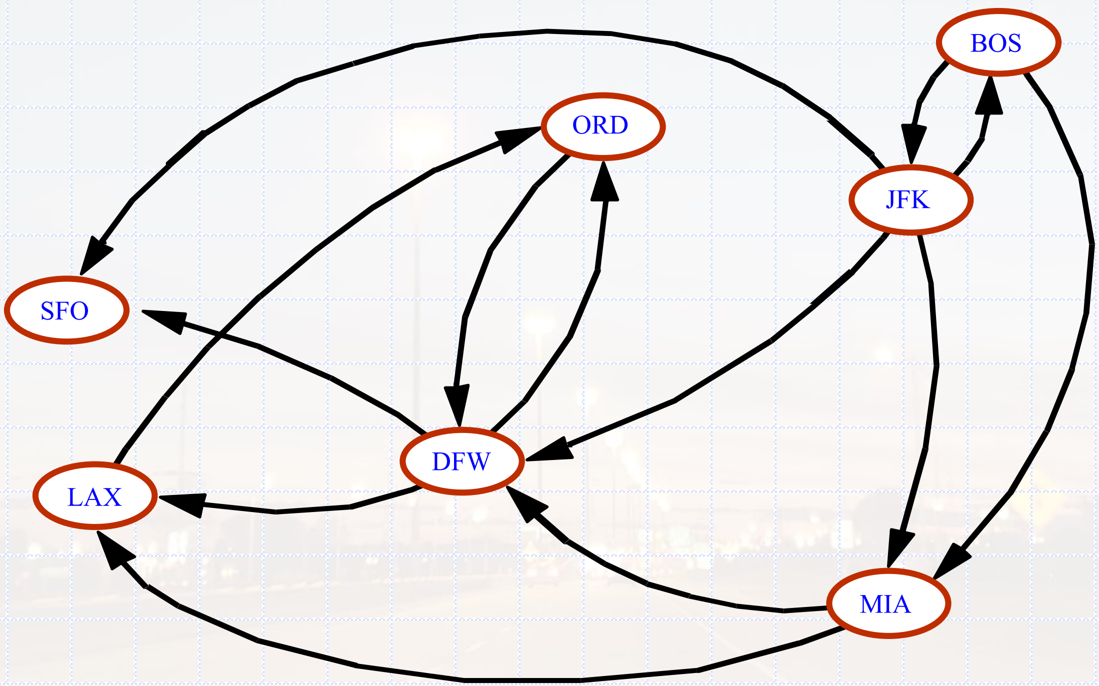
Directed Graphs (*digraphs*)

Dr. Aiman Hanna
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of :
Data Structures and Algorithms in Java, 5th edition. John Wiley& Sons, 2010. ISBN 978-0-470-38326-1.
Data Structures and the Java Collections Framework by William J. Collins, 3rdedition, ISBN 978-0-470-48267-4.
Both books are published by Wiley.

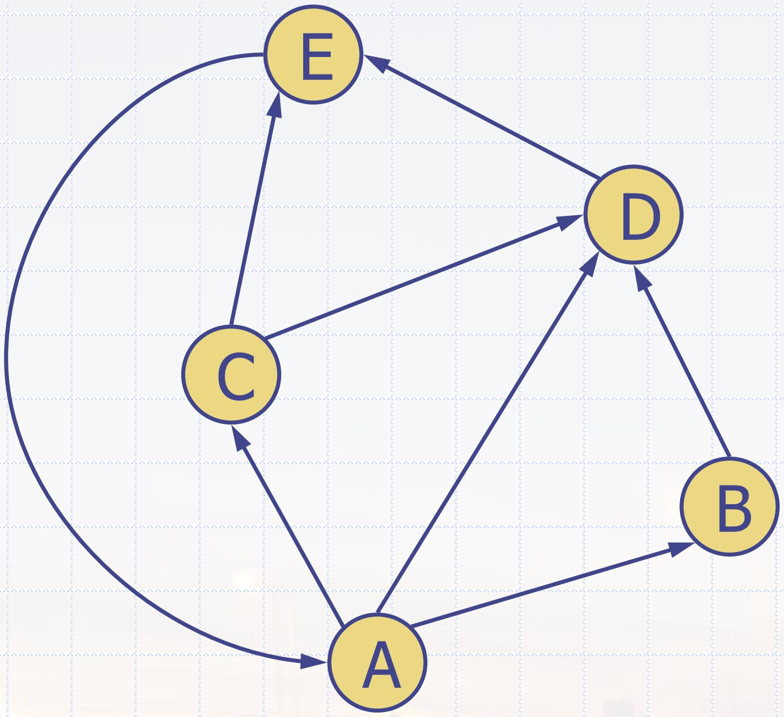
Coverage

- Directed Graphs (digraphs)
  - Traversal of digraphs
  - Transitive Closure
  - Directed Acyclic Graphs (DAGs)

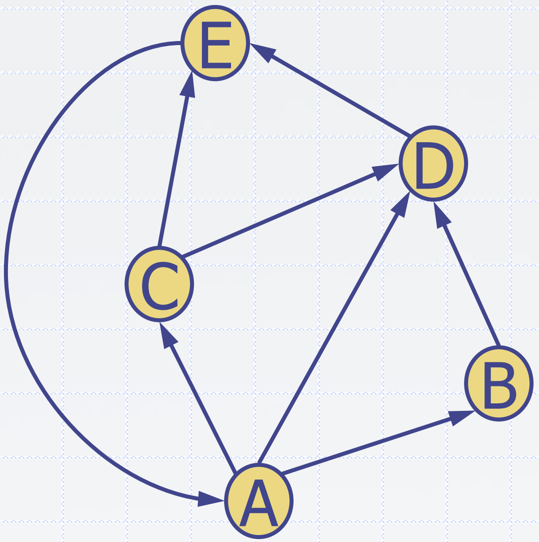


Digraphs

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



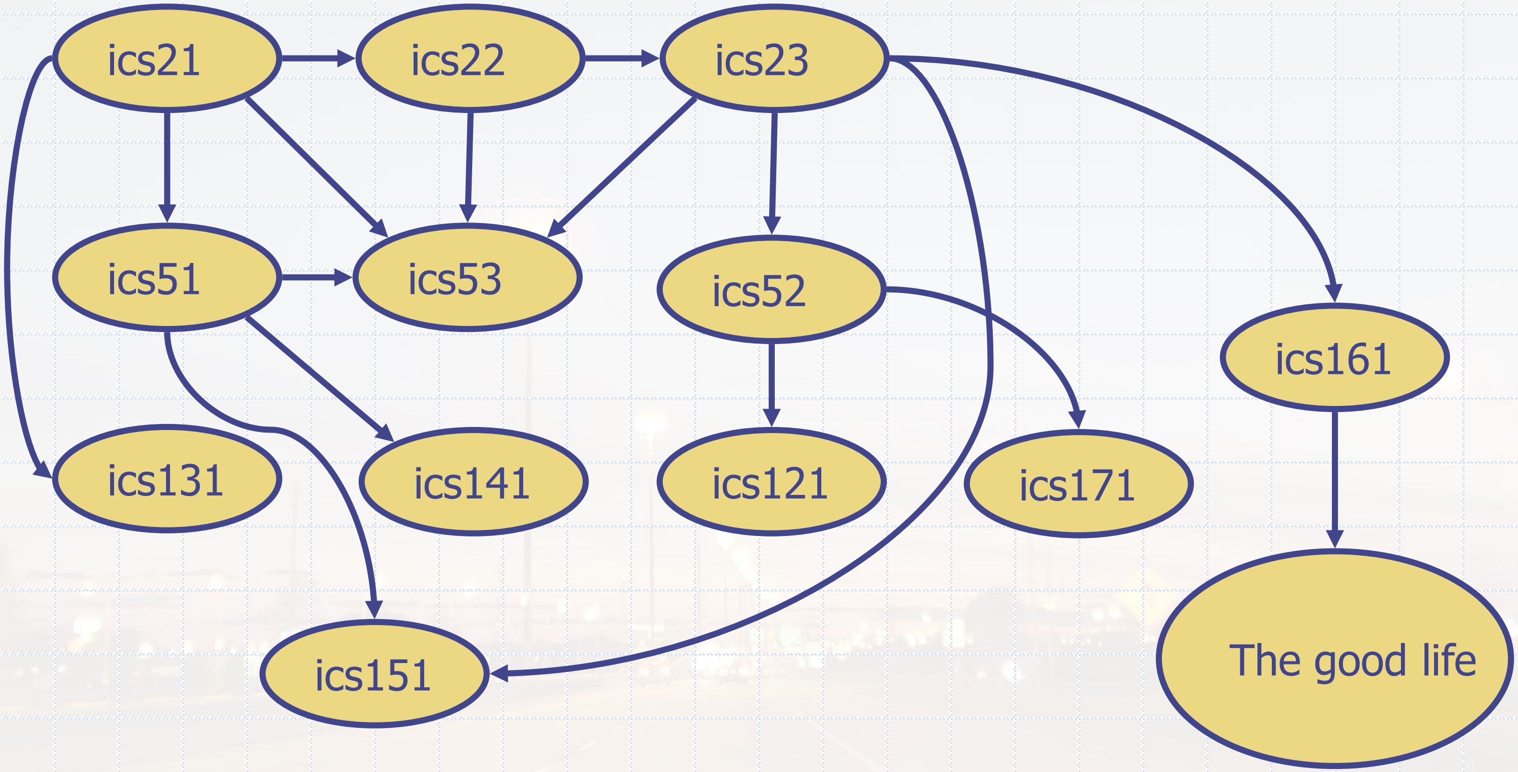
Digraph Properties



- 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
  - We usually refer to direct graph G as $\vec{G}$
- If G is simple, $m \leq n \cdot (n-1)$
  - We can have two directed edges between each two nodes (these are in opposite directions and not parallel since the graph is simple). Consequently, maximum degree at any vertex is $2(n-1)$; $n-1$ incoming and $n-1$ outgoing
- 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

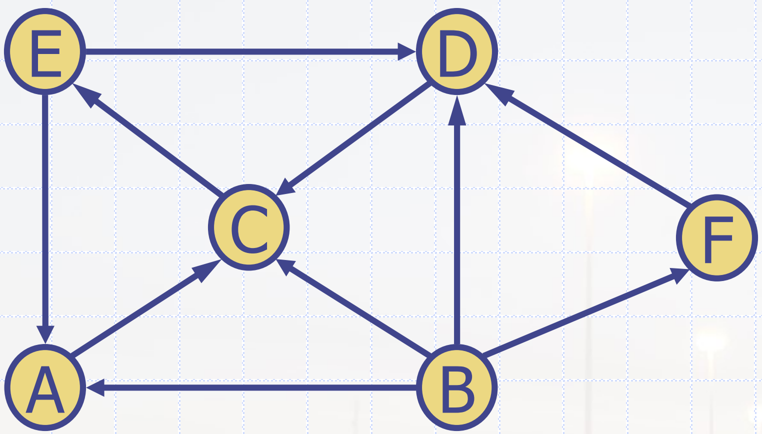


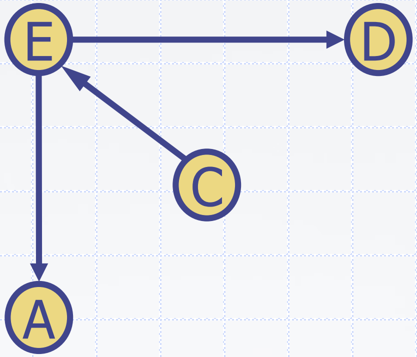
Reachability

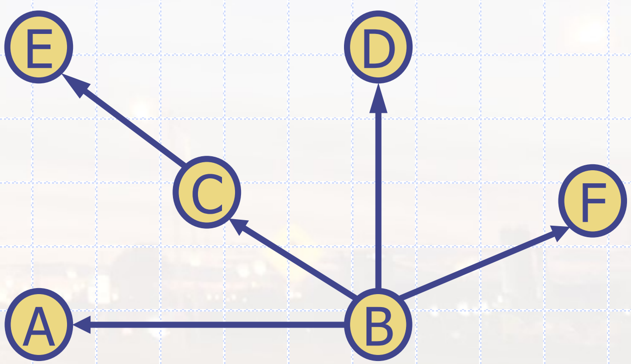
- One of the fundamental issues with digraphs is the notion of reachability
- Given two vertices w and u in graph G, we say that w ***reaches*** u, or u is ***reachable*** from w, if G has a directed path from w to u
- We also say that vertex v reaches an edge (w, z) if v reaches w and w is the origin of that edge

Reachability

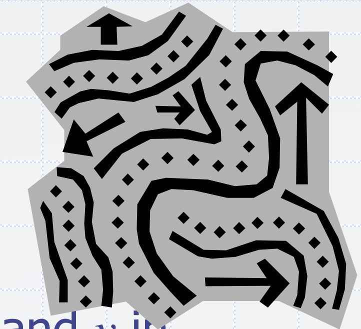
- Examples: Which vertex reaches which vertices?





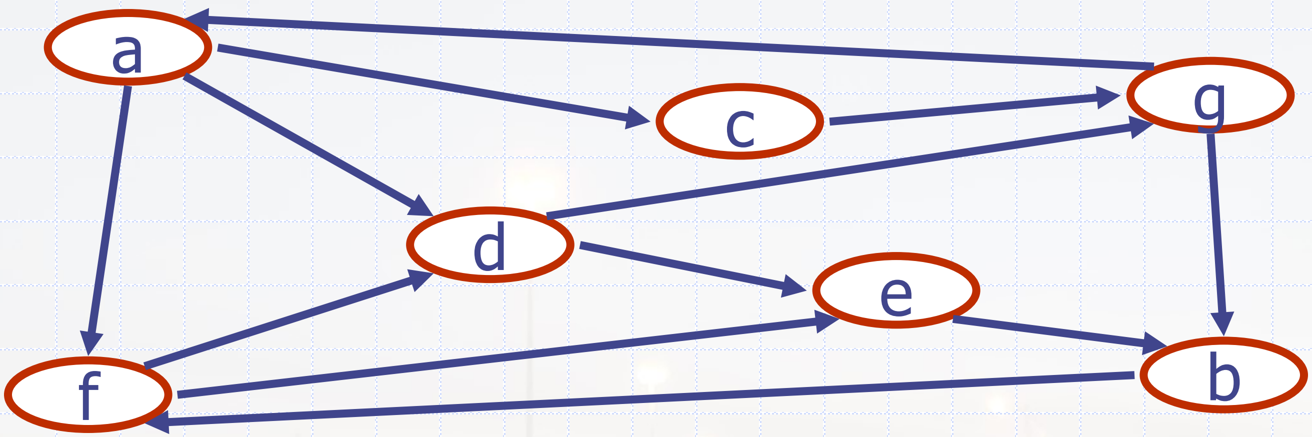


Strong Connectivity



- A digraph G is strongly connected if for any two vertices u and v in the graph, u reaches v and v reaches u

- In other words, each vertex can reach all other vertices

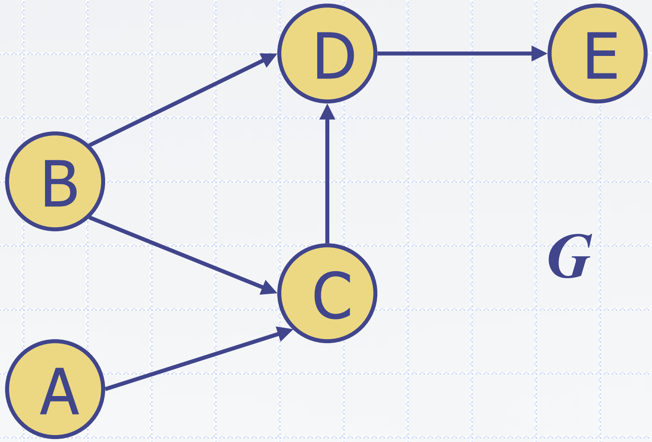


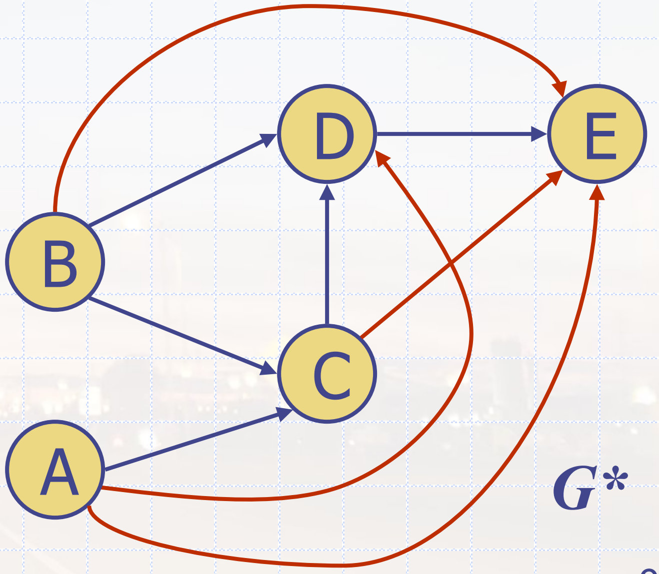
- A *directed cycle* is a cycle where all the edges are traversed in their respective directions

- A graph is *acyclic* if it has no directed cycles

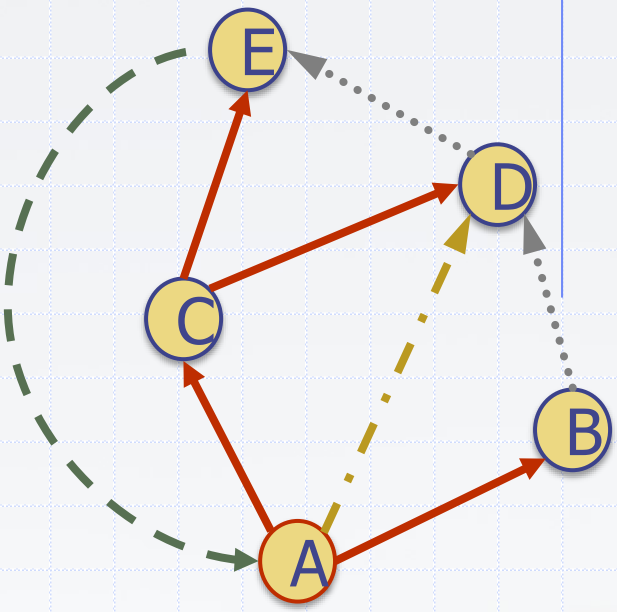
Transitive Closure

- 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





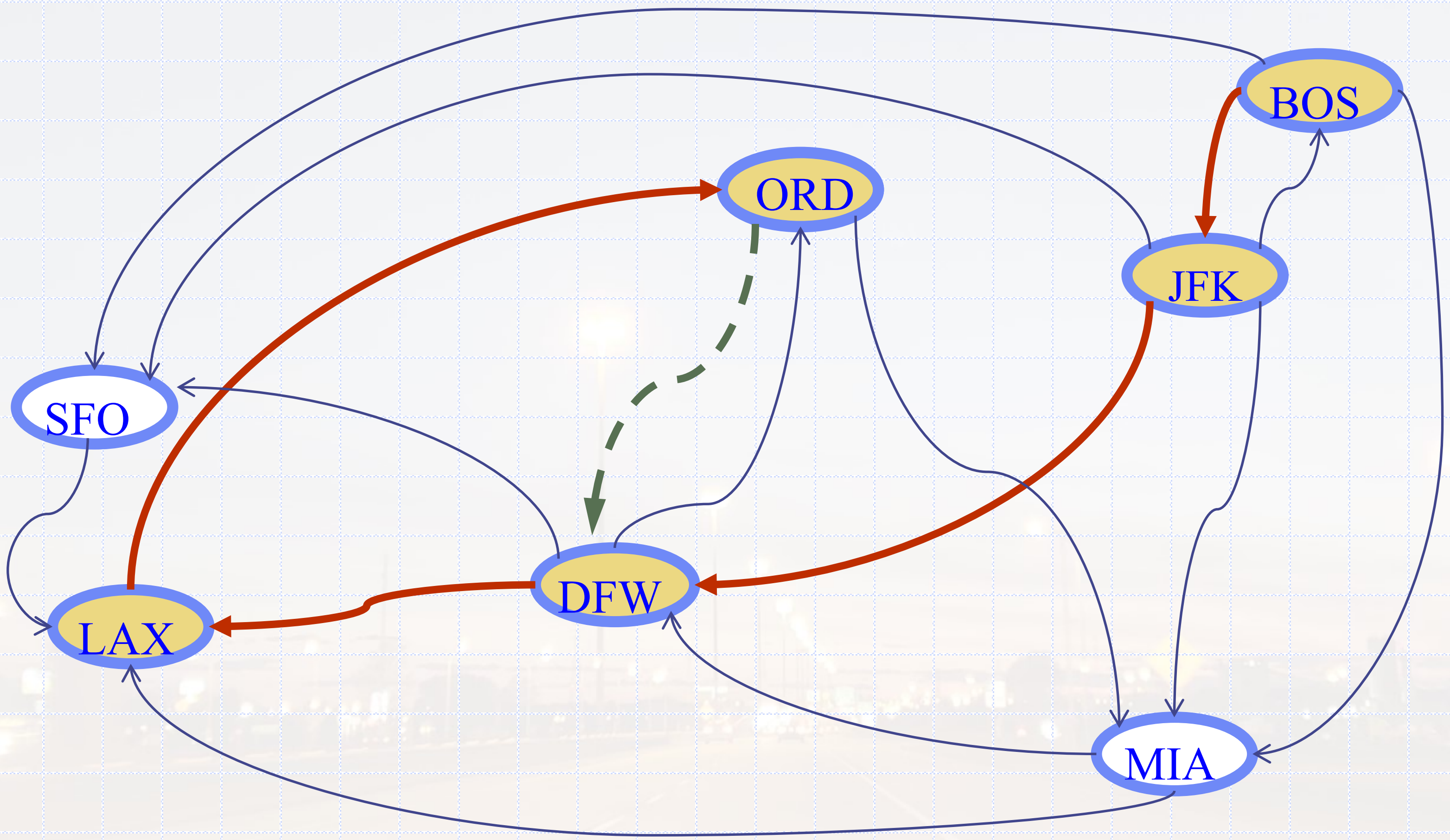
Digraph Traversal



- 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 (the last three are ***non-tree edges***)
  - **discovery edges** (edges that led us to discover new vertices; these are the ***tree edges***)
  - **back edges** (connect a vertex to its ancestor)
  - **forward edges** (connect a vertex to its descendant)
  - **cross edges** (connect an edge to another one who is neither ancestor no descendant)
- A directed DFS starting at a vertex s determines the vertices reachable from s

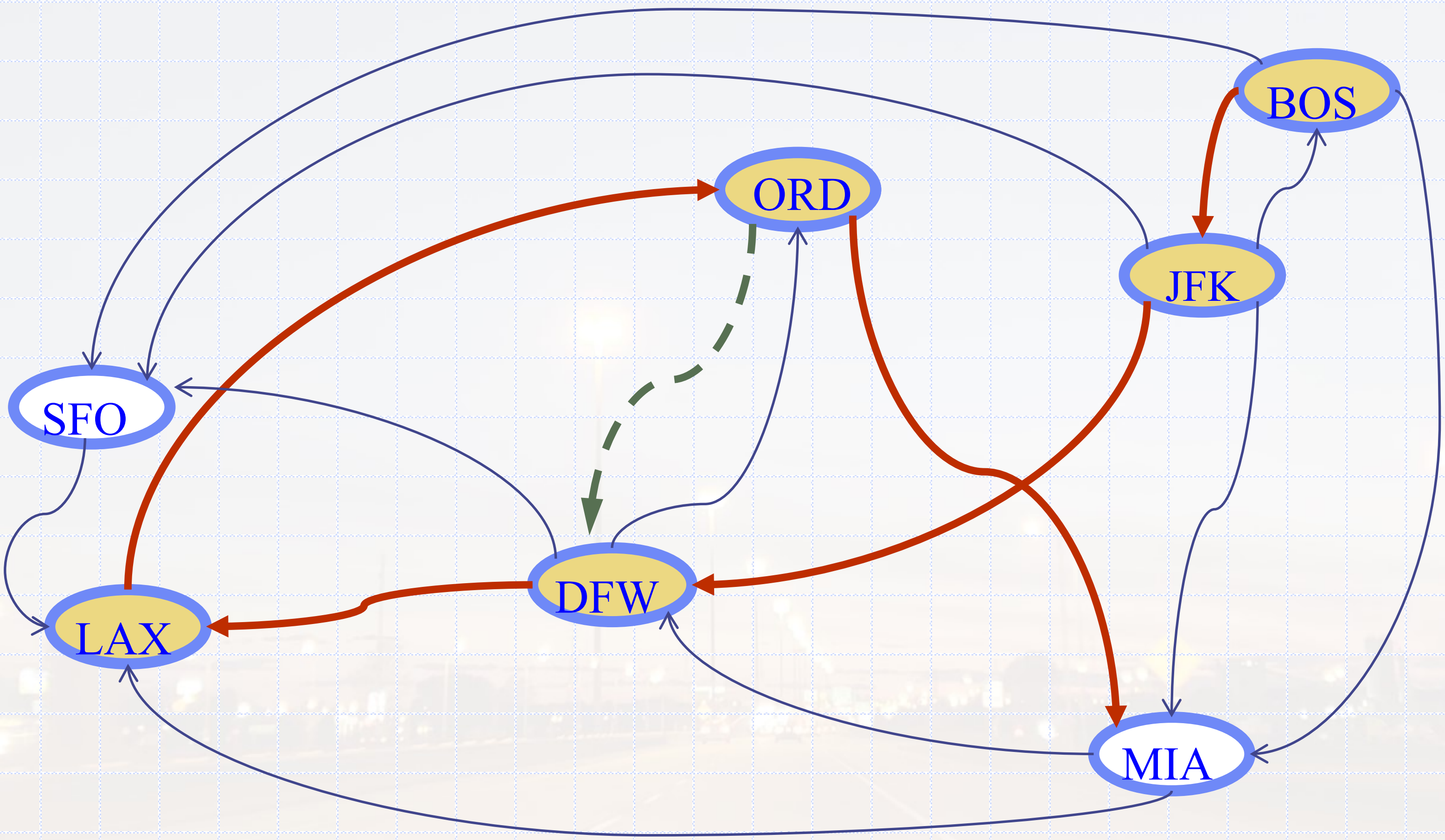
Digraph Traversal

- Example: Showing intermediate step where the traversal started from BOS and the “already previously visited” *DFW* is reached for the first time



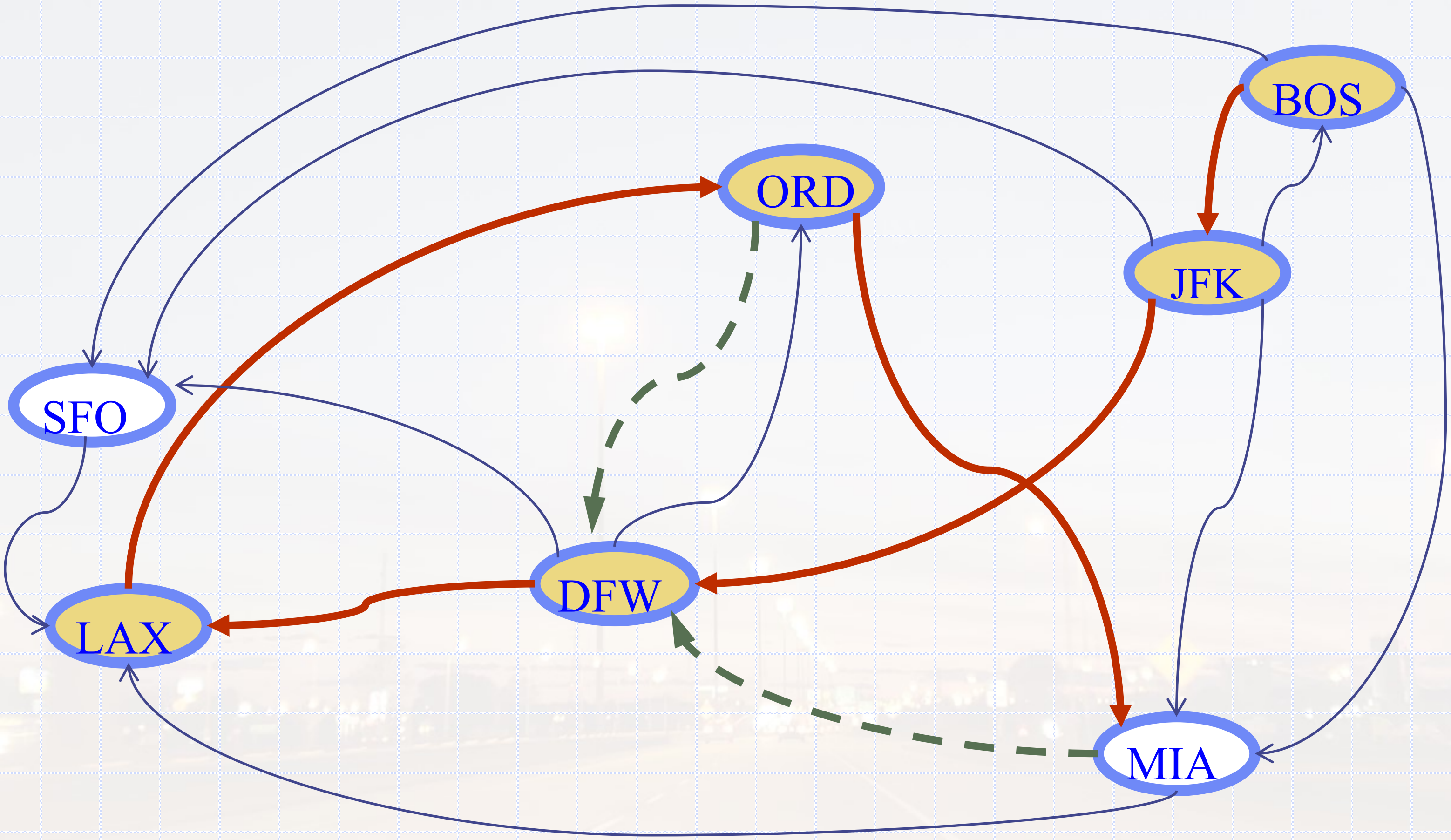
Digraph Traversal

- Example (continues…):



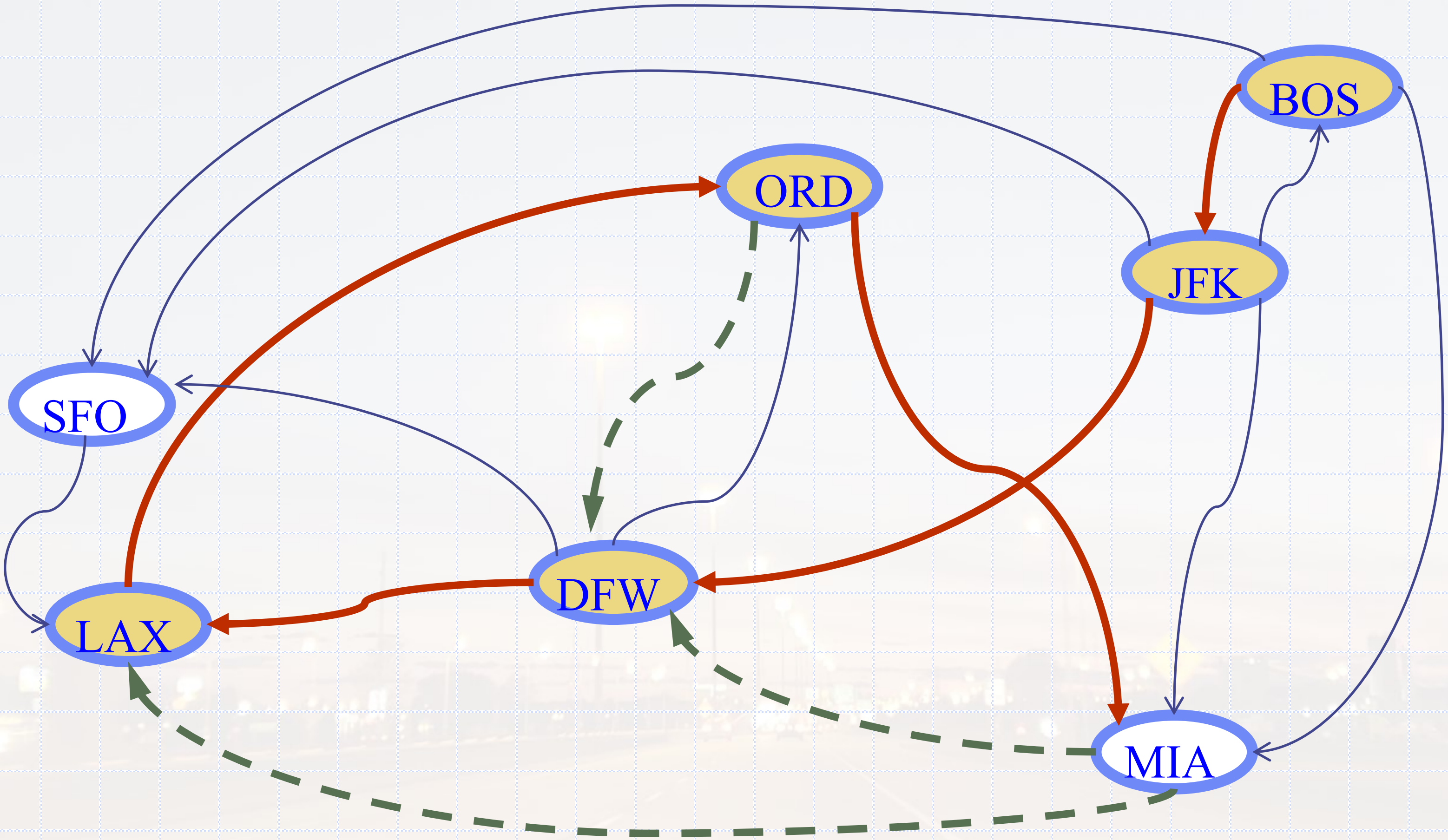
Digraph Traversal

- Example (continues…):



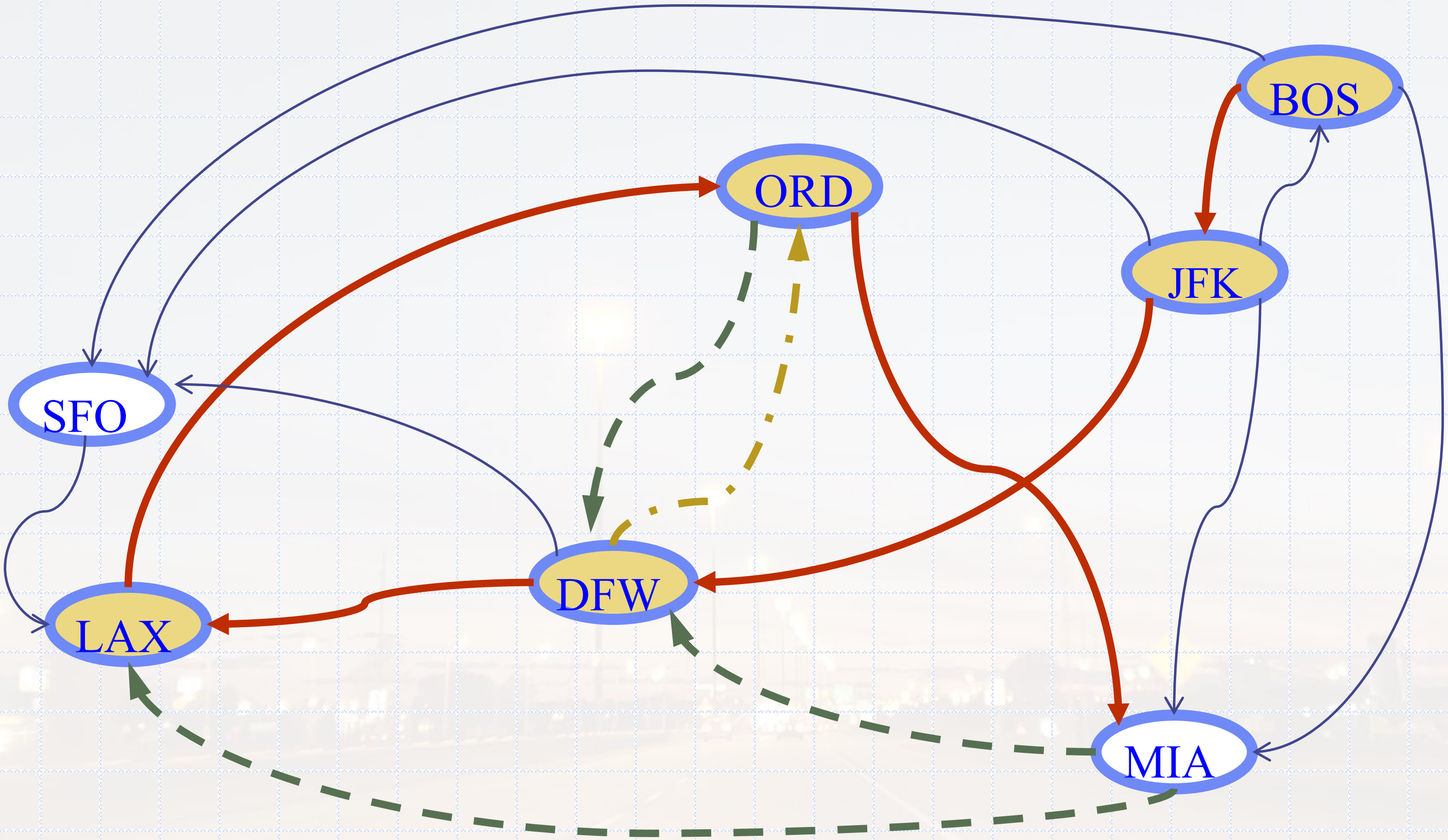
Digraph Traversal

- Example (continues…):



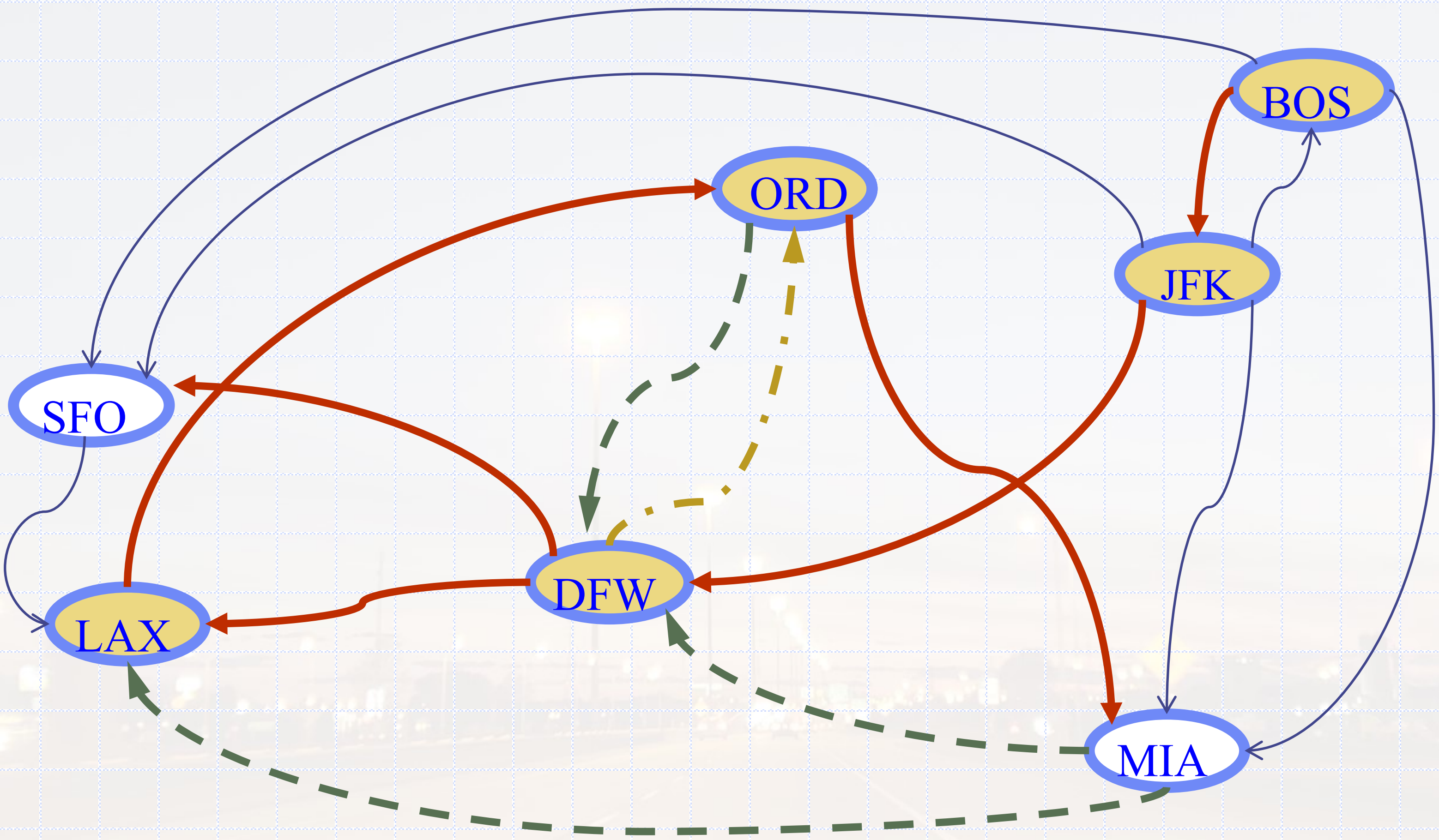
Digraph Traversal

- Example (continues…):



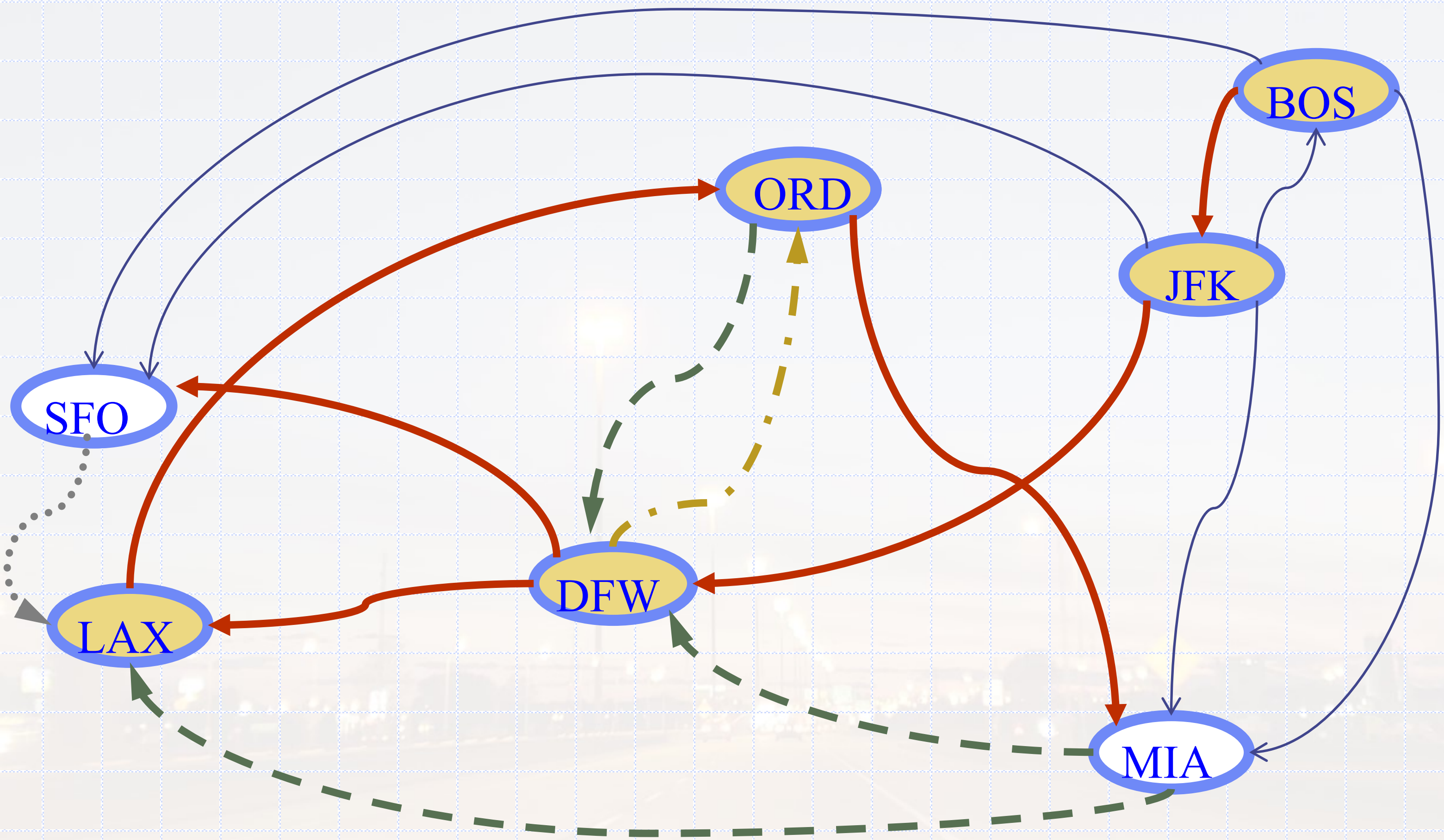
Digraph Traversal

- Example (continues…):



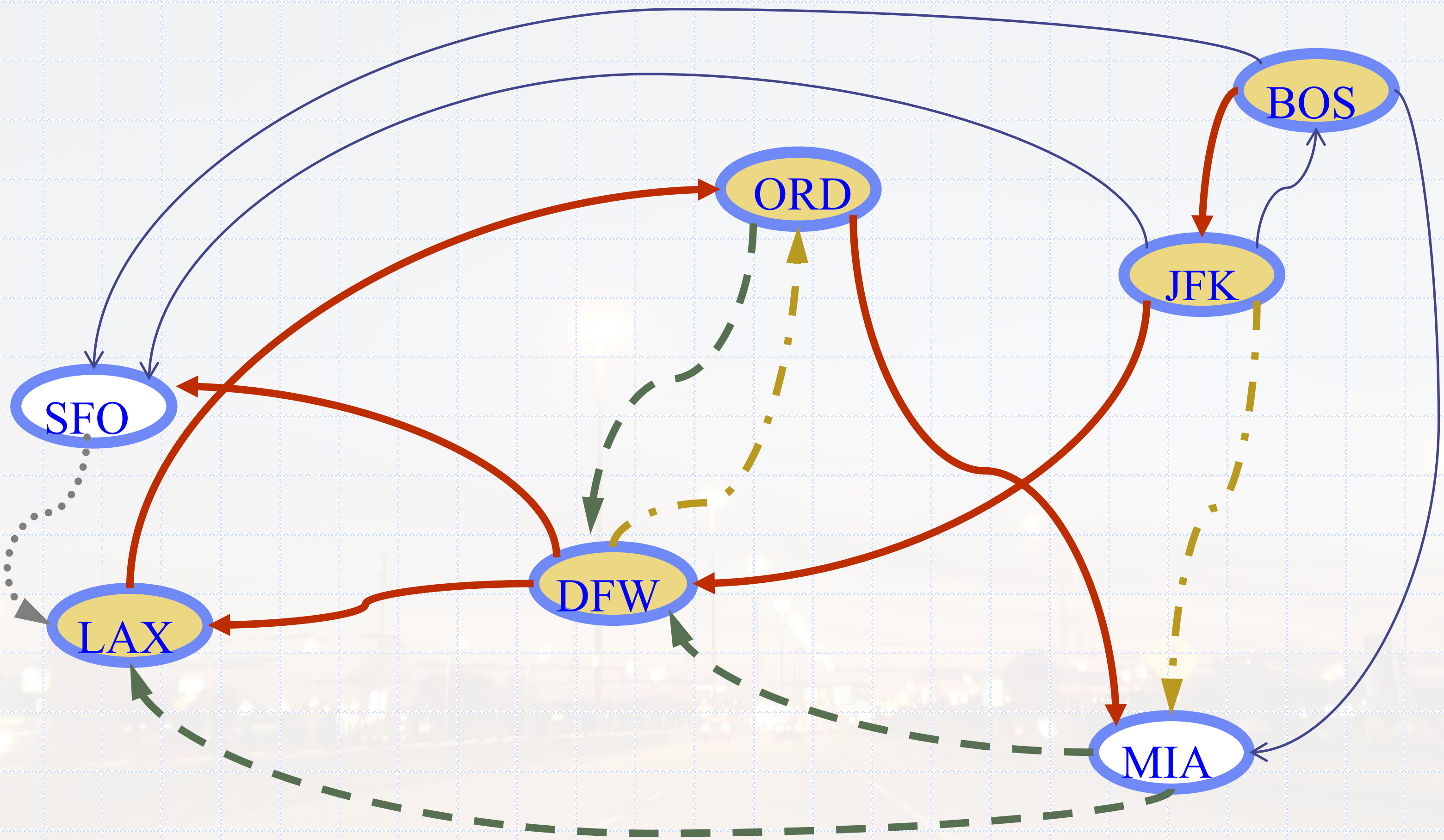
Digraph Traversal

- Example (continues…):



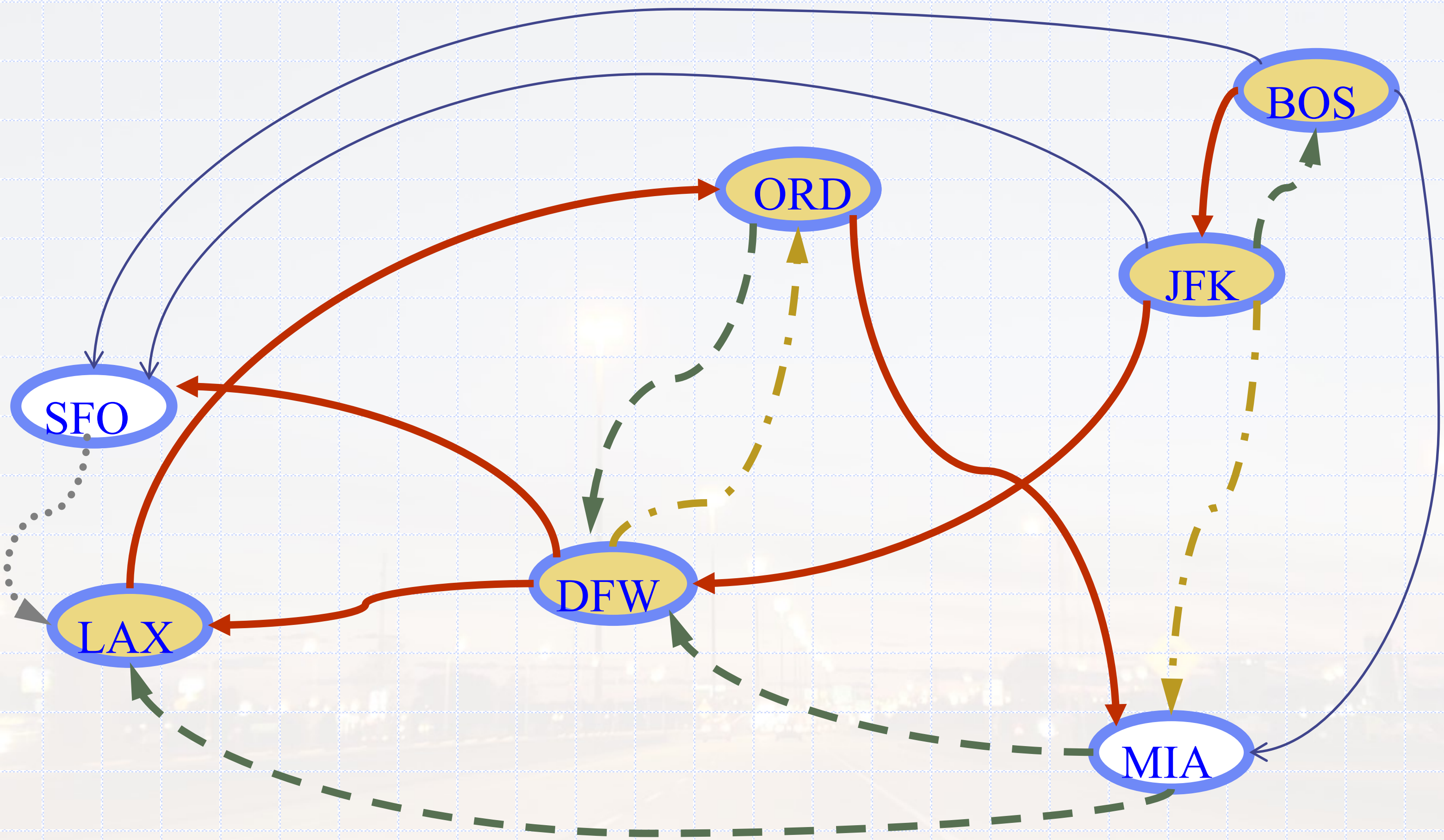
Digraph Traversal

- Example (continues…):



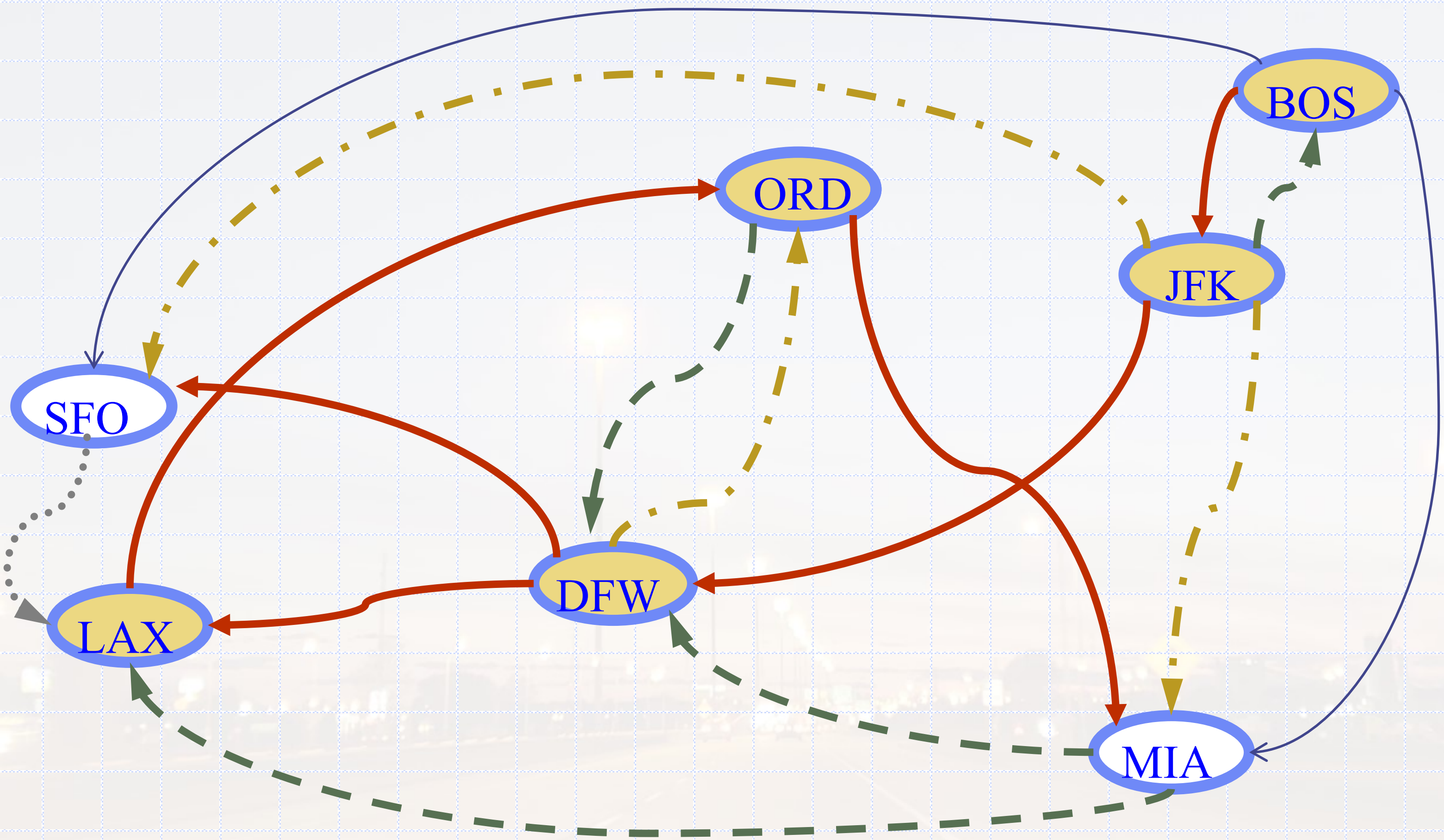
Digraph Traversal

- Example (continues…):



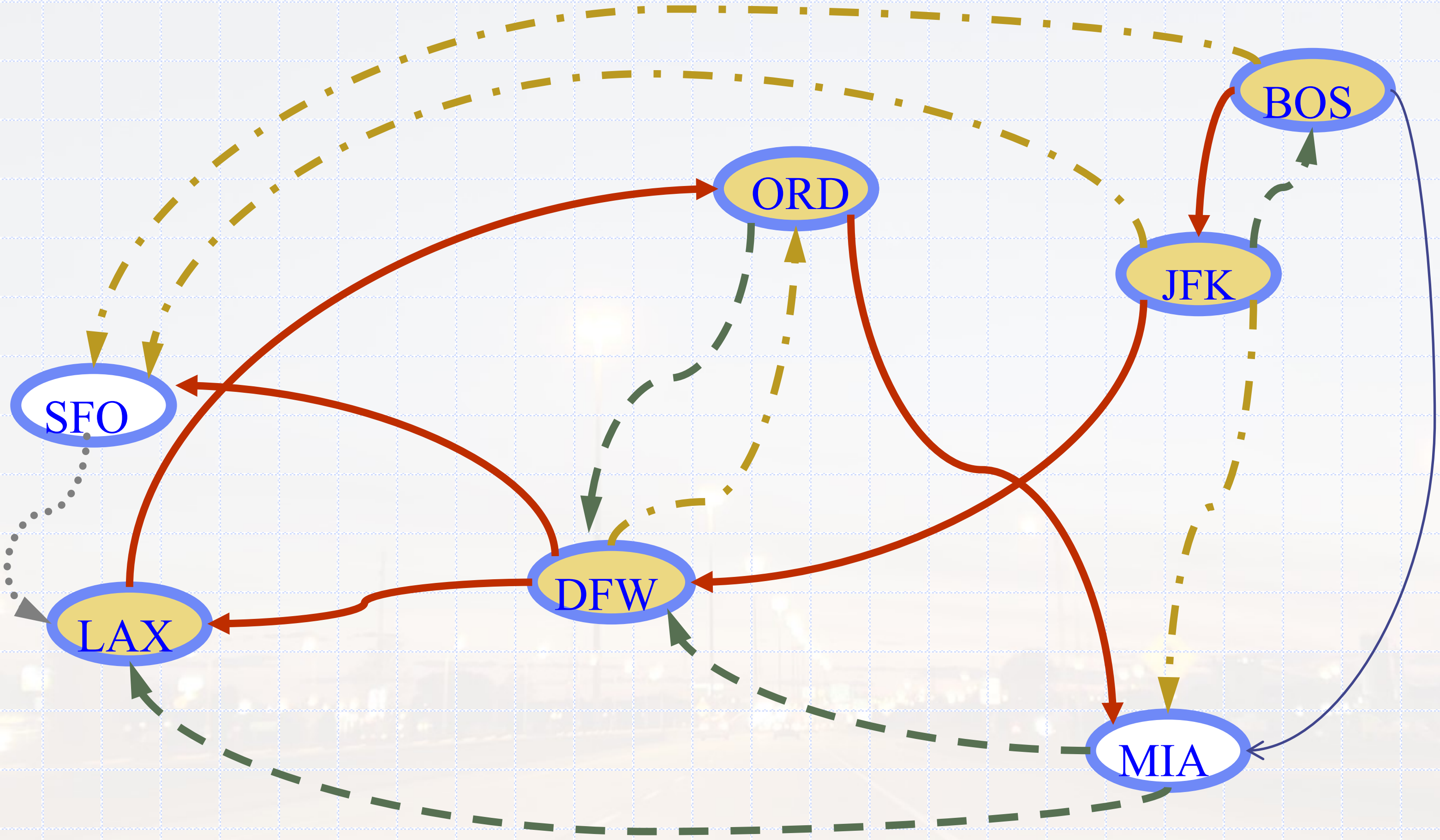
Digraph Traversal

- Example (continues…):



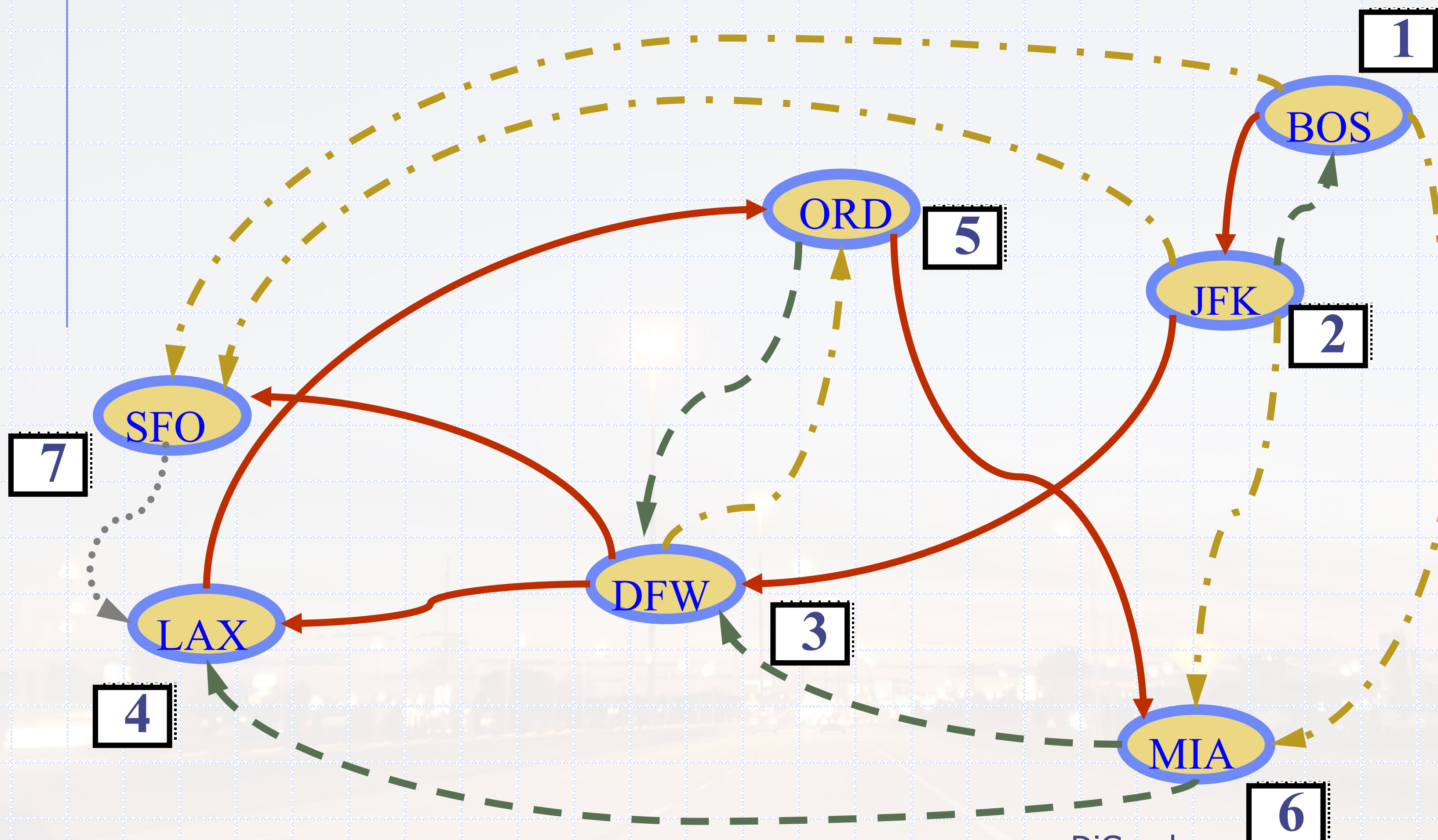
Digraph Traversal

- Example (continues…):



Digraph Traversal

- Example (continues…): Completed DFS



Digraph DFS Analysis

- Directed DFS over a graph G:
  - Starting at vertex s, DFS visits all vertices that are reachable from s (Notice that these may NOT be all vertices of G)
    - Can be proven by contradiction as we have done with undirected DFS
  - Starting for a vertex s, DFS runs in $\boldsymbol{O(n_s + m_s)}$, where n_s and m_s are the reachable vertices and edges from s.
    - A recursive call is needed once for each vertex, and each edge is traversed once from its origin
  - Can find/compute the transitive closure of G. This can be done as follows:
    - Find all the vertices reachable form a vertex s
    - Add edges from that vertex s to the ones that it reaches if such edges do not exist
    - Repeat the operation for each vertex v in the graph

Digraph DFS Analysis (Continue…)

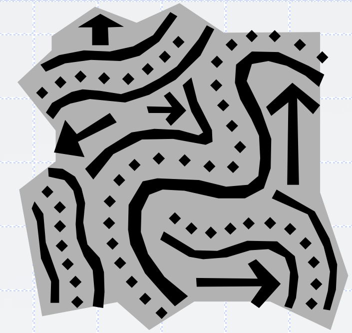
- Directed DFS over a graph G:
  - Test if G is strongly connected. This can be done as follows:
    - Perform repeated DFS traversal operations starting from each vertex in G
    - If each DFS visits all the vertices in G then G is strongly connected
- DFS complexity to find transitive closure or strong connectivity is:
  - Each run takes $O(n + m)$
  - We perform these runs n times

  → $O(n(n + m))$

Digraph DFS Analysis (Continue...)

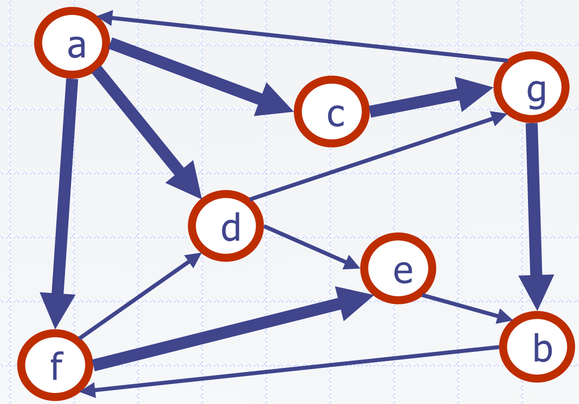
- In fact, strong connectivity can be tested much faster than that (by only using 2 DFSs) as follows:
  - Start at any vertex s and perform DFS
  - If s does not visit all the vertices then the graph is not strongly connected
  - If s reaches all vertices then reverse all edges in G (or change the algorithm to treat them as if they were reversed) and run DFS again starting from s
  - If this second run reaches all the vertices, then G is strongly connected; otherwise it is not (since not each vertex can reach s!)
    - Proof: from first run s can reaches all vertices. From second run, all vertices reach s, but s can reach all vertices (from first run!). Consequently each vertex can reach all other vertices in G
  - These will takes $\boldsymbol{O(2(n+m))} \rightarrow \boldsymbol{O(n+m)}$

Strong Connectivity Algorithm

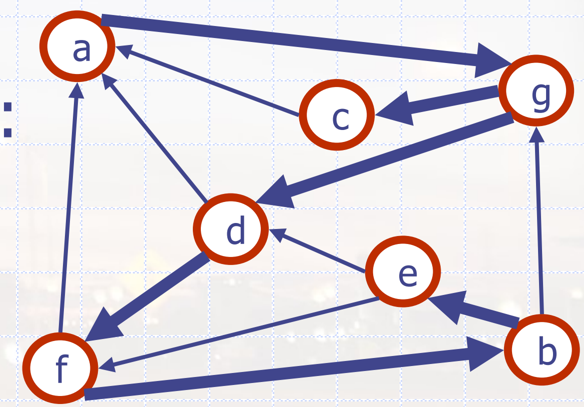


1. Pick a vertex v in G
2. Perform a DFS from v in G
   a. If there is a vertex w that is not visited, print "no strong connectivity"
3. Let G' be G with edges reversed
4. Perform a DFS from v in G'
   a. If there is a vertex w that is not visited, print "no strong connectivity"
   b. Else, print "Graph is strongly connected"

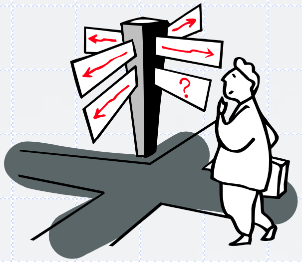
G:



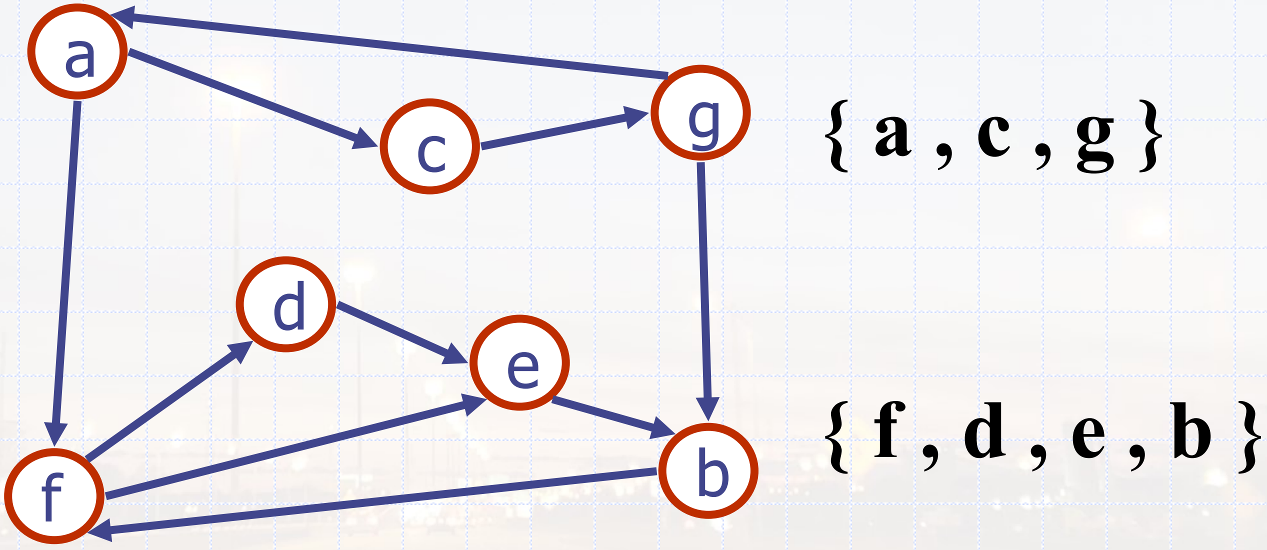
G′:



Strongly Connected Components



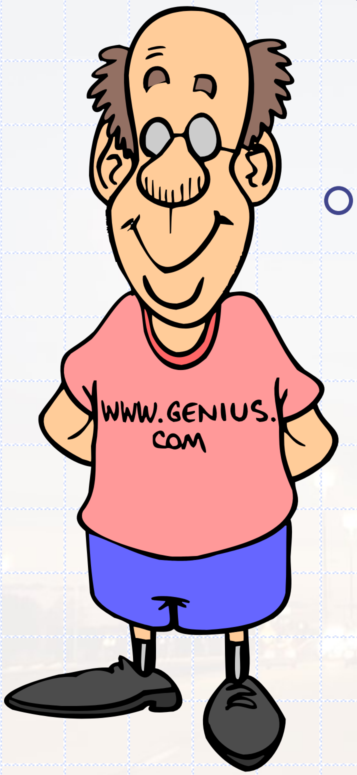
- Maximal subgraphs such that each vertex can reach all other vertices in these subgraphs
- Can also be done in $O(n+m)$ time using DFS, but is more complicated (similar to biconnectivity).



Computing the Transitive Closure

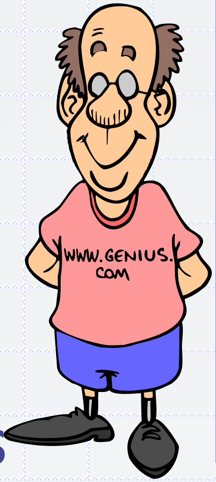
- We can perform DFS starting at each vertex
  - $O(n(n+m))$

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.



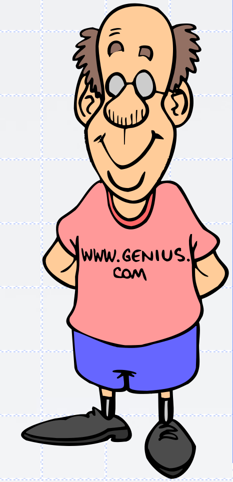
Alternatively ... Use dynamic programming: The Floyd-Warshall Algorithm

Floyd-Warshall Transitive Closure

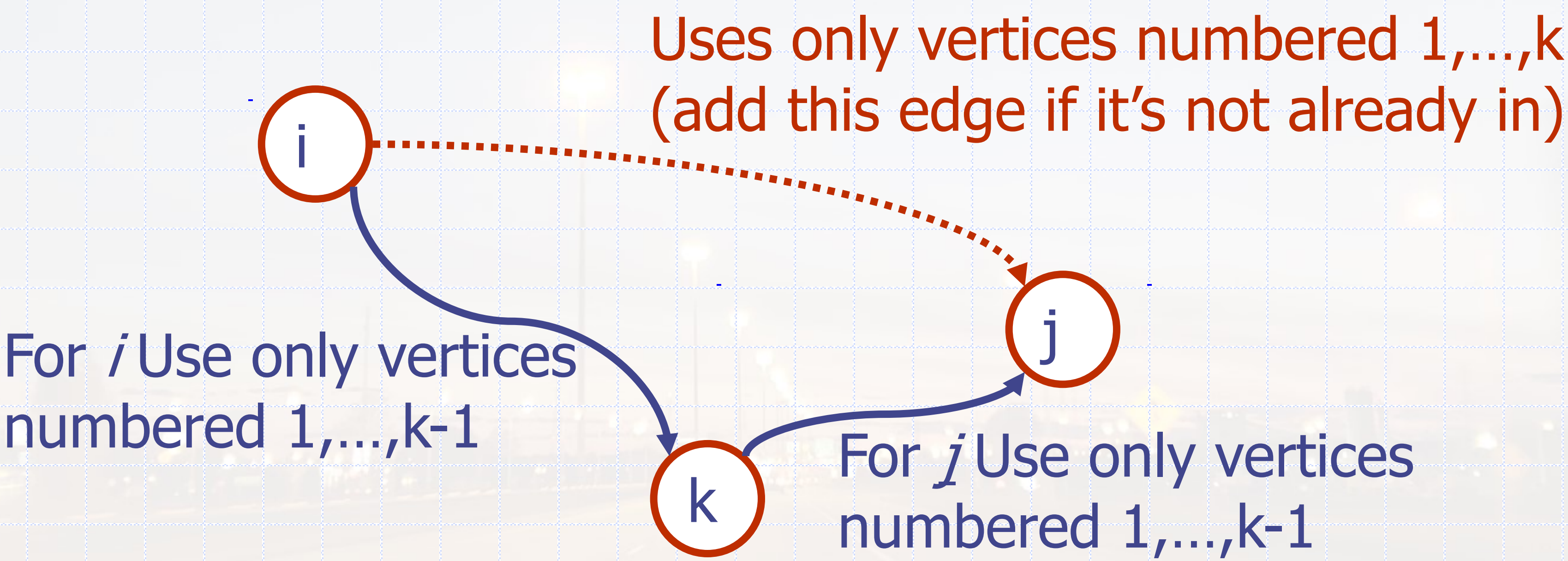


- The idea is as follows:
  - Let G be a digraph with n vertices and m edges
  - Compute the transitive closure in a series of rounds as follows:
    - Initialize $G_0 = G$
    - Arbitrary number the vertices v_1 to v_n
    - Start the computation from round 1
    - For any round k, we construct digraph G_k starting from $G_k = G_{k-1}$
    - If G_{k-1} contains both direct edges (v_i , v_k) and (v_k , v_j), then add a direct edge (v_i , v_j) to G_k if it is not already there
    - $G^* = G_n$

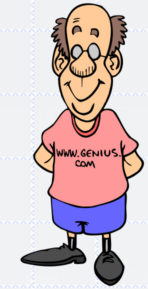
Floyd-Warshall Transitive Closure



- The algorithm is known as the ***Floyed-Warshall algorithm*** and it belongs to an algorithmic design pattern known as ***dynamic programming***

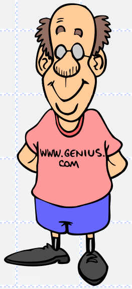


Floyd-Warshall's Algorithm



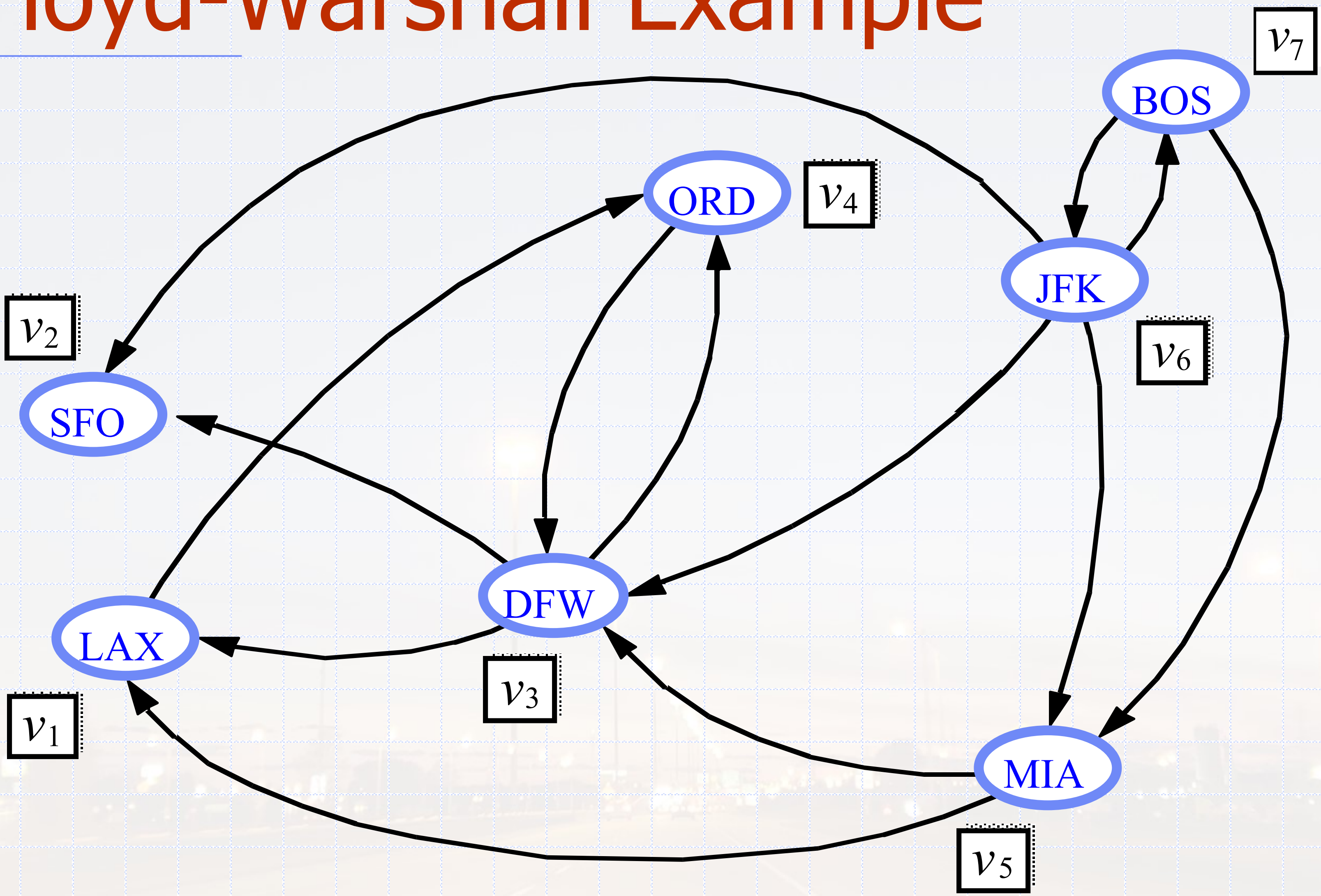
Algorithm *FloydWarshall*(G)
    Input digraph G
    Output transitive closure G^* of G
    $i \leftarrow 1$
    for all $v \in G.vertices()$
        denote v as v_i
        $i \leftarrow i + 1$
    $G_0 \leftarrow G$
    for $k \leftarrow 1$ **to** n **do**
        $G_k \leftarrow G_{k-1}$
        for $i \leftarrow 1$ **to** n $(i \neq k)$ **do**
            for $j \leftarrow 1$ **to** n $(j \neq i, j \neq k)$ **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's Analysis

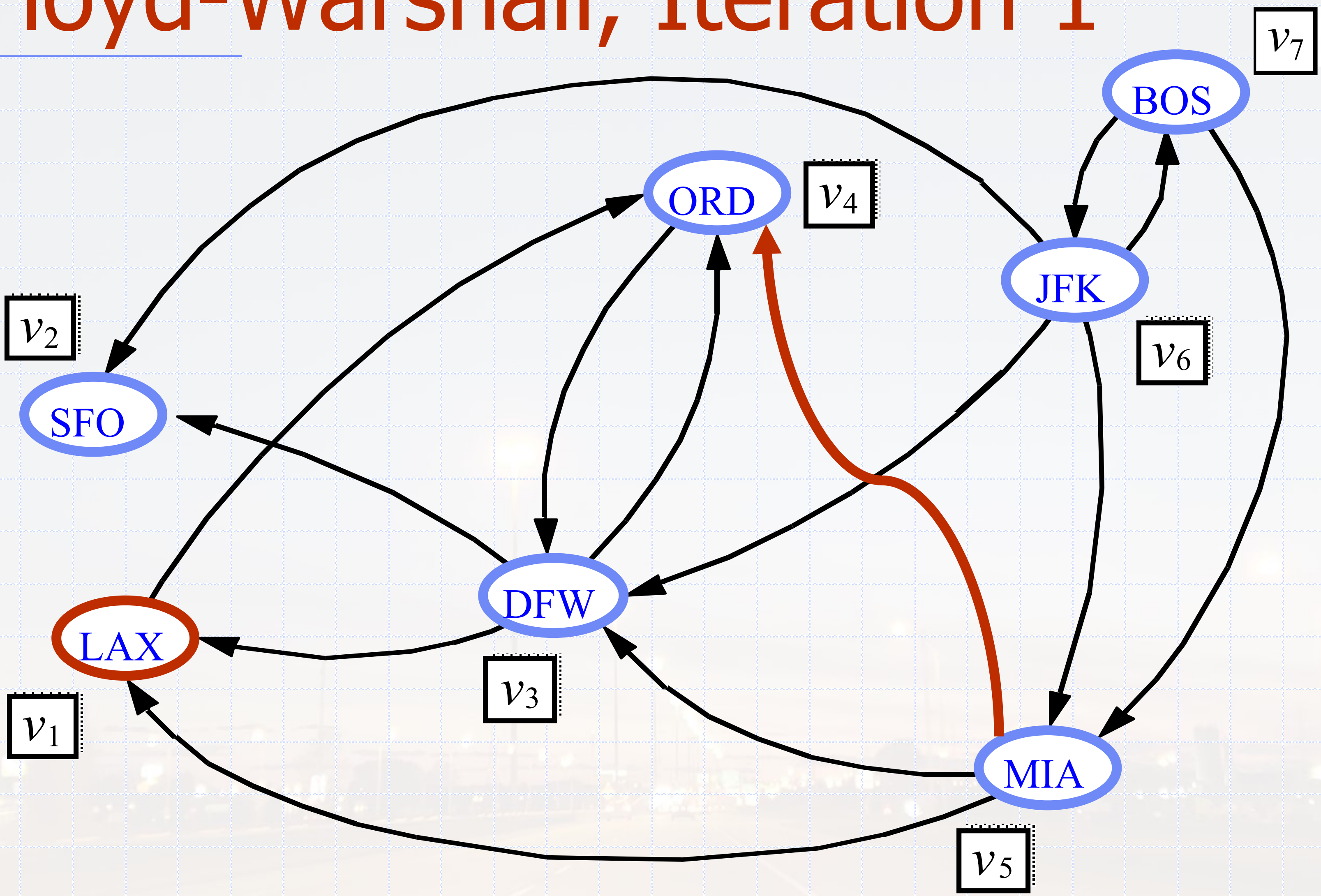


- Running time:
  - Assuming that ***areAdjacent()*** is $O(1)$ and ***insertDirectedEdge()*** is $O(1)$, which can be achieved using adjacency matrix, Floyed-Warshall has a complexity of $O(n^3)$
    - Notice the three loops in the algorithm
- Running DFS seems to be faster than Floyed-Warshall, however:
  - If the graph is represented by adjacency matrix then one run of DFS would take $O(n^2)$, which means to compute the transitive closure would require $O(n^3)$ since DFS needs to run n times
  - If this is the case, then running Floyed-Warshall once may be preferable than running DFS n times since after all both will have the same complexity of $O(n^3)$

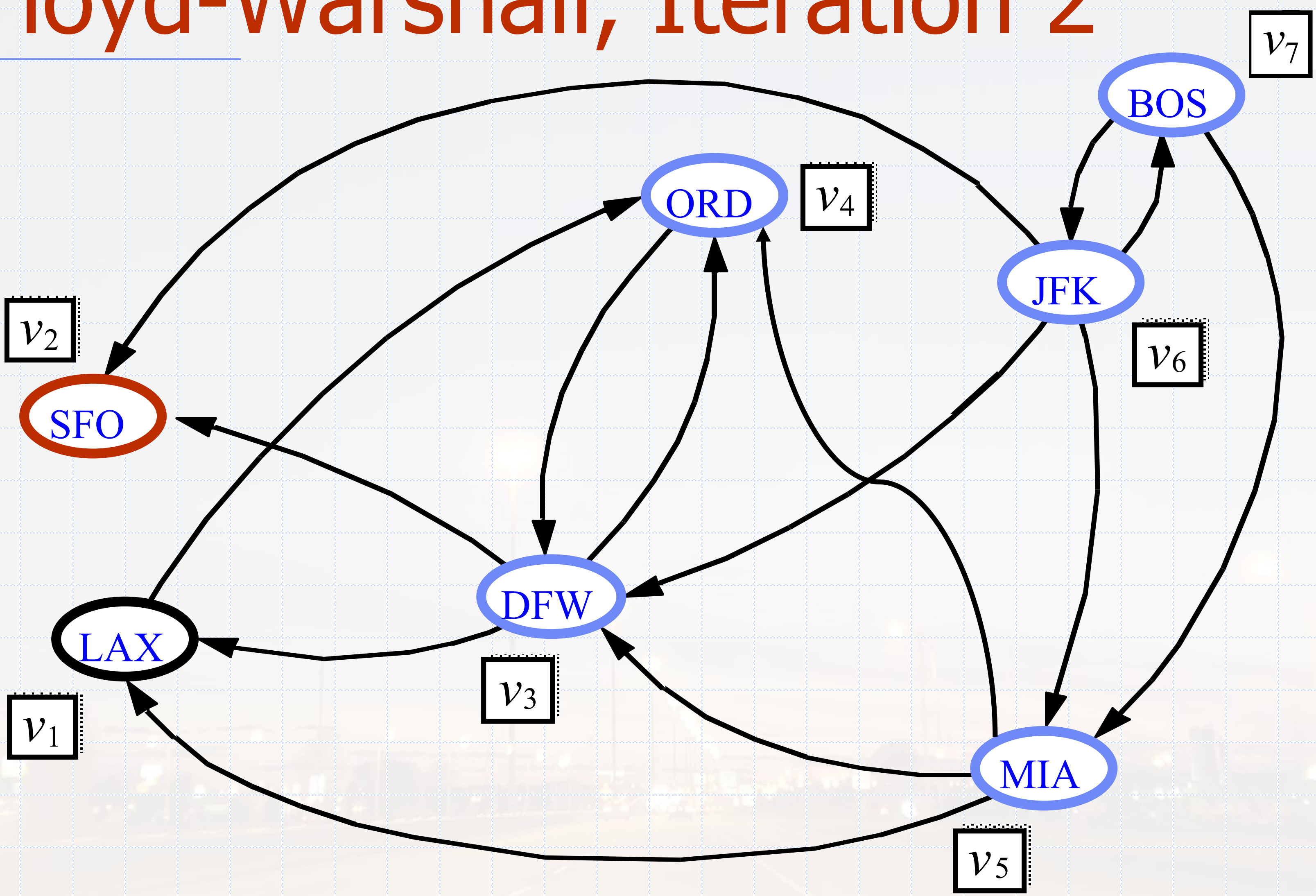
Floyd-Warshall Example



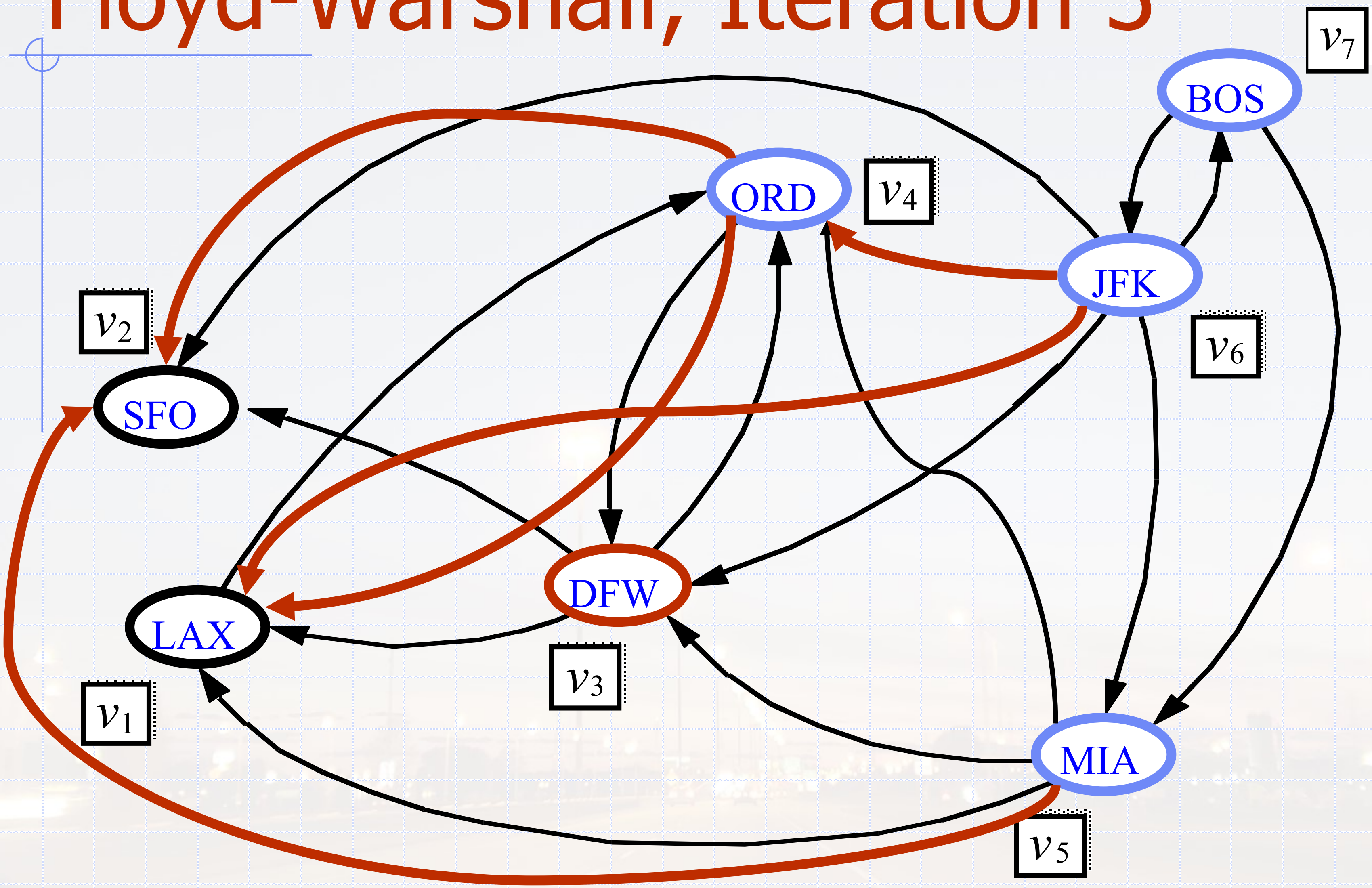
Floyd-Warshall, Iteration 1



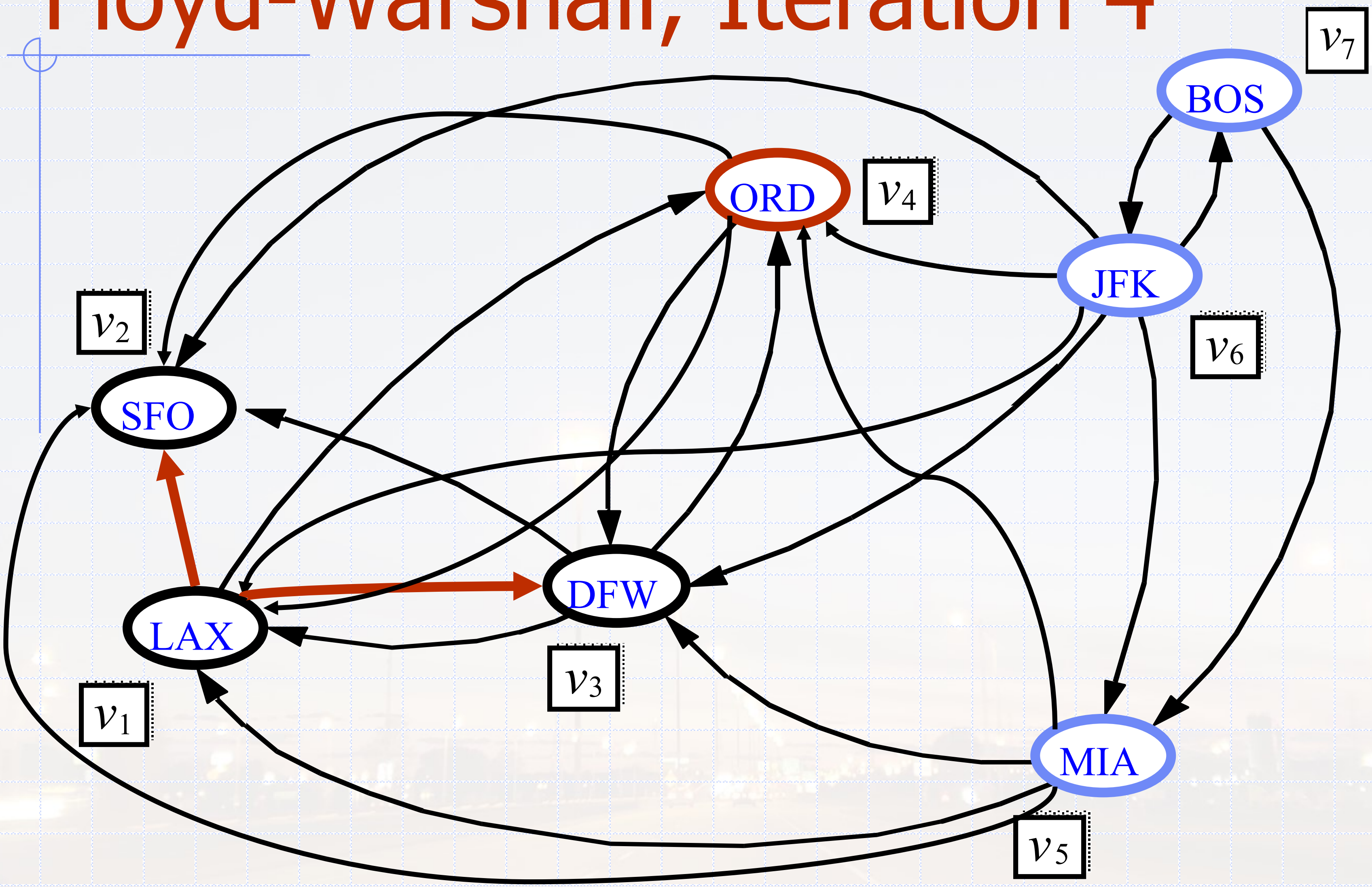
Floyd-Warshall, Iteration 2



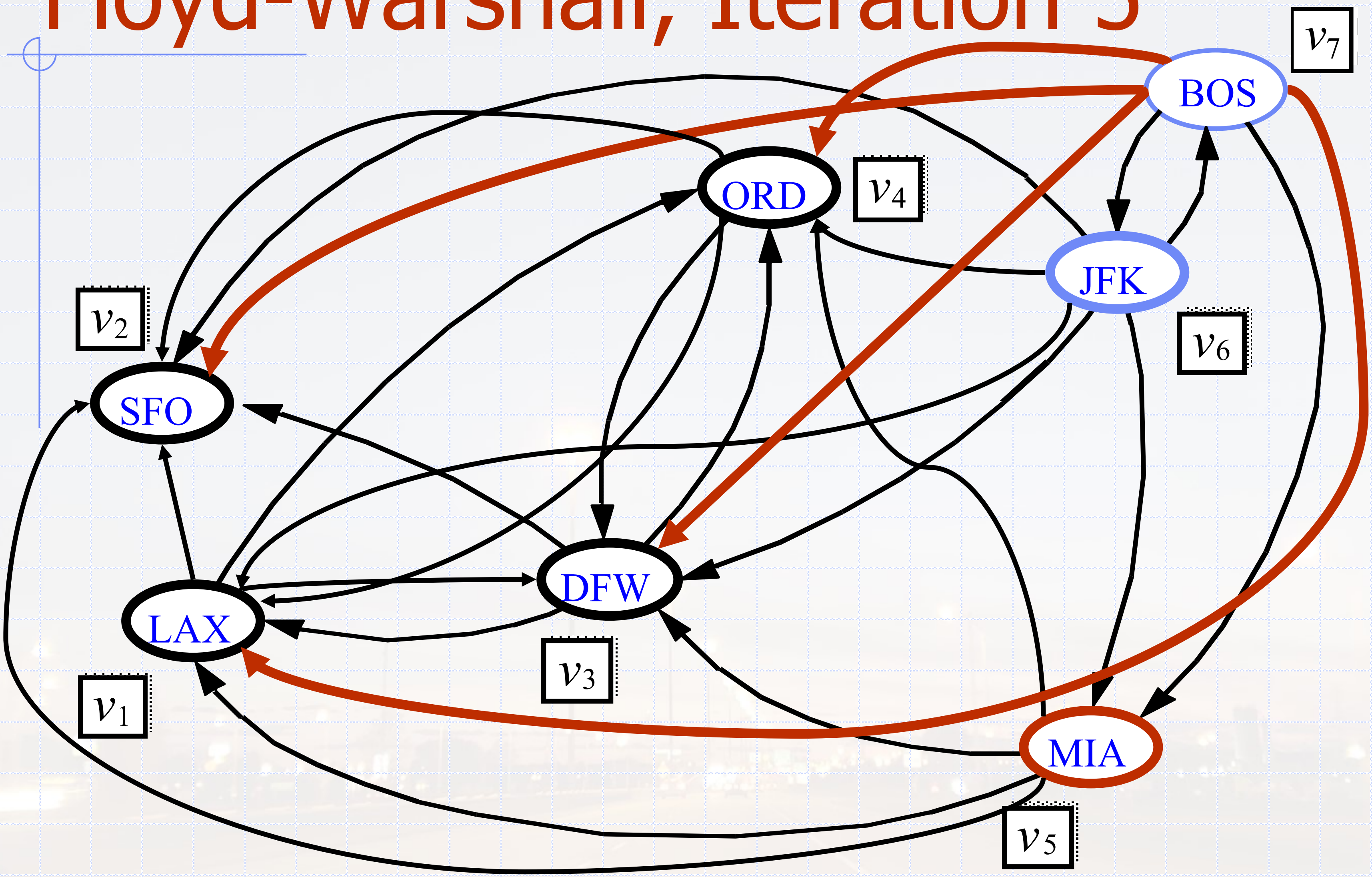
Floyd-Warshall, Iteration 3



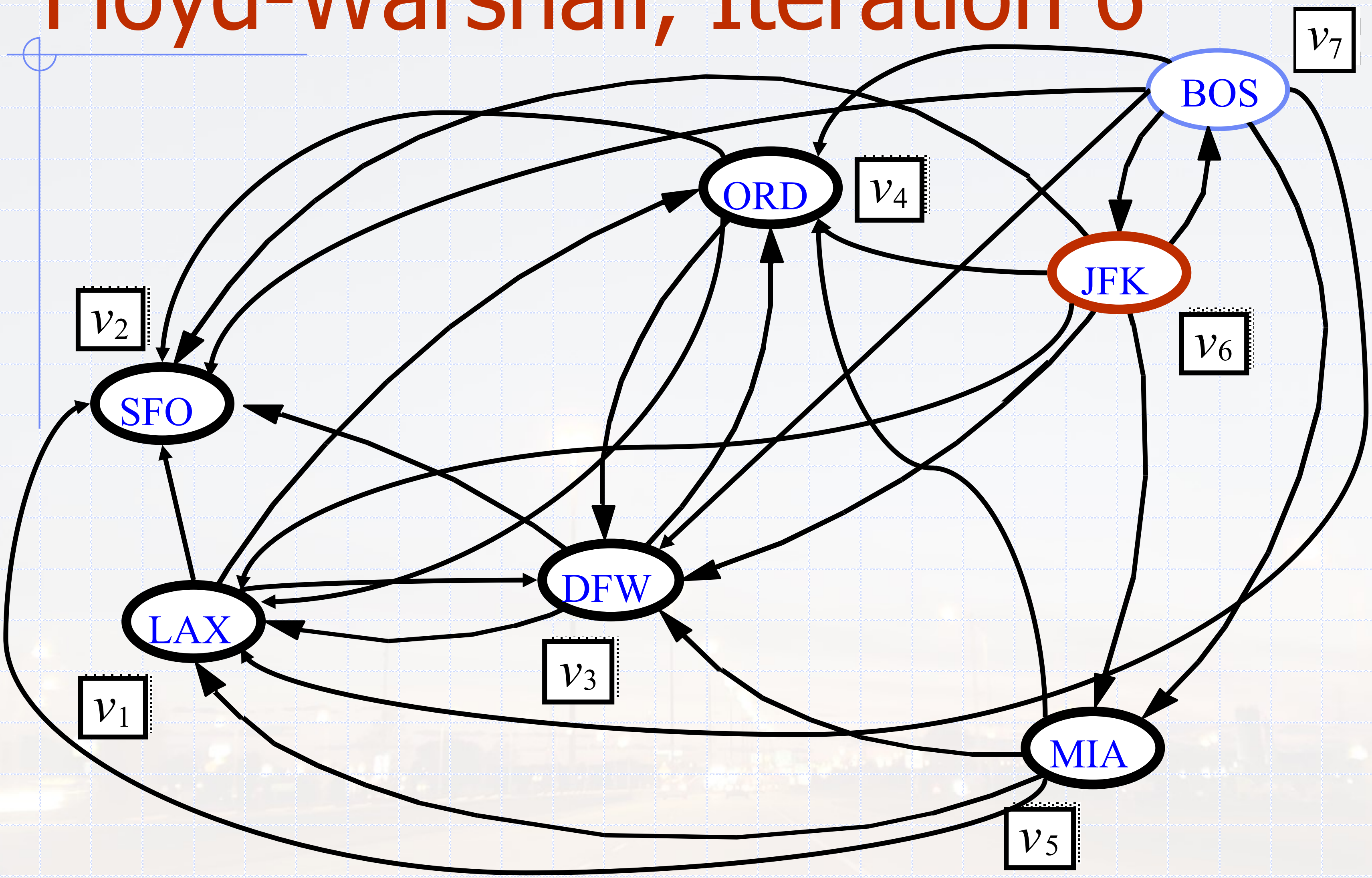
Floyd-Warshall, Iteration 4



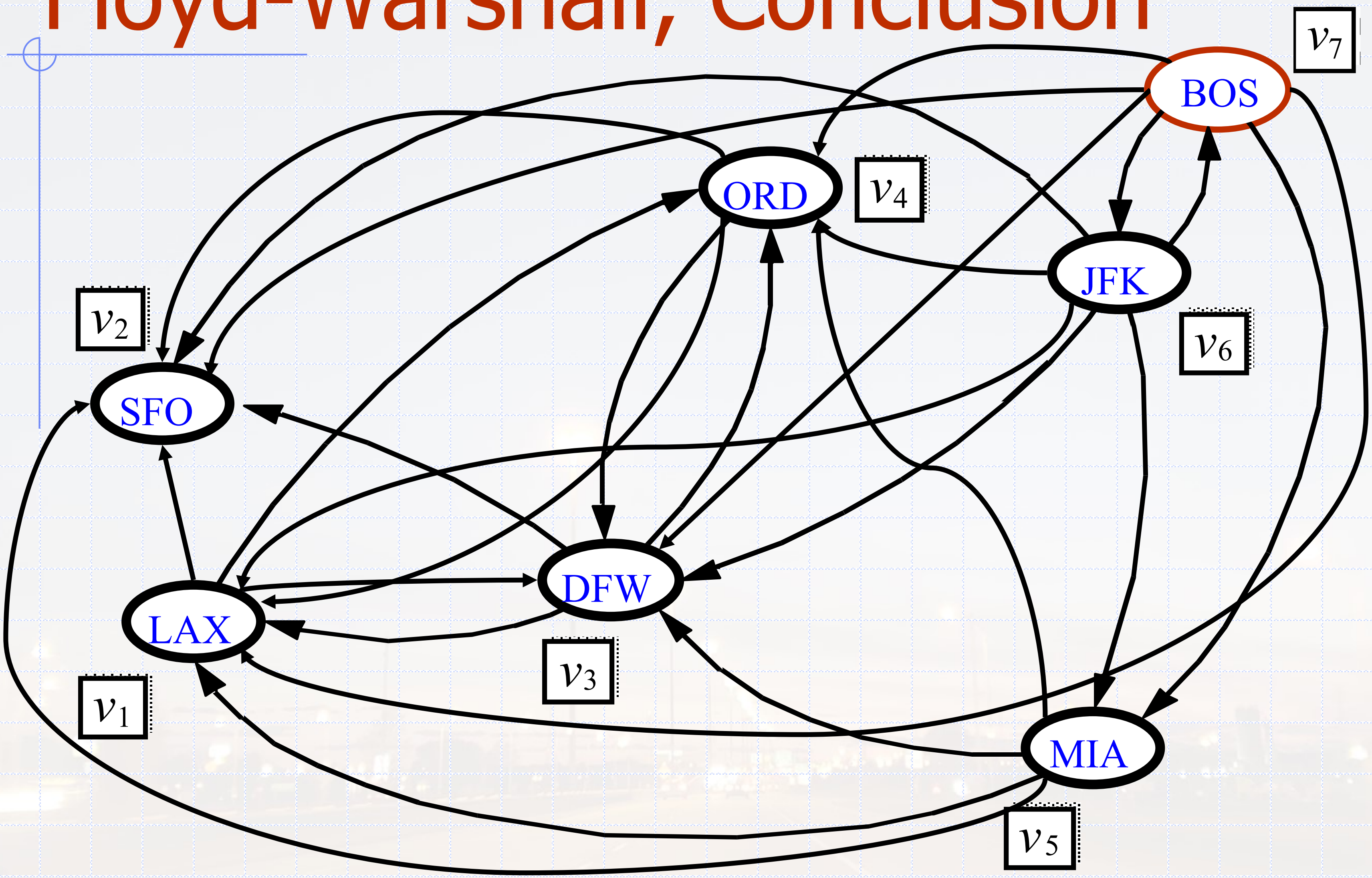
Floyd-Warshall, Iteration 5



Floyd-Warshall, Iteration 6



Floyd-Warshall, Conclusion



Directed Acyclic Graphs (DAGs)

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

- In practice, there are many applications of DAGs, including:
  - Inheritance relation between classes (i.e. in Java)
  - Prerequisites between courses in an academic program
  - Schedule constraints between tasks of a project (i.e. setup electric wires before testing lights, or fix walls in place before painting them)

DAGs and Topological Ordering

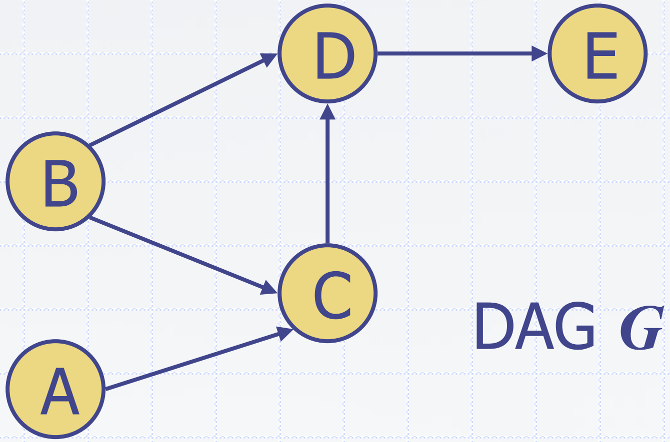
- Given a digraph G, a **topological ordering** of G is an ordering of

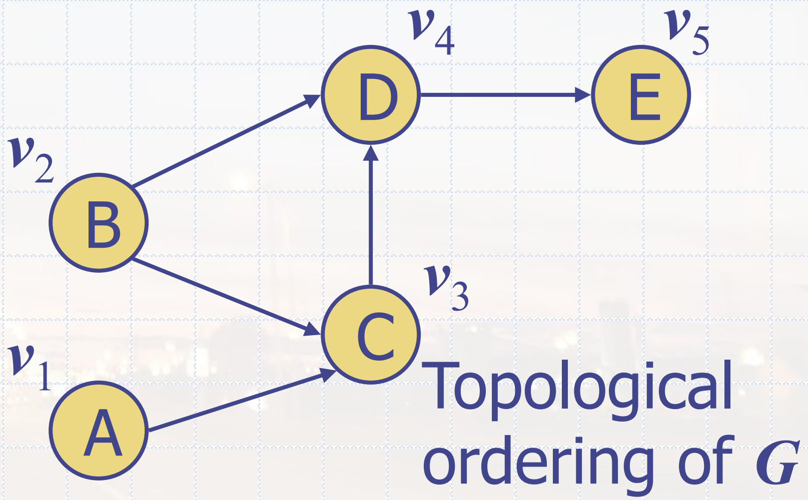
  $$v_1, \ldots, v_n$$

  of the vertices of G such that for every edge (v_i, v_j), we have $i < j$
- Example: Course v_1 must be taken before course v_3 and course v_2 must be taken before v_3 and v_4, and so on

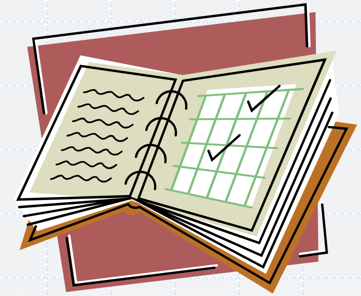
Theorem

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

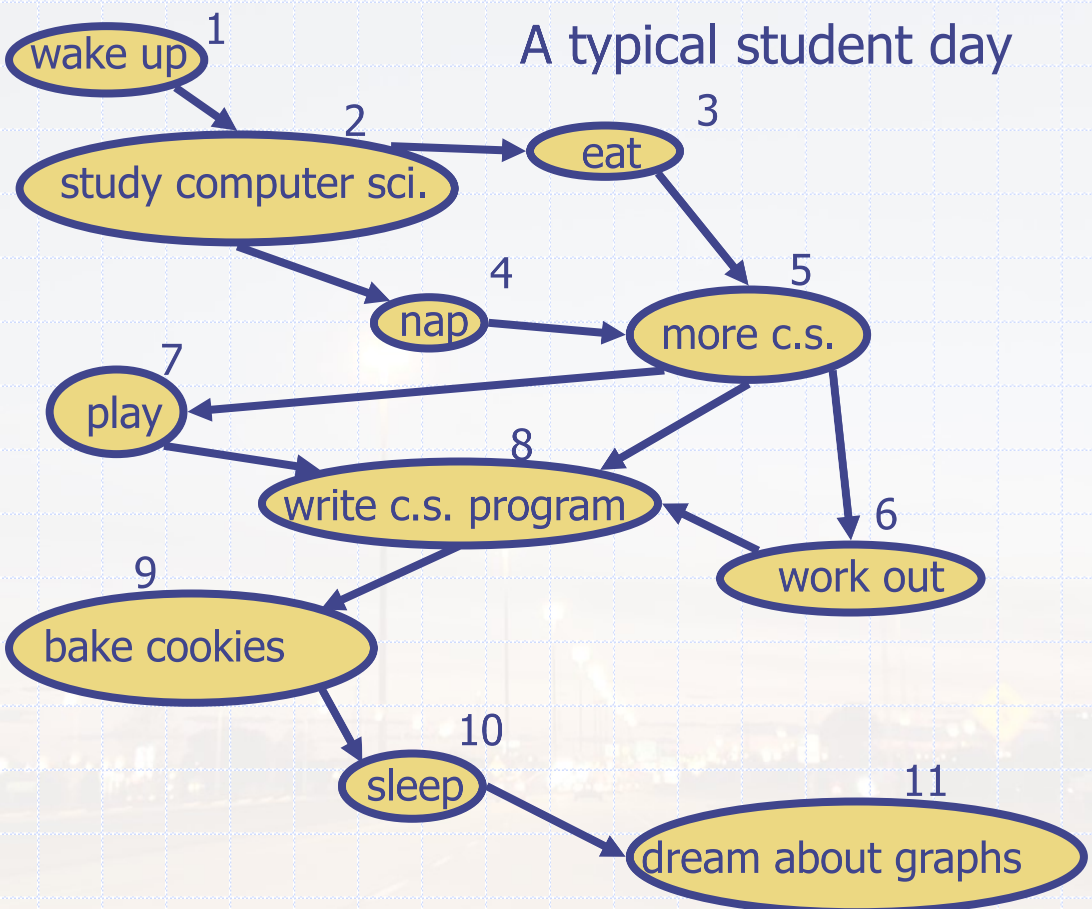




Topological Sorting



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

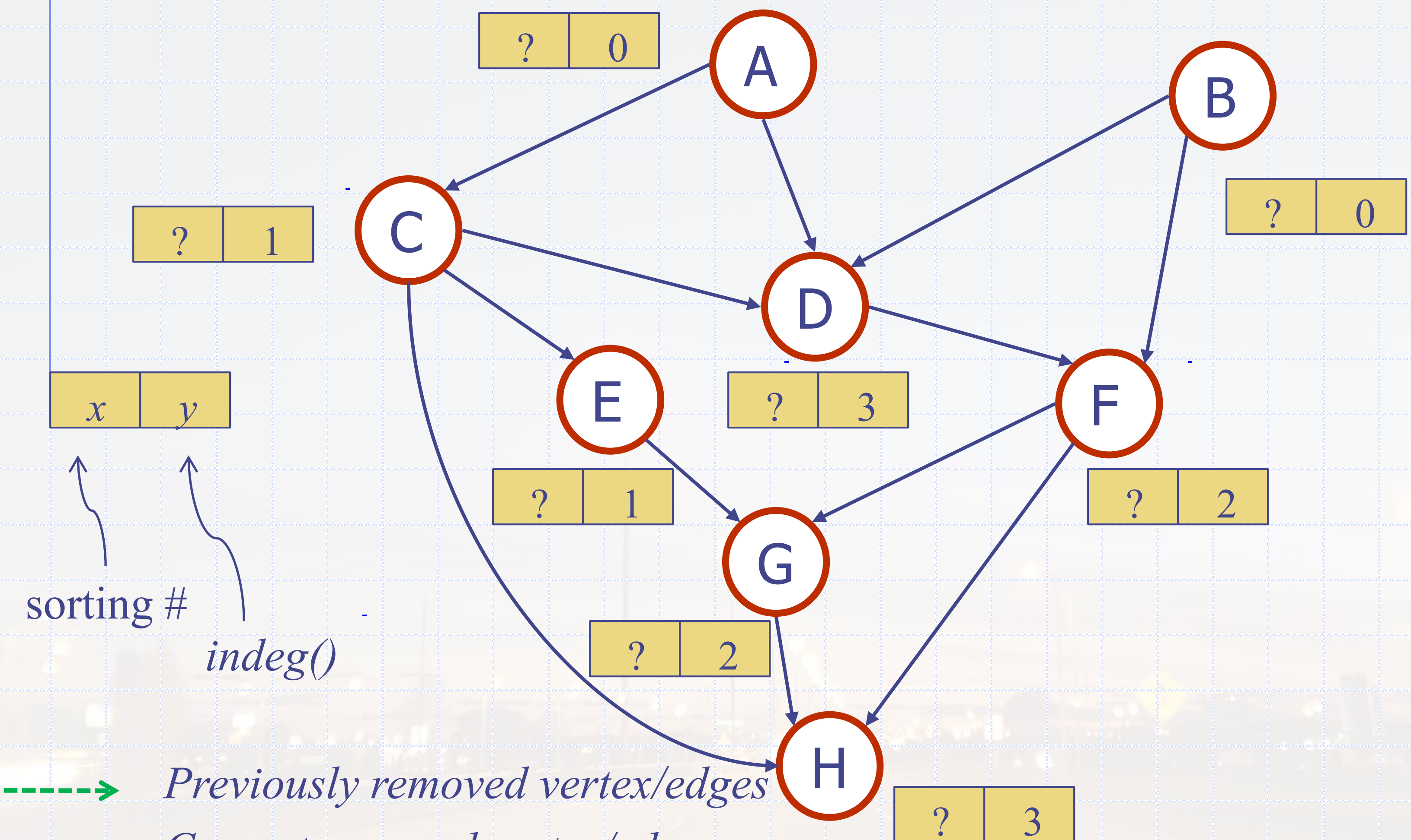


Algorithm for Topological Sorting

- Many algorithms can be used to calculate the topological ordering (to sort the vertices)
- One algorithm can be as follows: assume a DAG G:
  - Since G is acyclic then there must exist at least one vertex v such as v has no incoming edges ($indeg(v) = 0$)
  - If v is removed, then the resulting graph must still be acyclic, which means, there exist another vertex w in the remaining graph such as $indeg(w) = 0$
  - Give v sorting # 1, then remove it
  - Find another vertex w with $deg(w) = 0$, set its sorting # to 2 and remove
  - Repeat the above operations until all vertices are sorted
- Running time: $O(n + m)$
  - The algorithm traverses all the outgoing edges for each visited vertex once, so its running time is proportionate to the number of outgoing edges of the vertices

Topological Sorting Example

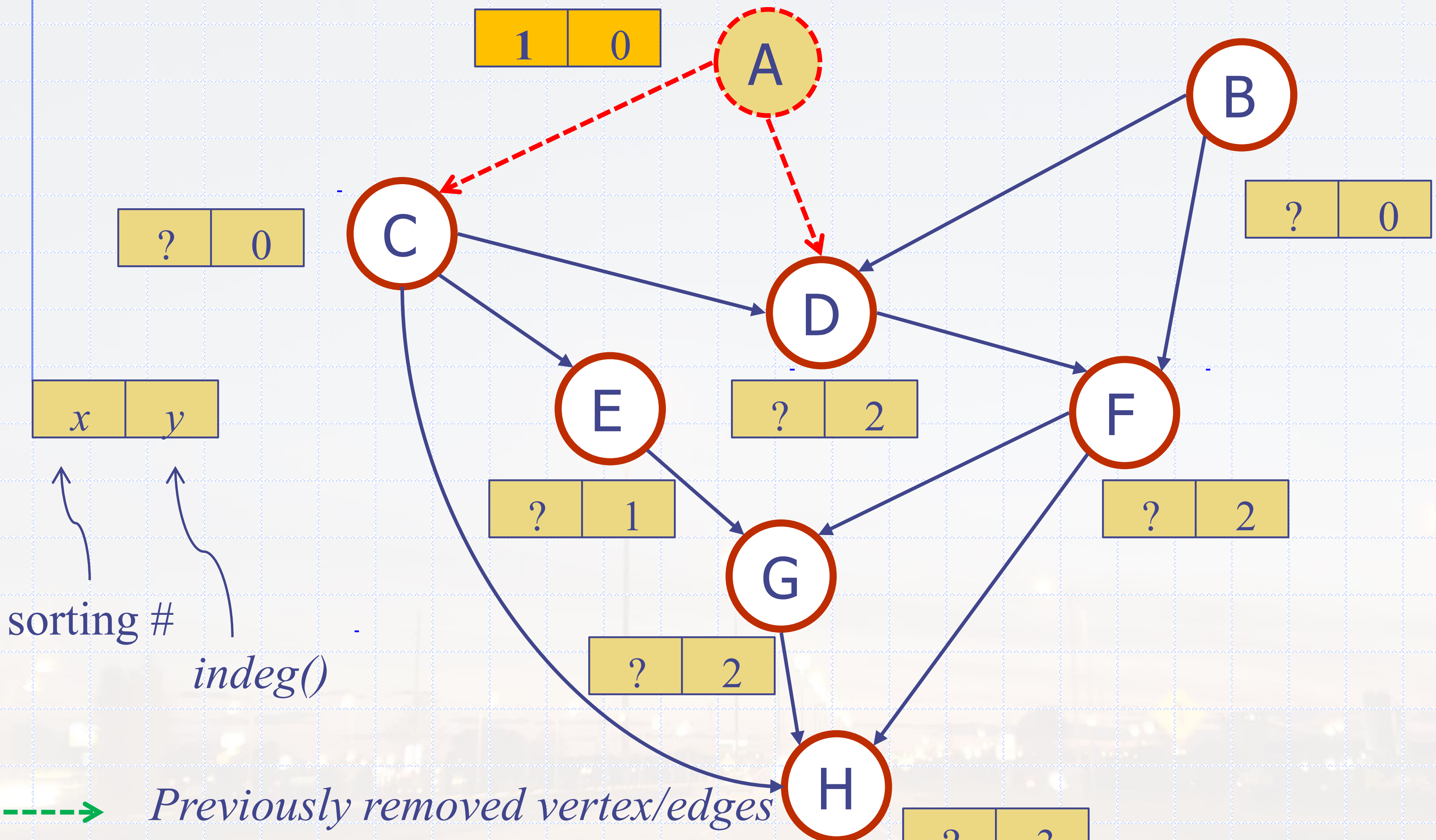
Note that there is more than one possible solution



Previously removed vertex/edges

Current removed vertex/edges

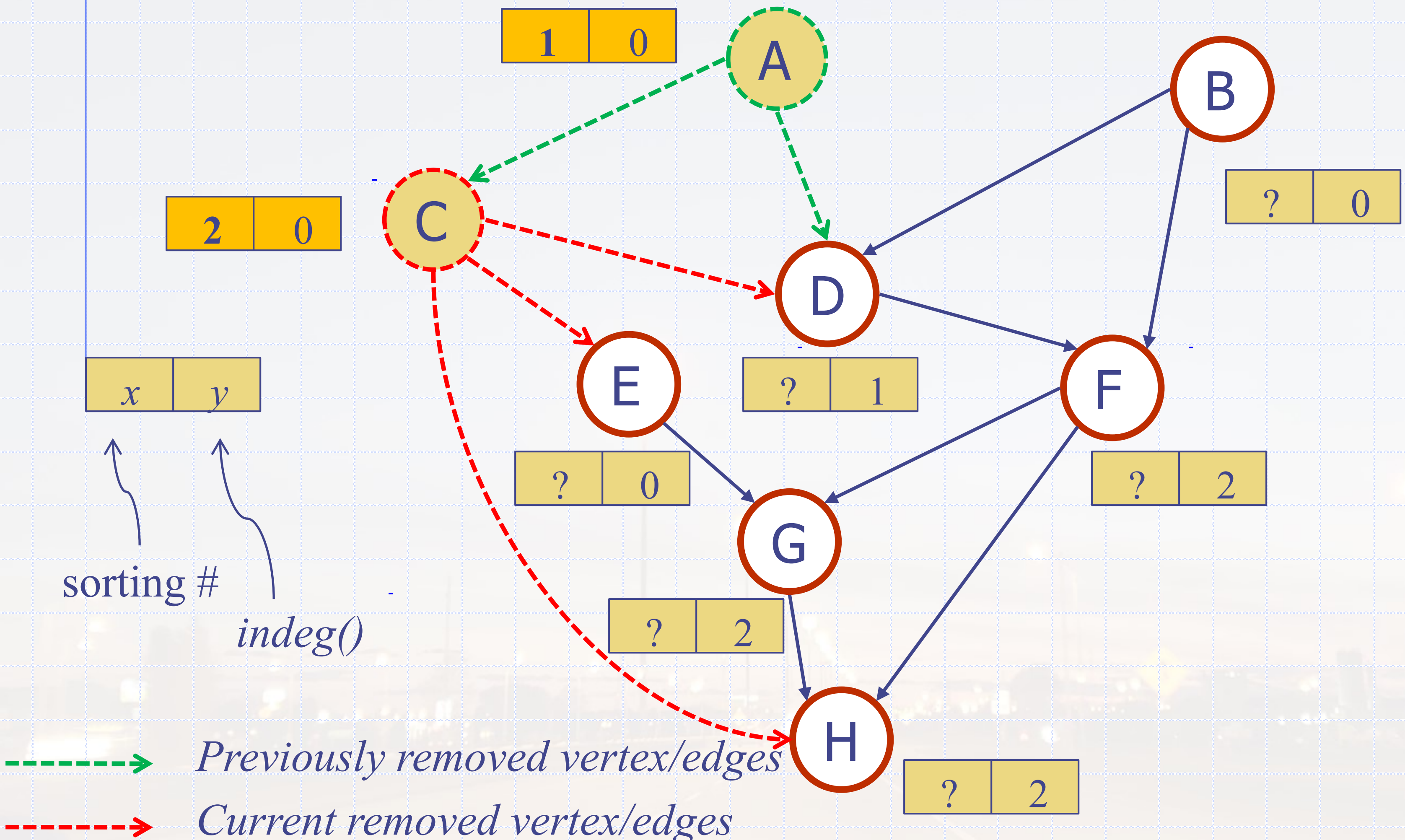
Topological Sorting Example (continues…)



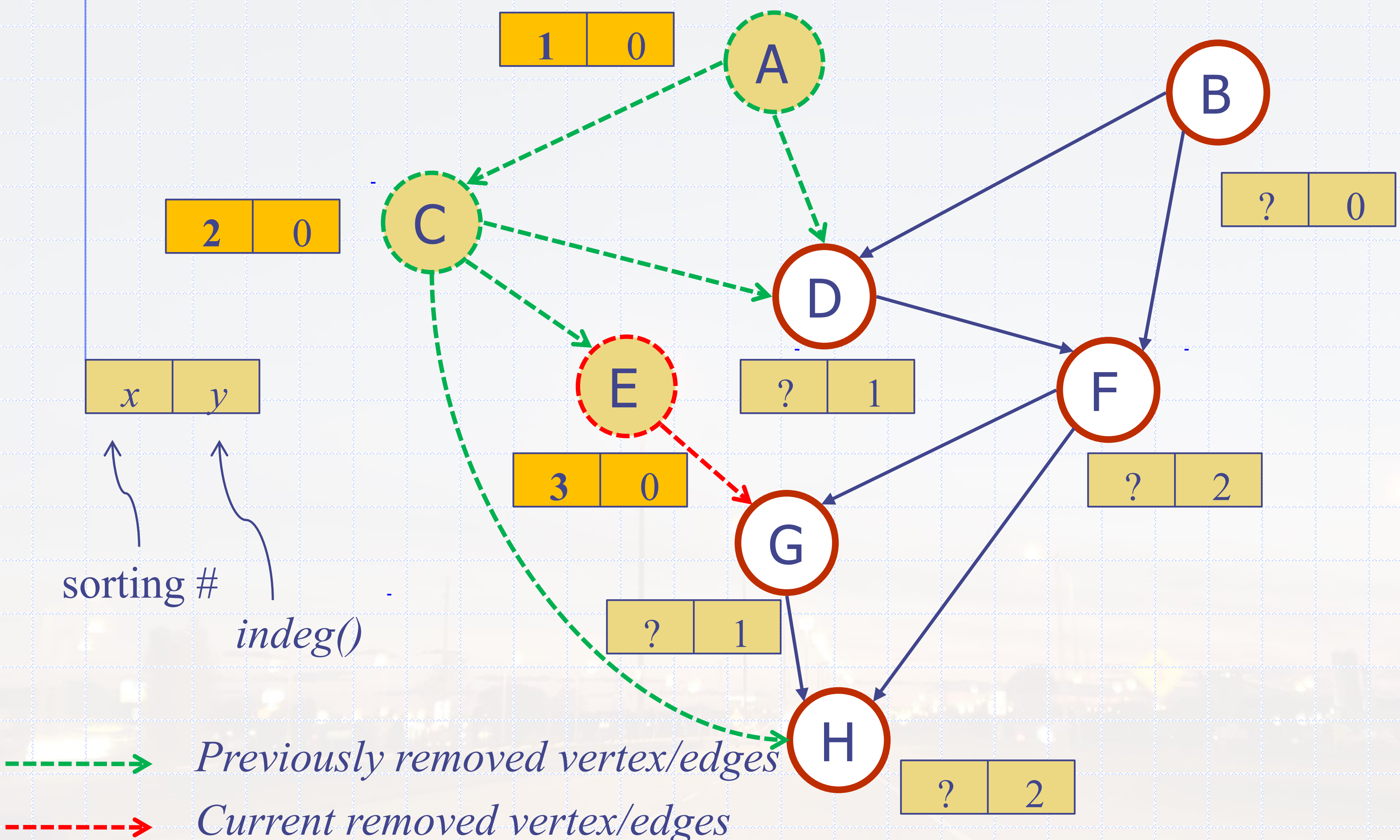
Previously removed vertex/edges

Current removed vertex/edges

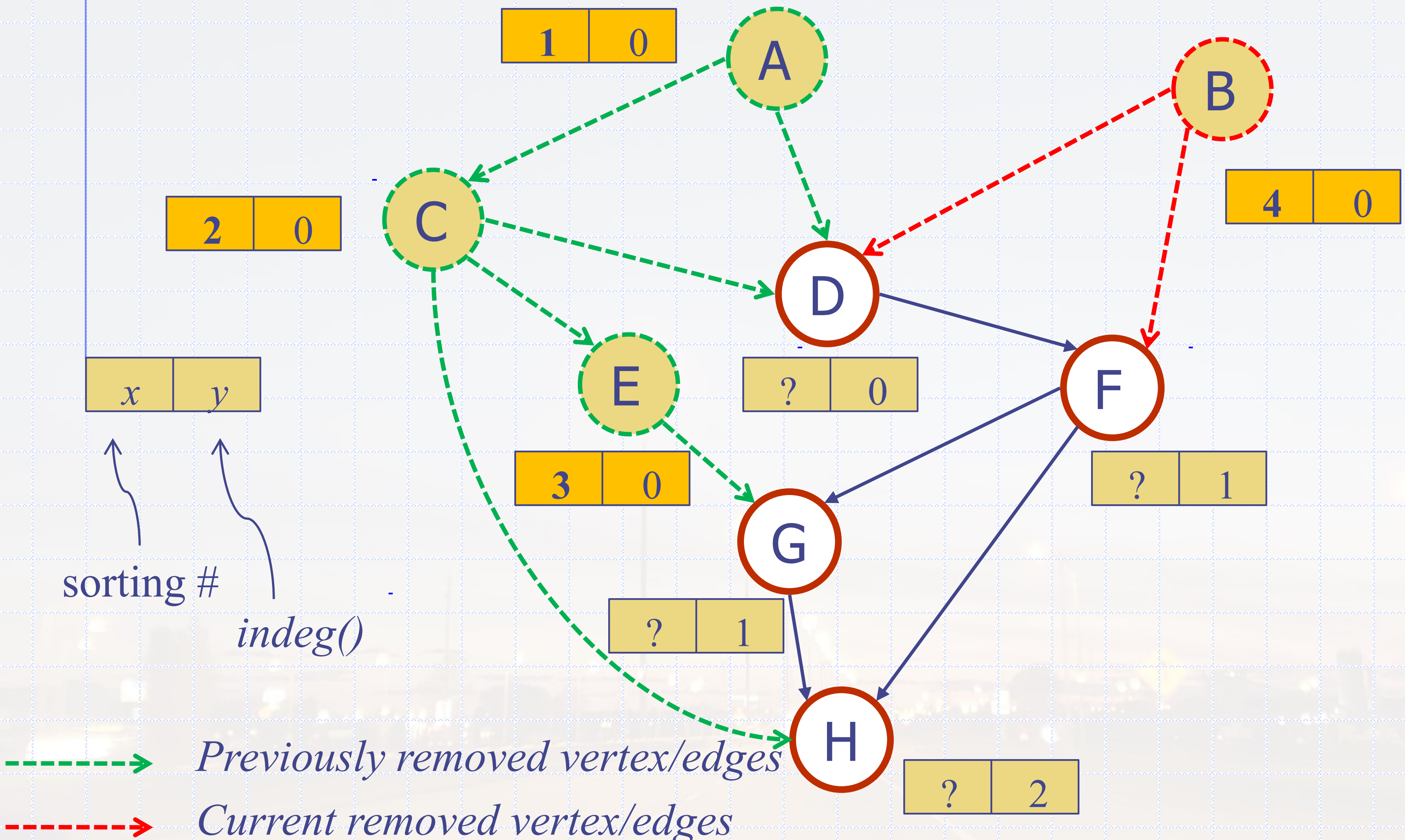
Topological Sorting Example (continues…)



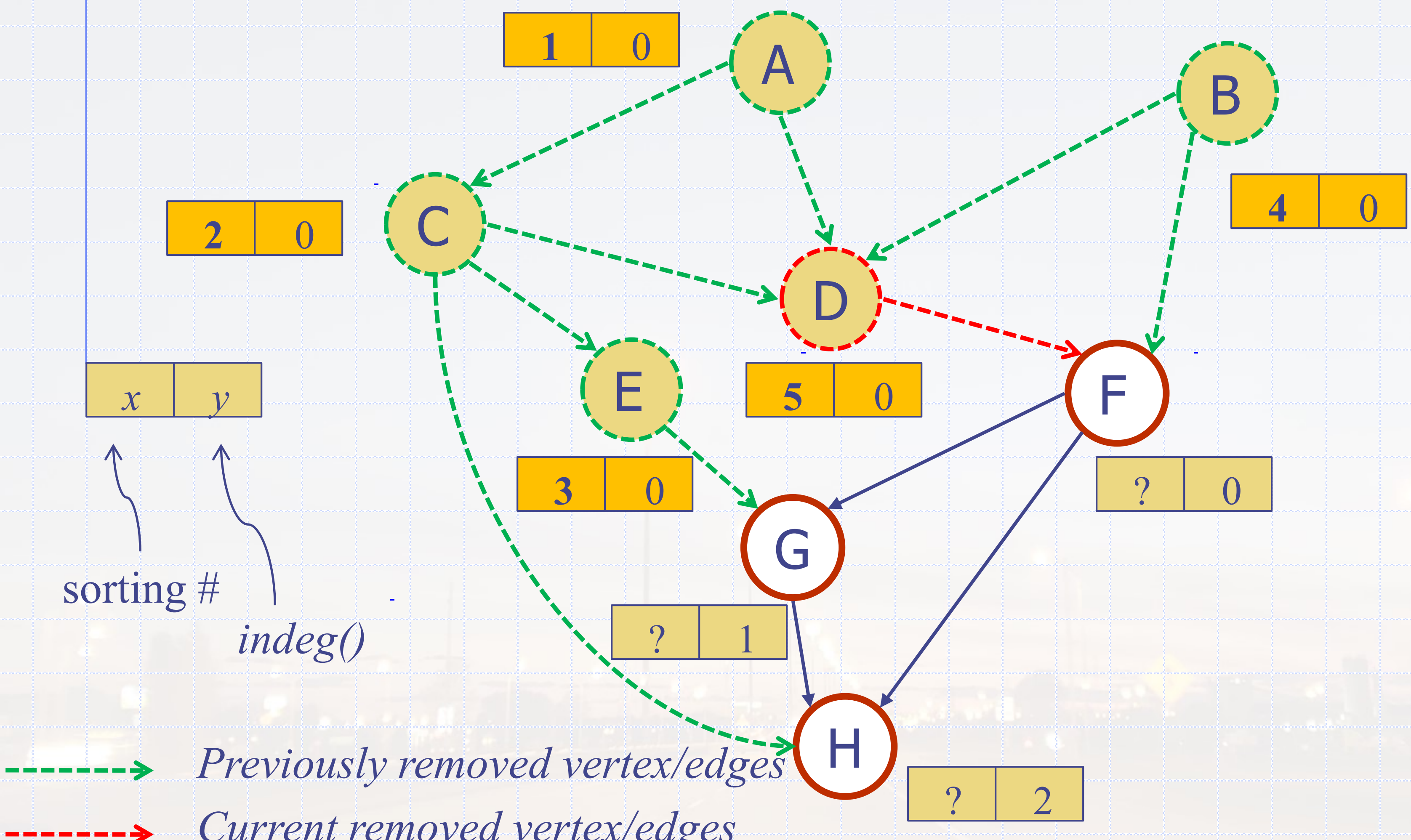
Topological Sorting Example (continues…)



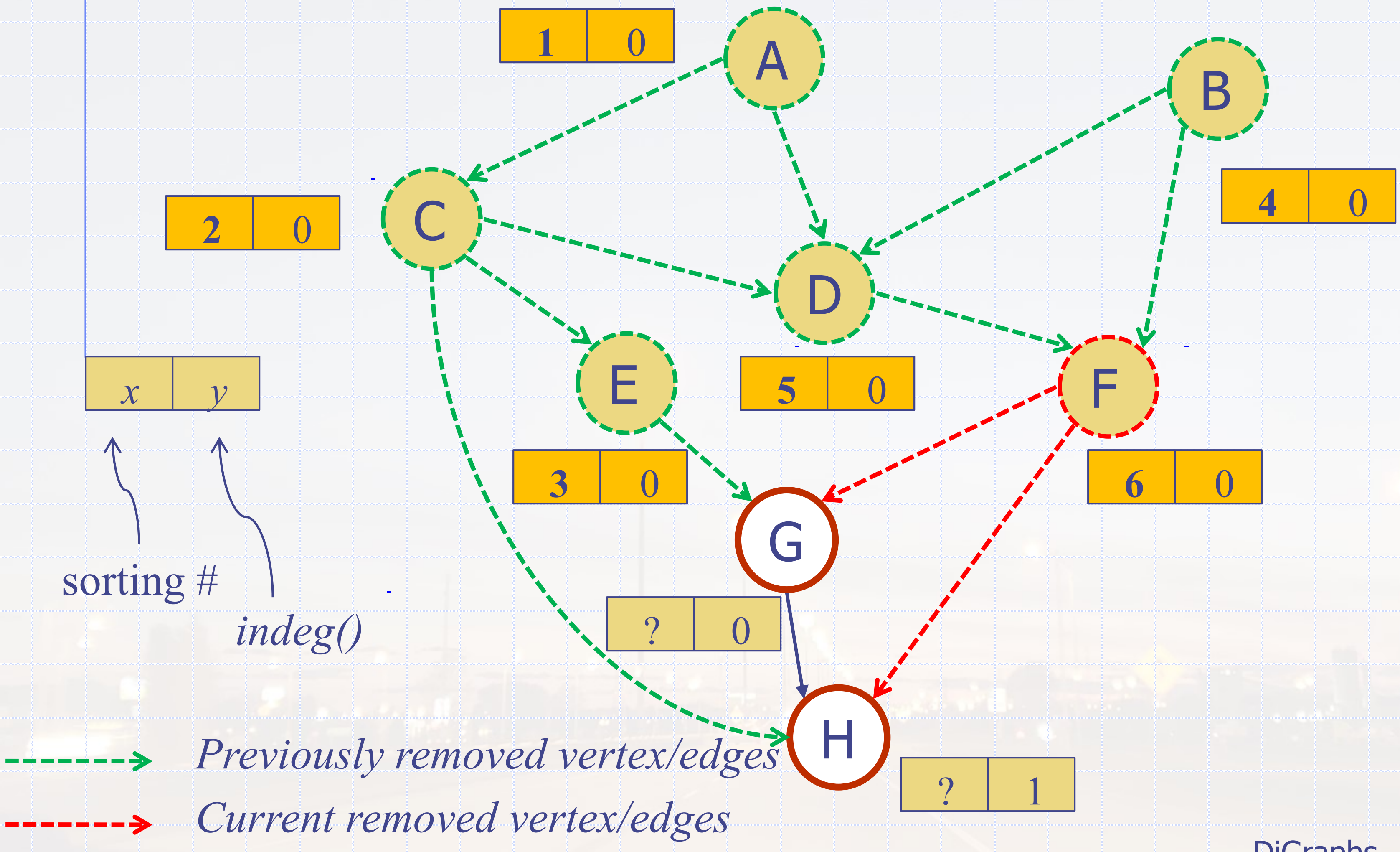
Topological Sorting Example (continues...)



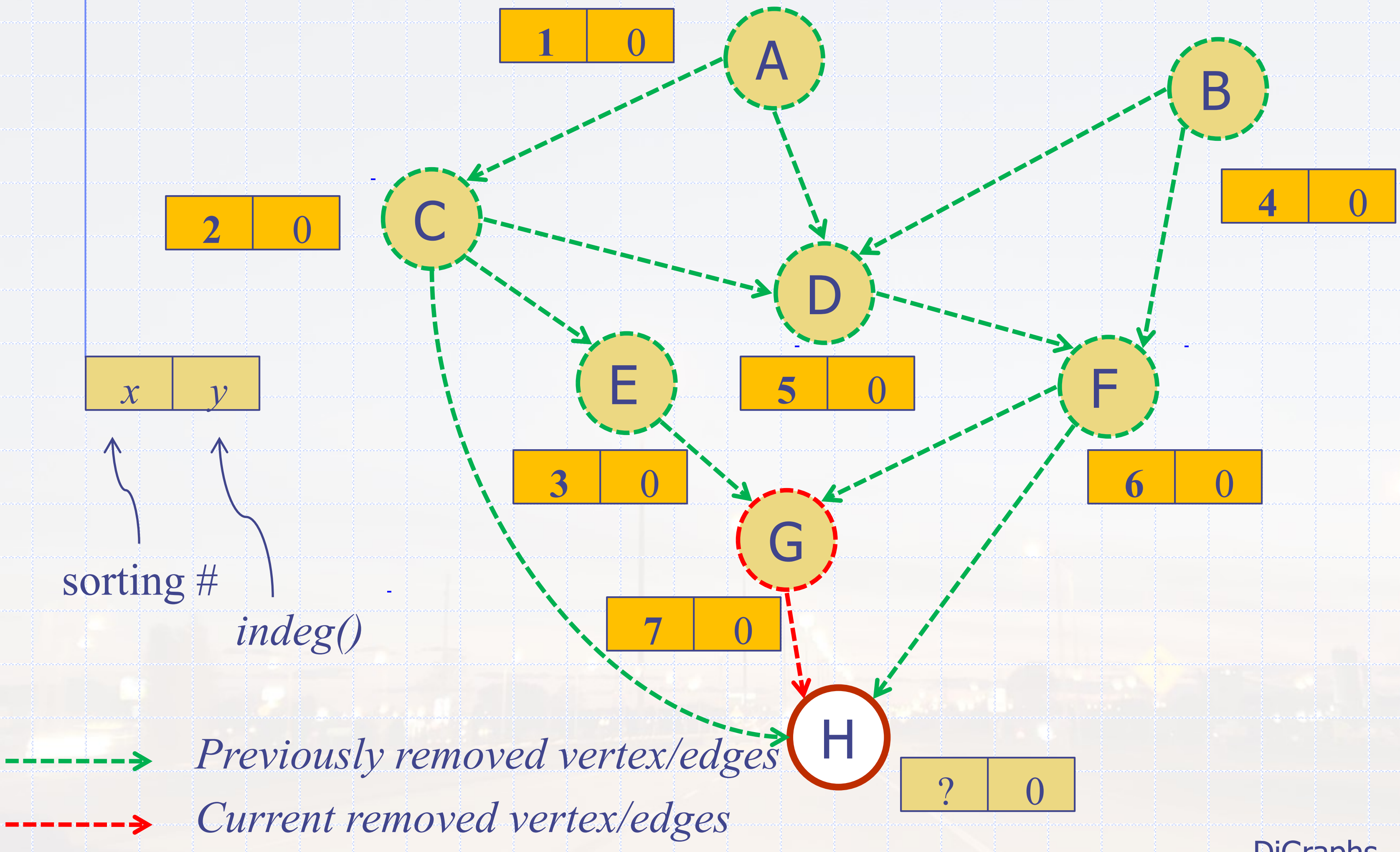
Topological Sorting Example (continues…)



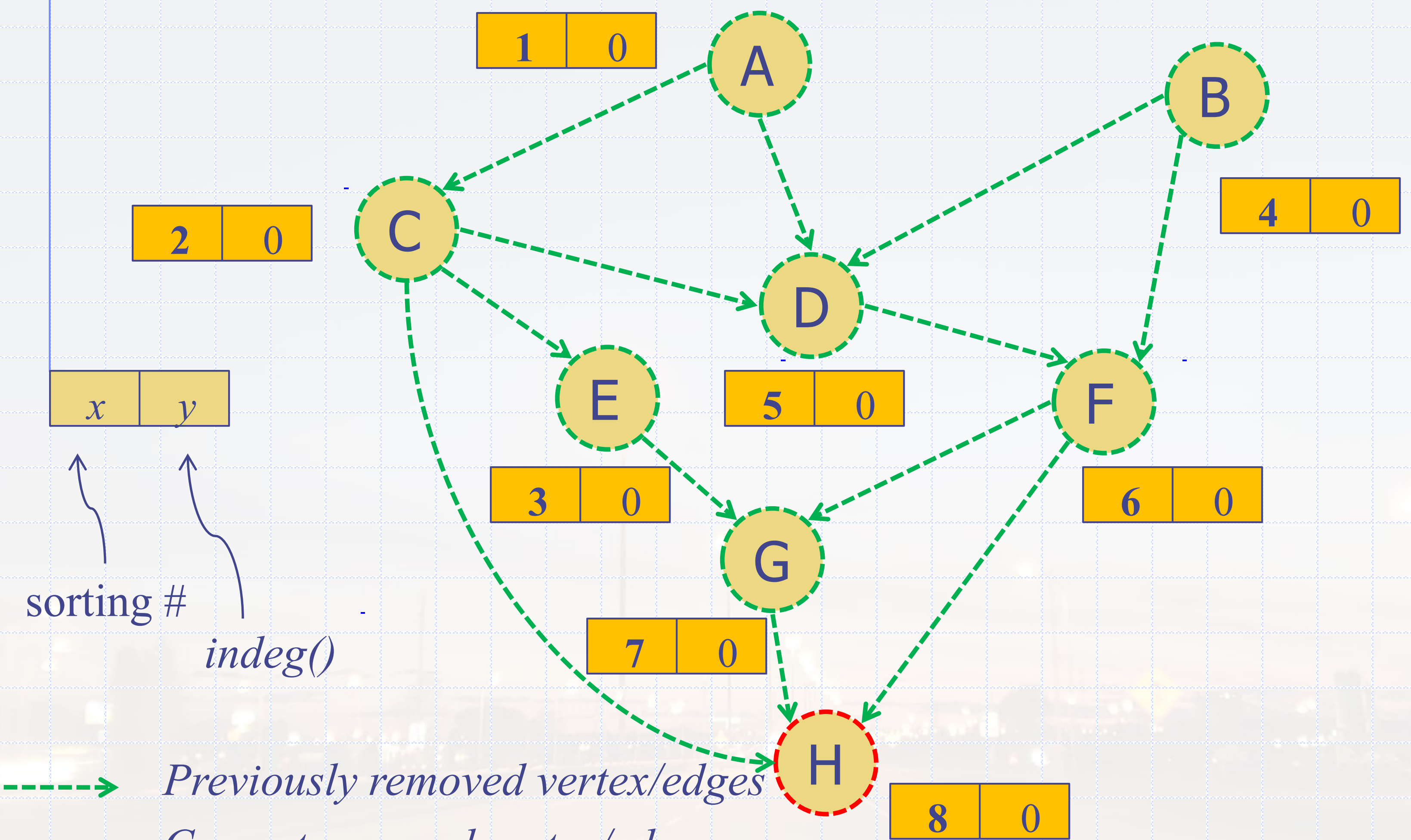
Topological Sorting Example (continues…)



Topological Sorting Example (continues…)



Topological Sorting Example (continues…)



Previously removed vertex/edges

Current removed vertex/edges

Other Algorithms for Topological Sorting

- Work it the other way around:
  - find a vertex with no outgoing edge ($outdeg(v) = 0$) and give it # n, then remove it
  - find another one w with ($outdeg(w) = 0$) and give it # n-1, then remove it
  - Repeat until all vertices are sorted
- Running time is still $O(n + m)$

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***

- *Note: This algorithm is different than the one in the book*

Implementation with DFS

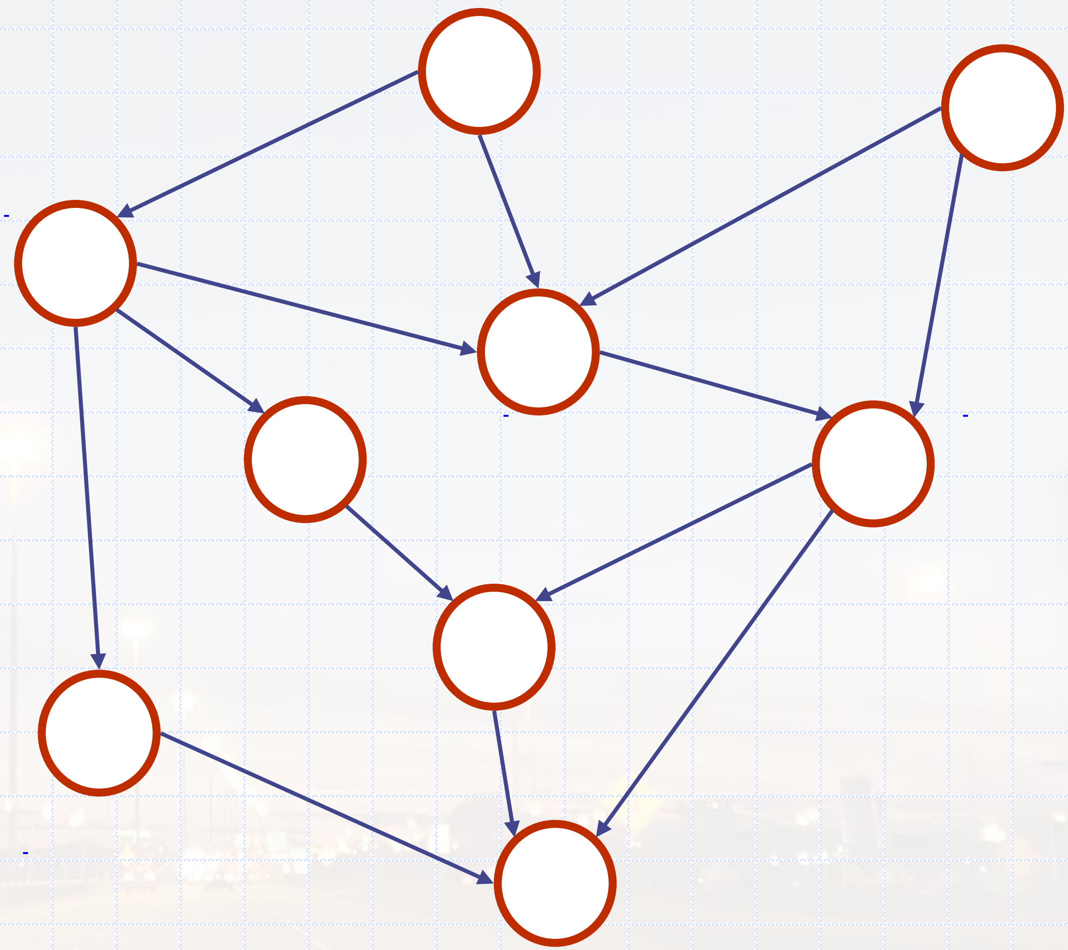
- Simulate the algorithm by using depth-first search
- O(n+m) time.

Algorithm *topologicalDFS*(*G*)
  Input dag *G*
  Output topological ordering of *G*
  $n \leftarrow$ *G.numVertices*()
  for all $u \in$ *G.vertices*()
    setLabel(*u, UNEXPLORED*)
  for all $v \in$ *G.vertices*()
    if *getLabel*(*v*) = *UNEXPLORED*
      topologicalDFS(*G, v*)

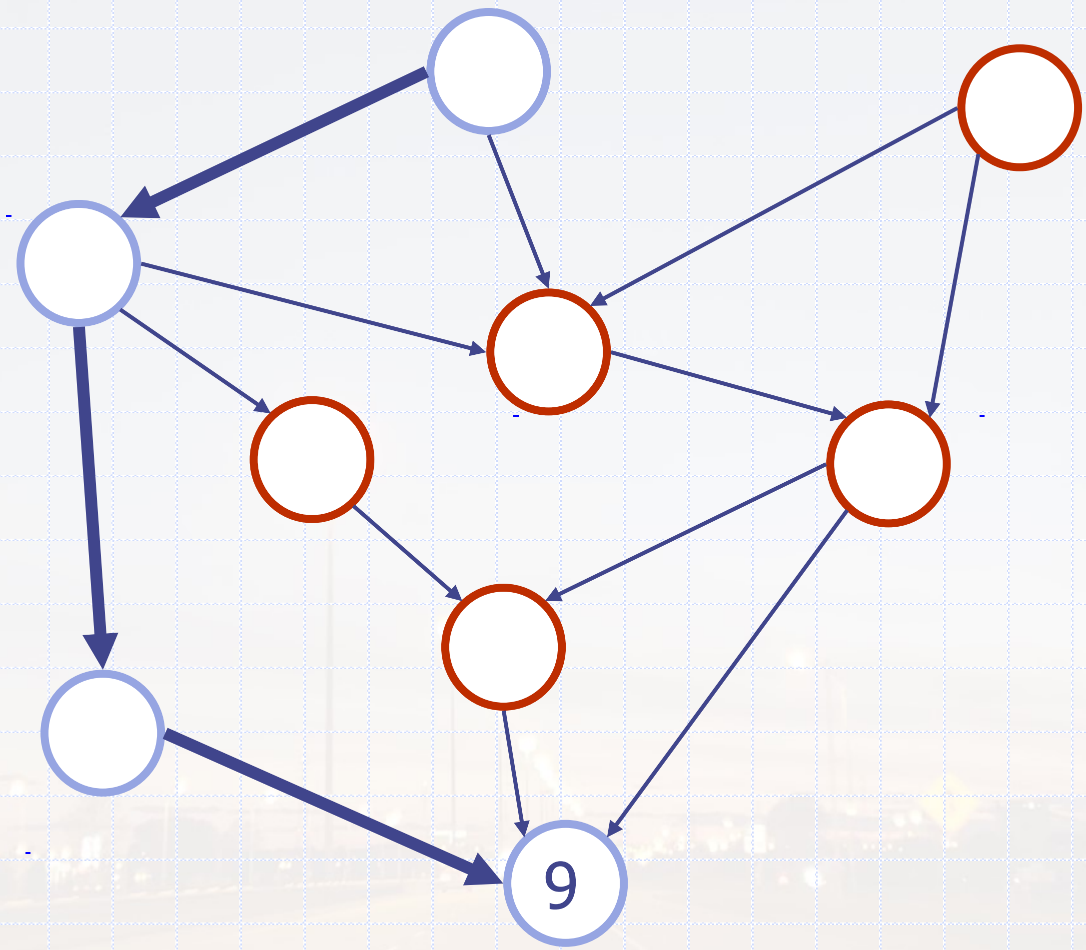
Algorithm *topologicalDFS*(*G, v*)
  Input graph *G* and a start vertex *v* of *G*
  Output labeling of the vertices of *G* in the connected component of *v*
  setLabel(*v, VISITED*)
  for all $e \in$ *G.outEdges*(*v*)
    { outgoing edges }
    $w \leftarrow$ *opposite*(*v,e*)
    if *getLabel*(*w*)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

DFS Topological Sorting Example

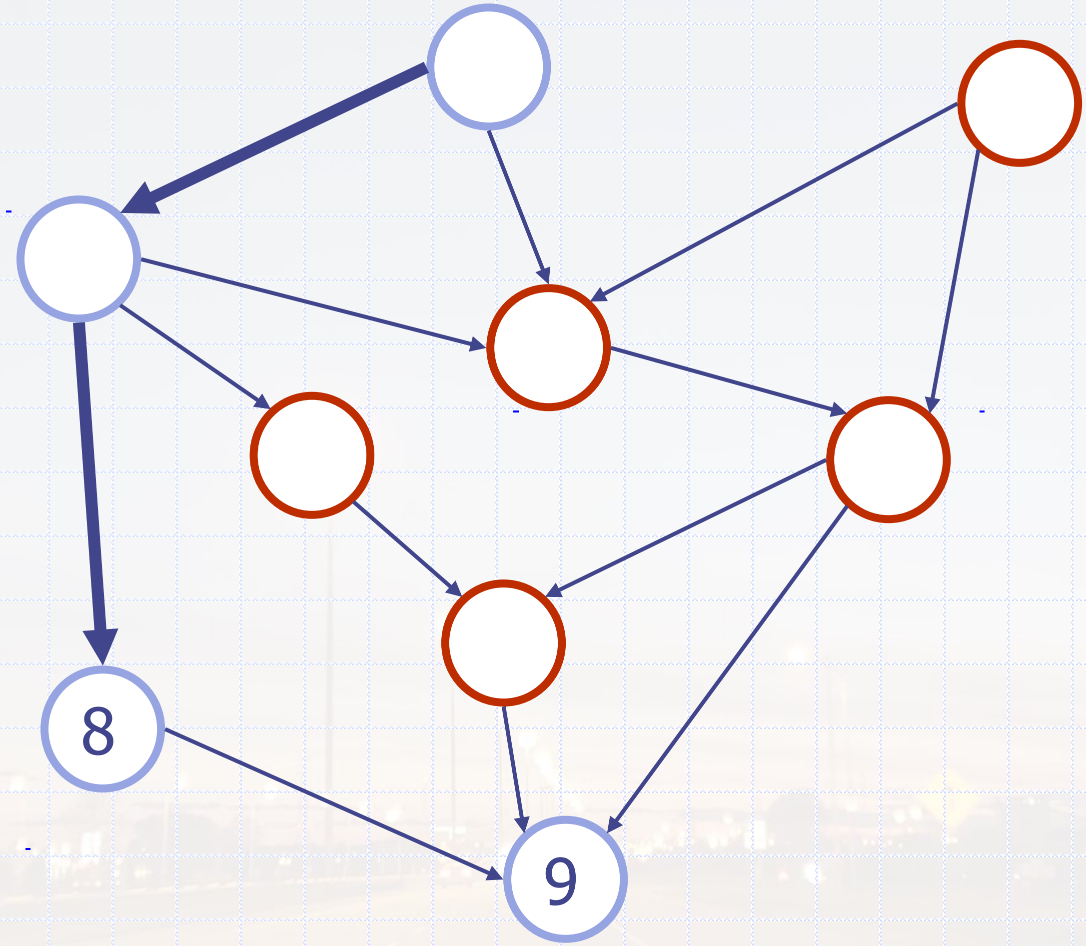
- The sorting # will always be assigned to a vertex just before the vertex rolls back to its previous vertex
- Notice however that the algorithm applies DFS also starting from each of the vertices that have only outgoing edges on the graph, which is important to guarantee all vertices will get their numbering
  - i,e. See how the sorting number for the vertex with final assigned # 1 is given



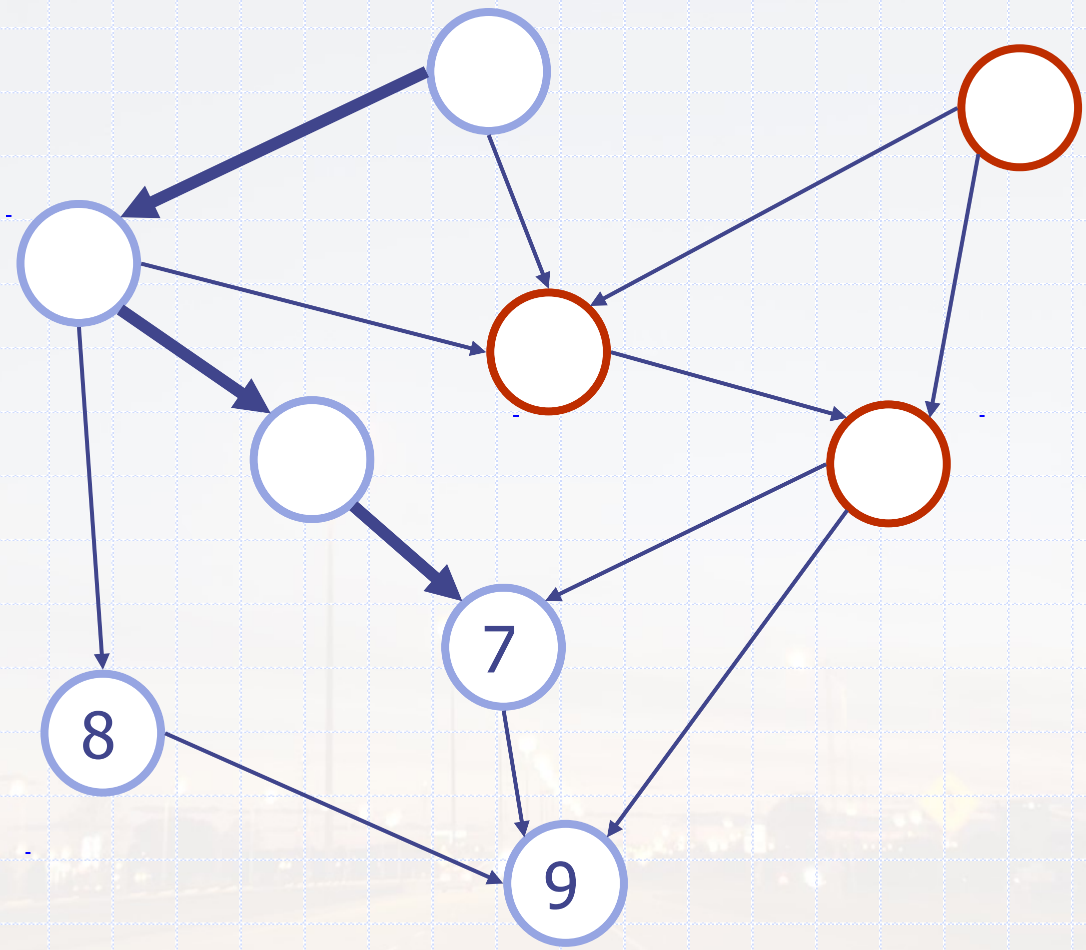
DFS Topological Sorting Example



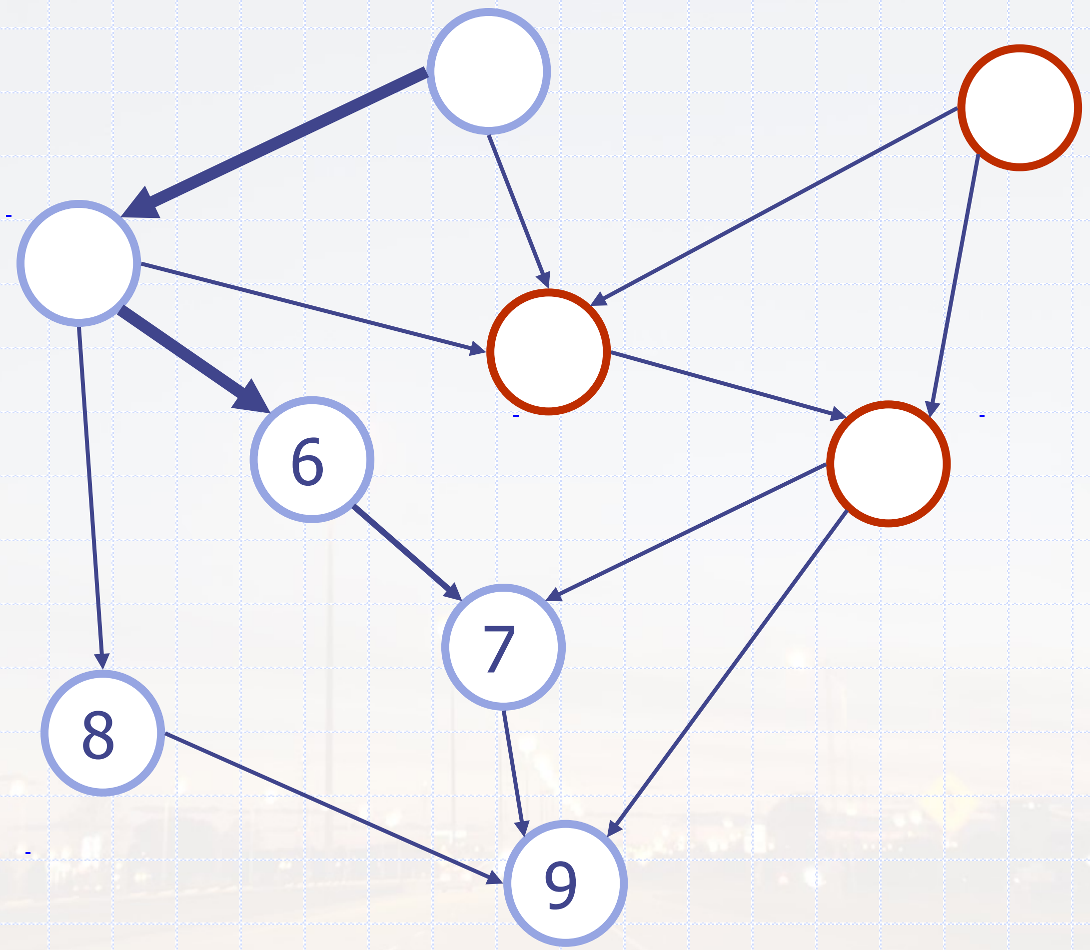
DFS Topological Sorting Example



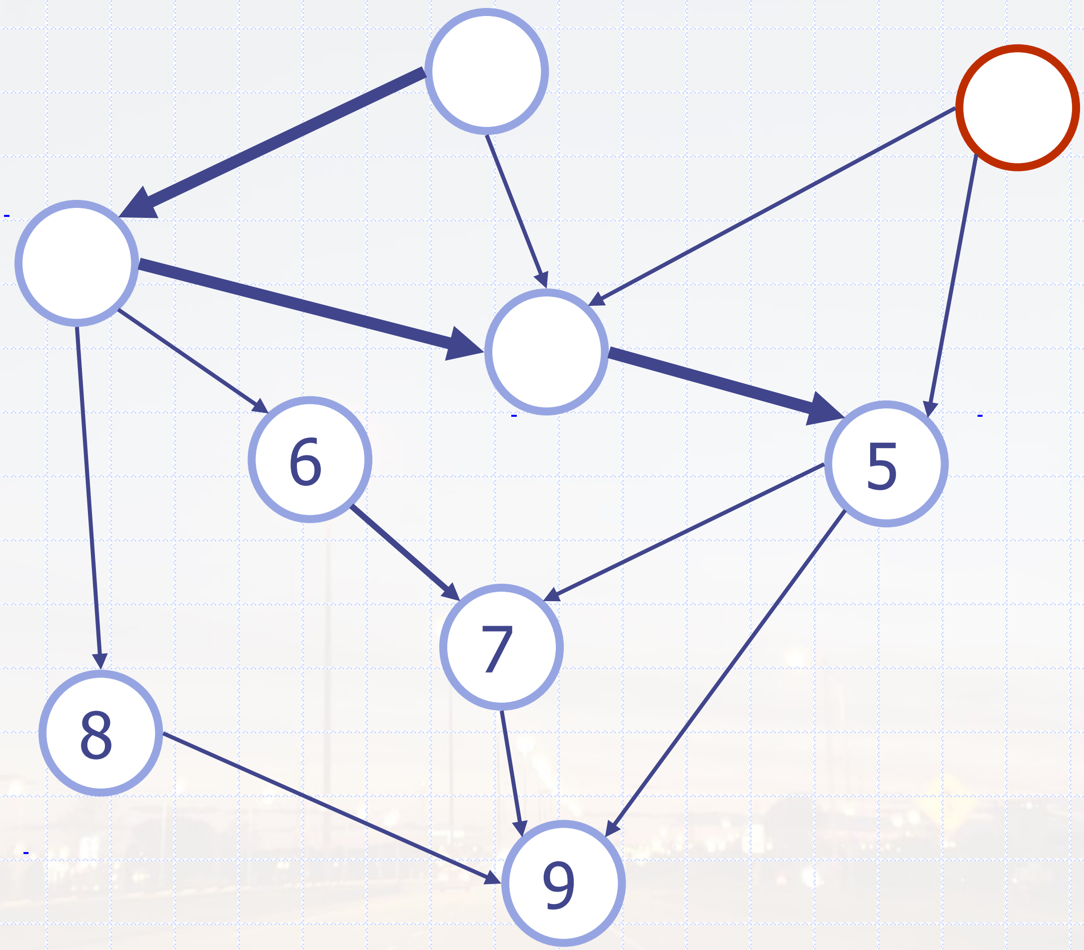
DFS Topological Sorting Example



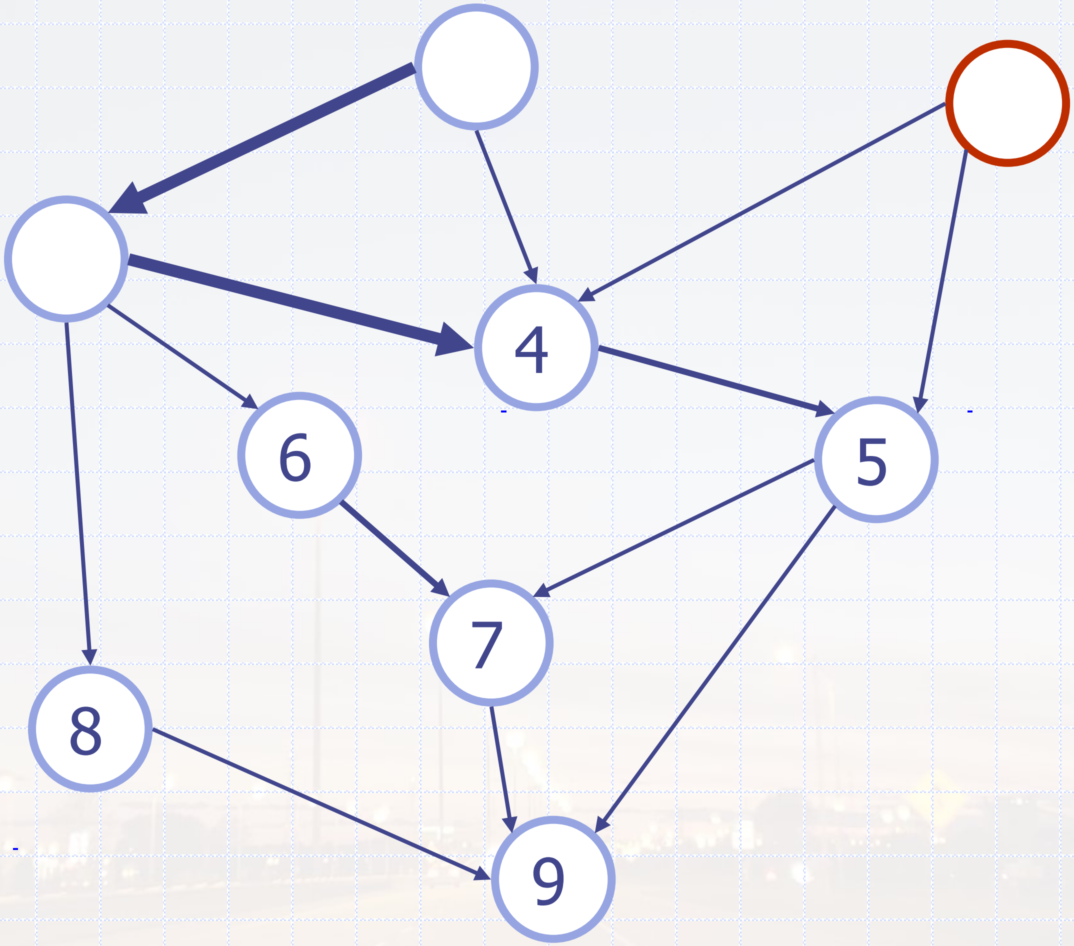
DFS Topological Sorting Example



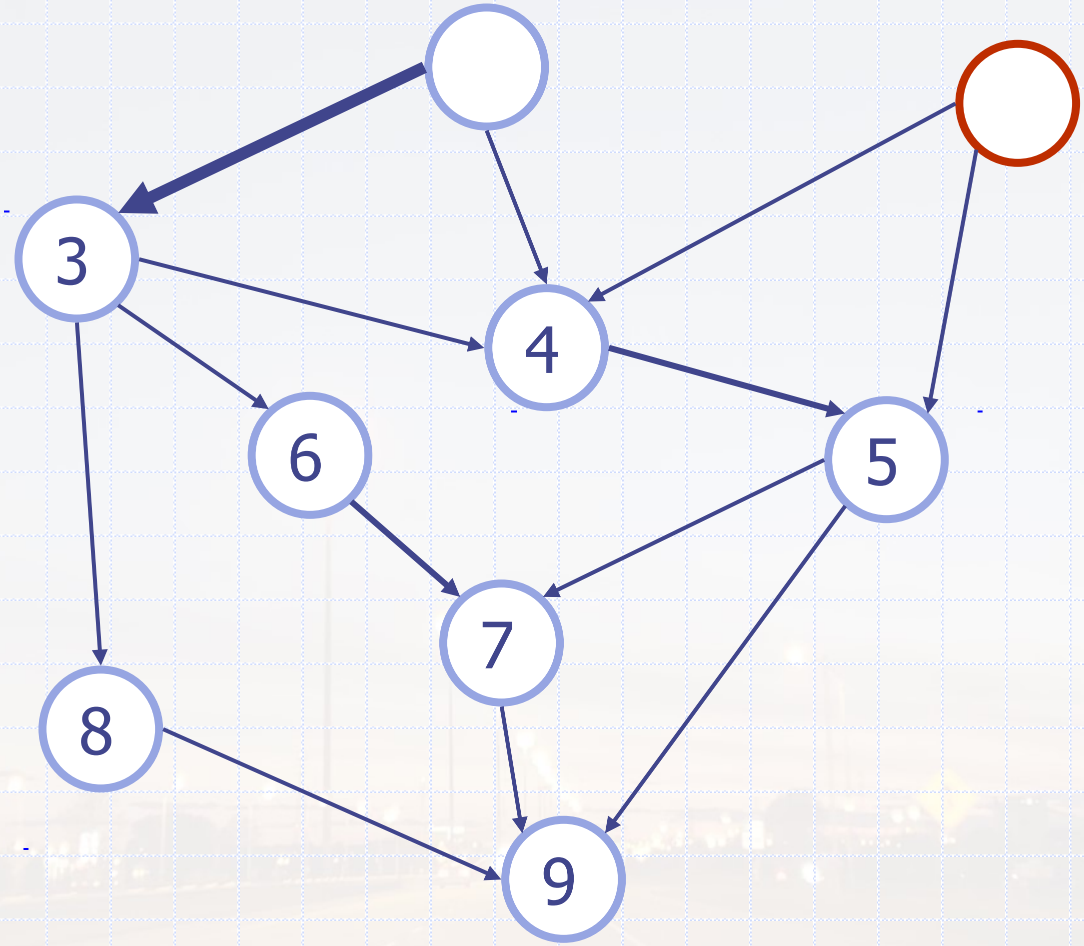
DFS Topological Sorting Example



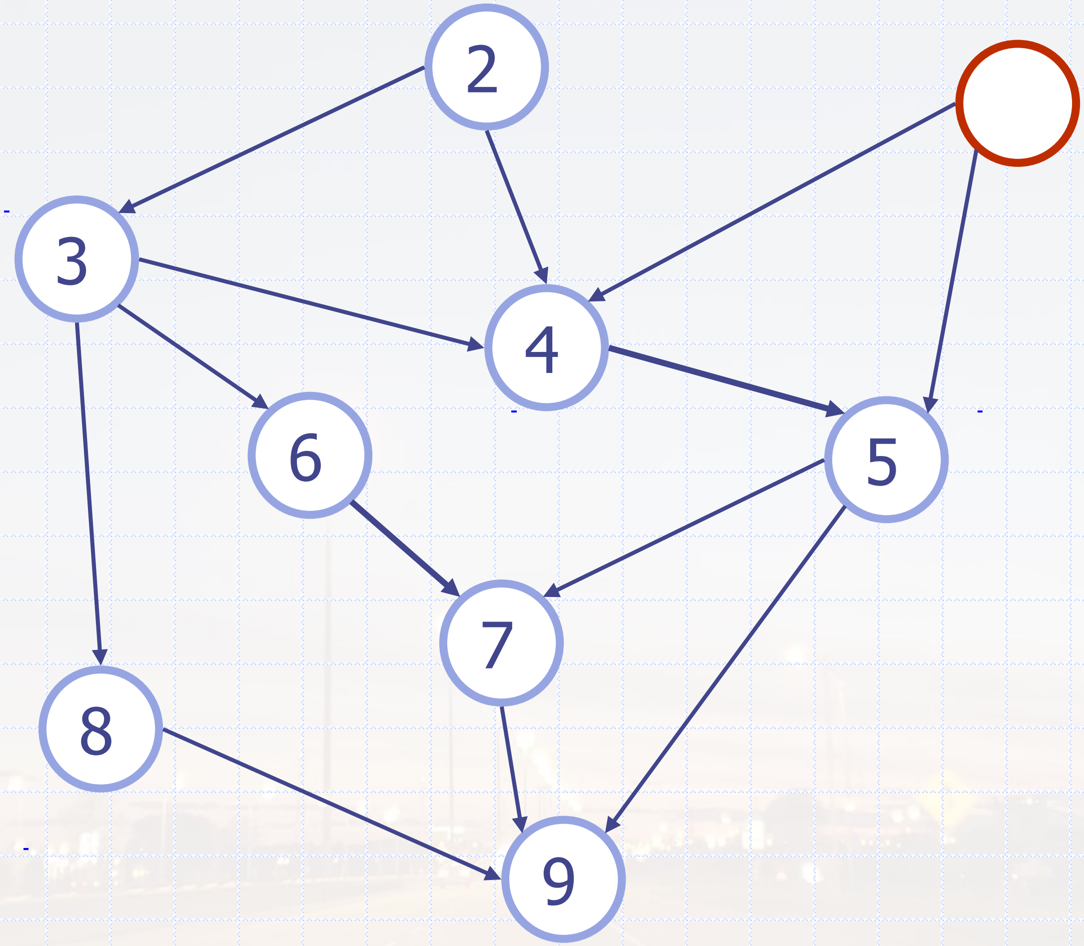
DFS Topological Sorting Example



DFS Topological Sorting Example



DFS Topological Sorting Example



DFS Topological Sorting Example