COMP9024: Data Structures and Algorithms

Graphs (II)

1

1

Contents

- Depth-First Search
- Breadth-First Search
- Transitive Closure
- Topological Sorting

2

2

Properties

Property 1

$\sum_v \deg(v) = 2m$

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges

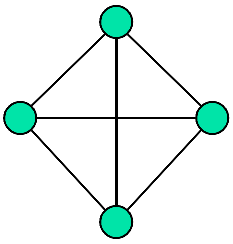
$m \leq n\,(n-1)/2$

Proof: each vertex has degree at most $(n-1)$

What is the bound for a directed graph?

Notation

n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v



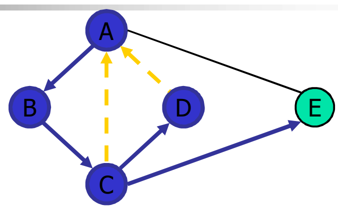
Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

3

3

Depth-First Search

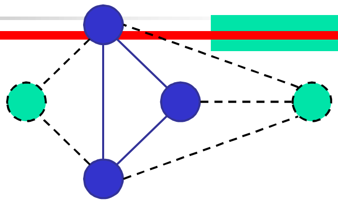


4

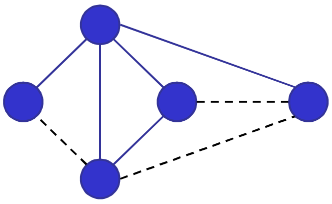
4

Subgraphs

- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G



Subgraph



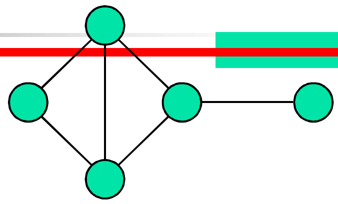
Spanning subgraph

5

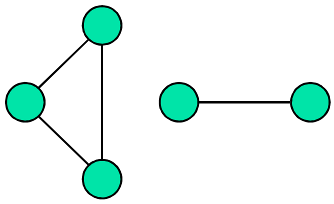
5

Connectivity

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G



Connected graph



Non connected graph with two connected components

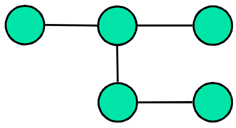
6

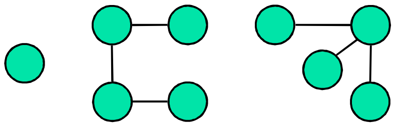
6

Trees and Forests

- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles

  This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees


Tree

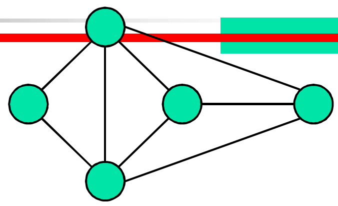

Forest

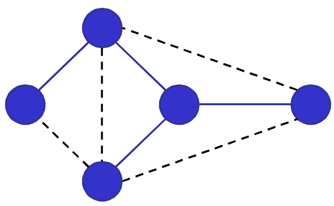
7

7

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest


Graph


Spanning tree

8

8

Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph
- Depth-first search is to graphs what Euler tour is to binary trees

9

9

DFS Algorithm

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

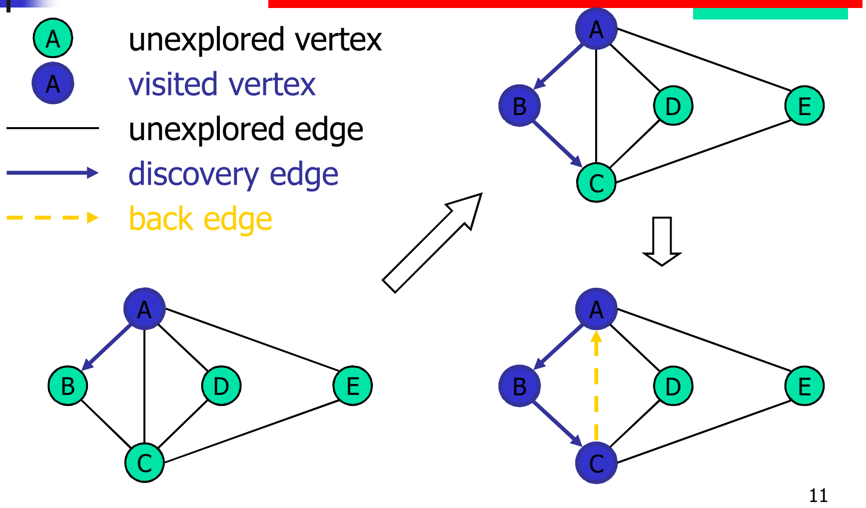
```
Algorithm DFS(G)
    Input graph G
    Output labeling of the edges of G
        as discovery edges and
        back edges
{ for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED);
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED);
  for all v ∈ G.vertices()
    if ( getLabel(v) = UNEXPLORED )
        DFS(G, v);
}
```

```
Algorithm DFS(G, v)
  Input graph G and a start vertex v of G
  Output labeling of the edges of G
     in the connected component of v
     as discovery edges and back edges
{ setLabel(v, VISITED);
  for all e ∈ G.incidentEdges(v)
    if ( getLabel(e) = UNEXPLORED )
      { w = opposite(v, e);
        if ( getLabel(w) = UNEXPLORED )
          { setLabel(e, DISCOVERY);
            DFS(G, w);
          }
        else
            setLabel(e, BACK);
      }
}
```

10

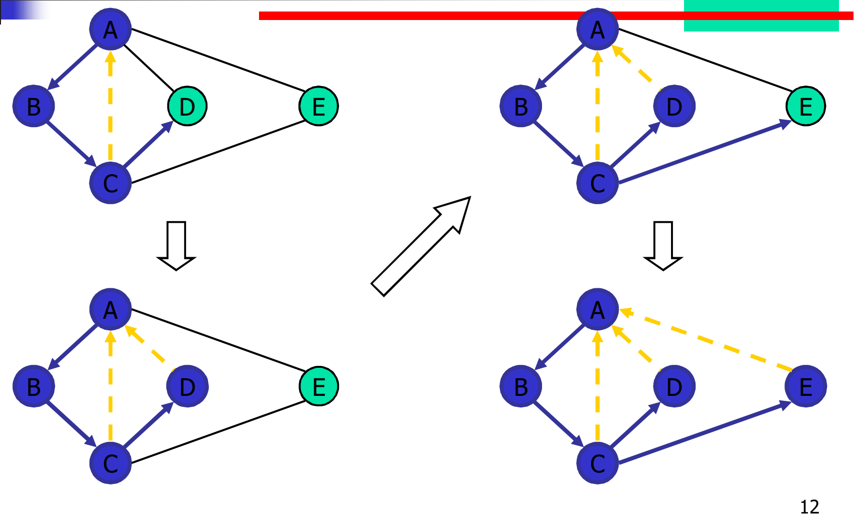
10

Example (1/2)



11

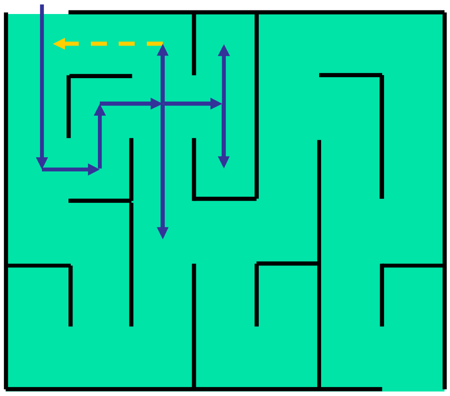
Example (2/2)



12

DFS and Maze Traversal

- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



13

13

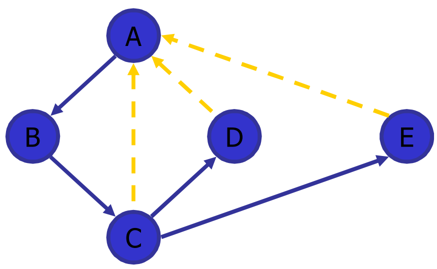
Properties of DFS

Property 1

DFS*(*G*, *v*)** visits all the vertices and edges in the connected component of ***v

Property 2

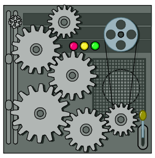
The discovery edges labeled by ***DFS*(*G*, *v*)** form a spanning tree of the connected component of ***v***



14

14

Analysis of DFS



- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

15

15

Path Finding

- We can specialize the DFS algorithm to find a path between two given vertices v and z using the template method pattern
- We call $DFS(G, v)$ with v as the start vertex
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex z is encountered, we return the path as the contents of the stack

```
Algorithm pathDFS(G, v, z)
 { setLabel(v, VISITED);
    S.push(v);
    if ( v = z )
       return S.elements();
    for all e ∈ G.incidentEdges(v)
       if ( getLabel(e) = UNEXPLORED )
         { w = opposite(v,e);
            if ( getLabel(w) = UNEXPLORED )
              { setLabel(e, DISCOVERY);
                 S.push(e);
                 pathDFS(G, w, z);
                 S.pop();
              }
            else
               setLabel(e, BACK);
         }
     S.pop();
 }
```

16

16

Cycle Finding

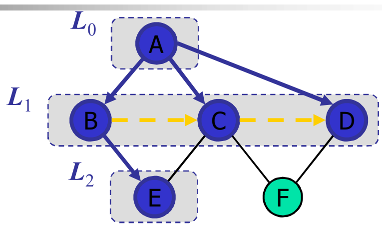
- We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

```
Algorithm cycleDFS(G, v)
  { setLabel(v, VISITED);
    S.push(v);
    for all  e ∈ G.incidentEdges(v)
      if ( getLabel(e) = UNEXPLORED )
        { w = opposite(v,e);
          S.push(e);
          if ( getLabel(w) = UNEXPLORED )
            { setLabel(e, DISCOVERY);
              cycleDFS(G, w);
              S.pop(); }
          else
            { T = new empty stack
              repeat
                { o = S.pop();
                  T.push(o); }
              until ( o = w );
              return T.elements(); }
    S.pop();
  }
```

17

17

Breadth-First Search



18

18

Breadth-First Search

- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- BFS on a graph with n vertices and m edges takes $O(n + m)$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

19

19

BFS Algorithm

The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

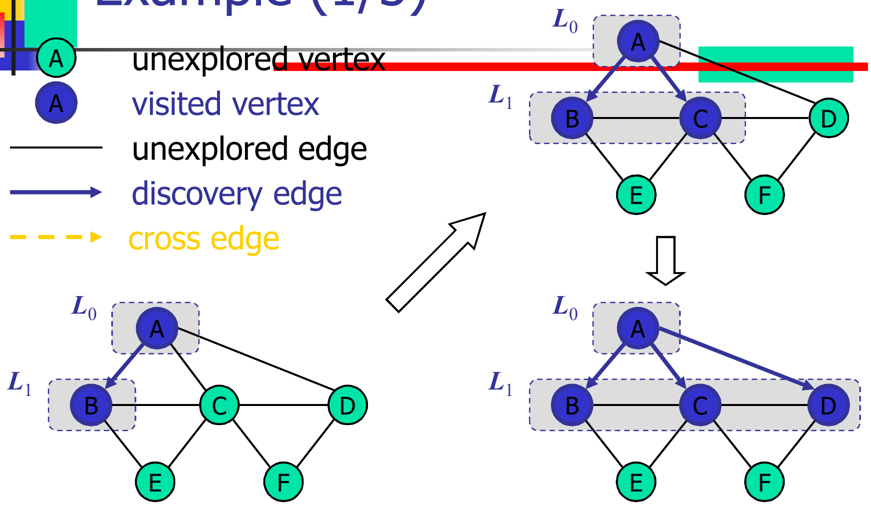
```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  {
    for all u ∈ G.vertices()
      setLabel(u, UNEXPLORED);
    for all e ∈ G.edges()
      setLabel(e, UNEXPLORED);
    for all v ∈ G.vertices()
      if ( getLabel(v) = UNEXPLORED )
        BFS(G, v);
  }
```

```
Algorithm BFS(G, s)
 { L0 = new empty sequence;
   L0.insertLast(s);
   setLabel(s, VISITED);
   i = 0;
   while ( ¬Li.isEmpty() )
    { Li+1 = new empty sequence;
      for all v ∈ Li.elements()
        for all e ∈ G.incidentEdges(v)
          if ( getLabel(e) = UNEXPLORED )
           { w = opposite(v,e);
             if ( getLabel(w) = UNEXPLORED )
              { setLabel(e, DISCOVERY);
                setLabel(w, VISITED);
                Li+1.insertLast(w); }
             else
                setLabel(e, CROSS);
           }
      i = i +1;
    }
 }
```

20

20

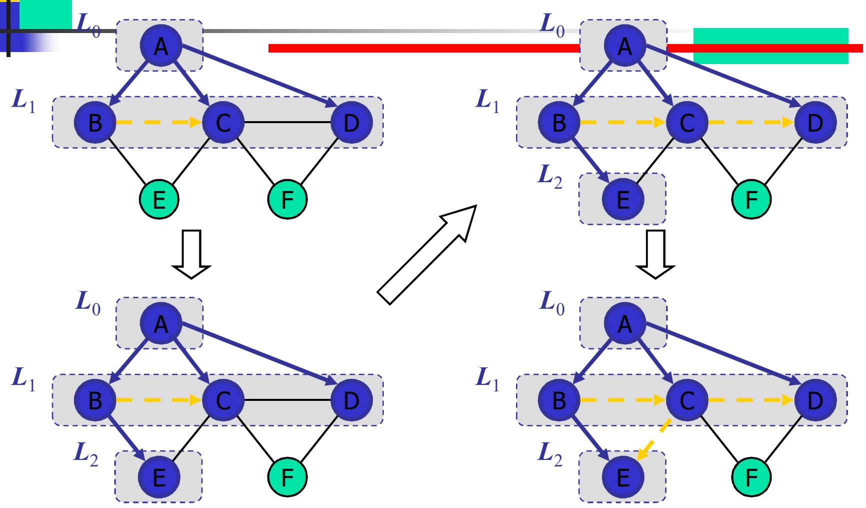
Example (1/3)



21

21

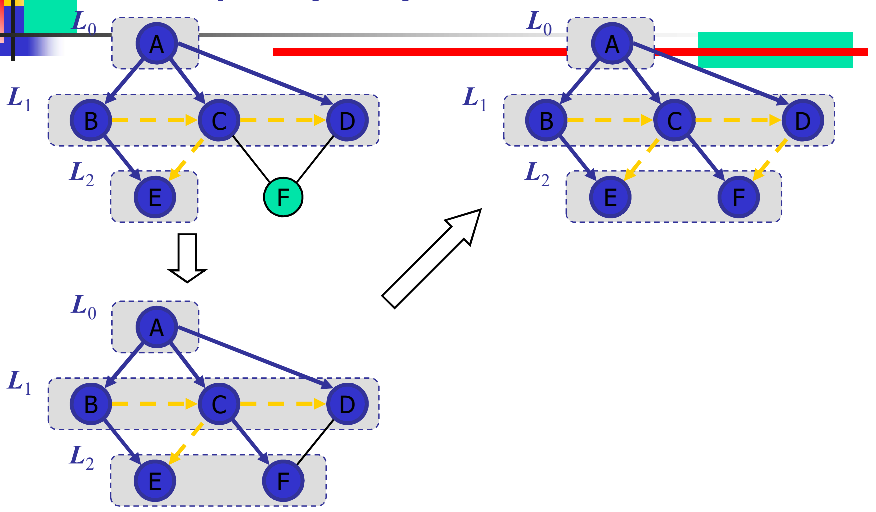
Example (2/3)



22

22

Example (3/3)



Properties

Notation

G_s: connected component of s

Property 1

$BFS(G, s)$ visits all the vertices and edges of G_s

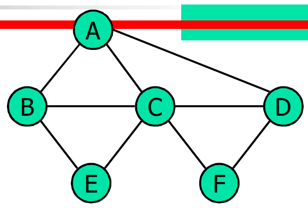
Property 2

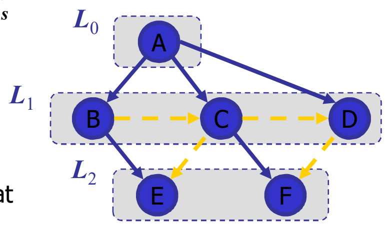
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

Property 3

For each vertex v in L_i

- The path of T_s from s to v has i edges
- Every path from s to v in G_s has at least i edges





Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges is called once for each vertex
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

25

25

Applications

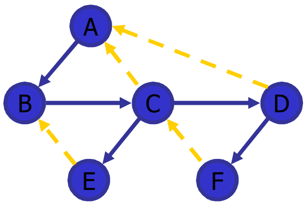
- Using the template method pattern, we can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, find a path in G between them with the minimum number of edges, or report that no such path exists

26

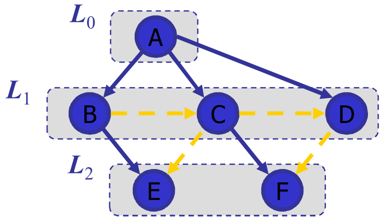
26

DFS vs. BFS (1/2)

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	



DFS



BFS

27

27

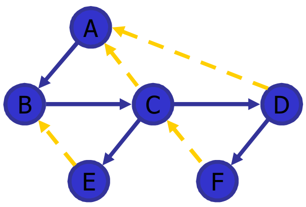
DFS vs. BFS (2/2)

Back edge (v,w)

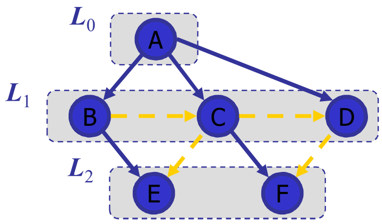
- w is an ancestor of v in the tree of discovery edges

Cross edge (v,w)

- w is in the same level as v or in the next level in the tree of discovery edges



DFS

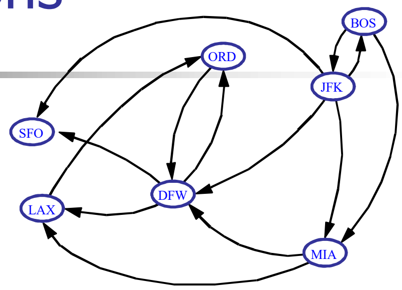


BFS

28

28

Directed Graphs

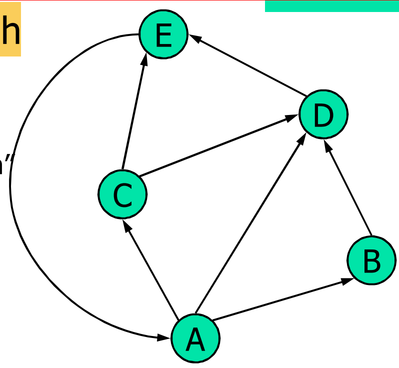


29

29

Digraphs

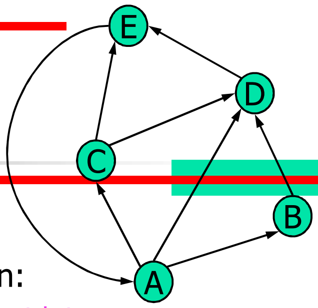
- A **digraph** is a graph whose edges are all directed
  - Short for "directed graph"
- Applications
  - one-way streets
  - flights
  - task scheduling



30

30

Digraph Properties



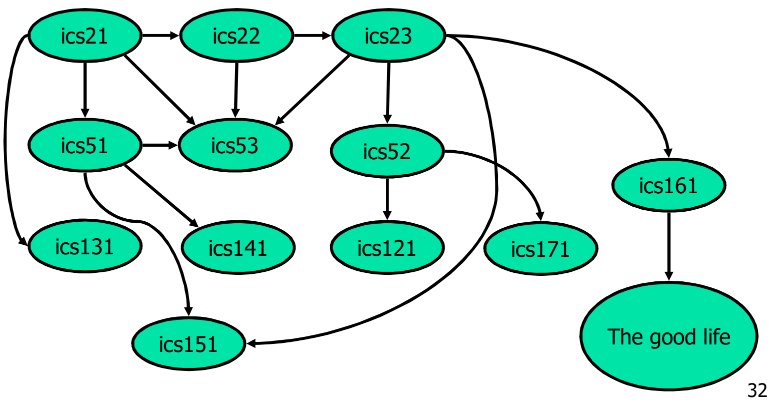
- A graph G=(V,E) such that
  - Each edge goes in one direction:
    - Edge (a,b) goes from a to b, but not b to a.
- If G is simple, m $\leq$ n*(n-1).
- If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of in-edges and out-edges in time proportional to their size.

31

31

Digraph Application

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

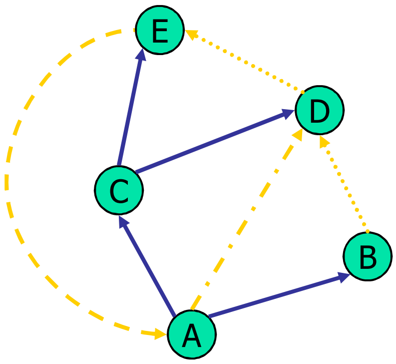


32

32

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 a a vertex s determines the vertices reachable from s

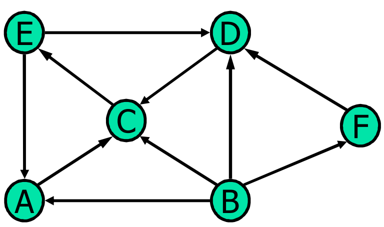


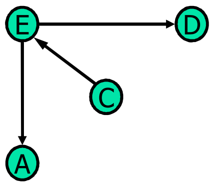
33

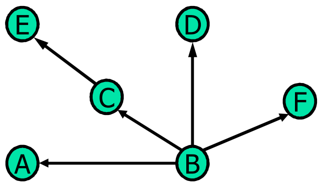
33

Reachability

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





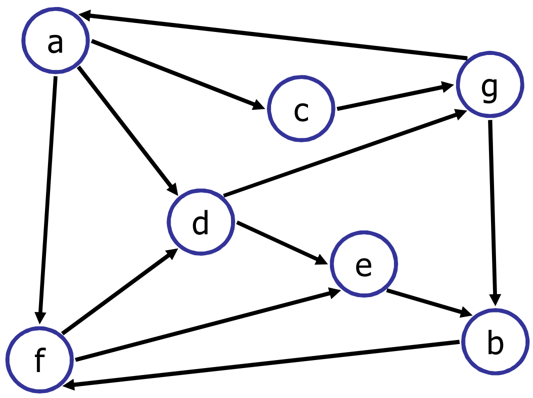


34

34

Strong Connectivity

- Each vertex can reach all other vertices



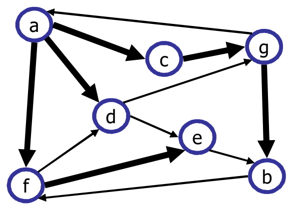
35

35

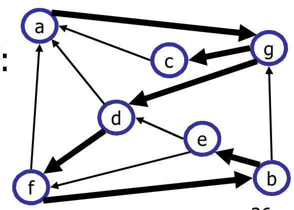
Strong Connectivity Algorithm

- 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':

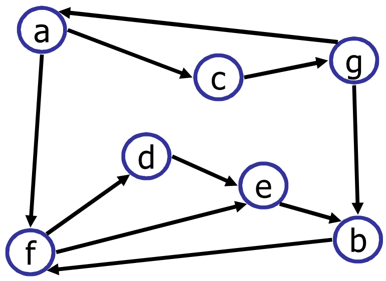


36

36

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).

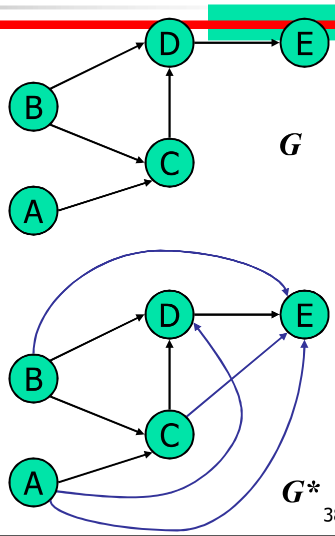


{ a , c , g }

{ f , d , e , b }

Transitive Closure

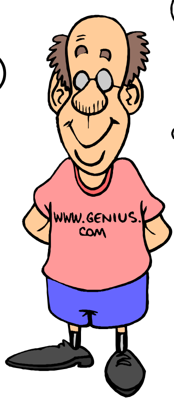
- Given a digraph G, the transitive closure of G is the digraph G^* such that
  - G^* has the same vertices as G
  - if G has a directed path from u to v ($u \neq v$), G^* has a directed edge from u to v
- The transitive closure provides reachability information about a digraph



Computing the Transitive Closure

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

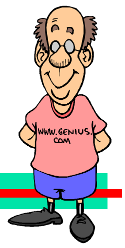
If there's a way to get from A to B and from B to C, then there's a way to get from A to C.



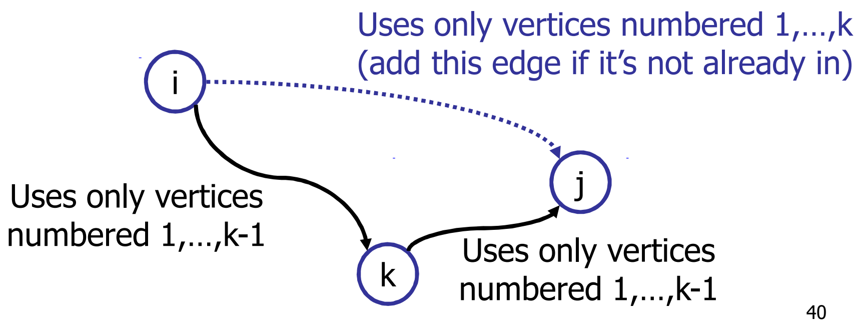
- Alternatively ... Use dynamic programming: The Floyd-Warshall Algorithm

39

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:



40

Floyd-Warshall's Algorithm

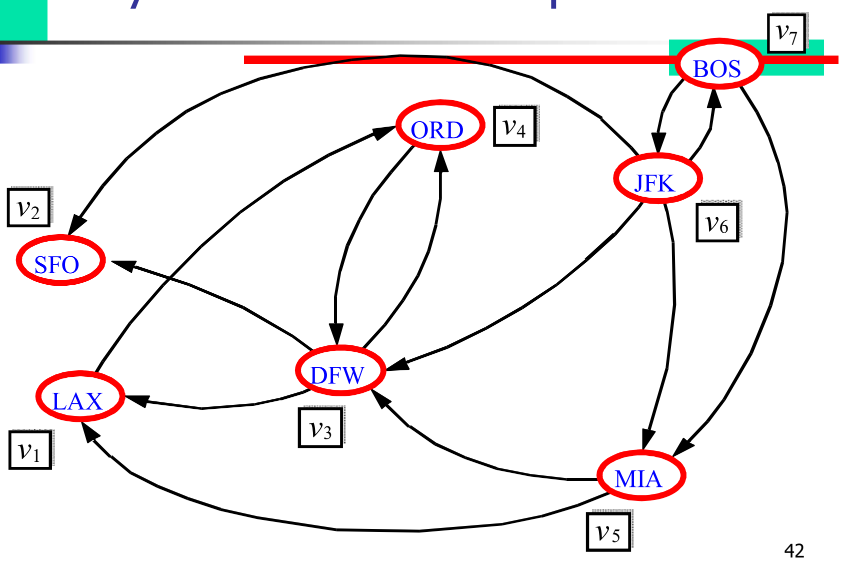
- Floyd-Warshall's algorithm numbers the vertices of G as $v_1, \ldots, v_n$ and computes a series of digraphs $G_0, \ldots, G_n$
  - $G_0 = G$
  - G_k has a directed edge (v_i, v_j) if G has a directed path from v_i to v_j with intermediate vertices in the set $\{v_1, \ldots, v_k\}$
- We have that $G_n = G^*$
- In phase k, digraph G_k is computed from G_{k-1}
- Running time: O(n³), 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 $v \in G.vertices()$
    { 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 $(i \neq k)$ **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 $\neg G_k.areAdjacent(v_i, v_j)$
              $G_k.insertDirectedEdge(v_i, v_j)$;
    }
  return G_n
}

41

41

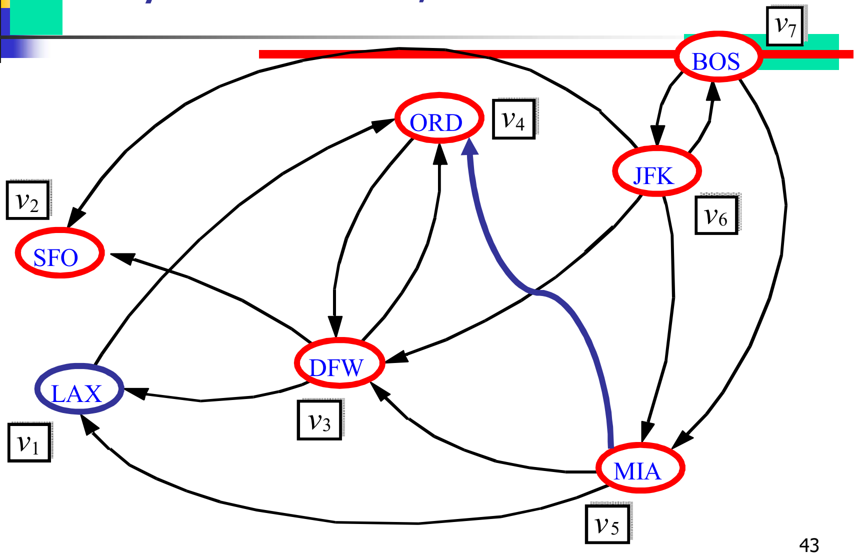
Floyd-Warshall Example



42

42

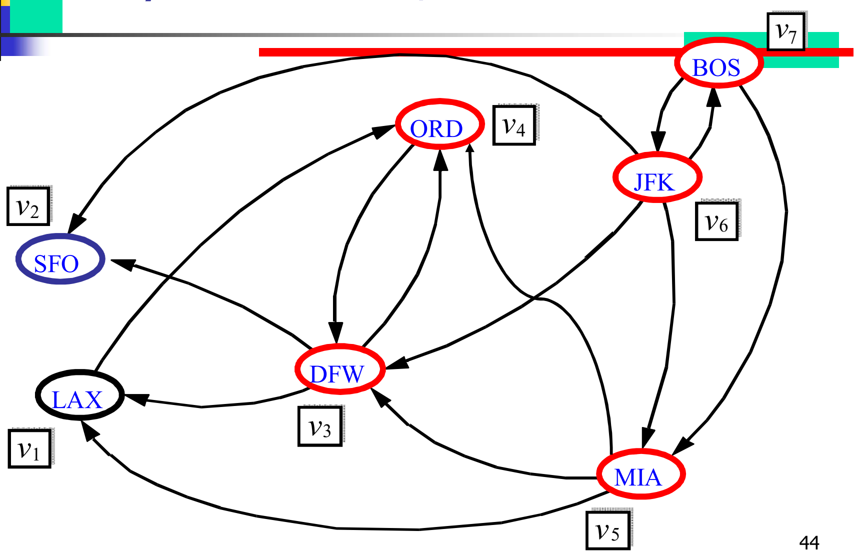
Floyd-Warshall, Iteration 1



43

43

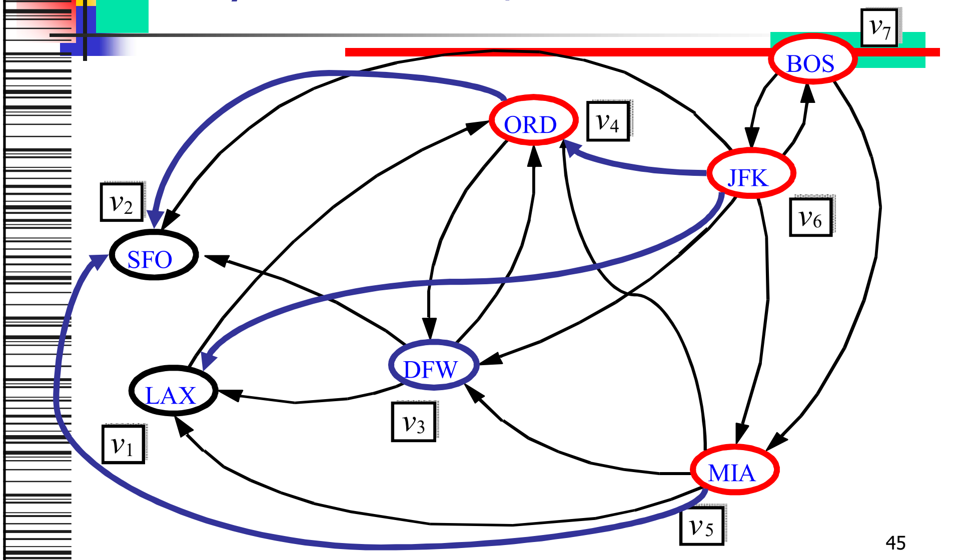
Floyd-Warshall, Iteration 2



44

44

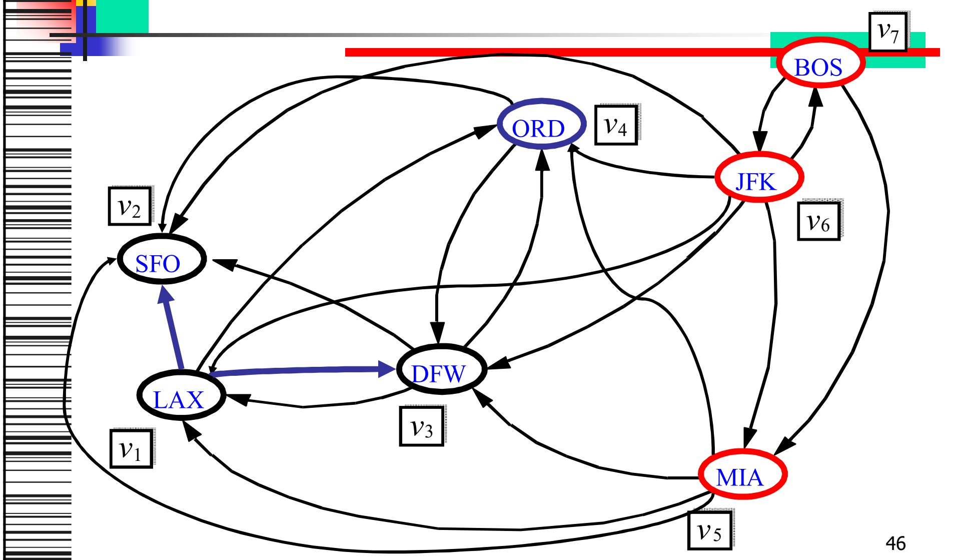
Floyd-Warshall, Iteration 3



45

45

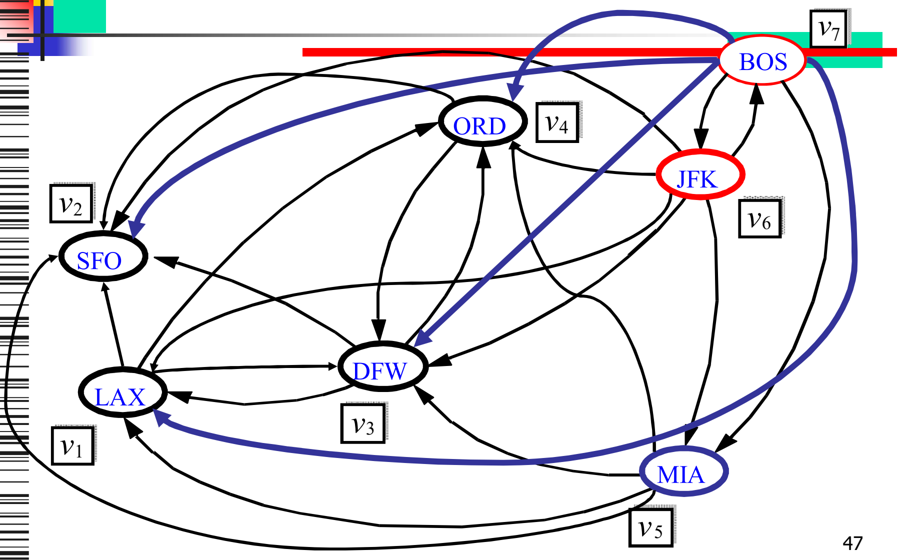
Floyd-Warshall, Iteration 4



46

46

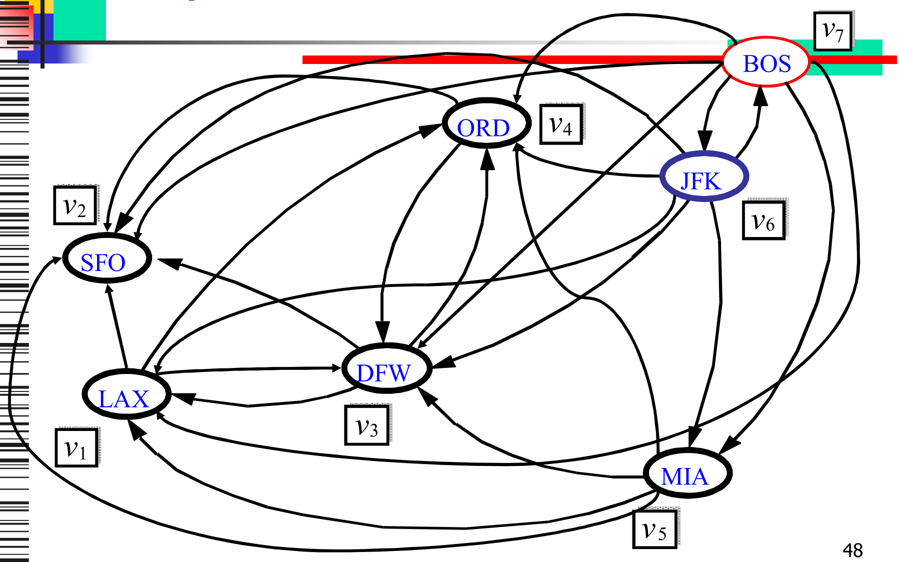
Floyd-Warshall, Iteration 5



47

47

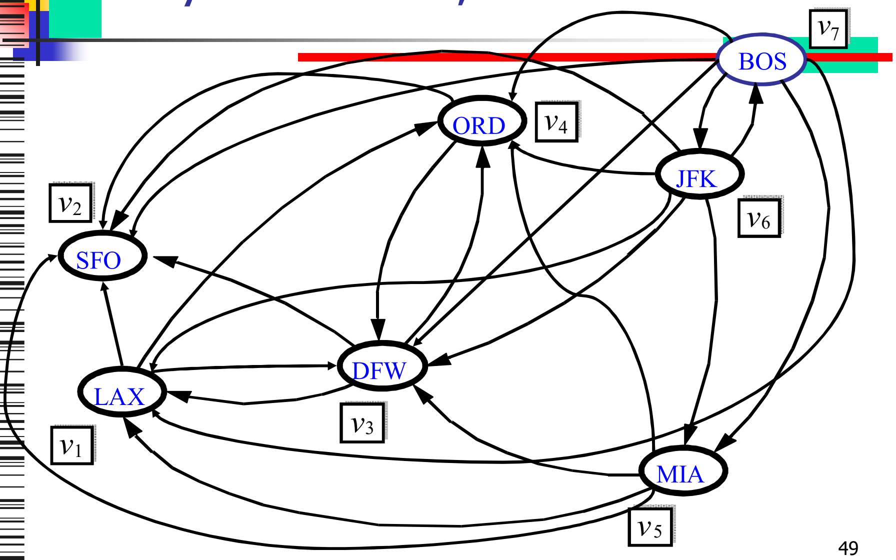
Floyd-Warshall, Iteration 6



48

48

Floyd-Warshall, Conclusion



49

49

DAGs and Topological Ordering

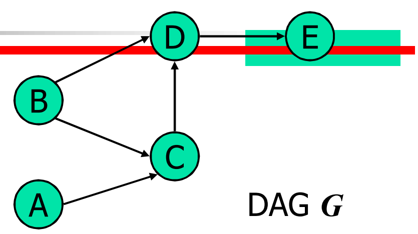
- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

  $v_1, \ldots, v_n$

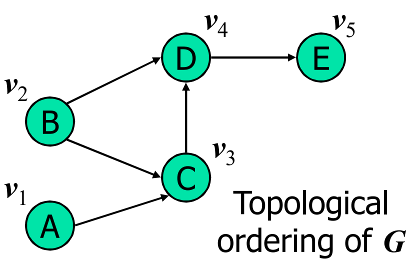
  of the vertices such that for every edge (v_i, v_j), we have $i < j$
- Example: in a task scheduling digraph, a topological ordering is a task sequence that satisfies the precedence constraints

Theorem

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



DAG G



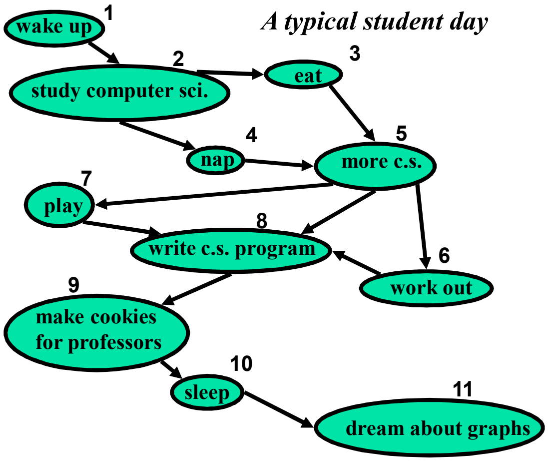
Topological ordering of G

50

50

Topological Sorting

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



51

51

Algorithm for Topological Sorting

```
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…?

52

52

Topological Sorting Algorithm using DFS

- Simulate the algorithm by using depth-first search

Algorithm *topologicalDFS*(*G*)
  Input dag *G*
  Output topological ordering of *G*
{ *n* = *G.numVertices*();
  for all *u* ∈ *G.vertices*()
    setLabel(*u, UNEXPLORED*);
  for all *e* ∈ *G.edges*()
    setLabel(*e, UNEXPLORED*);
  for all *v* ∈ *G.vertices*()
    if (*getLabel*(*v*) = *UNEXPLORED*)
      topologicalDFS(*G, v*);
}

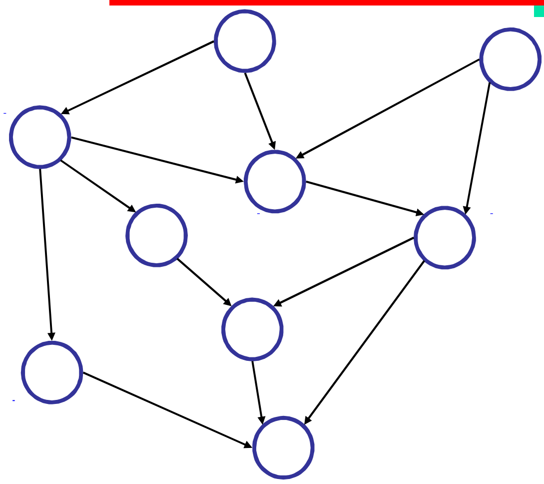
Algorithm *topologicalDFS*(*G, v*)
  Input graph *G* and a start vertex *v* of *G*
  Output labeling of the vertices of *G* in the connected component of *v*
{ *setLabel*(*v, VISITED*);
  for all *e* ∈ *G.incidentEdges*(*v*)
    if (*getLabel*(*e*) = *UNEXPLORED*)
      { *w* = *opposite*(*v,e*);
        if (*getLabel*(*w*) = *UNEXPLORED*)
          { *setLabel*(*e, DISCOVERY*);
            topologicalDFS(*G, w*); }
      }
  Label *v* with topological number ***n;***
  n **=** ***n*** **– 1;**
  return;
}

- O(n+m) time.

53

53

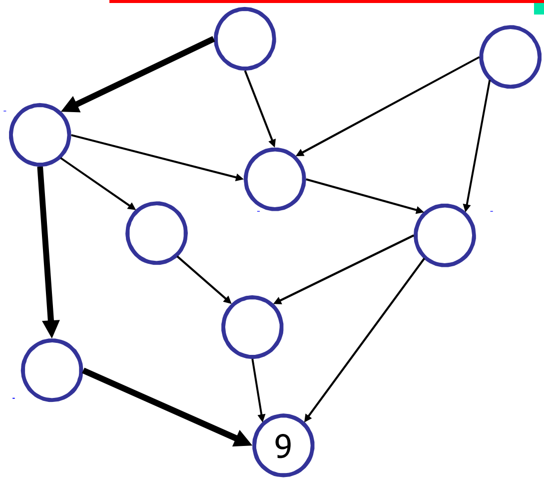
Topological Sorting Example (1/10)



54

54

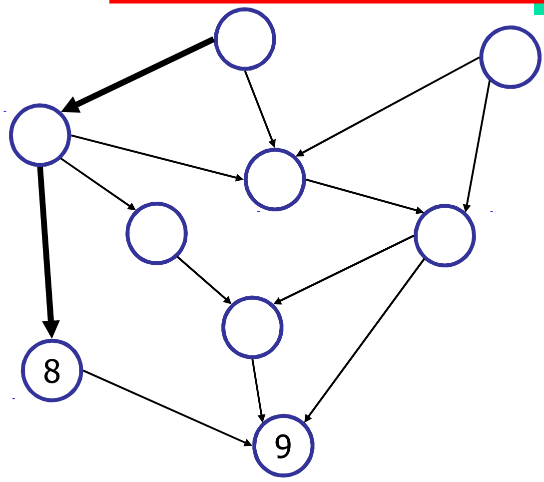
Topological Sorting Example (2/10)



55

55

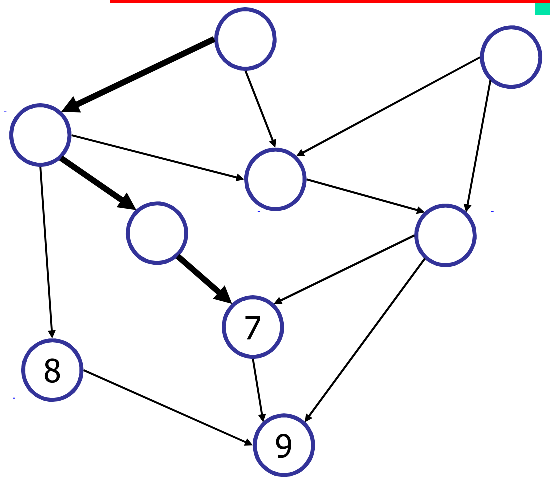
Topological Sorting Example (3/10)



56

56

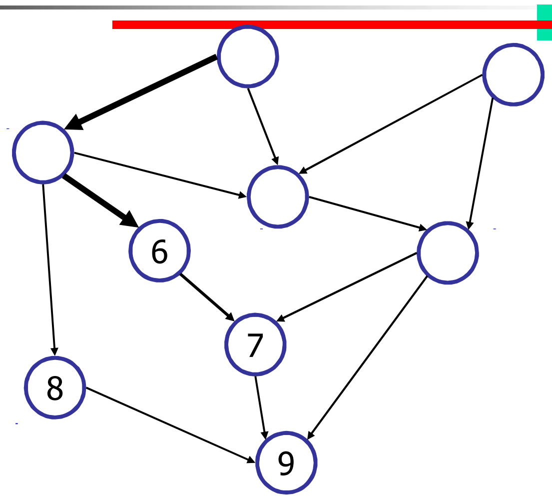
Topological Sorting Example (4/10)



57

57

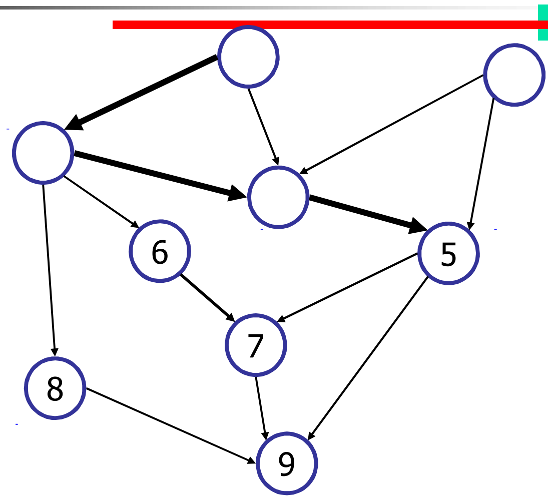
Topological Sorting Example (5/10)



58

58

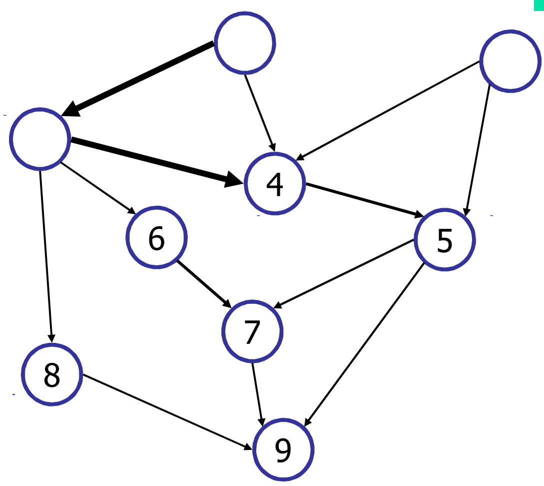
Topological Sorting Example (6/10)



59

59

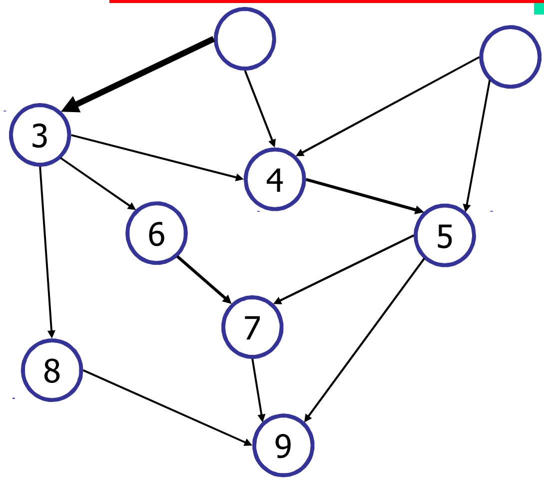
Topological Sorting Example (7/10)



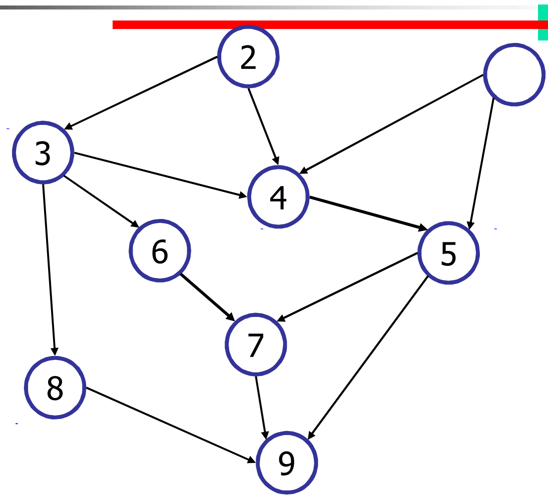
60

60

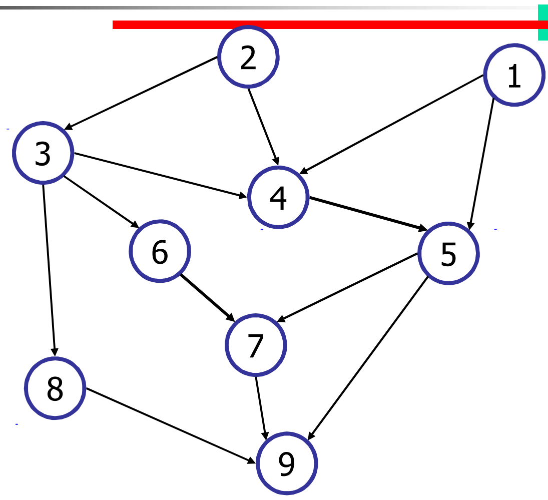
Topological Sorting Example (8/10)



Topological Sorting Example (9/10)



Topological Sorting Example (10/10)



Summary

- Depth-First Search
- Breadth-First Search
- Transitive Closure
- Topological Sorting
- Suggested reading (Sedgewick):
  - Ch.18
  - Ch.19