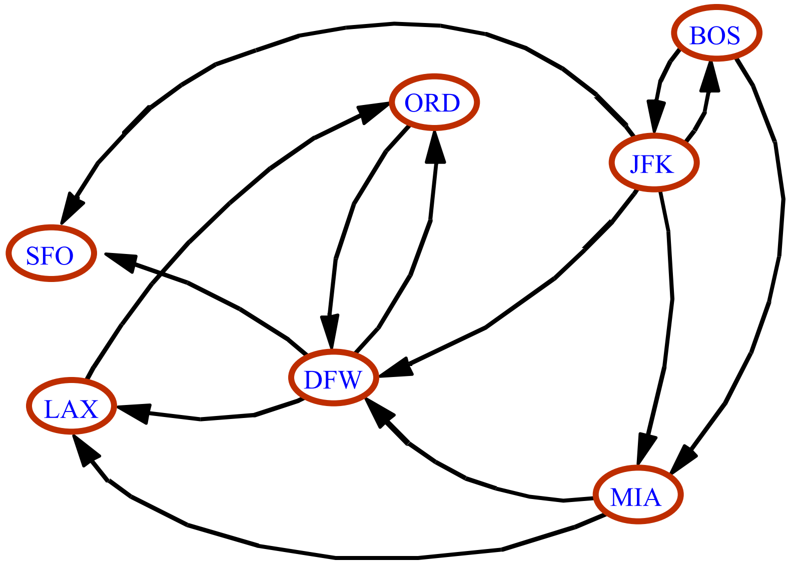
Directed Graphs

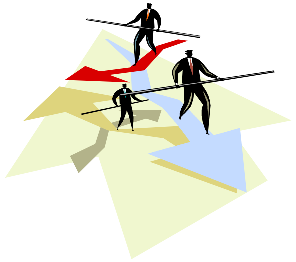
Algorithms & Data Structures
ITCS 6114/8114

Dr. Dewan Tanvir Ahmed
Department of Computer Science
University of North Carolina at Charlotte

Directed Graphs



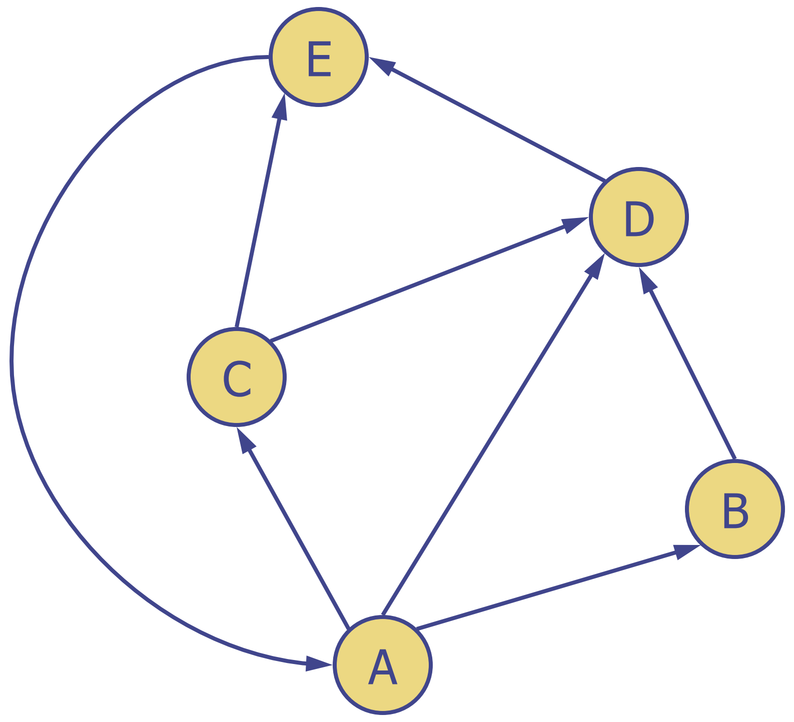
Outline and Reading (§6.4)



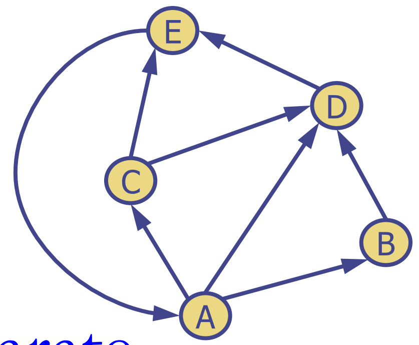
- Reachability (§6.4.1)
  - Directed DFS
  - Strong connectivity
  - Connected Components [Cormen]
- Transitive closure (§6.4.2)
  - The Floyd-Warshall Algorithm
- Directed Acyclic Graphs (DAG's) [Cormen]
  - Topological Sorting

Digraphs

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



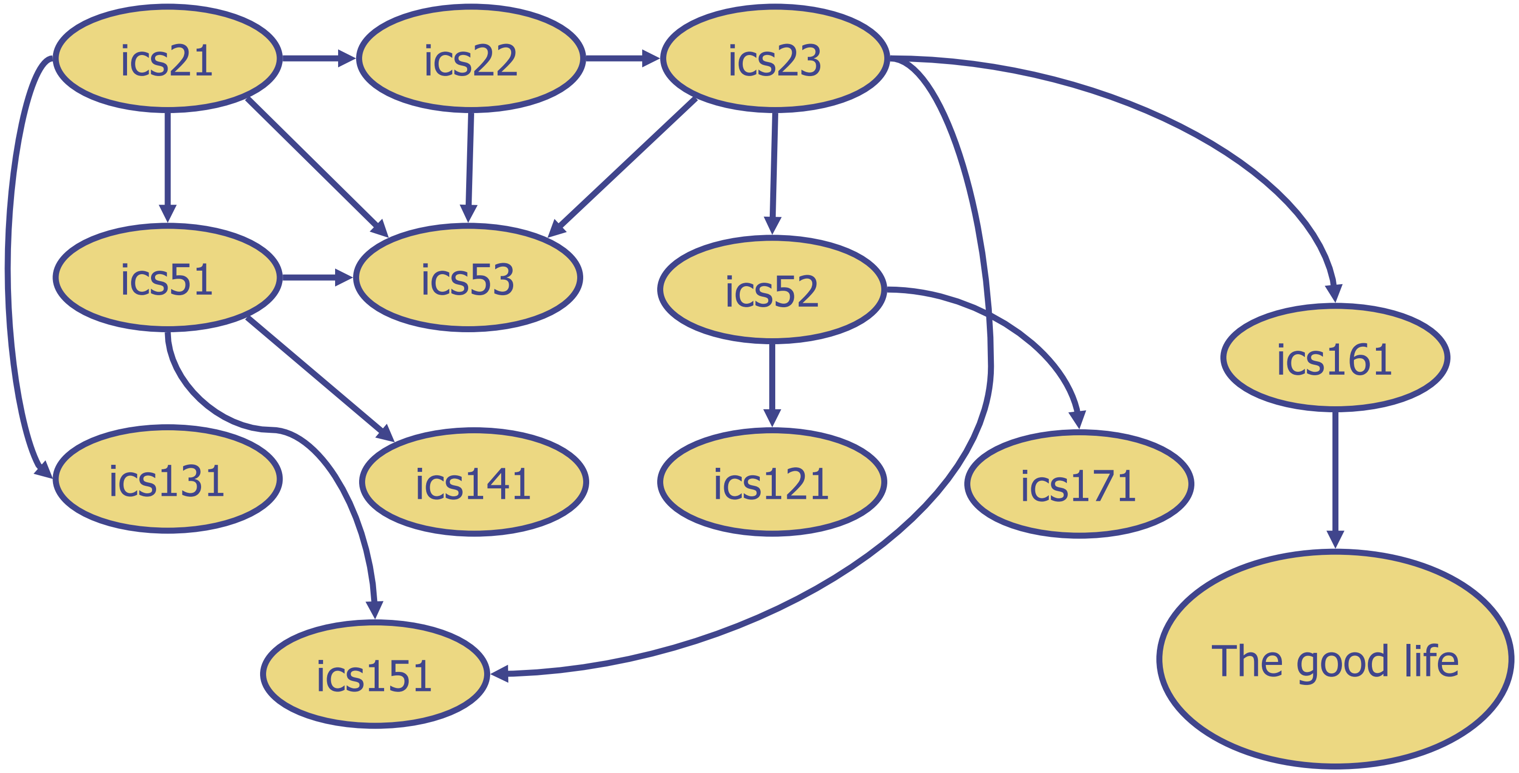
Digraph Properties



- Each edge goes in one direction:
  - edge (a,b) goes from a to b, but not b to a.
- If G is simple, $m \leq n(n-1)$.
- 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.

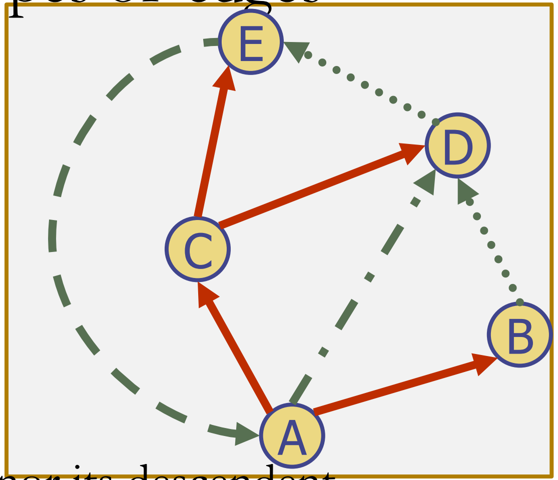
Digraph Application

- 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
  - Tree/discovery edges
  - back edges
    - Connect a vertex to an ancestor in the DFS tree
  - forward edges
    - Connect a vertex to a descendent in the DFS tree
  - cross edges
    - Connect a vertex to a vertex that is neither its ancestor nor its descendent



- A directed DFS starting at a vertex ***s*** determines the vertices reachable from ***s***

Reachability

- Theorem 6.20:

  Let $\vec{G}$ be a digraph. Depth-first search on $\vec{G}$ starting at a vertex s visits all the vertices of $\vec{G}$ that are reachable from s.

  Also, the DFS tree contains directed paths from s to every vertex reachable from s.

Reachability

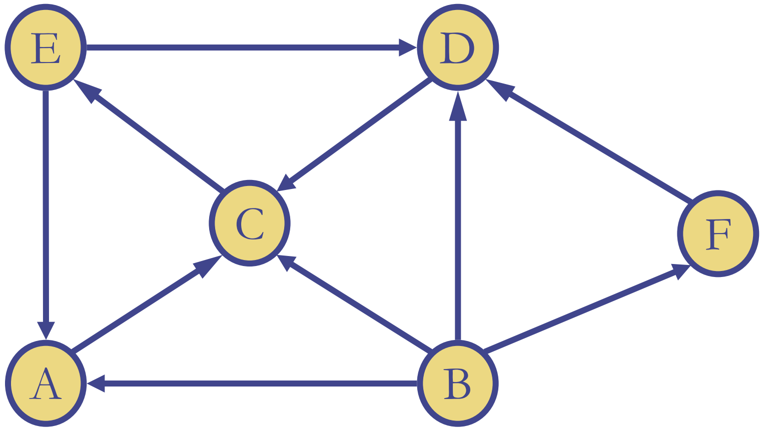
- Theorem 6.21:

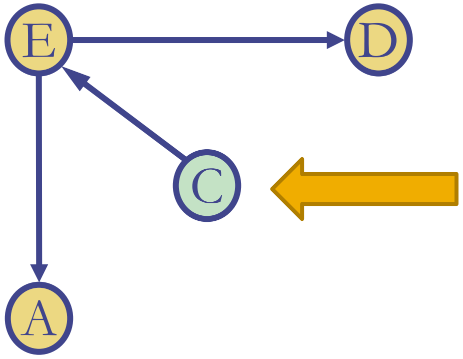
  Let be a $\vec{G}$ digraph with n vertices and m edges. The following problems can be solved by an algorithm that traverses n times using DFS, runs in $O(n(n+m))$ time, and uses $O(n)$ auxiliary space:

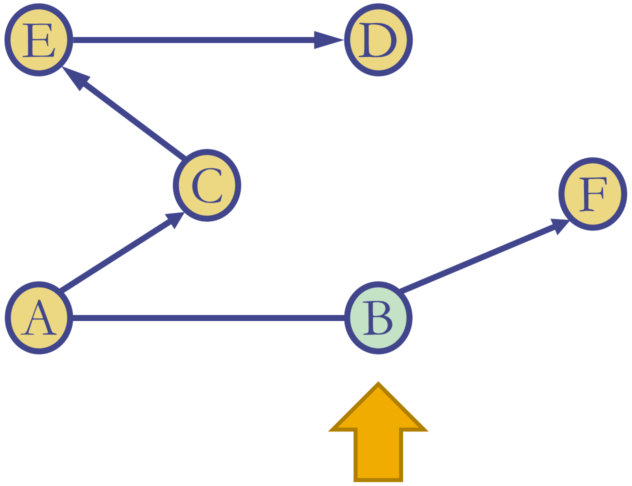
  - Computing, for each vertex v of $\vec{G}$, the subgraph reachable from v
  - Testing whether $\vec{G}$ is strongly connected
  - Computing the transitive closure $\vec{G}^*$ of $\vec{G}$.

Reachability

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

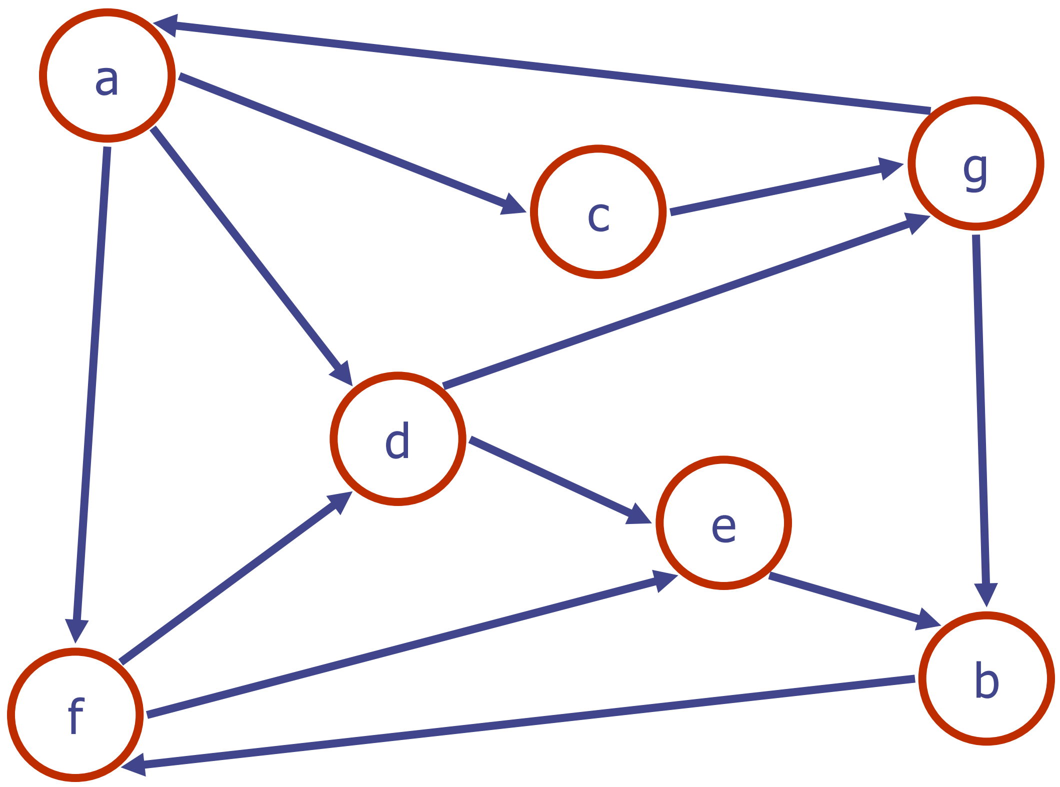




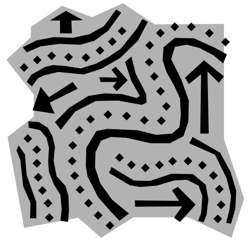


Strong Connectivity

- Each vertex can reach all other vertices



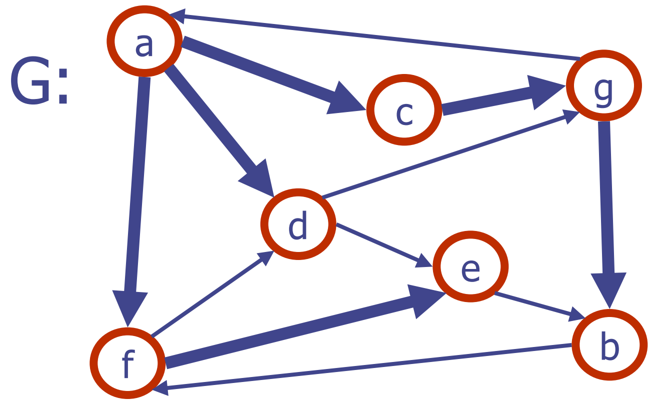
Strong Connectivity Algorithm

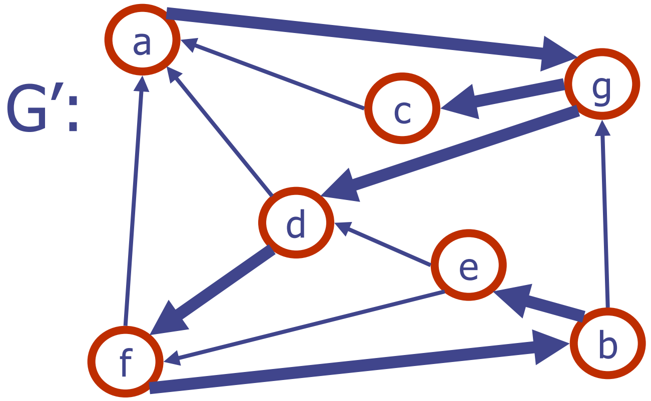


- Pick a vertex v in G.
- Perform a DFS from v in G.
  - If there's a w not visited, print "no".
- Let G' be G with edges reversed.
- Perform a DFS from v in G'.
  - If there's a w not visited, print "no".
  - else, print "yes".

If every vertex of *G'* is visited by this second DFS, then the graph is strongly connected, for each of the vertices visited in this DFS can reach s.

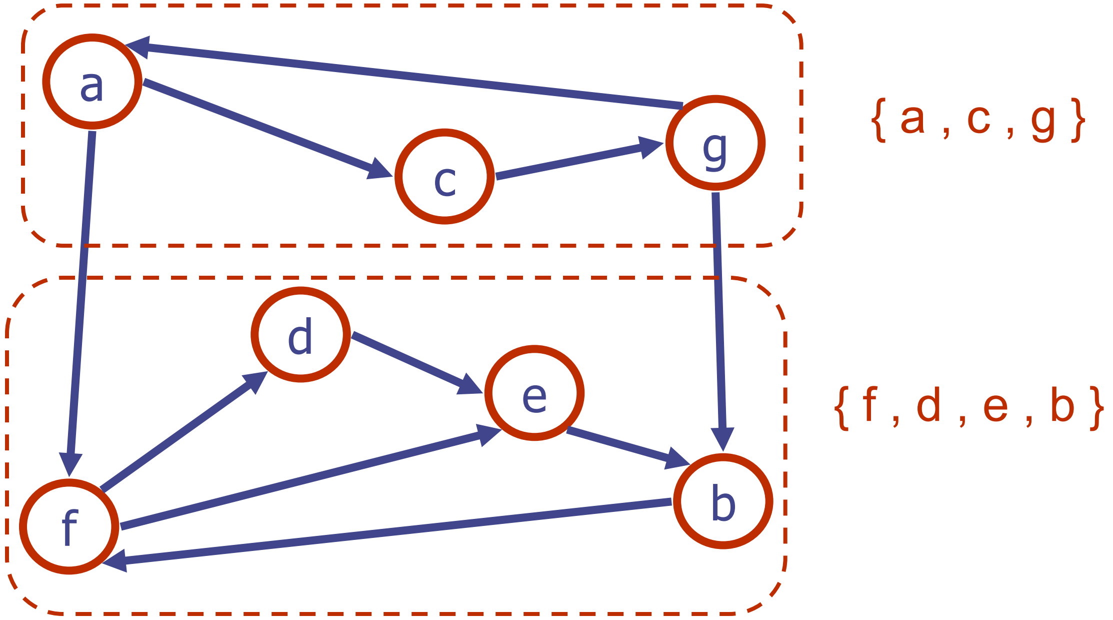
- Running time: $O(n + m)$.



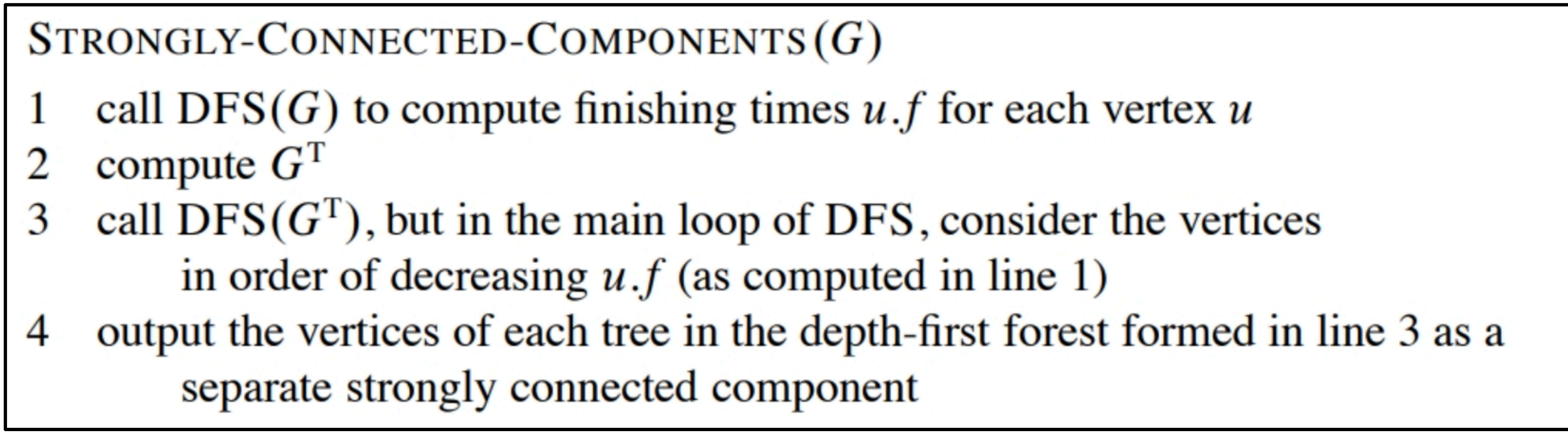


Strongly Connected Components [22.5]

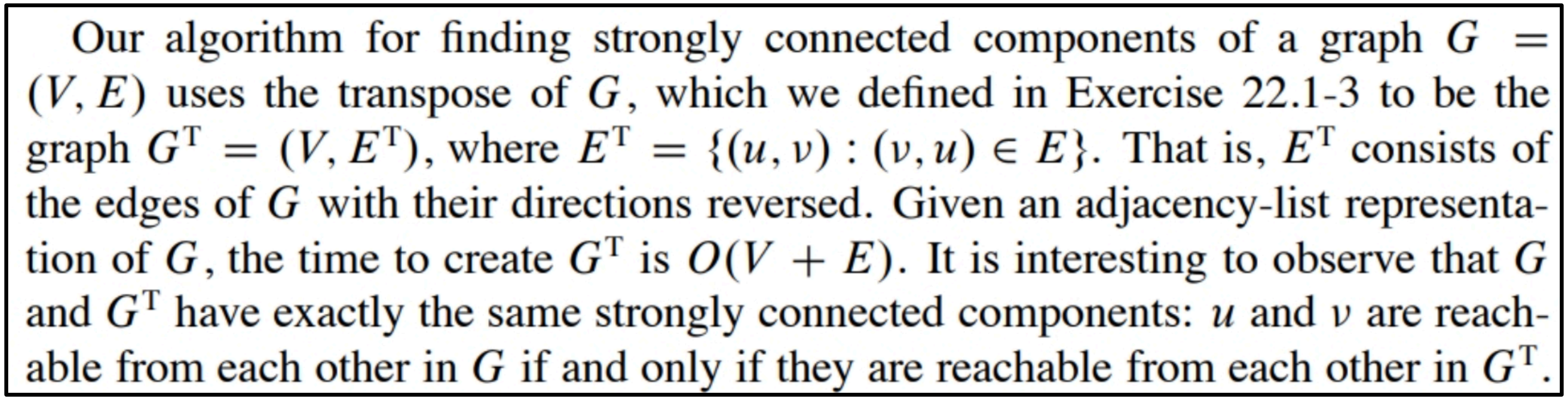
- Maximal subgraphs such that each vertex can reach all other vertices in the subgraph
- Can also be done in $O(n+m)$ time using DFS, but is more complicated (similar to biconnectivity).



Strongly connected components



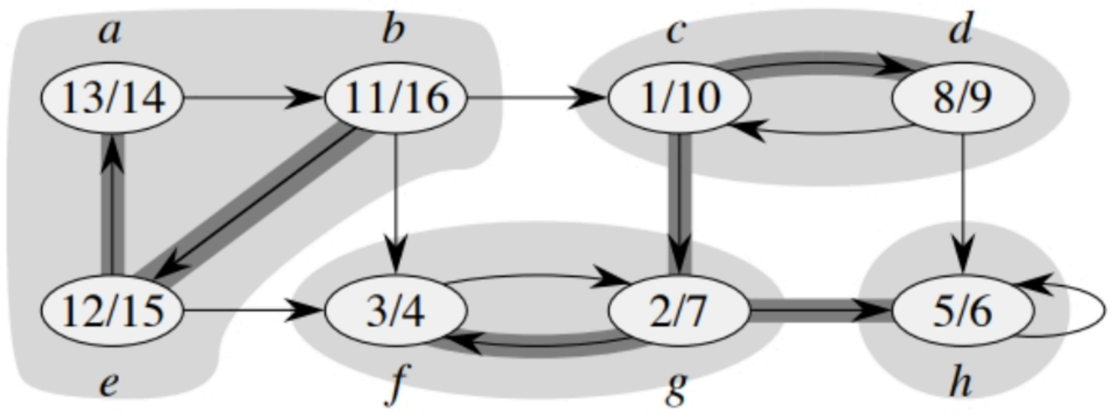
```
STRONGLY-CONNECTED-COMPONENTS(G)
1  call DFS(G) to compute finishing times u.f for each vertex u
2  compute G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
       in order of decreasing u.f (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
       separate strongly connected component
```



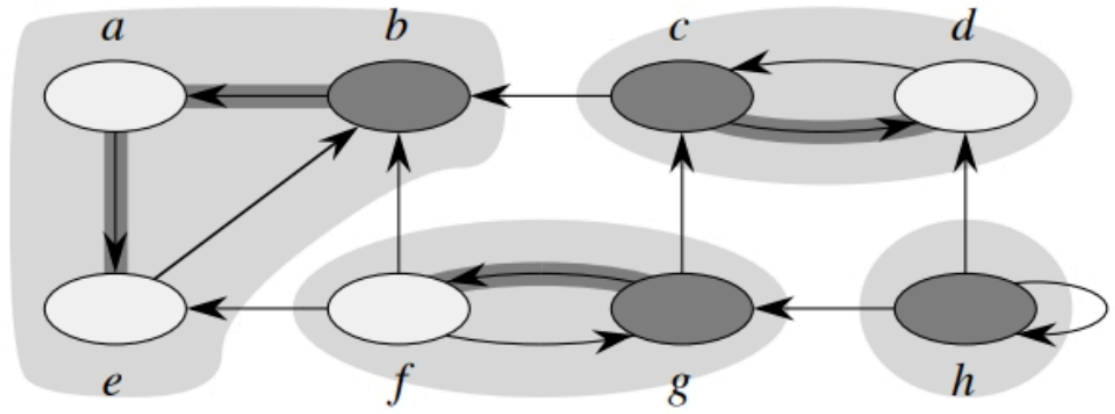
Our algorithm for finding strongly connected components of a graph $G = (V, E)$ uses the transpose of G, which we defined in Exercise 22.1-3 to be the graph $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, where $E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$. That is, E^{T} consists of the edges of G with their directions reversed. Given an adjacency-list representation of G, the time to create G^{T} is $O(V + E)$. It is interesting to observe that G and G^{T} have exactly the same strongly connected components: u and v are reachable from each other in G if and only if they are reachable from each other in G^{T}.

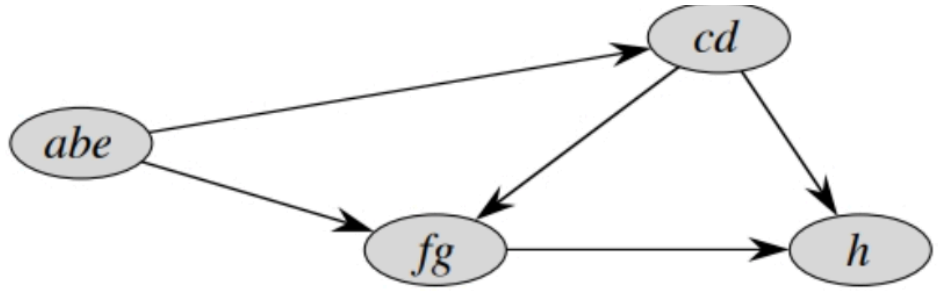
Strongly connected components

(a)



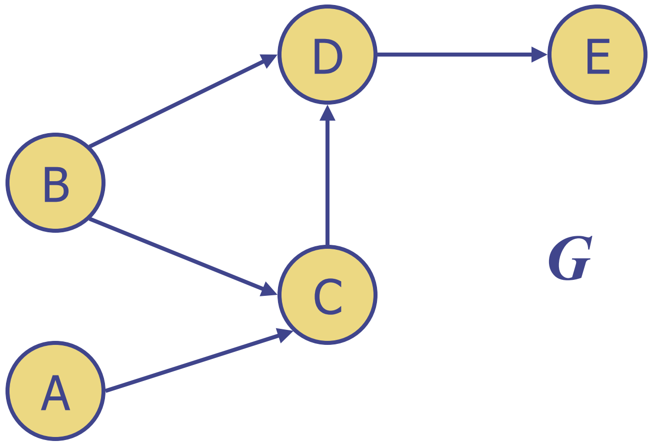
(b)

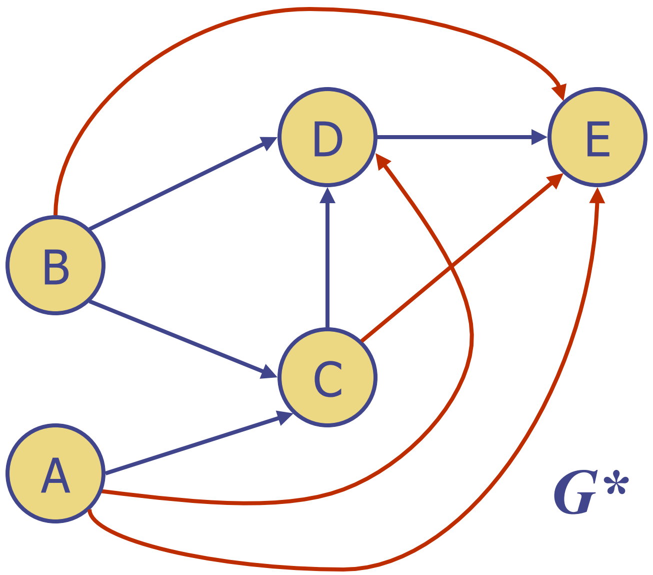




Transitive Closure

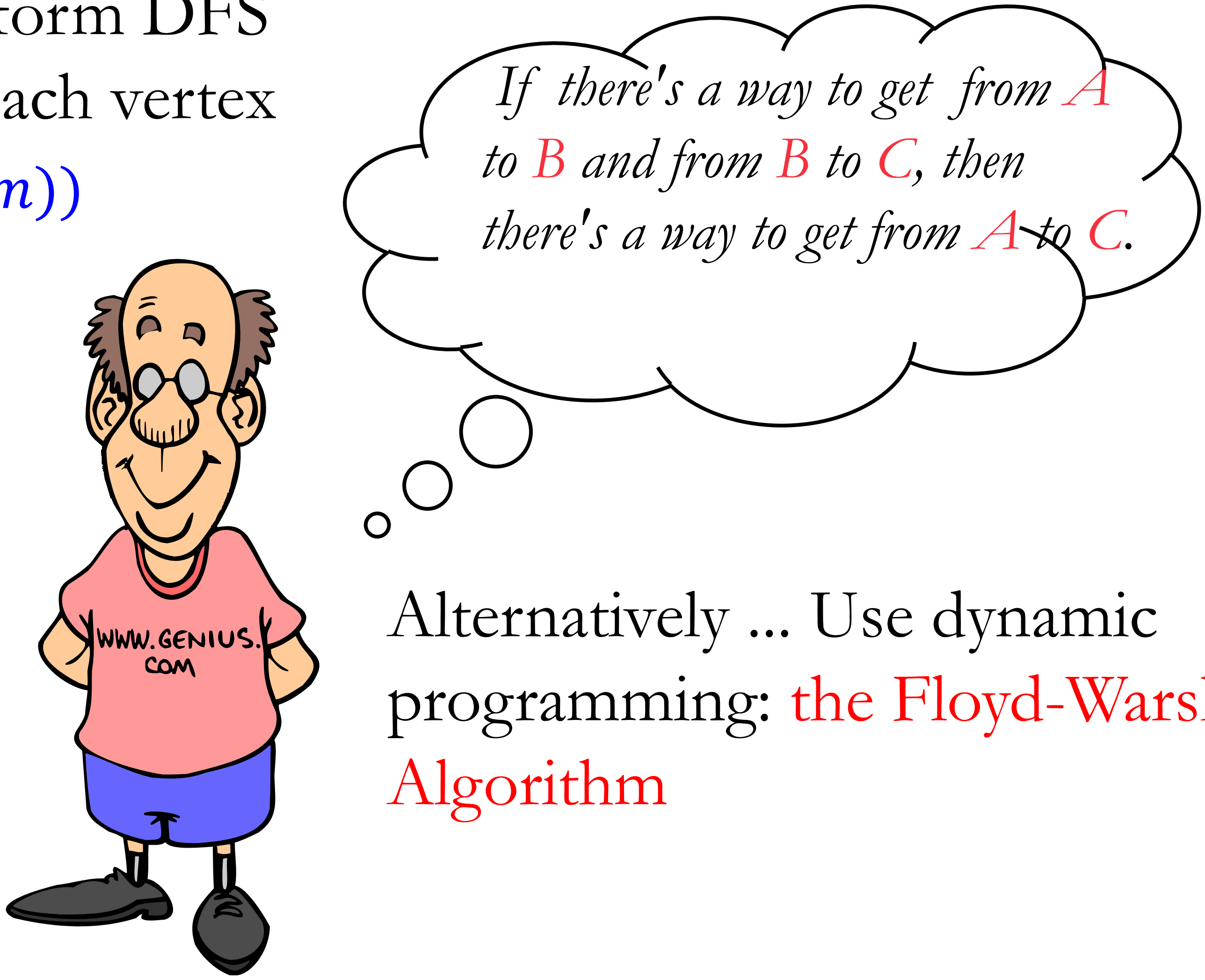
- Given a digraph ***G***, the transitive closure of ***G*** is the digraph ***G**** such that
  - ***G**** has the same vertices as ***G***
  - if ***G*** has a directed path from ***u*** to ***v*** ($u \neq v$), ***G**** has a directed edge from ***u*** to ***v***
- The transitive closure provides reachability information about a digraph





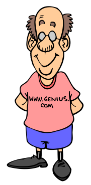
Computing the Transitive Closure

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

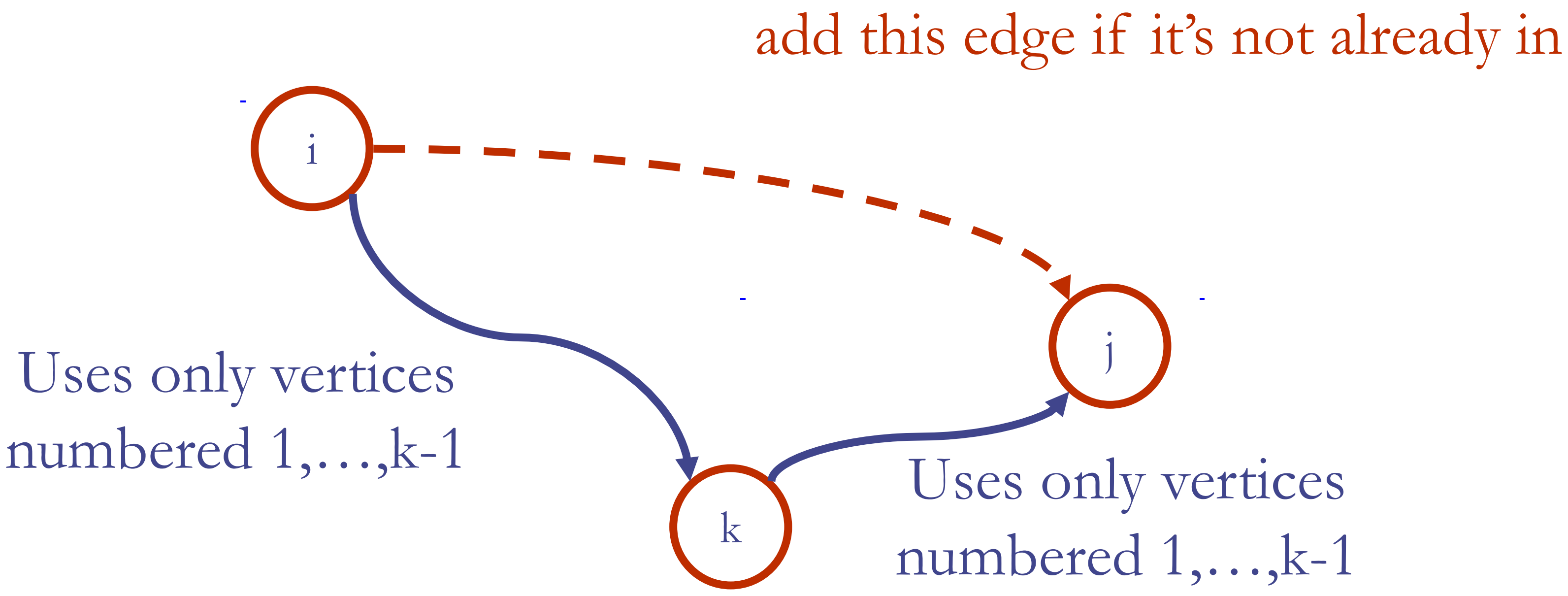

If there's a way to get from A to B and from B to C, then there's a way to get from A to C.
WWW.GENIUS.COM

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

Floyd-Warshall Transitive Closure



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


add this edge if it's not already in
i
j
k
Uses only vertices numbered 1,…,k-1
Uses only vertices numbered 1,…,k-1

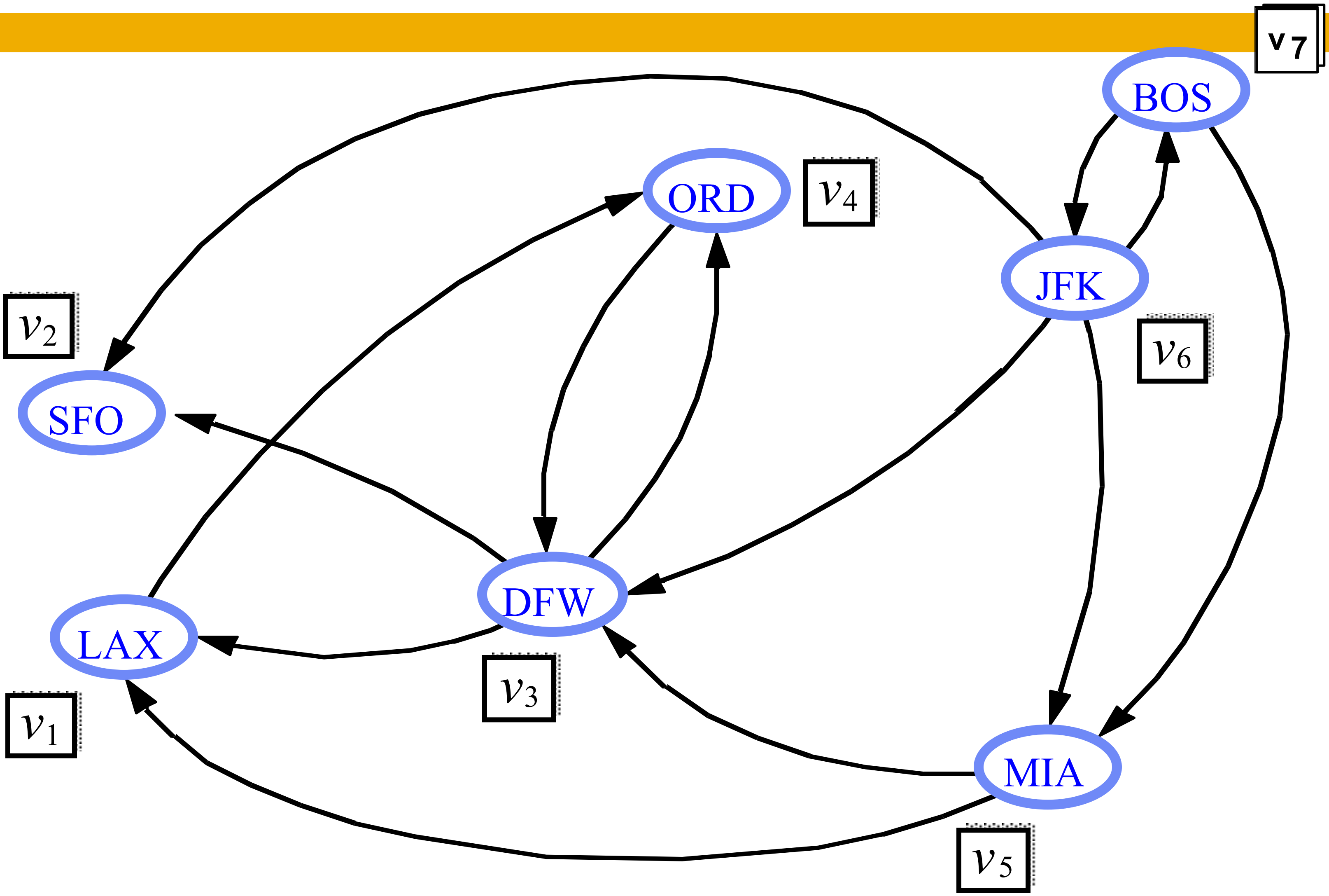
Floyd-Warshall's Algorithm

- Floyd-Warshall's algorithm numbers the vertices of G as $\boldsymbol{v}_1, \dots, \boldsymbol{v}_{\boldsymbol{n}}$ and computes a series of digraphs $\boldsymbol{G}_0, \dots, \boldsymbol{G}_{\boldsymbol{n}}$
  - $\boldsymbol{G}_0 = \boldsymbol{G}$
  - $\boldsymbol{G}_{\boldsymbol{k}}$ has a directed edge $(\boldsymbol{v}_{\boldsymbol{i}}, \boldsymbol{v}_{\boldsymbol{j}})$ if $\boldsymbol{G}$ has a directed path from $\boldsymbol{v}_{\boldsymbol{i}}$ to $\boldsymbol{v}_{\boldsymbol{j}}$ with intermediate vertices in the set $\{\boldsymbol{v}_1, \dots, \boldsymbol{v}_{\boldsymbol{k}}\}$
- We have that $\boldsymbol{G}_{\boldsymbol{n}} = G^*$
- 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)

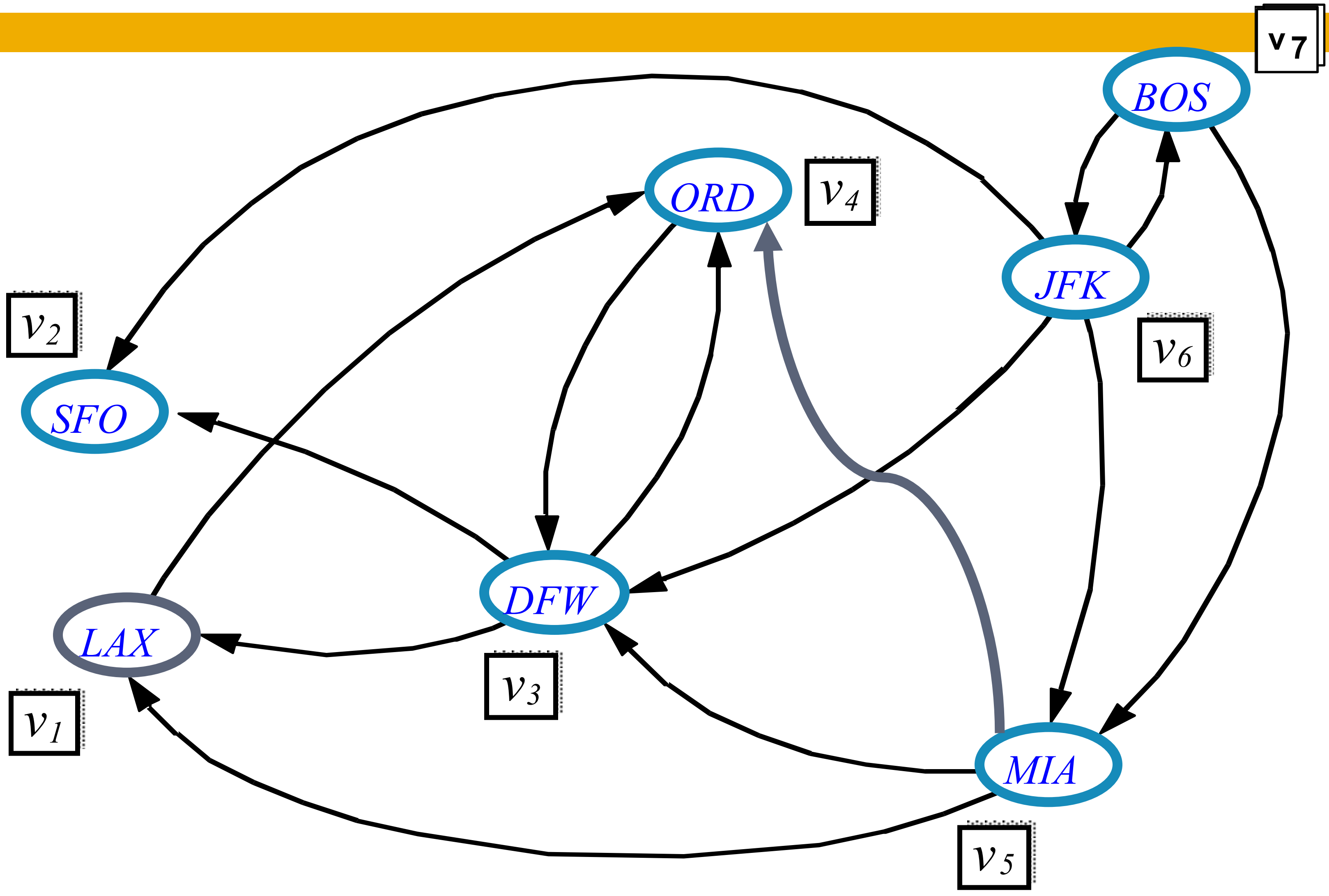
Floyd-Warshall's Algorithm

```
Algorithm FloydWarshall(G)
  Input digraph G
  Output transitive closure G* of G
  let $v_1, \dots, v_n$ be an arbitrary numbering of the vertices of G
  $G_0 \leftarrow G$
  for k ← 1 to n do
    $G_k \leftarrow G_{k-1}$
    for i ← 1 to n, $(i \neq k)$ do
      for j ← 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 !$G_k$.areAdjacent($v_i$, $v_j$)
              $G_k$.insertDirectedEdge($v_i, v_j, k$)
    return $G_n$
```

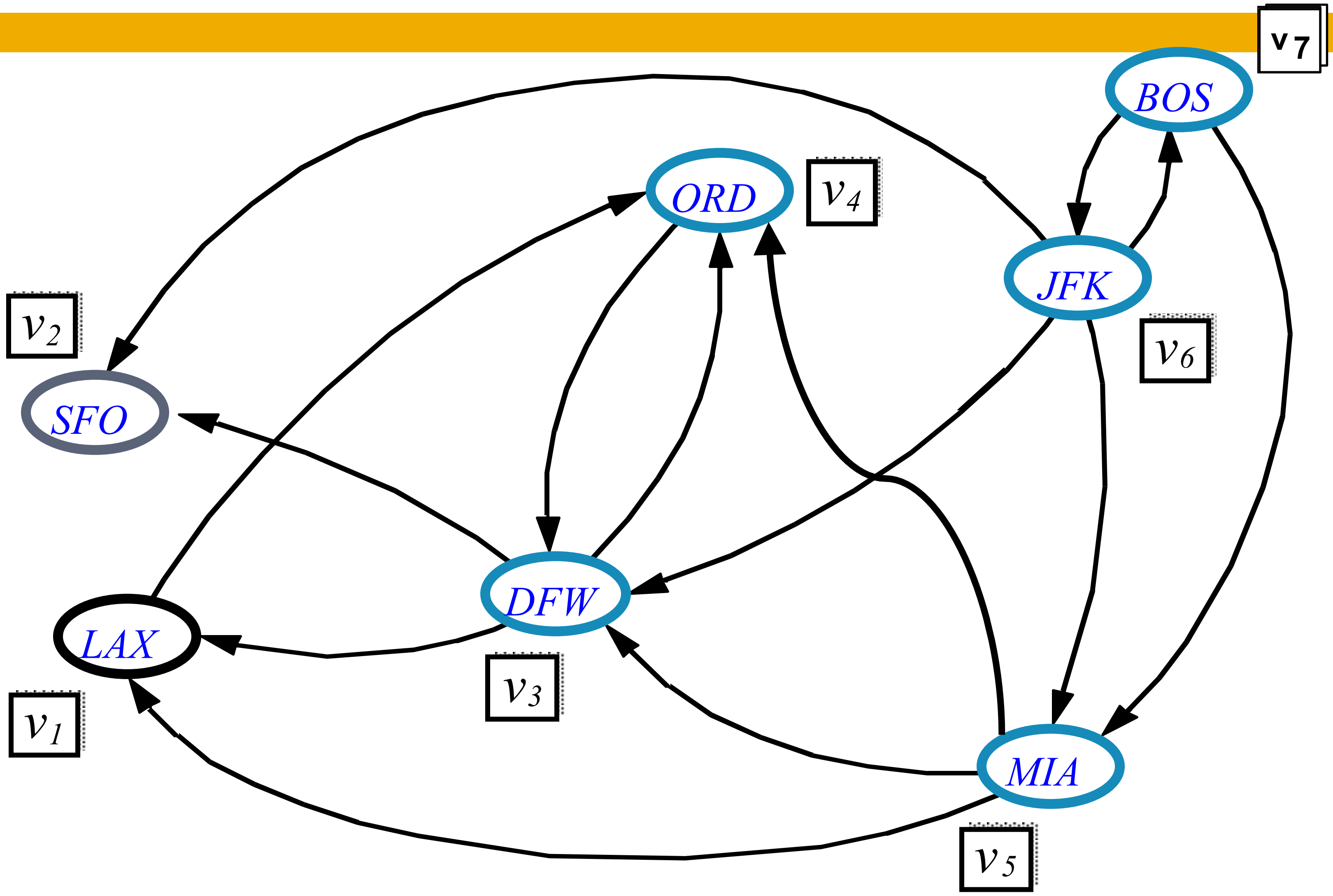
Floyd-Warshall Example



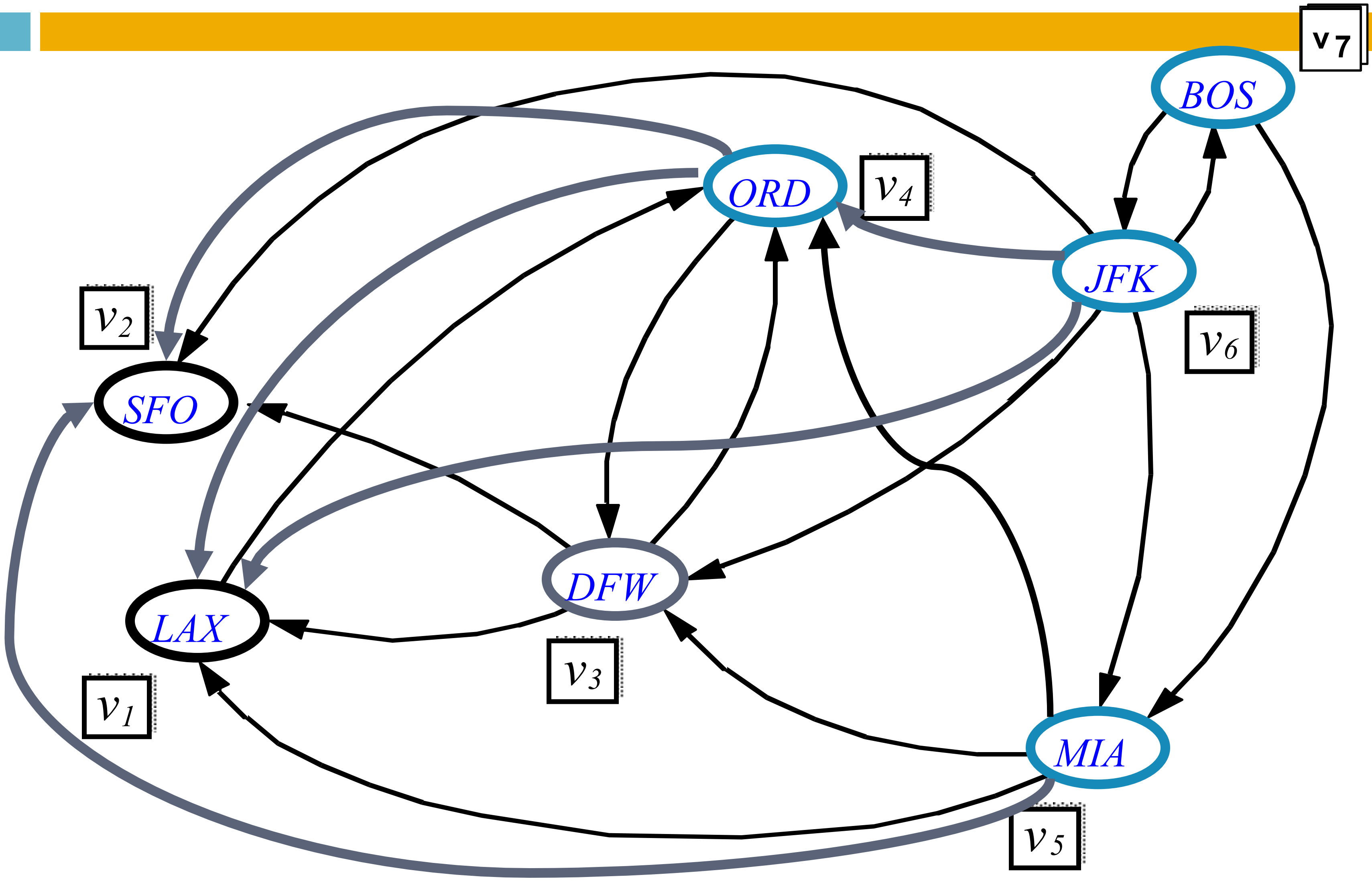
Floyd-Warshall, Iteration 1



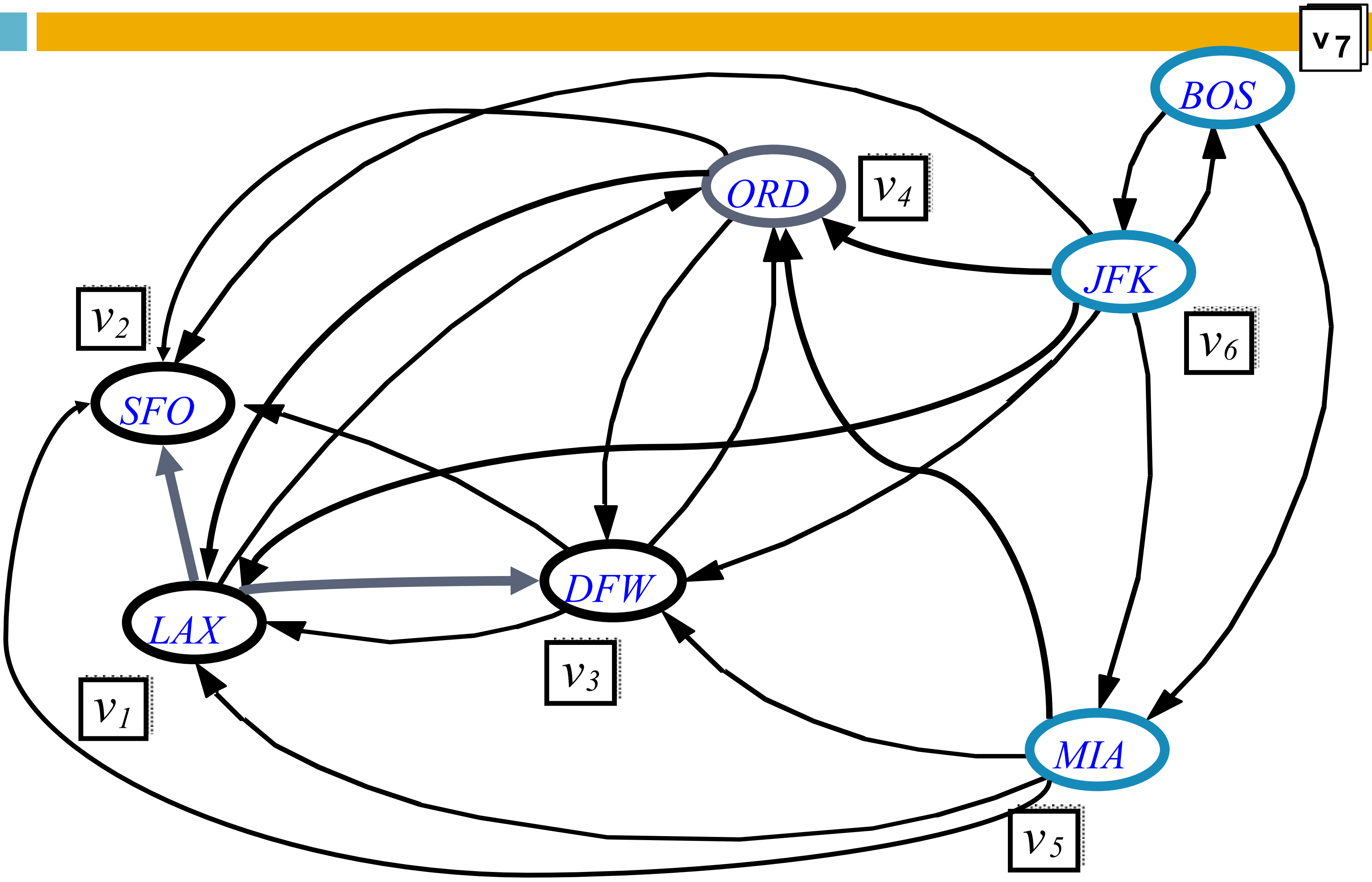
Floyd-Warshall, Iteration 2



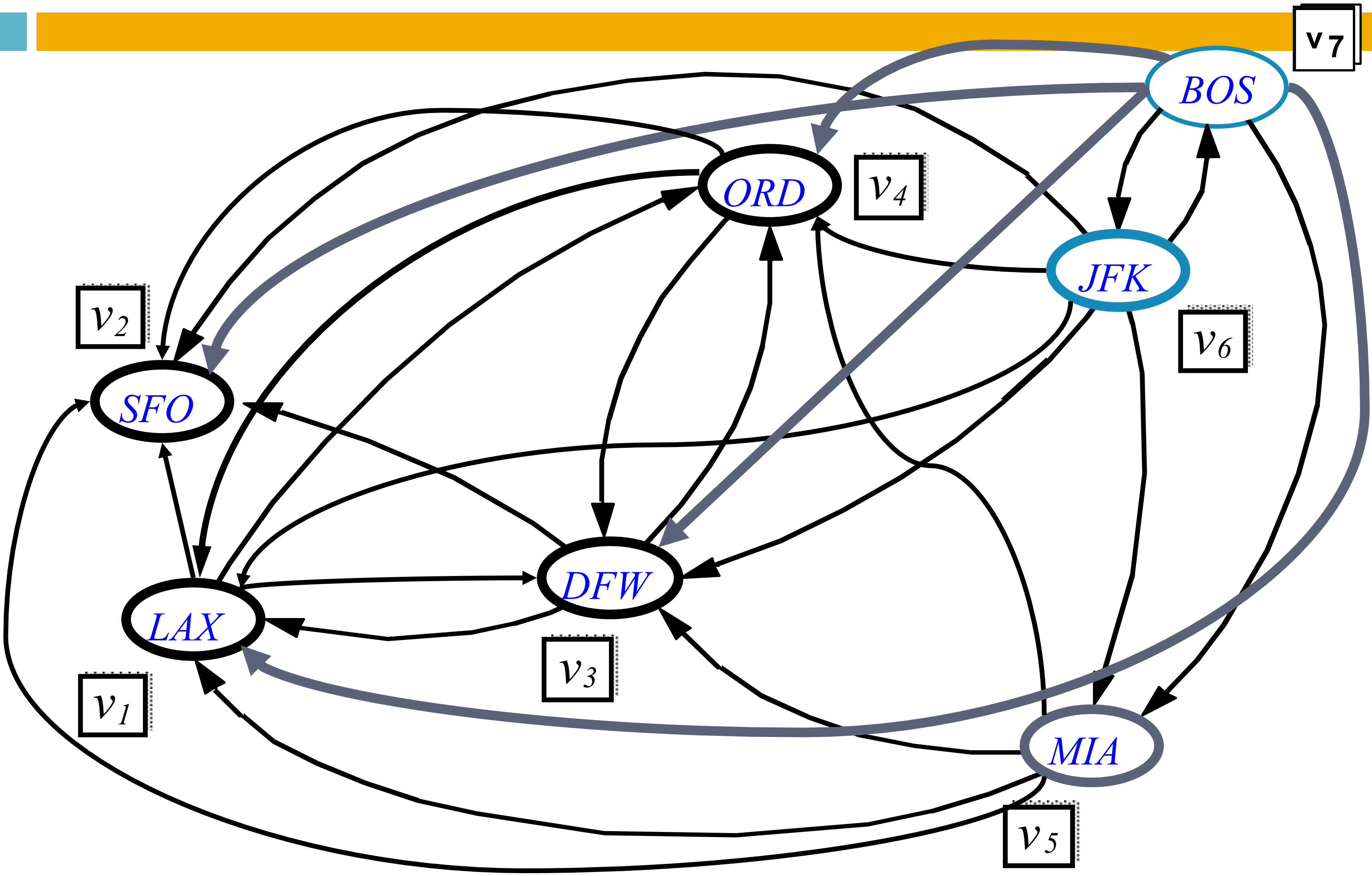
Floyd-Warshall, Iteration 3



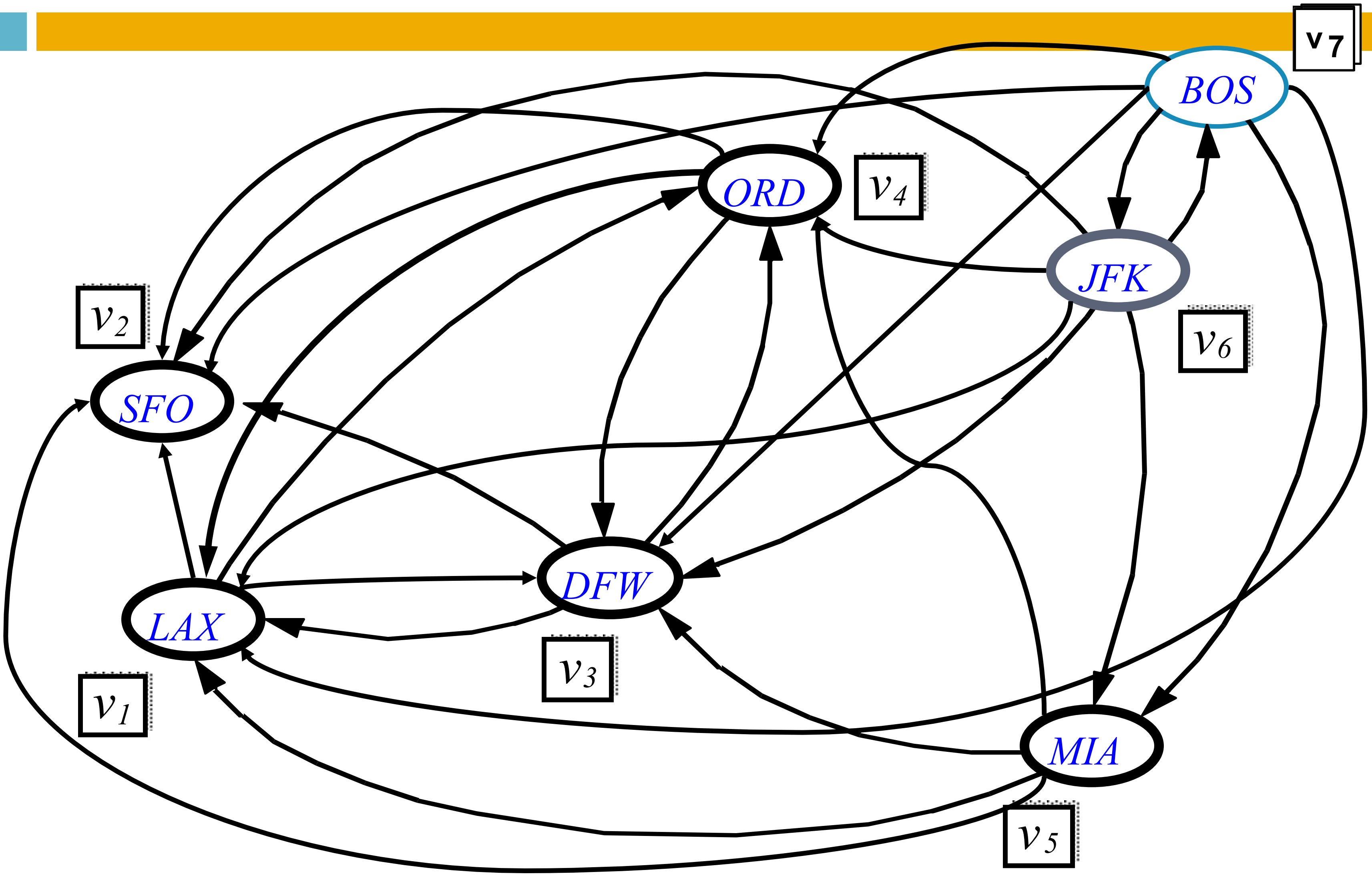
Floyd-Warshall, Iteration 4



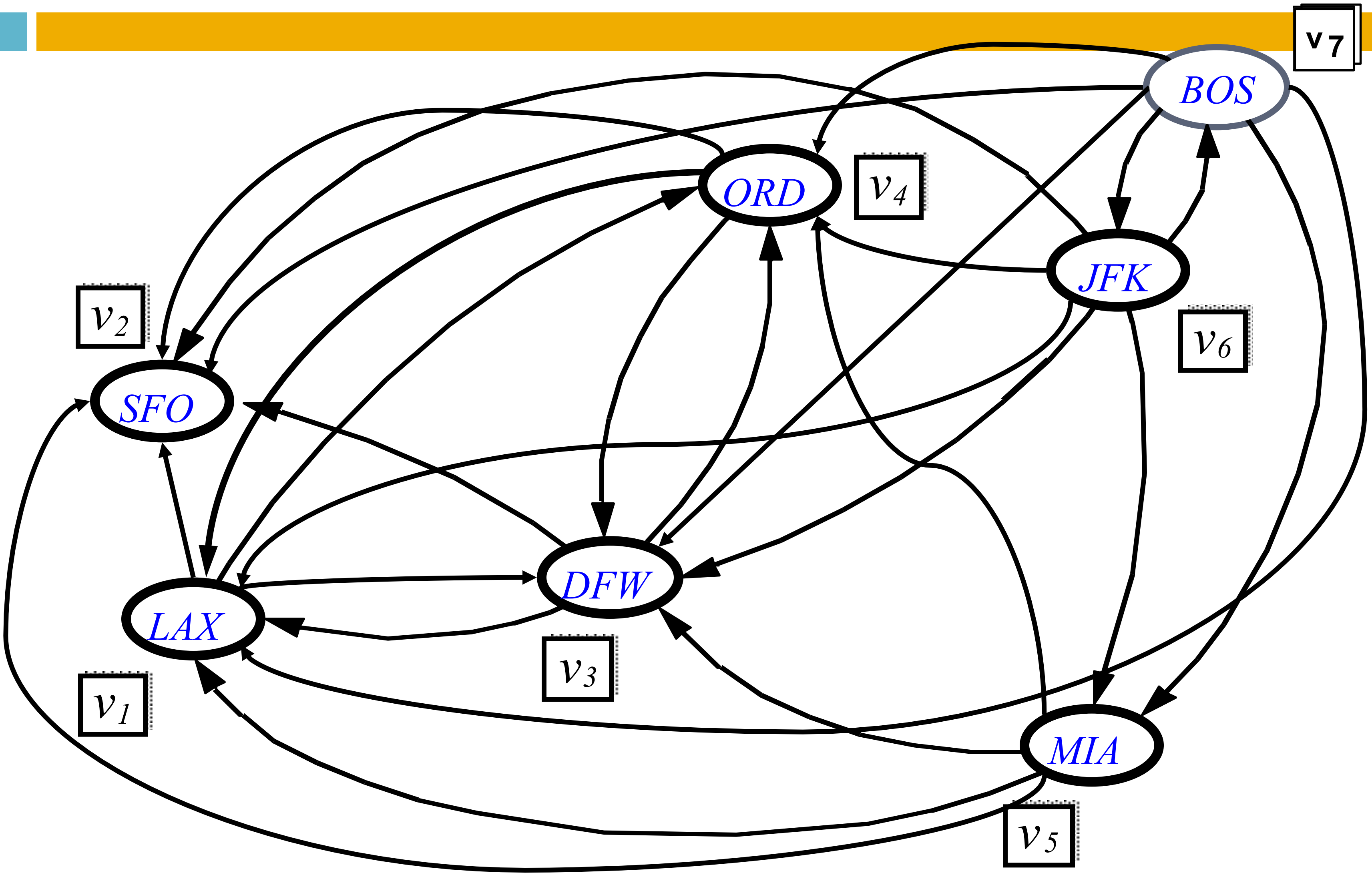
Floyd-Warshall, Iteration 5



Floyd-Warshall, Iteration 6



Floyd-Warshall, Conclusion



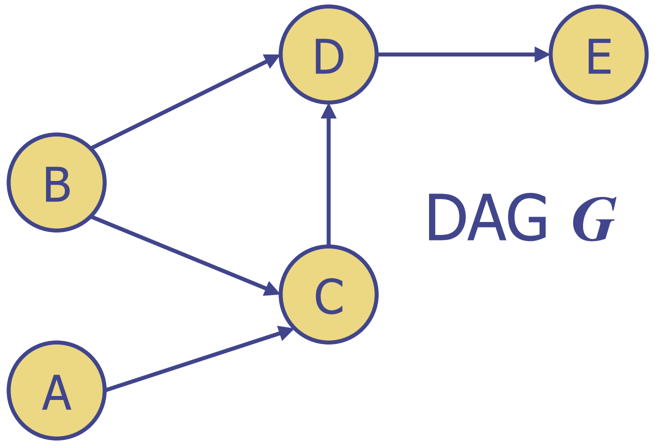
Topological Sort [22.4]

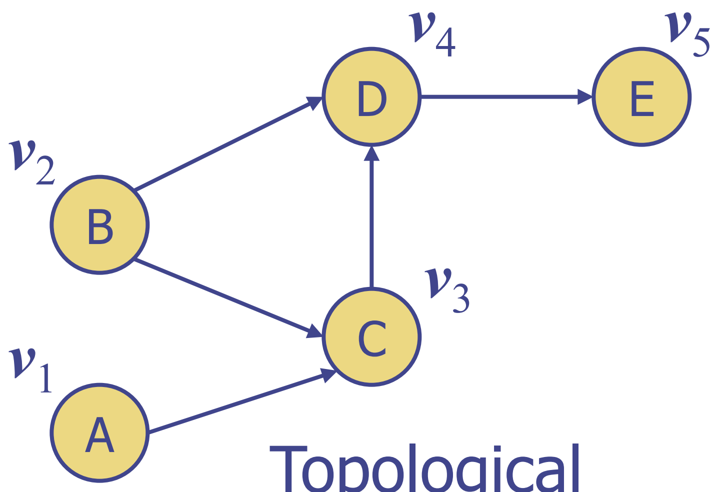
- A topological sort of a dag G = (V, E) is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.
- If the graph contains a cycle, then no linear ordering is possible.
- We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. T

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

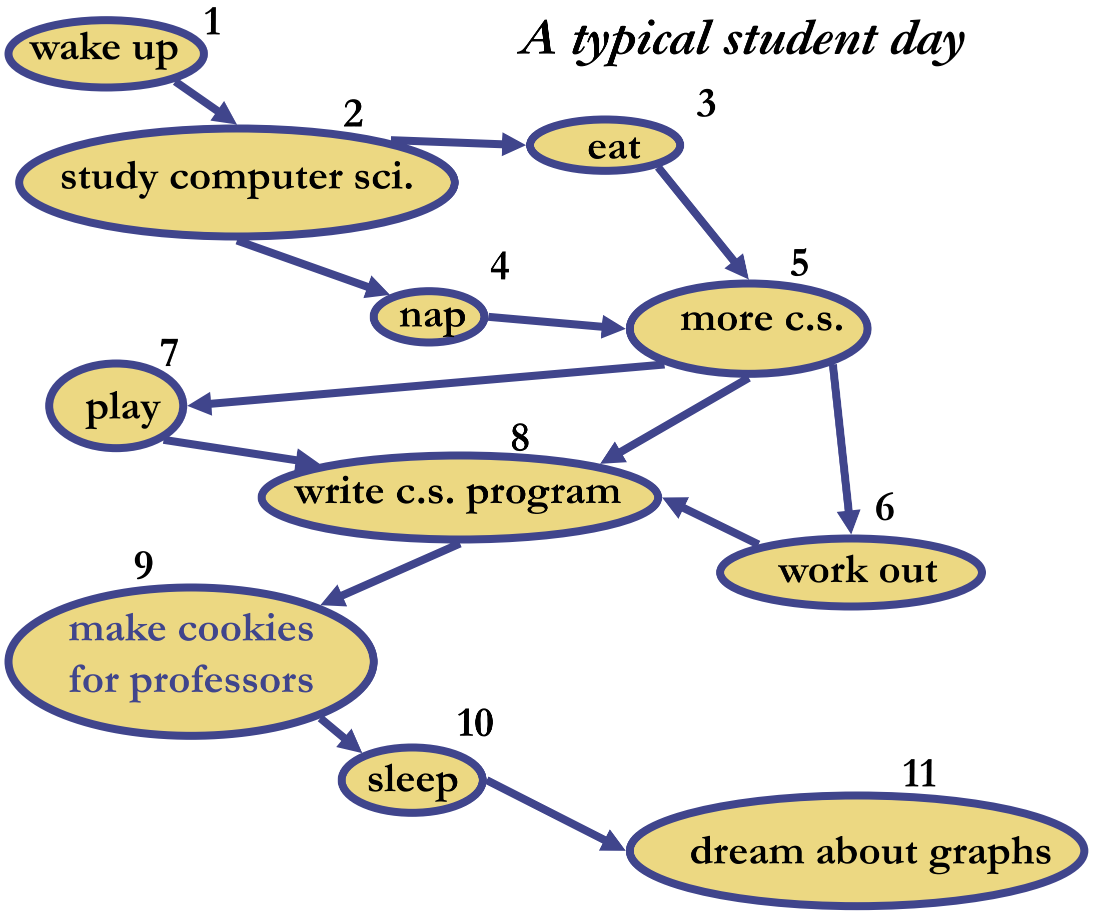




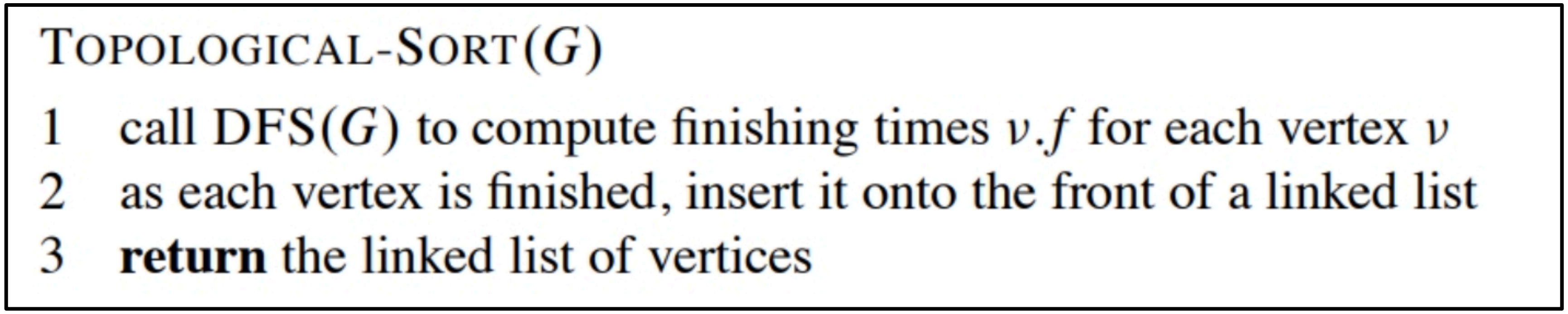
Topological ordering of G

Topological Sorting

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

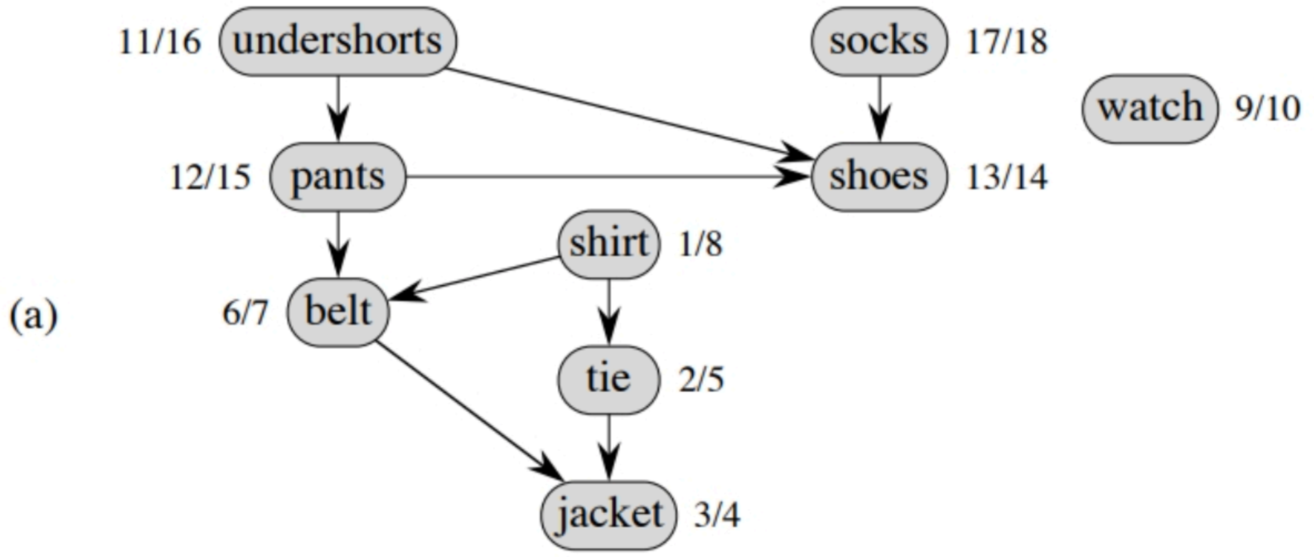


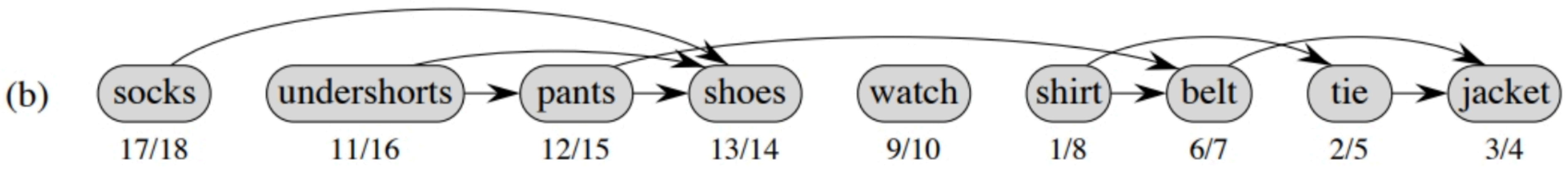
Topological Sorting



```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times v.f for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

Figure 22.7 **(a)** Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v. The discovery and finishing times from a depth-first search are shown next to each vertex. **(b)** The same graph shown topologically sorted, with its vertices arranged from left to right in order of decreasing finishing time. All directed edges go from left to right.





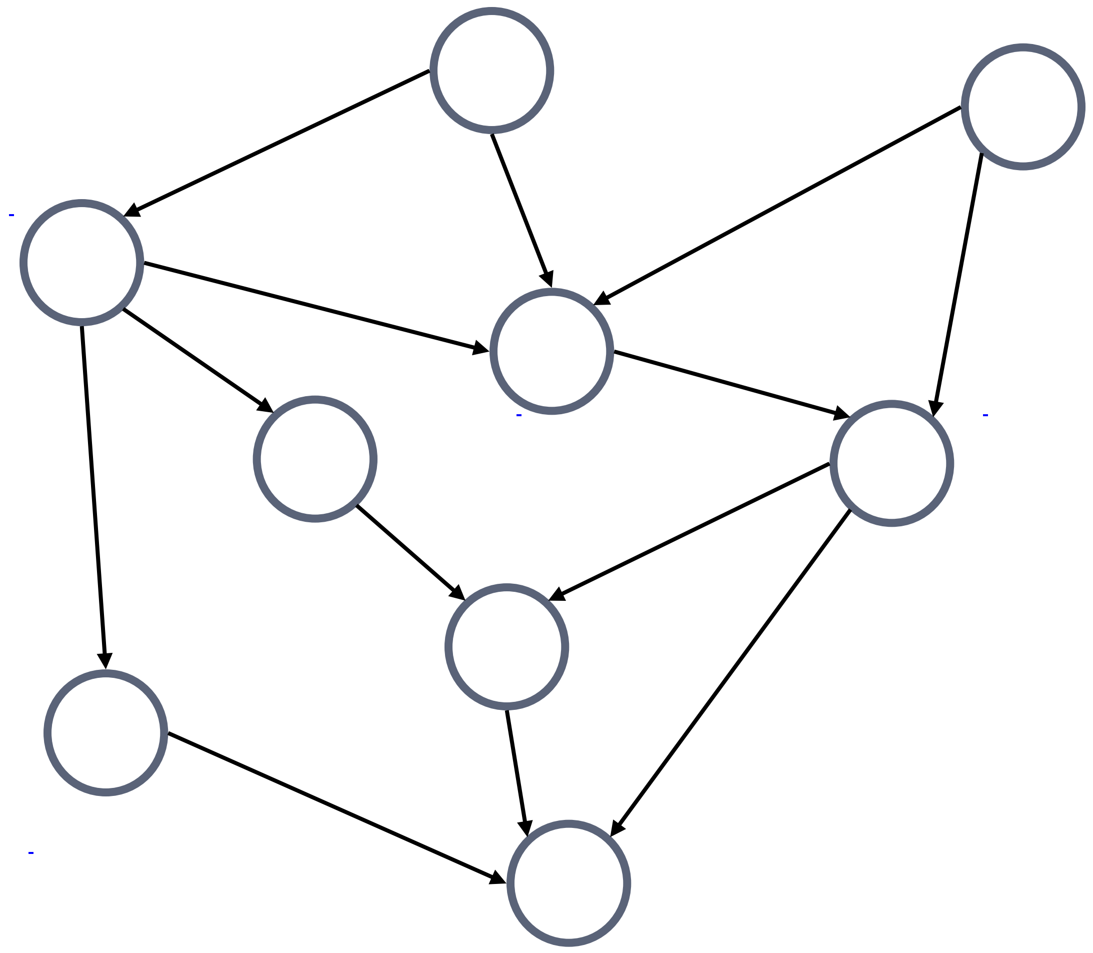
Topological Sorting

- Another way of thinking
- Note: This algorithm is different than the one in

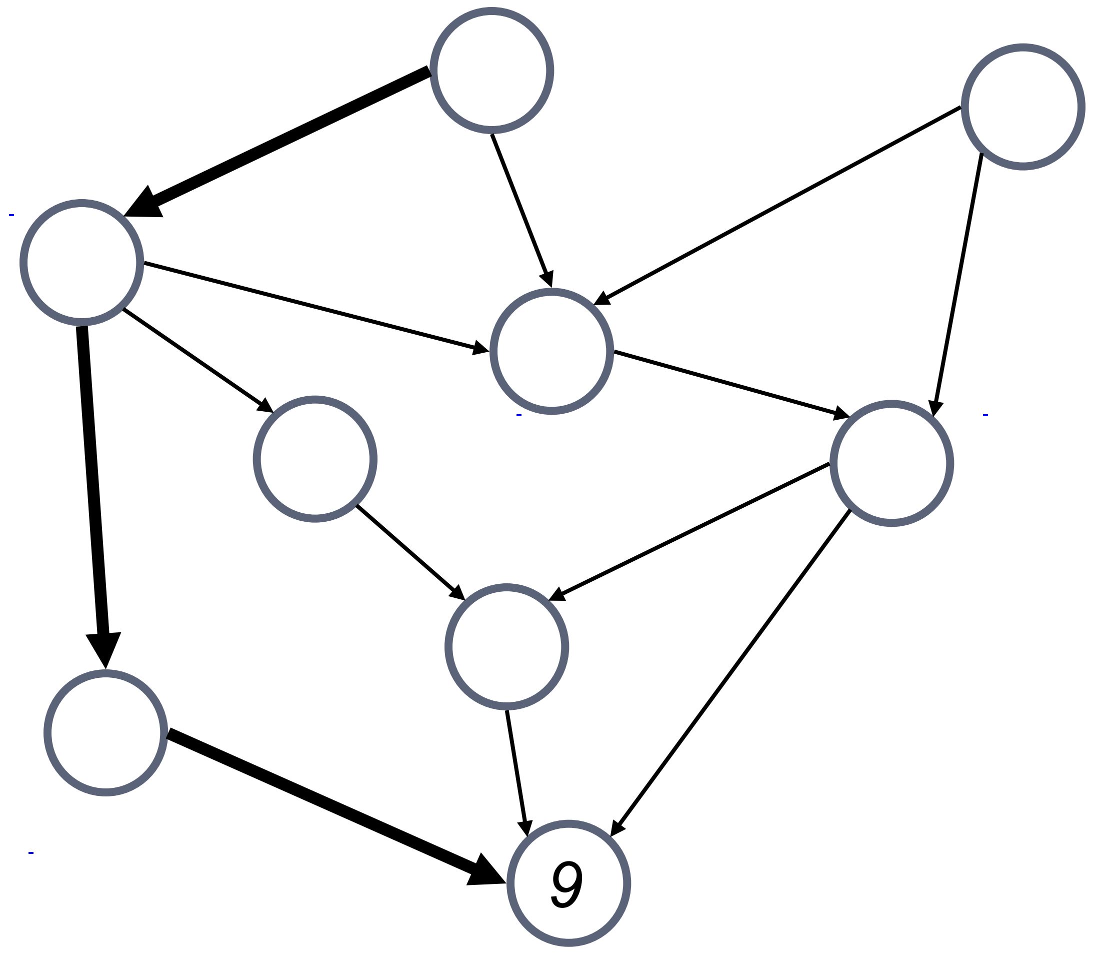
```
Method TopologicalSort(G)
    H ← G    // Temporary copy of G
    n ← G.numVertices()
    while H is not empty do
        Let v be a vertex with no outgoing edges
        Label v ← n
        n ← n - 1
        Remove v from H
```

- Running time: O(n + m). How…?

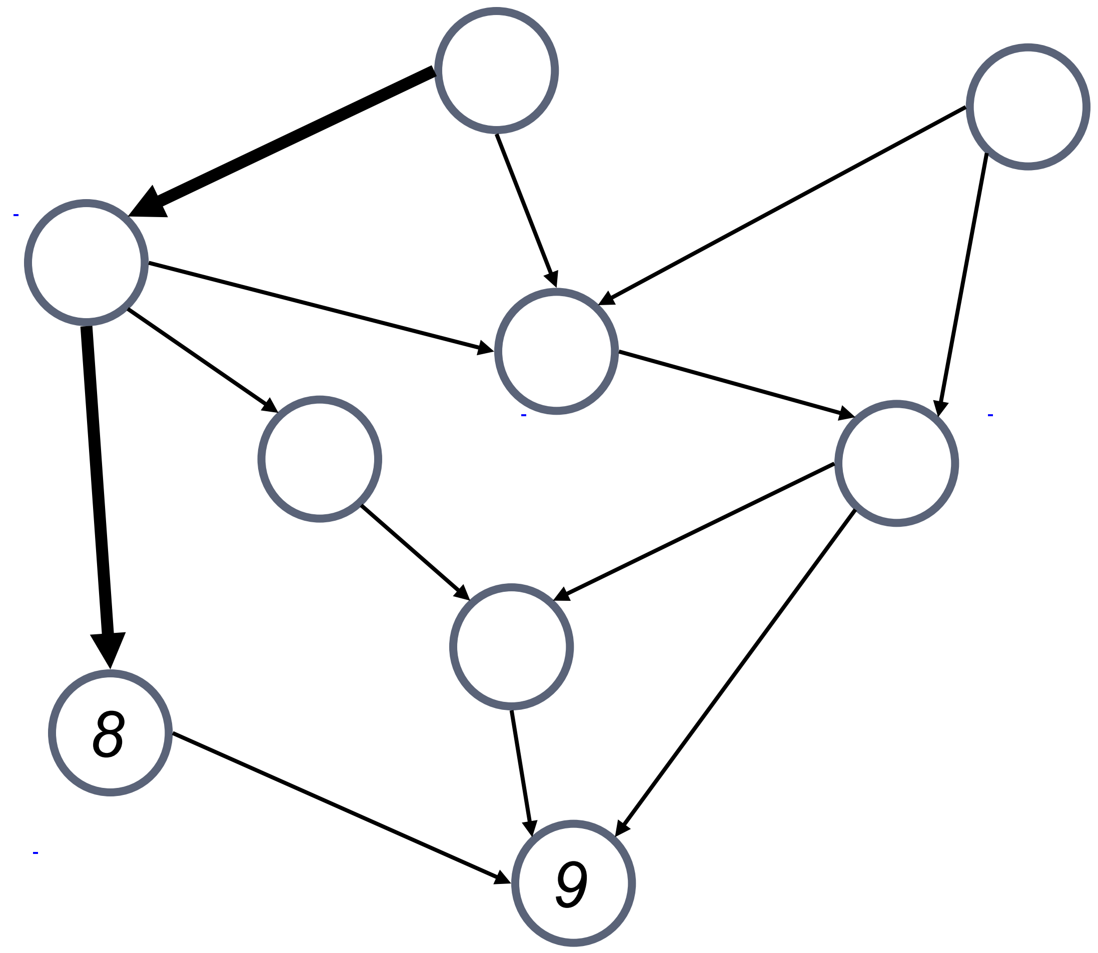
Topological Sorting Example



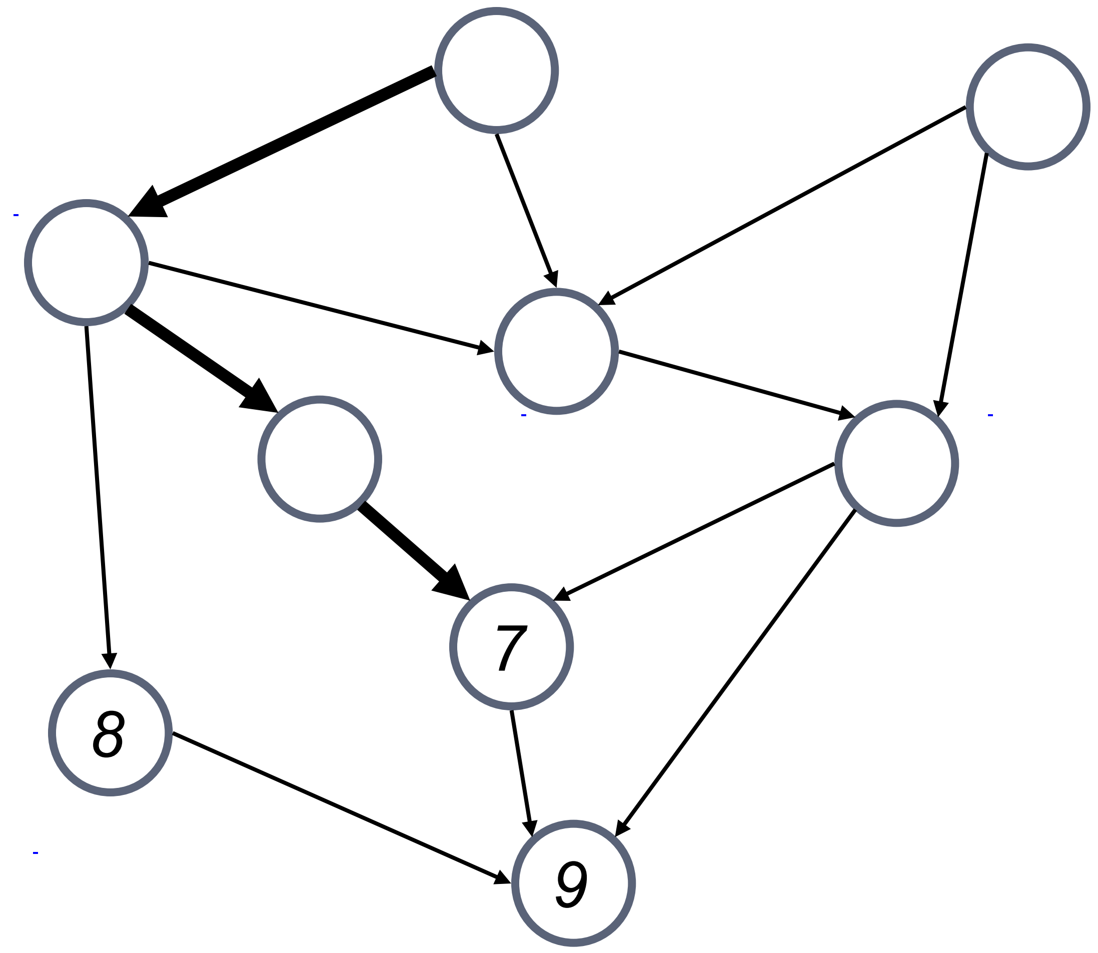
Topological Sorting Example



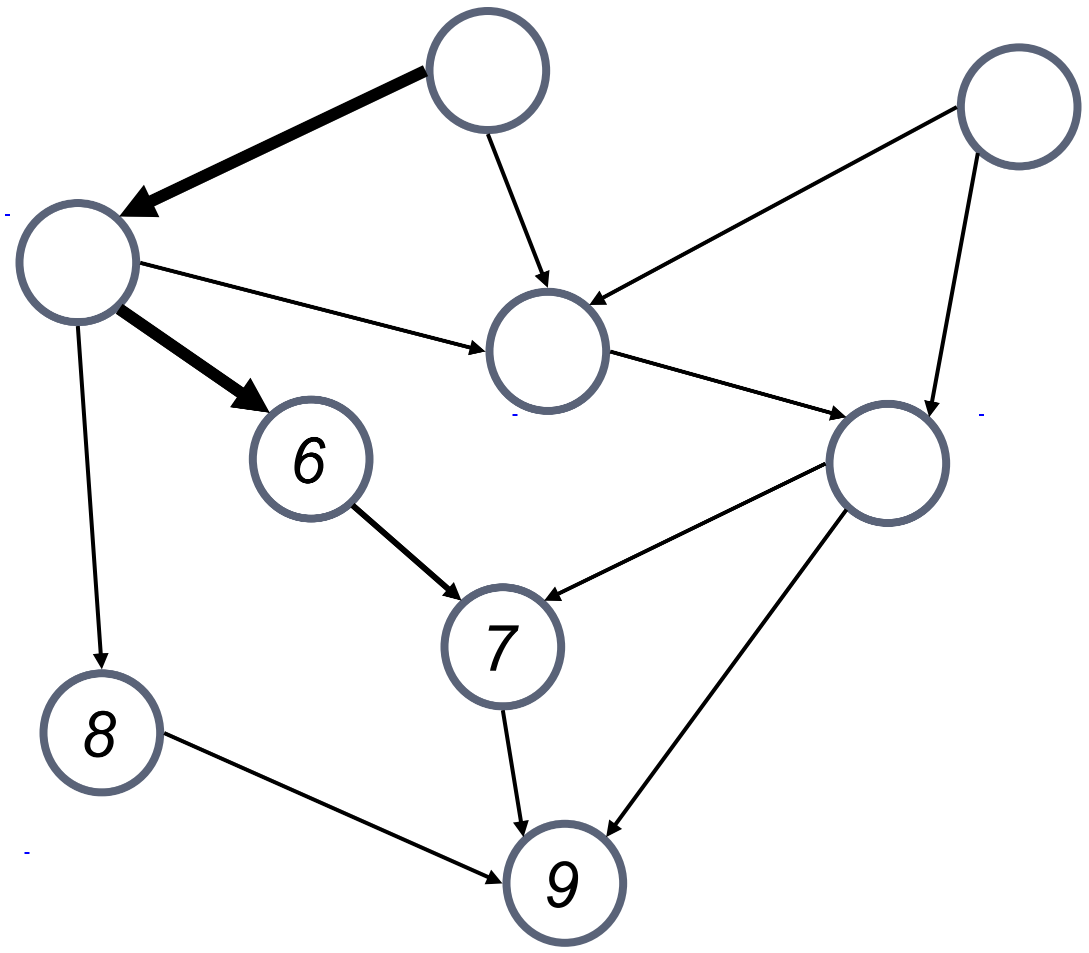
Topological Sorting Example



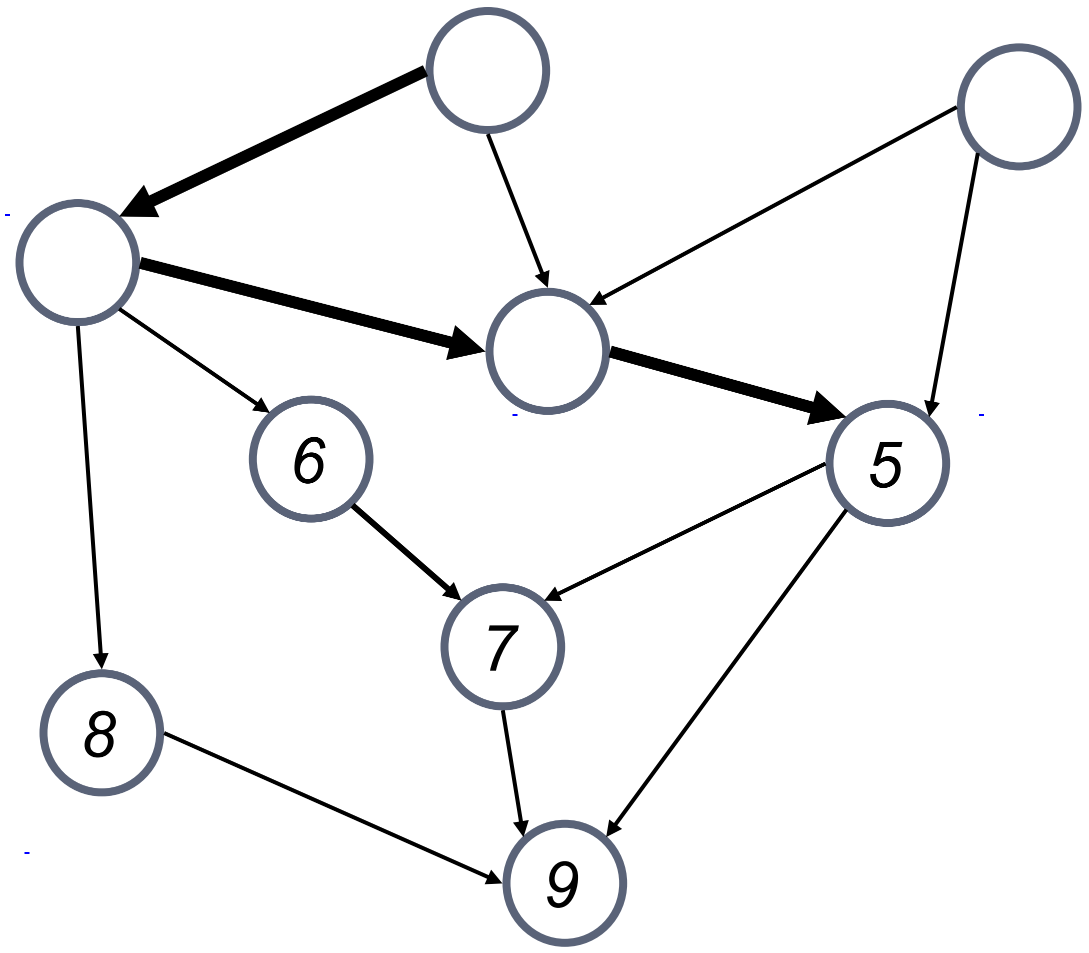
Topological Sorting Example



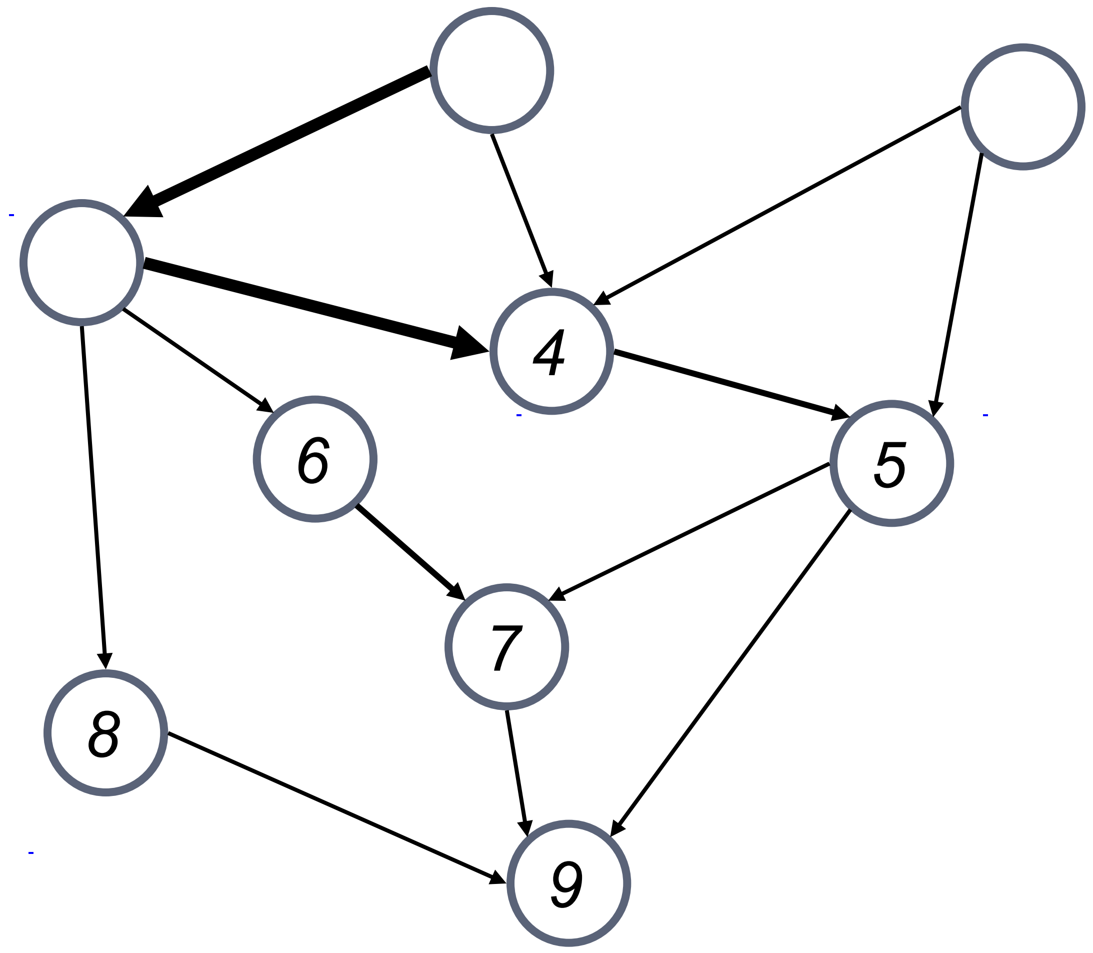
Topological Sorting Example



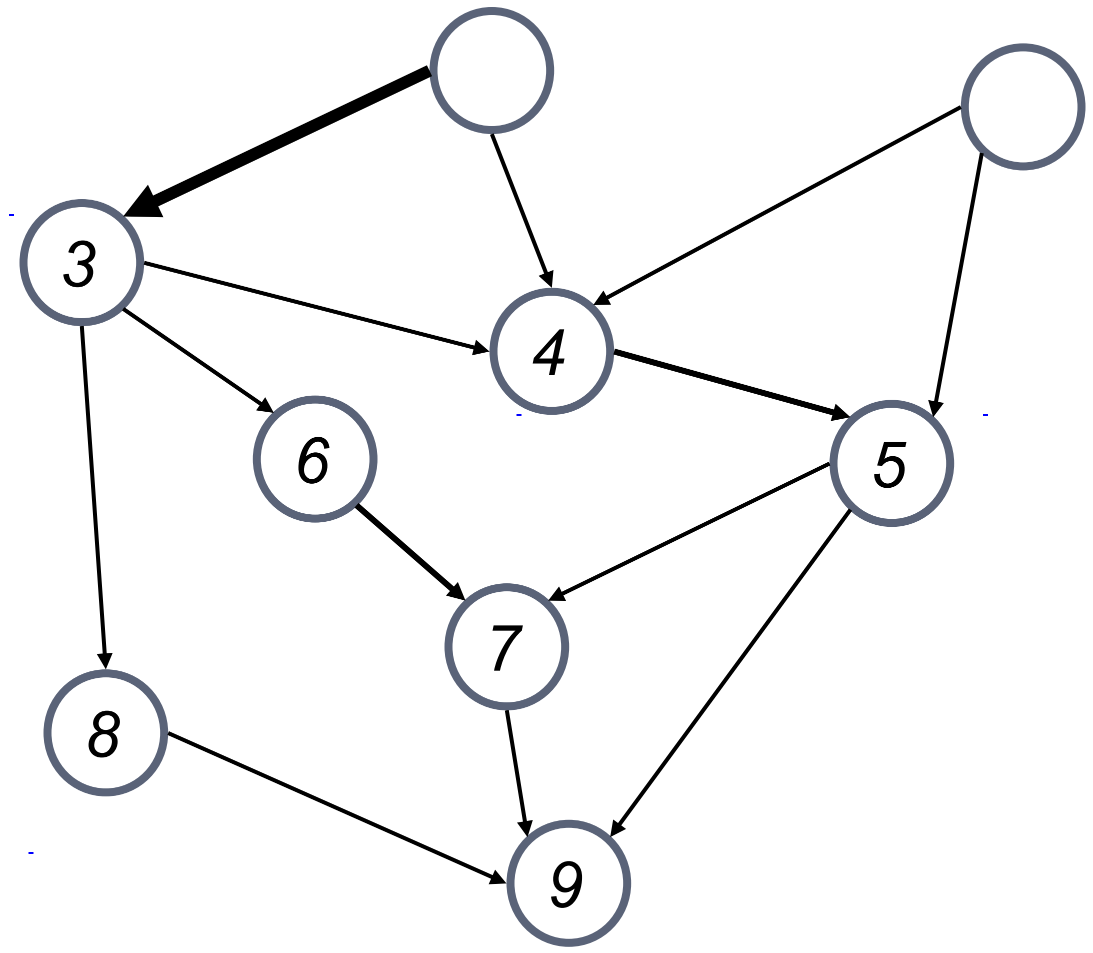
Topological Sorting Example



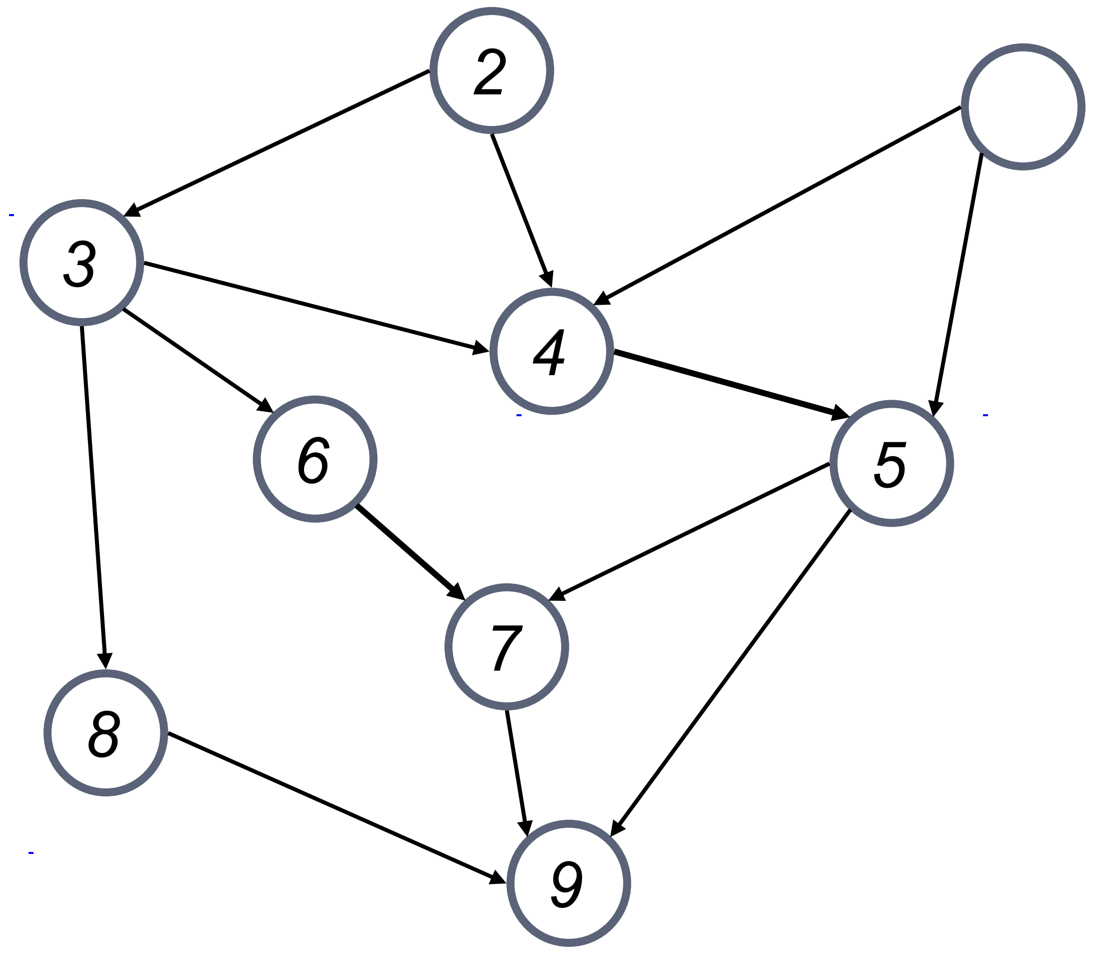
Topological Sorting Example



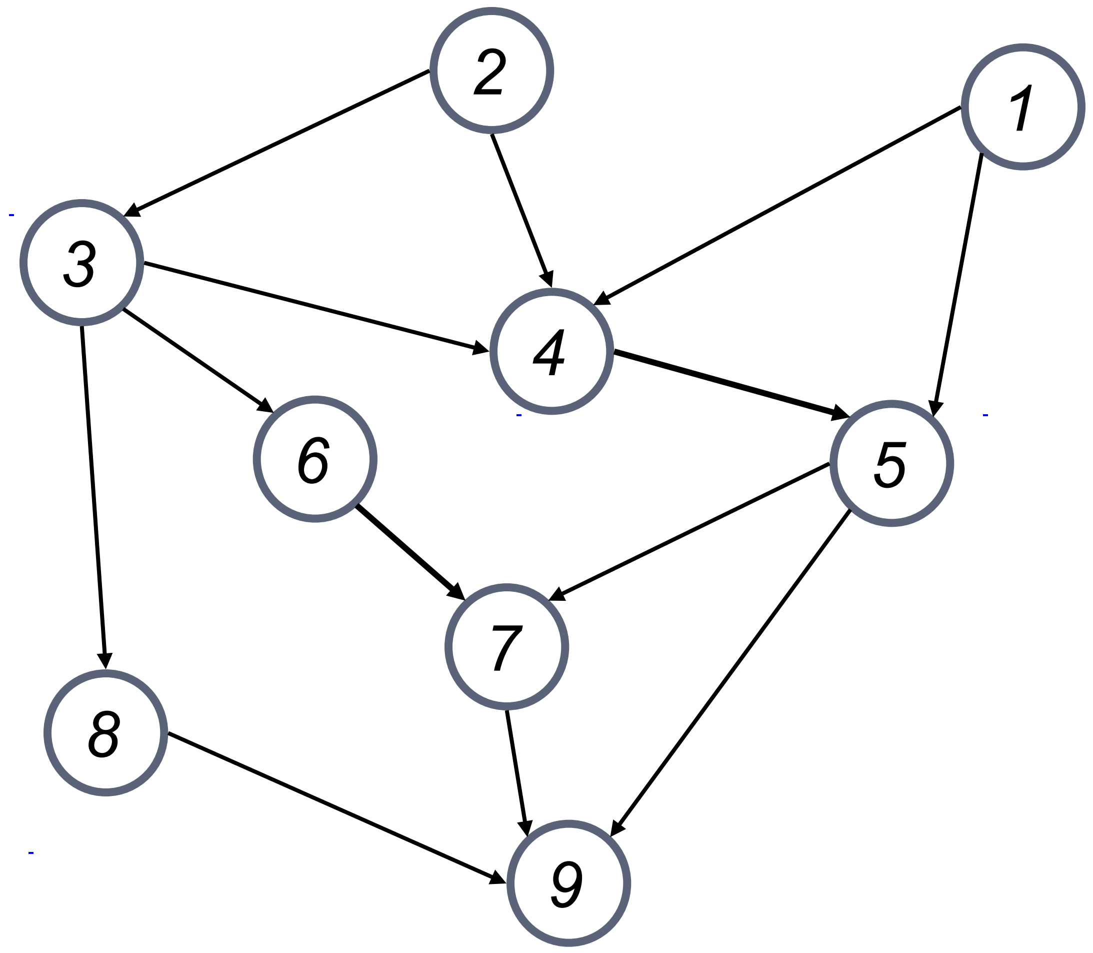
Topological Sorting Example



Topological Sorting Example



Topological Sorting Example



References

- Algorithm Design: Foundations, Analysis, and Internet Examples. Michael T. Goodrich and Roberto Tamassia. John Wiley & Sons.
- Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.

Thank you!