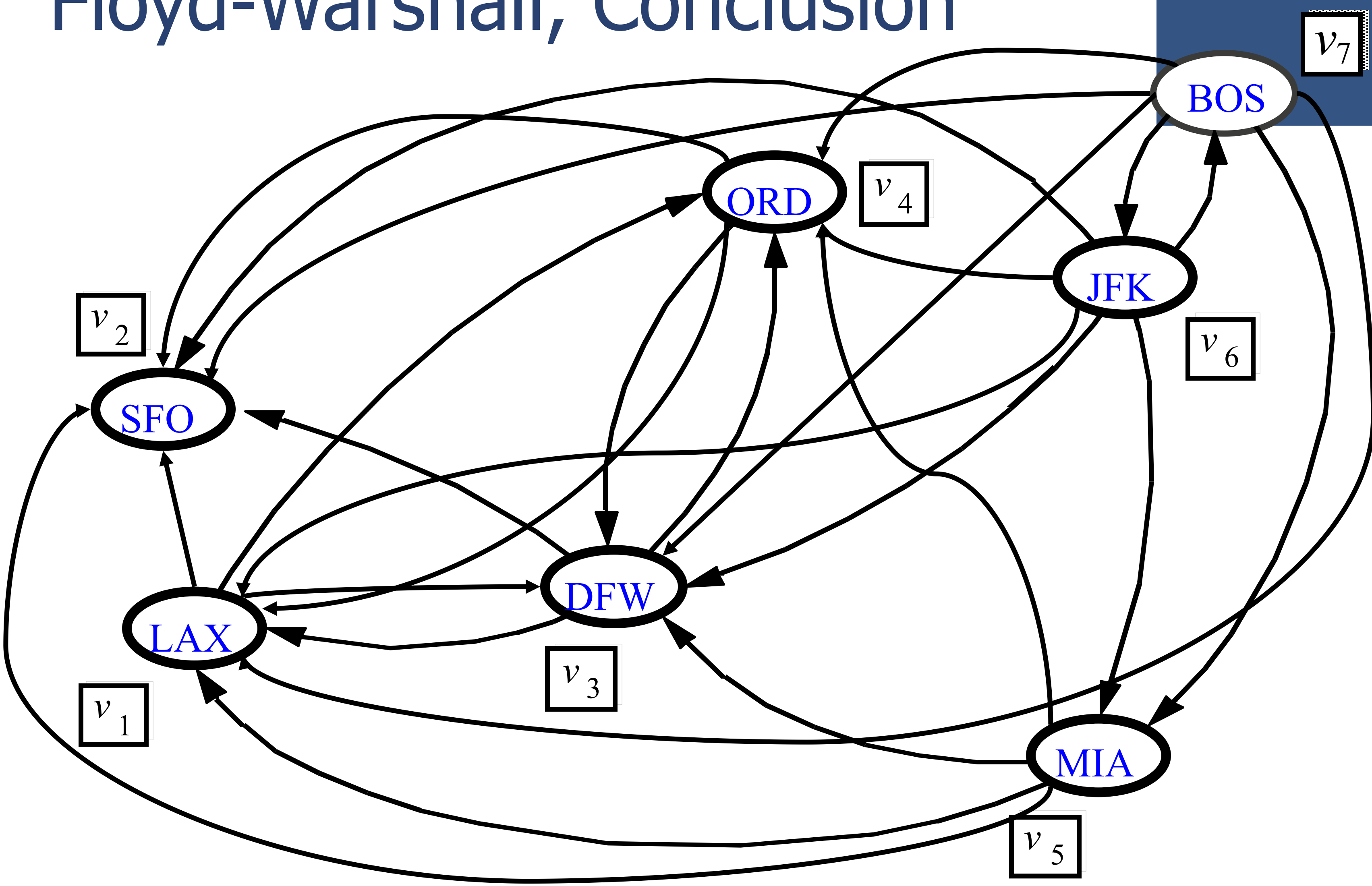
Floyd-Warshall, Iteration 5



Floyd-Warshall, Iteration 6

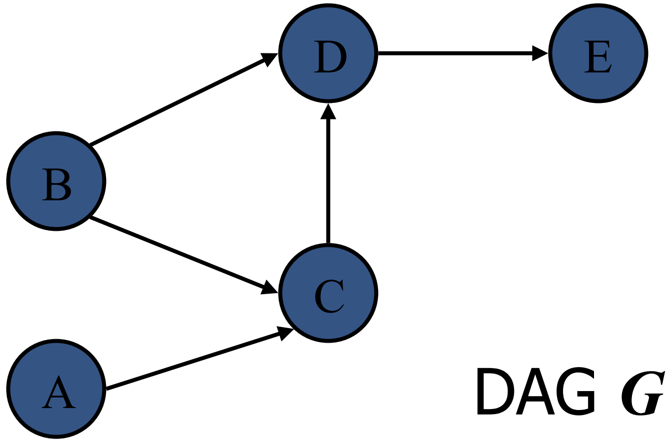


Floyd-Warshall, Conclusion



DAGs and Topological Ordering

- A directed acyclic graph (DAG) is a digraph that has no directed cycles



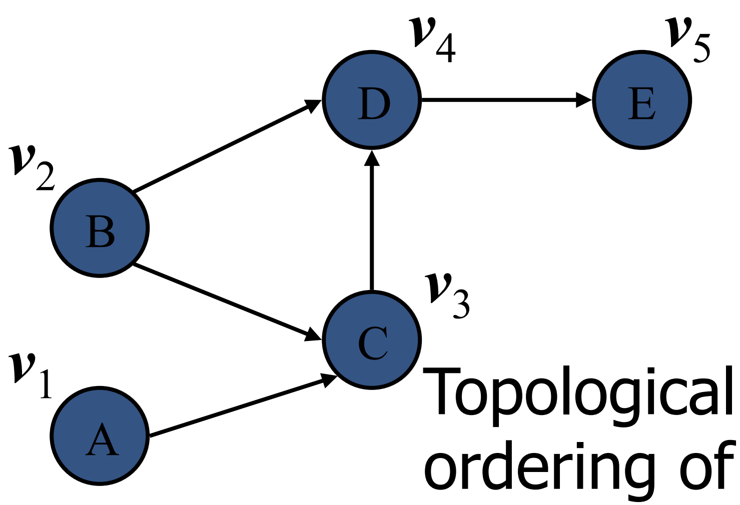
DAG G

DAGs and Topological Ordering

- 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

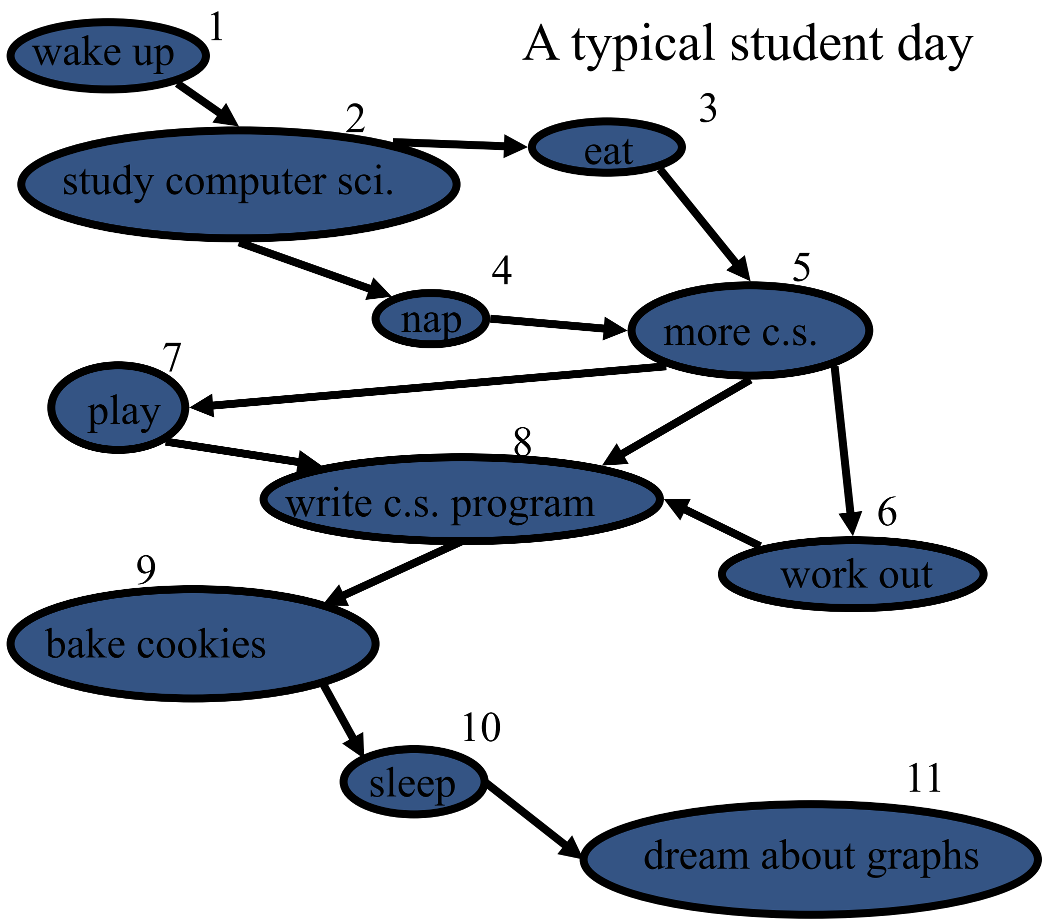
A digraph admits a topological ordering if and only if it is a DAG



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)
    $H \leftarrow G$ // 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)

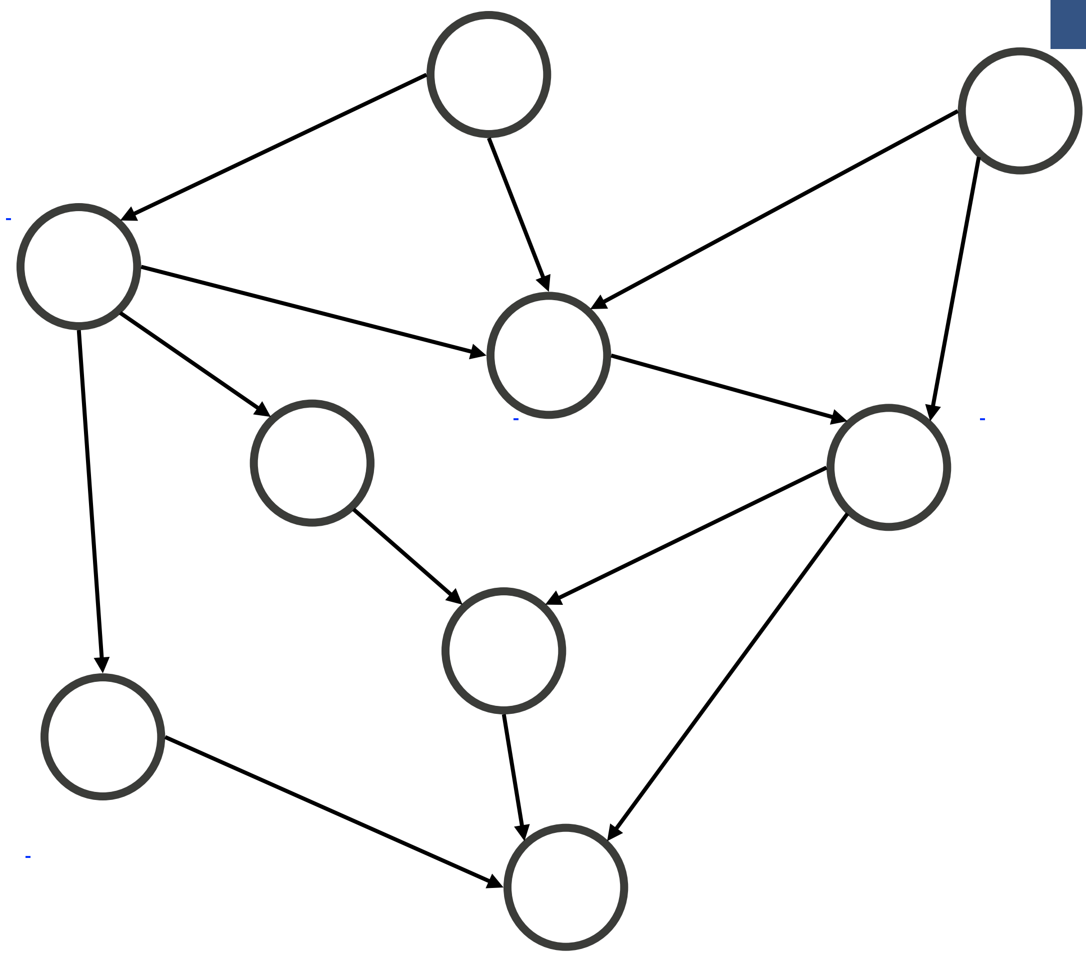
Implementation with DFS

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

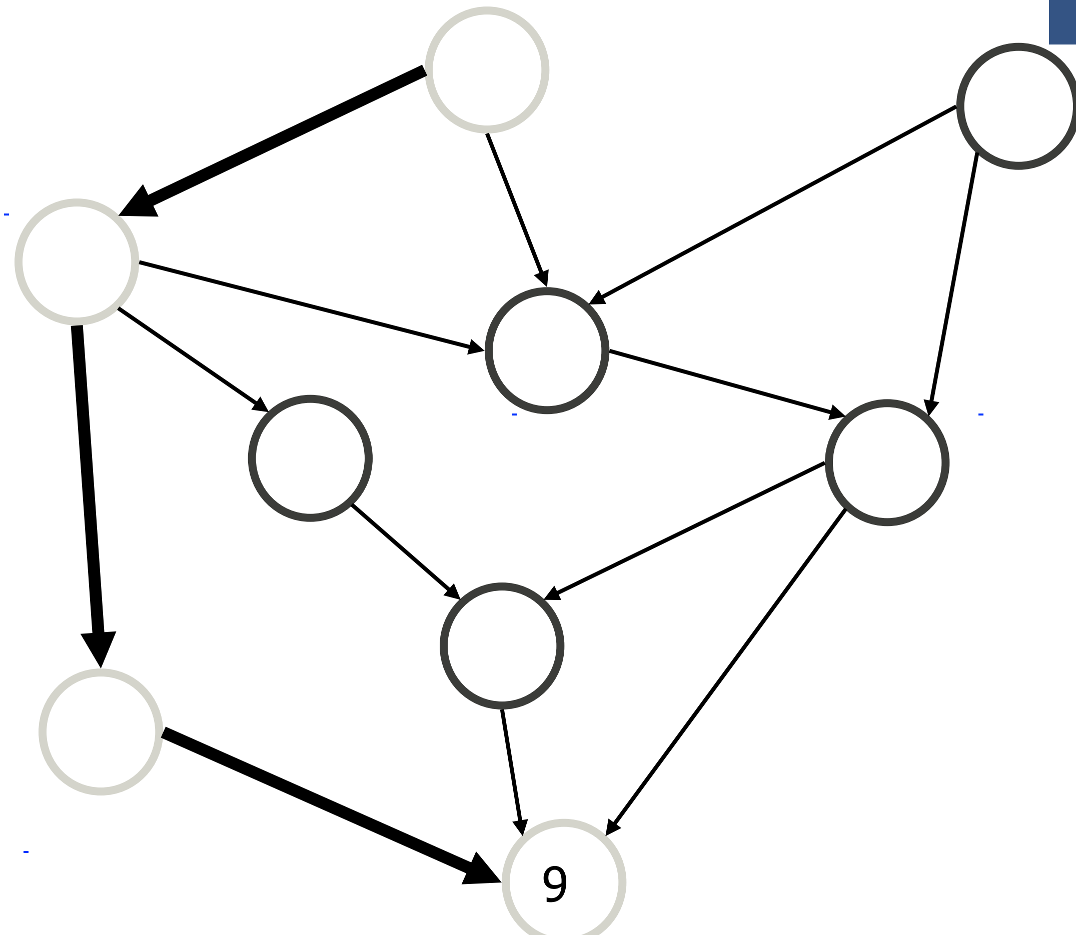
```
Algorithm topologicalDFS(G)
    Input dag G
    Output topological ordering of G
  n ← G.numVertices()
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      topologicalDFS(G, v)
```

```
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
```

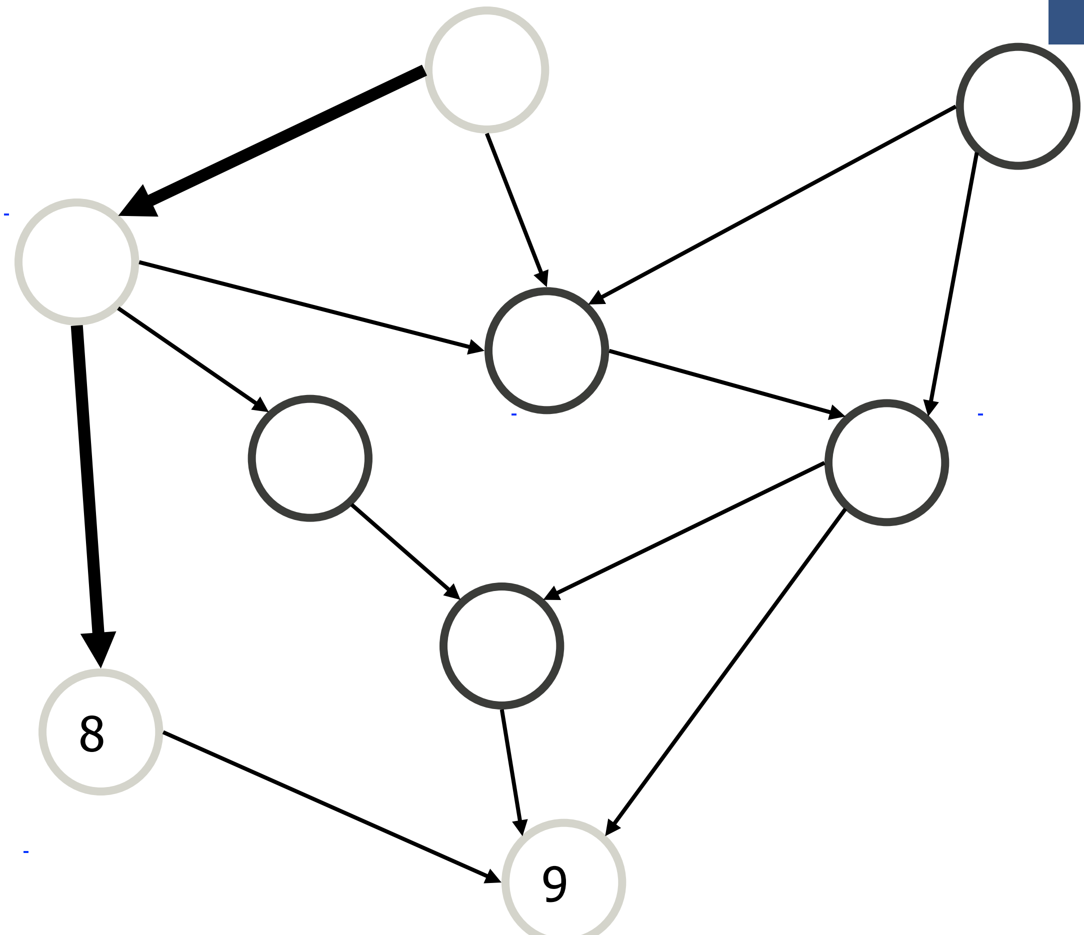
Topological Sorting Example



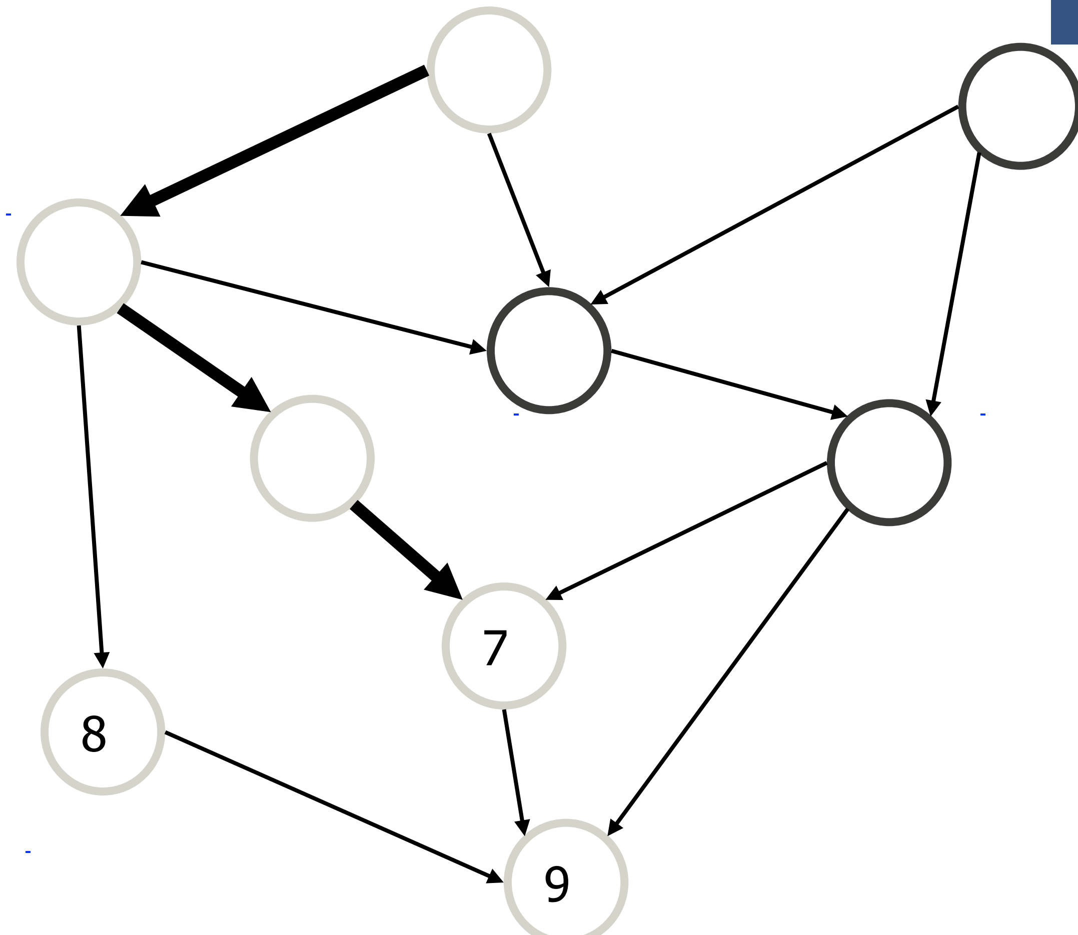
Topological Sorting Example



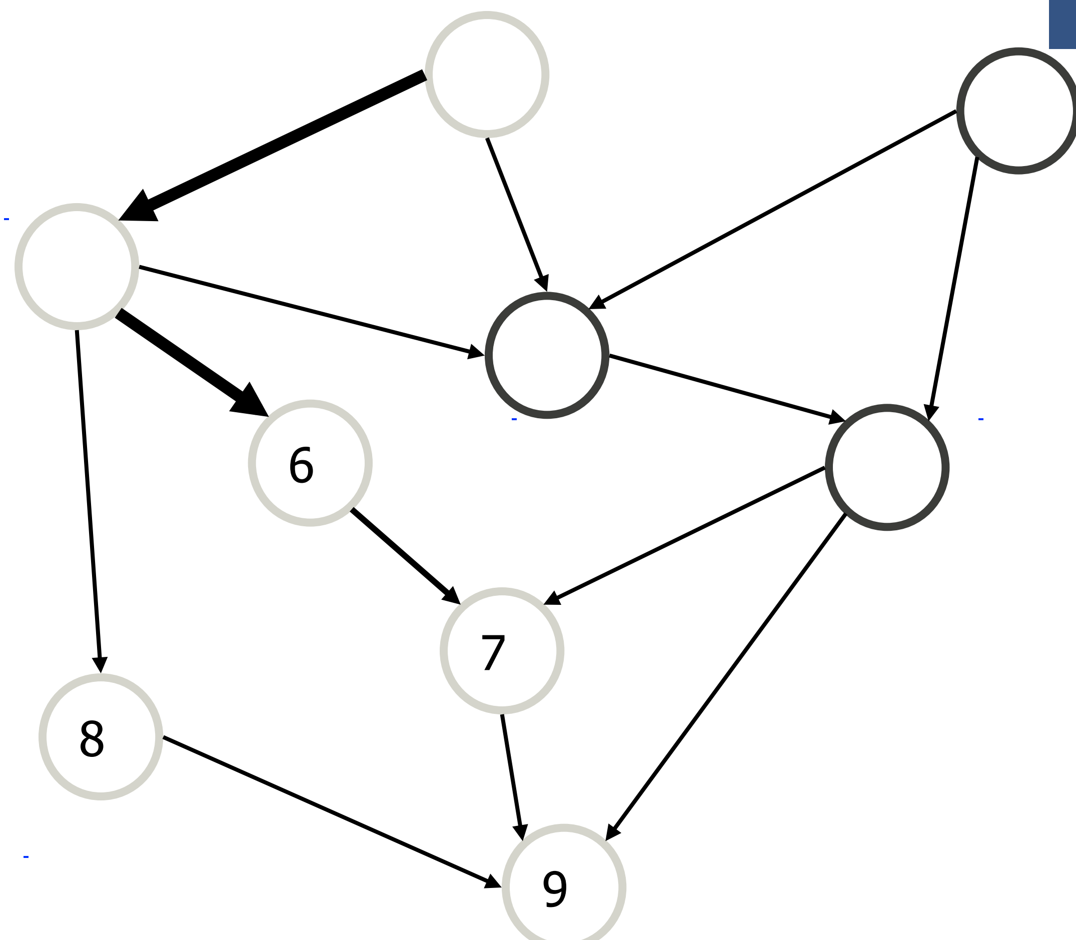
Topological Sorting Example



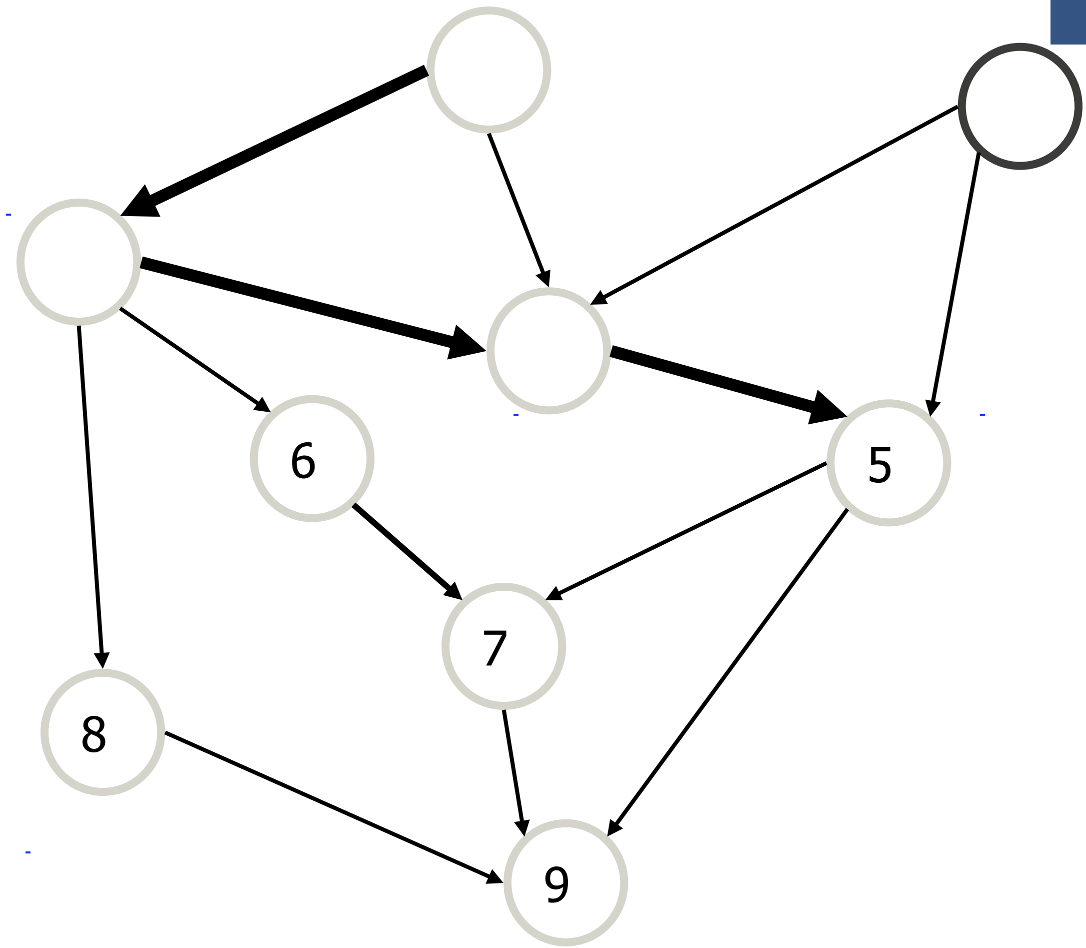
Topological Sorting Example



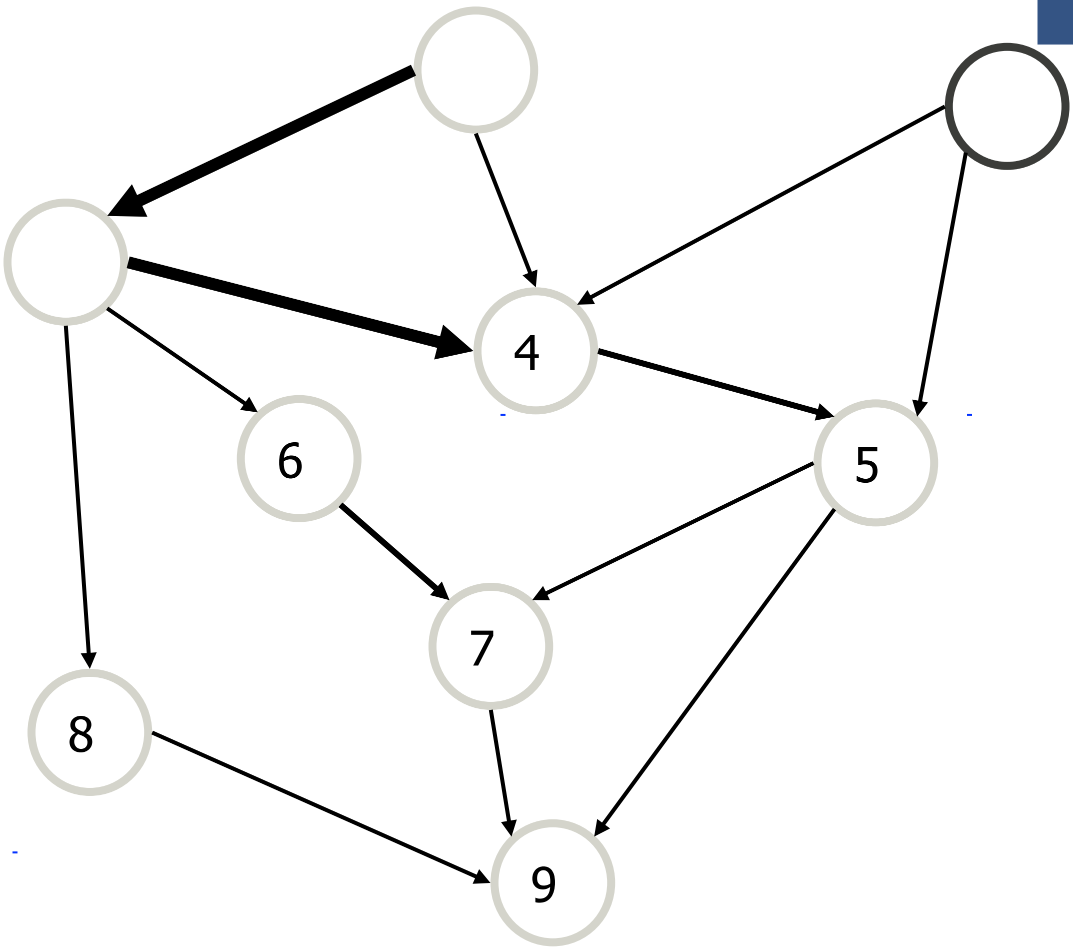
Topological Sorting Example



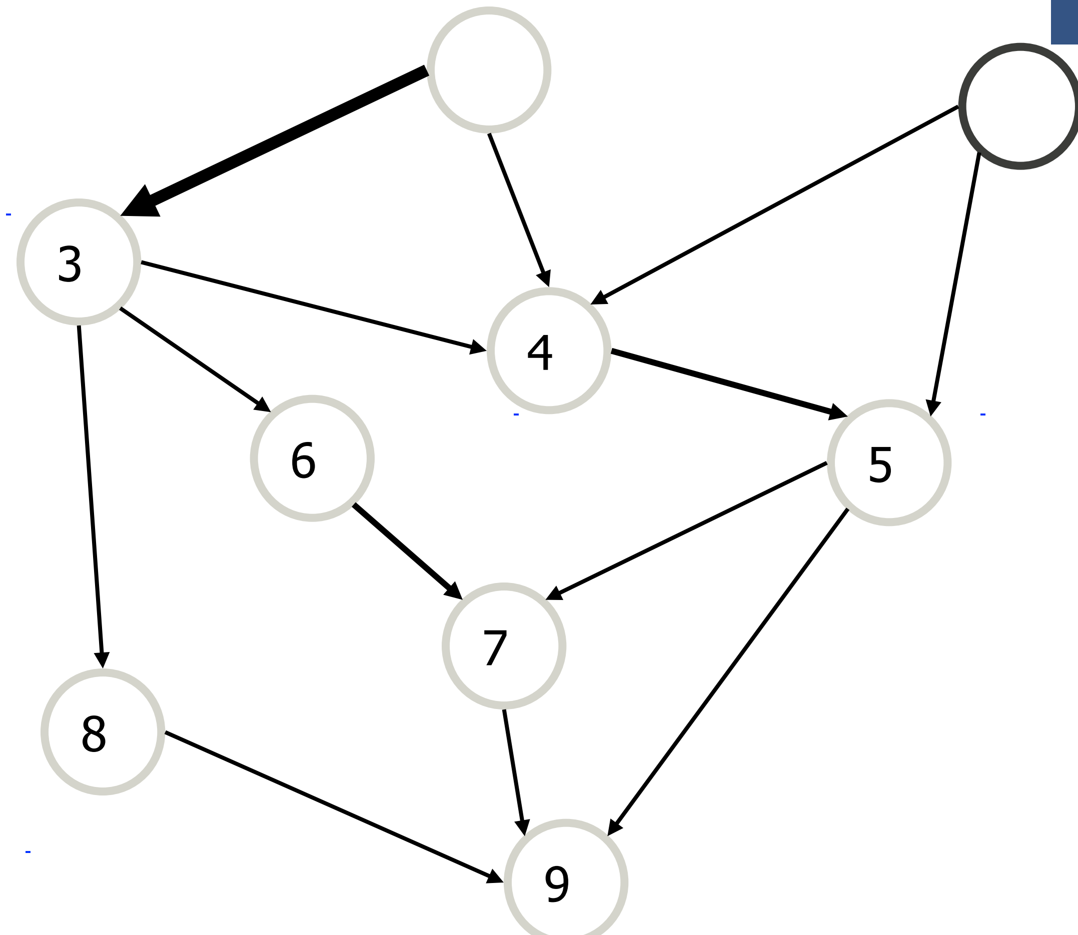
Topological Sorting Example



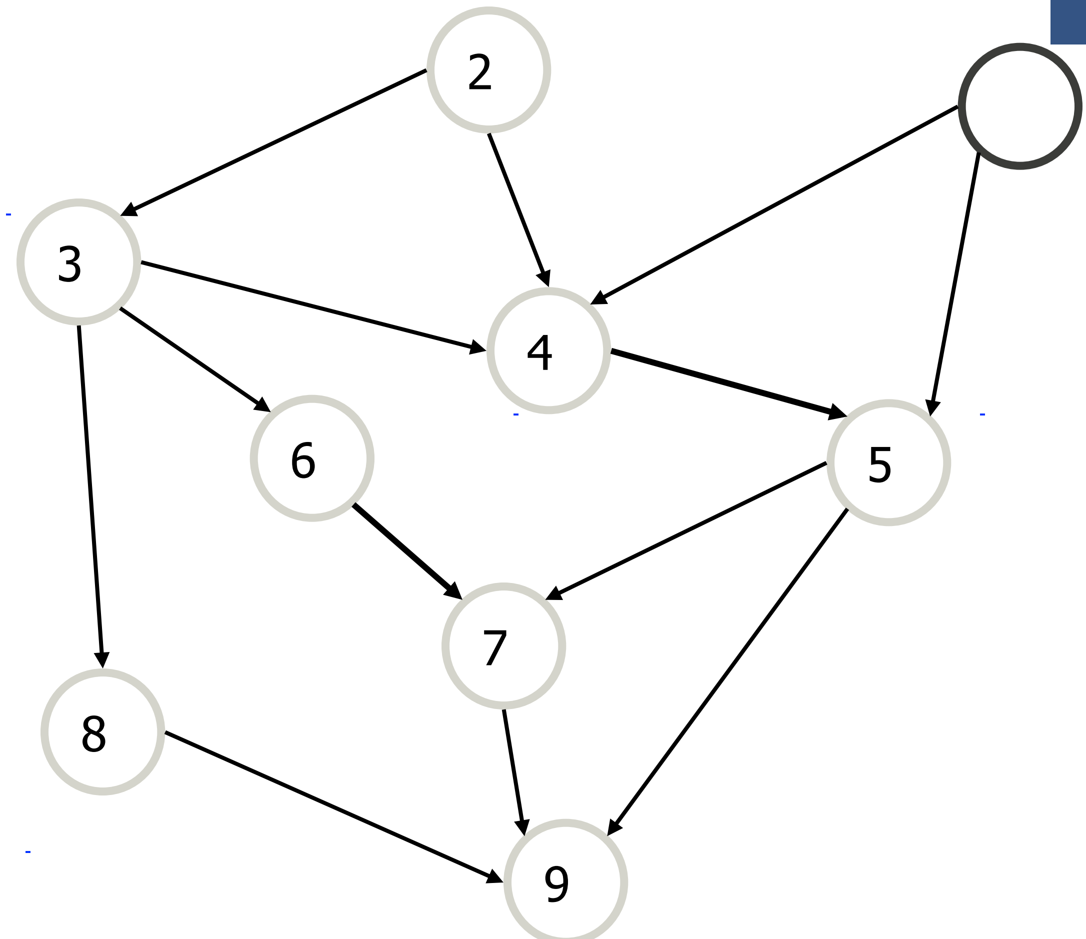
Topological Sorting Example



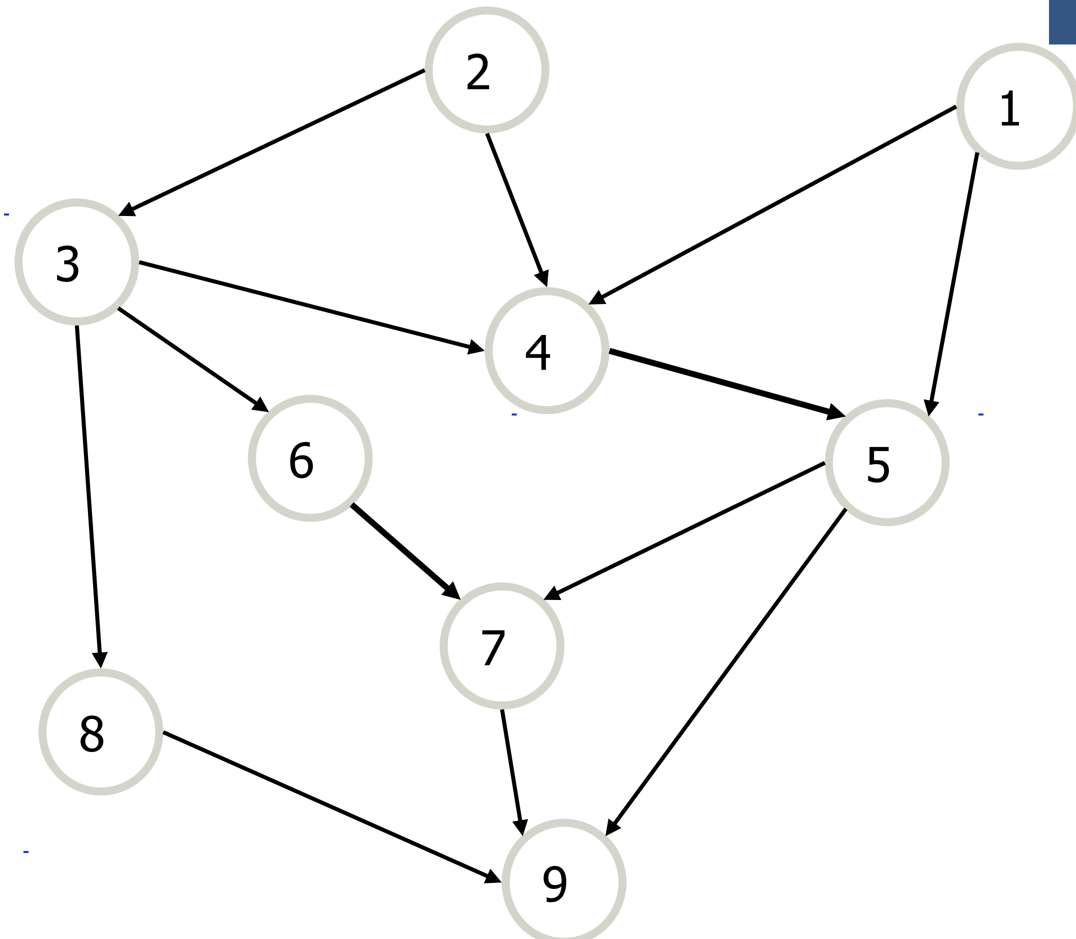
Topological Sorting Example



Topological Sorting Example



Topological Sorting Example



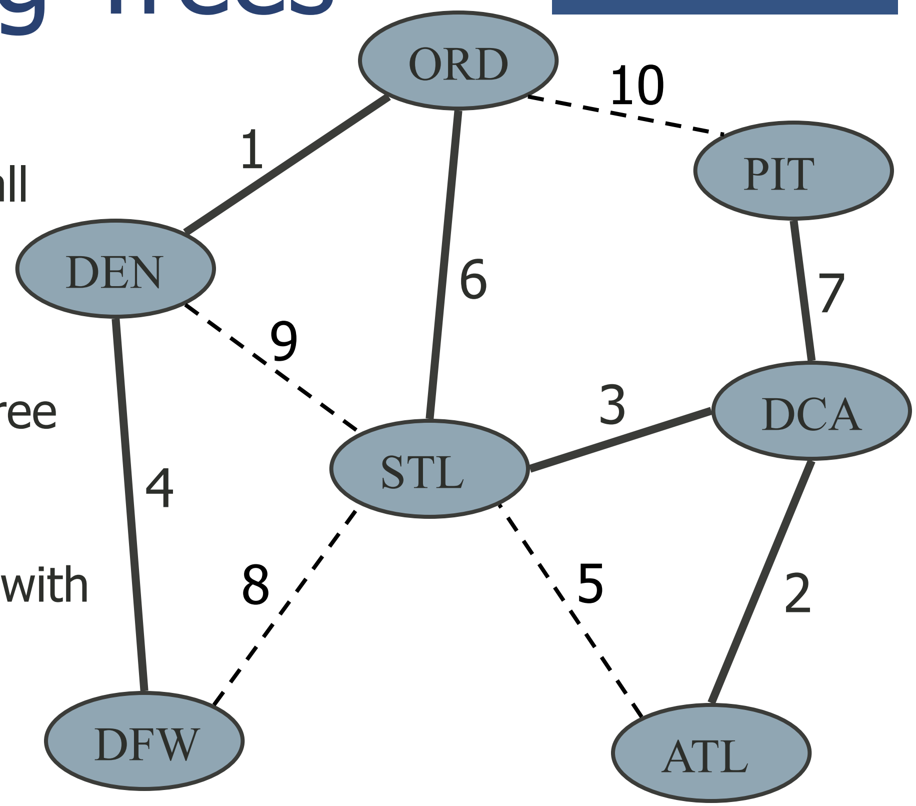
A Quiz

- Liang wants to plan his course schedule. The course prerequisites are:
  - CS15: (none)
  - CS16: CS15
  - CS22: (none)
  - CS31: CS15
  - CS32: CS16, CS31
  - CS126: CS22, CS32, CS16
  - CS127: CS16
  - CS141: CS22, CS16
  - CS169:CS32

Please help Liang to find the sequence of courses that allows him to satisfy all the prerequisities.

Minimum Spanning Trees

- Spanning subgraph
  - Subgraph of a graph G containing all the vertices of G
- Spanning tree
  - Spanning subgraph that is itself a tree
- Minimum spanning tree (MST)
  - Spanning tree of a weighted graph with minimum total edge weight
- Applications
  - Communications networks
  - Transportation networks



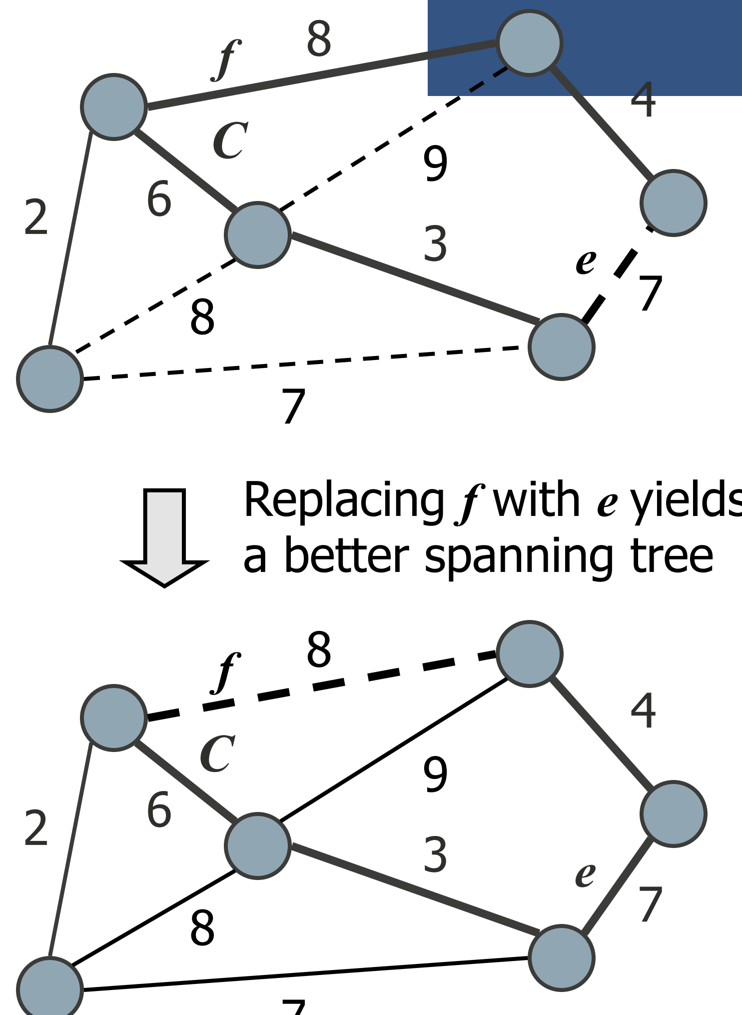
Cycle Property

Cycle Property:

- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and C let be the cycle formed by e with T
- For every edge f of C, $weight(f) \leq weight(e)$

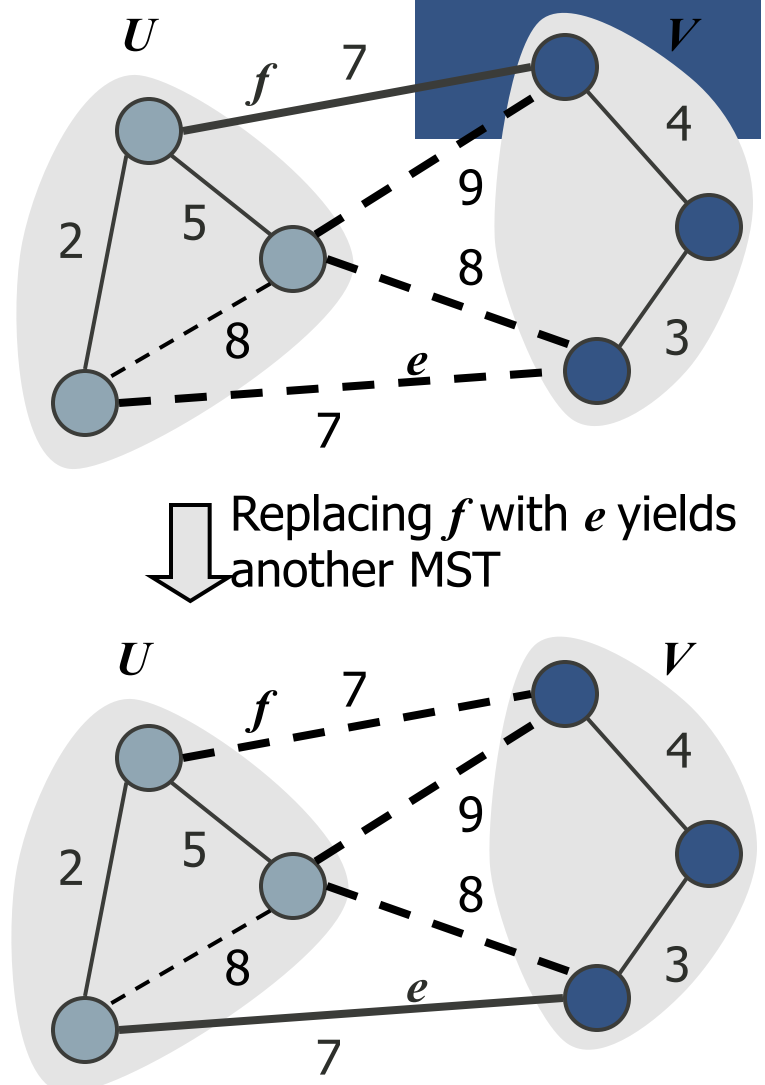
Proof:

- By contradiction
- If $weight(f) > weight(e)$ we can get a spanning tree of smaller weight by replacing e with f



Partition Property

- Partition Property:
  - Consider a partition of the vertices of G into subsets U and V
  - Let e be an edge of minimum weight across the partition
  - There is a minimum spanning tree of G containing edge e

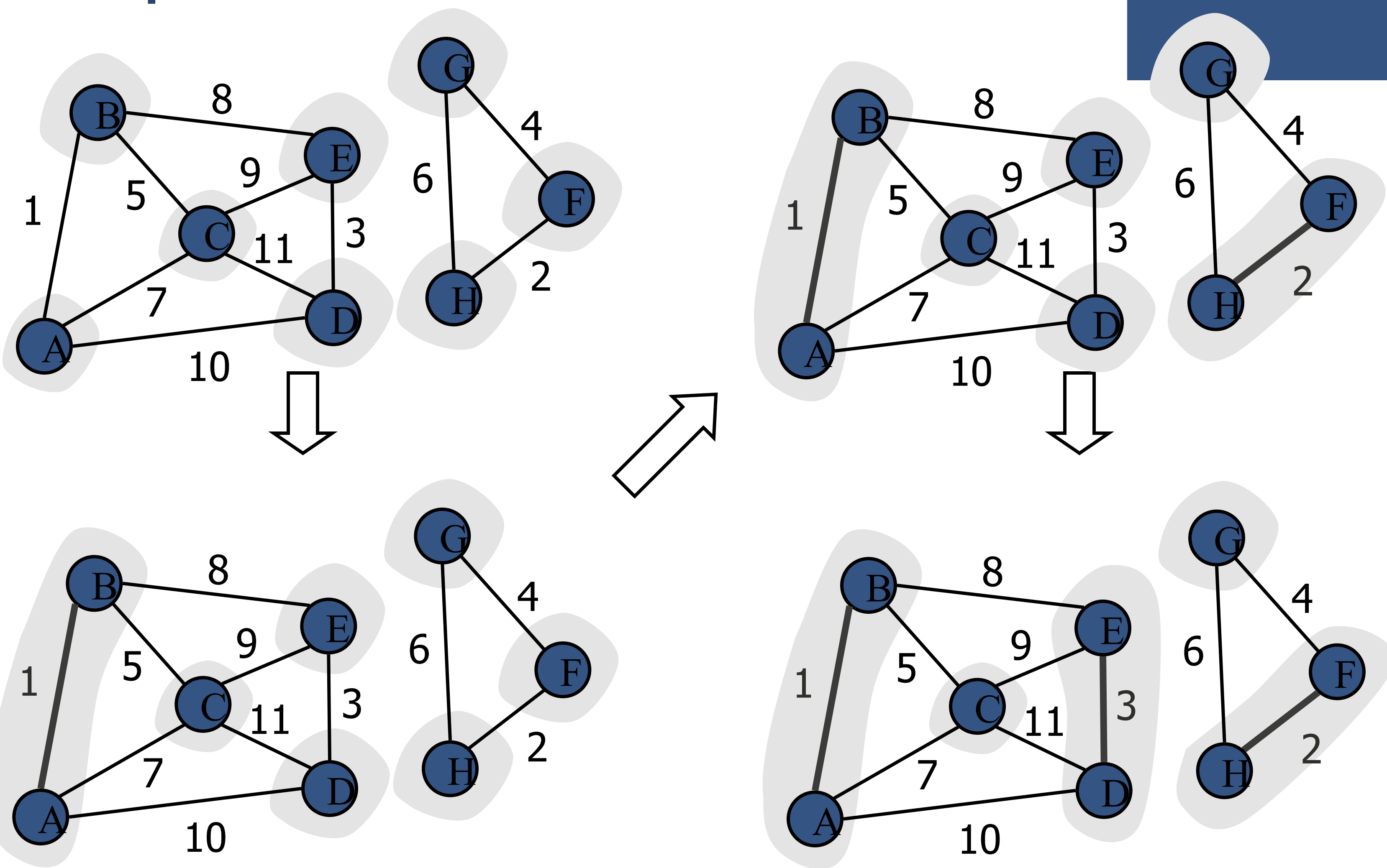


Kruskal's Algorithm

- Maintain a partition of the vertices into clusters
  - Initially, single-vertex clusters
  - Keep an MST for each cluster
  - Merge "closest" clusters and their MSTs
- A priority queue stores the edges outside clusters
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - One cluster and one MST

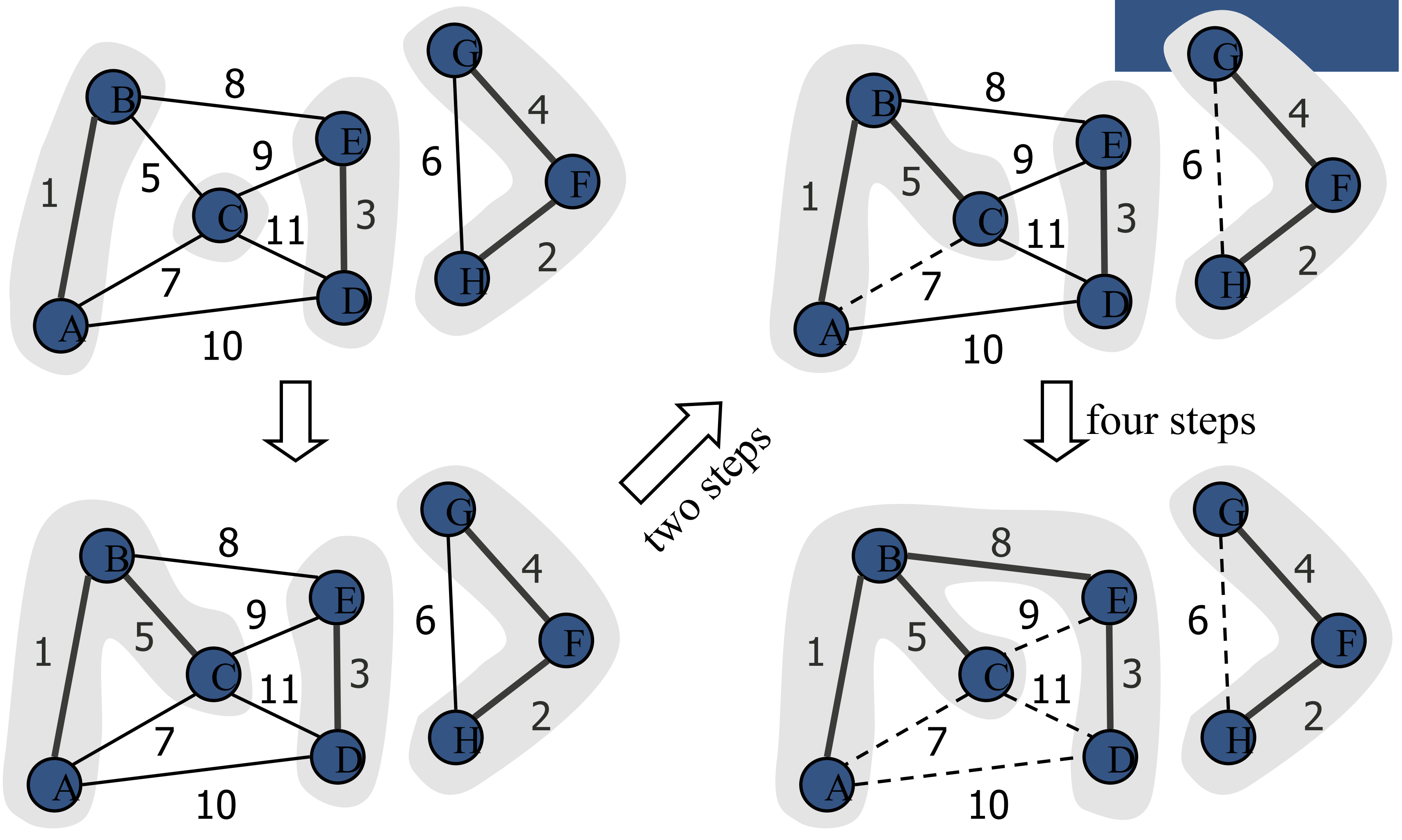
Algorithm *KruskalMST*(G)
  for each vertex v in G **do**
    Create a cluster consisting of v
  let Q be a priority queue.
  Insert all edges into Q
  $T \leftarrow \emptyset$
  {T is the union of the MSTs of the clusters}
  while T has fewer than $n - 1$ edges **do**
    $e \leftarrow Q.removeMin().getValue()$
    $[u, v] \leftarrow G.endVertices(e)$
    $A \leftarrow getCluster(u)$
    $B \leftarrow getCluster(v)$
    if $A \neq B$ **then**
      Add edge e to T
      $mergeClusters(A, B)$
  return T

Example



G
B 8 E 4
1 5 9 6 F
C 11 3
7 D H 2
A 10
G
B 8 E 4
1 5 9 6 F
C 11 3
7 D H 2
A 10
G
B 8 E 4
1 5 9 6 F
C 11 3
7 D H 2
A 10
G
B 8 E 4
1 5 9 6 F
C 11 3
7 D H 2
A 10

Example (contd.)



Prim-Jarnik's Algorithm

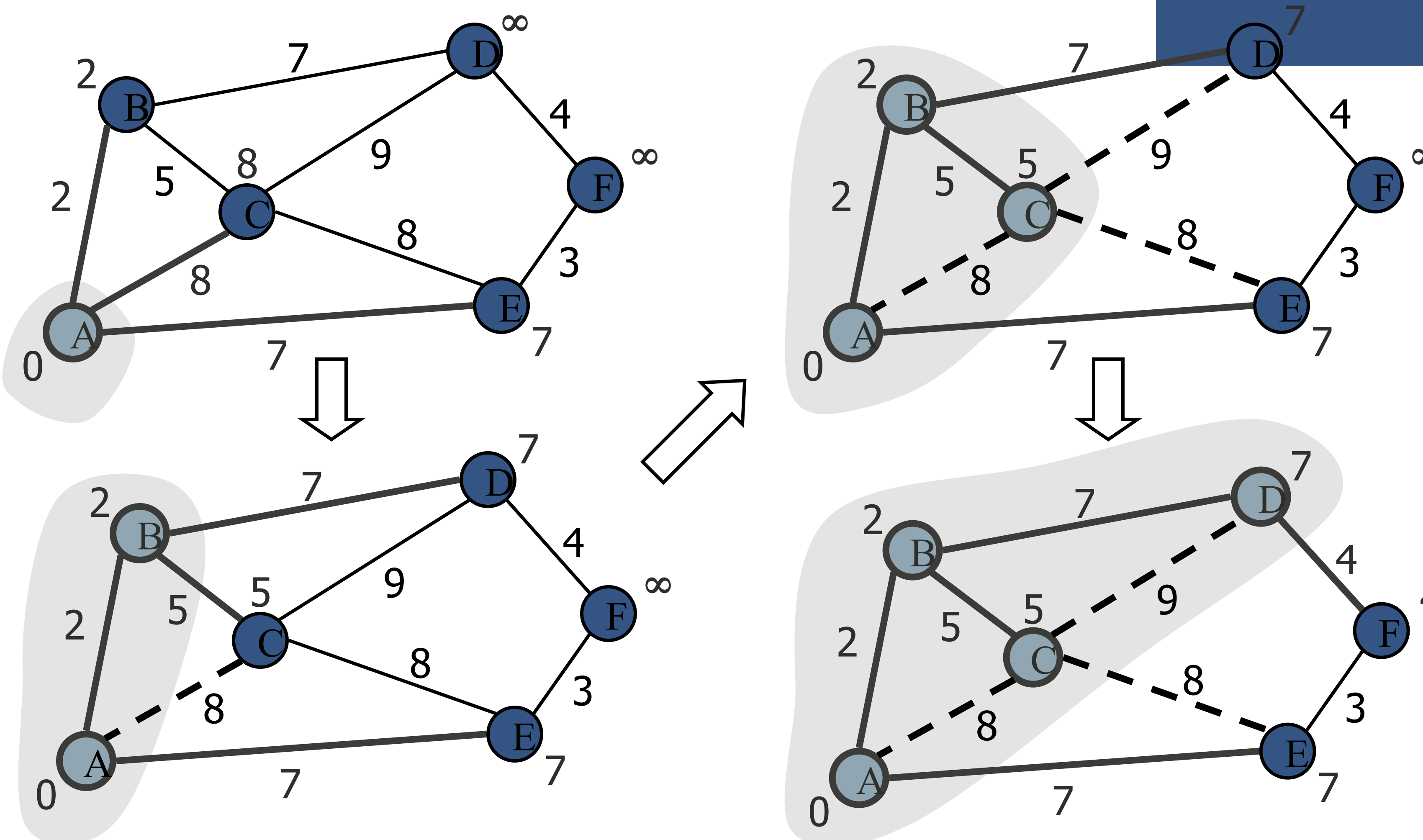
- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- We store with each vertex v label $d(v)$ representing the smallest weight of an edge connecting v to a vertex in the cloud
- At each step:
  - We add to the cloud the vertex u outside the cloud with the smallest distance label
  - We update the labels of the vertices adjacent to u

Prim-Jarnik's Algorithm (cont.)

- A heap-based adaptable priority queue with location-aware entries stores the vertices outside the cloud
  - Key: distance
  - Value: vertex
  - Recall that method *replaceKey*(l,k) changes the key of entry l
- We store three labels with each vertex:
  - Distance
  - Parent edge in MST
  - Entry in priority queue

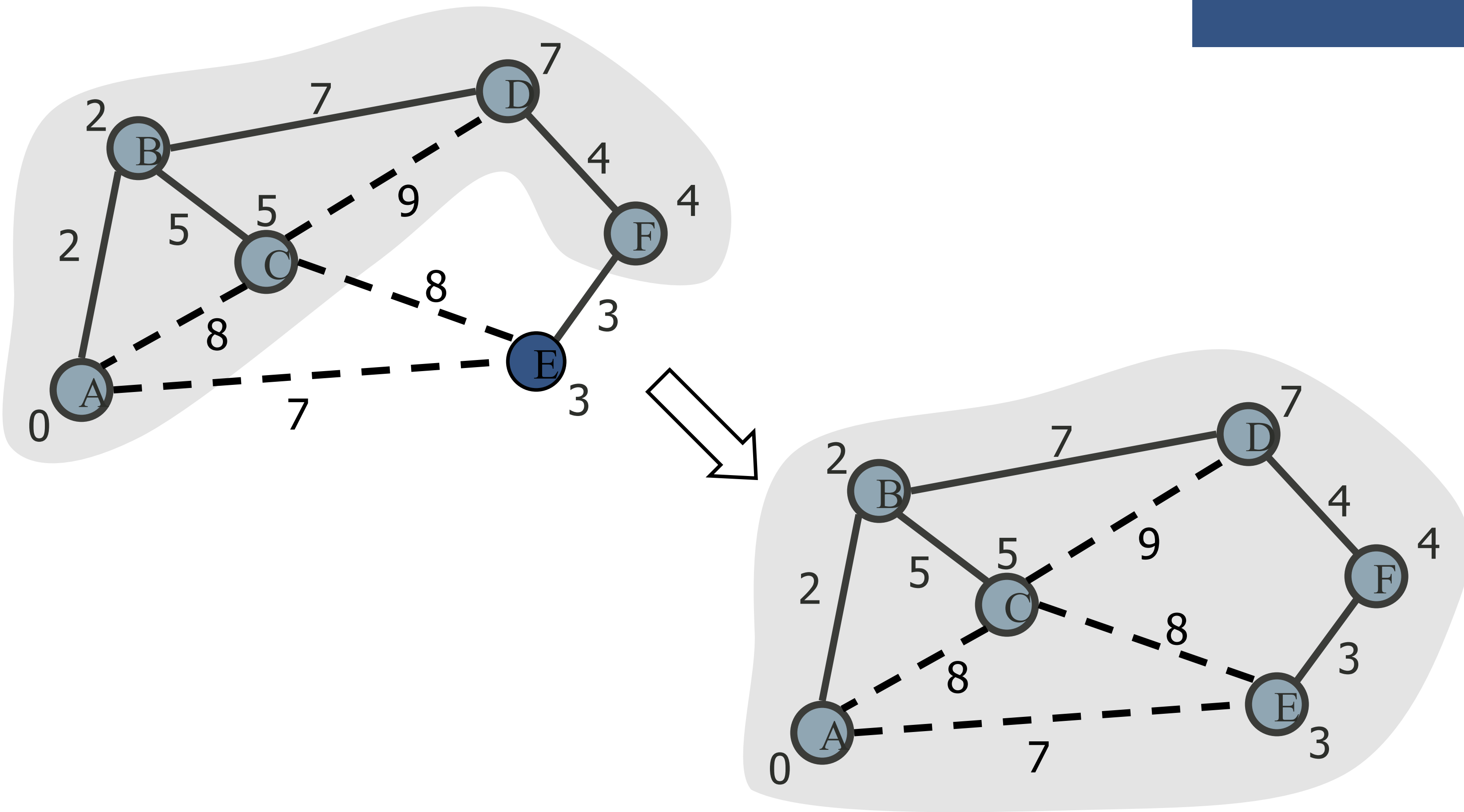
```
Algorithm PrimJarnikMST(G)
  Q ← new heap-based priority queue
  s ← a vertex of G
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
    setParent(v, ∅)
    l ← Q.insert(getDistance(v), v)
    setLocator(v,l)
  while ¬Q.isEmpty()
    l ← Q.removeMin()
    u ← l.getValue()
    for all e ∈ G.incidentEdges(u)
      z ← G.opposite(u,e)
      r ← weight(e)
      if r < getDistance(z)
        setDistance(z, r)
        setParent(z,e)
        Q.replaceKey(getEntry(z), r)
```

Example



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

Example (contd.)



Baruvka's Algorithm

- Like Kruskal's Algorithm, Baruvka's algorithm grows many clusters at once and maintains a forest T
- Each iteration of the while loop halves the number of connected components in forest T
- The running time is $O(m \log n)$

Algorithm $BaruvkaMST(G)$
  $T \leftarrow V$ {just the vertices of G}
  while T has fewer than $n - 1$ edges **do**
    for each connected component C in T **do**
      Let edge e be the smallest-weight edge from C to another component in T
      if e is not already in T **then**
        Add edge e to T
  return T

Example of Baruvka's Algorithm (animated)