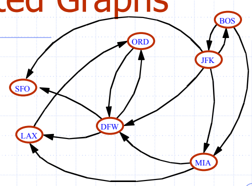
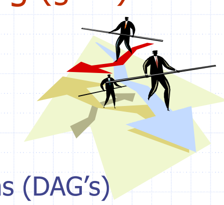
DFS on Directed Graphs



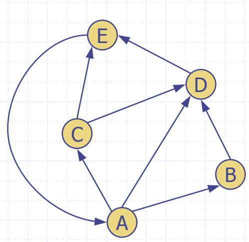
Outline and Reading (§6.4)



- Reachability (§6.4.1)
  - Directed DFS
  - Strong connectivity
- Directed Acyclic Graphs (DAG's) (§6.4.4)
  - Topological Sorting

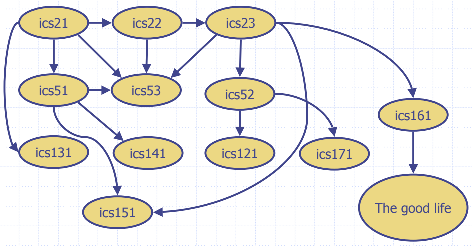
Digraphs

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



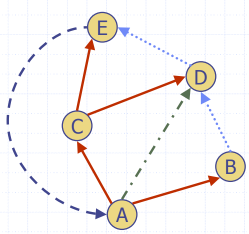
Digraph Application

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

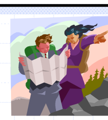


Directed DFS

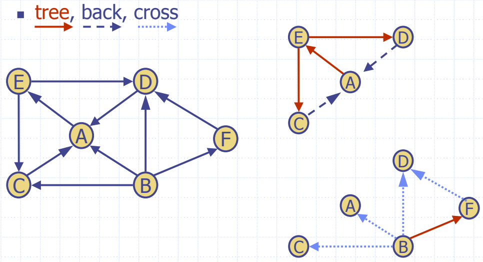
- DFS on digraphs traverses edges only along their proper direction
- In the directed DFS algorithm, we have four types of edges
  - discovery edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex *s* determines the vertices reachable from *s*



Directed DFS example



- DFS_Sweep starts at A, then B,...
  - tree, back, cross



DAGs and Topological Ordering

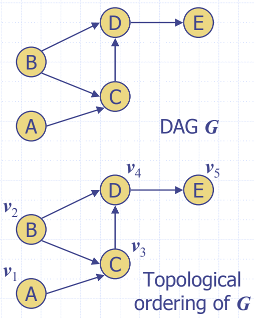
- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

  $v_1, \ldots, v_n$

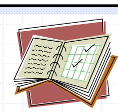
  of the vertices such that for every edge (v_i, v_j), we have $i < j$
- Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints

Theorem

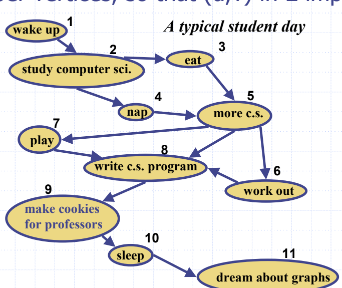
A digraph admits a topological ordering if and only if it is a DAG



Topological 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 Goodrich-Tamassia

```
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) [with smart implementation] How…?

Topological Sorting Algorithm using DFS

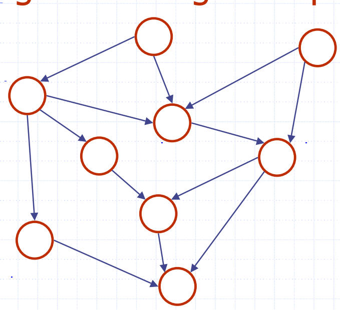
- Simulate the algorithm by using depth-first search

```
Algorithm topoDFS_Sweep(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)
```

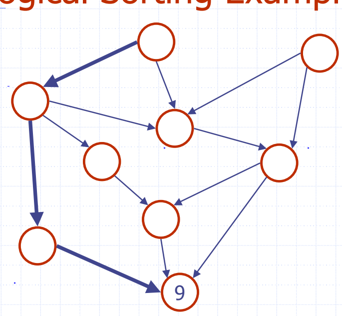
- O(n+m) time.

```
Algorithm topologicalDFS(G, v)
  Input graph G and a start vertex v of G
  Output a labeling of the vertices of G
    in the connected component of v in
    topological order
  setLabel(v, VISITED)
  for all e ∈ G.outIncidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        topologicalDFS(G, w)
      else
        {e is a forward or cross edge}
        setLabel(e, VISITED)
  Label v with topological number n
  n ← n - 1
```

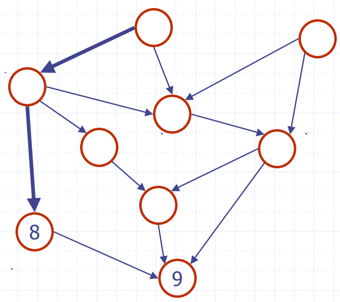

Topological Sorting Example



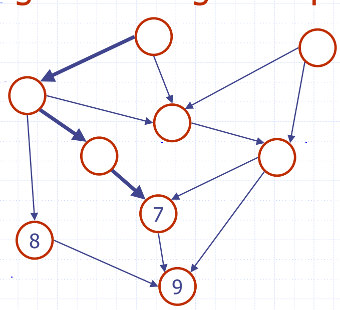
Topological Sorting Example



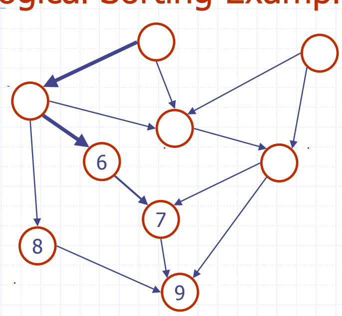
Topological Sorting Example



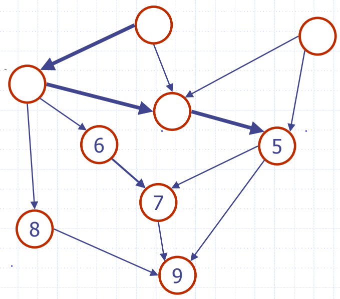
Topological Sorting Example



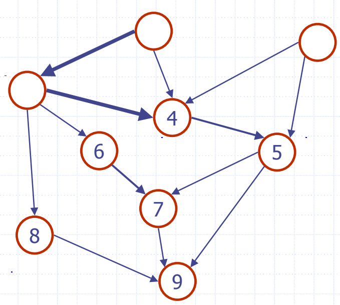
Topological Sorting Example



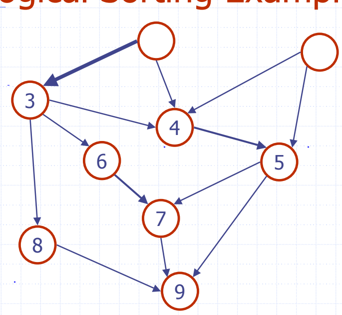
Topological Sorting Example



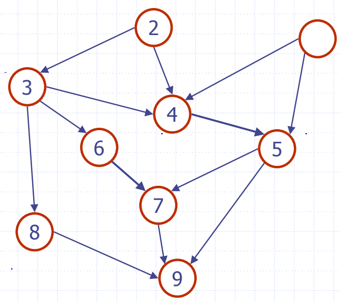
Topological Sorting Example



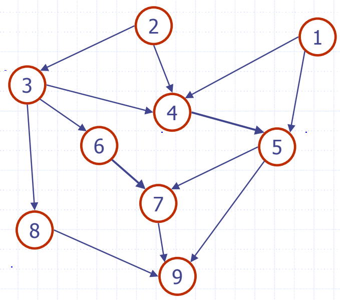
Topological Sorting Example



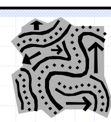
Topological Sorting Example



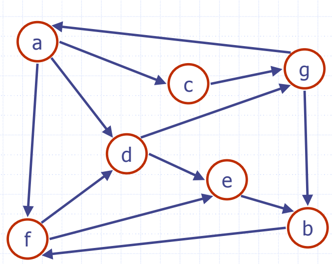
Topological Sorting Example



Strong Connectivity

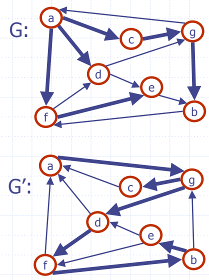


- Each vertex can reach all other vertices



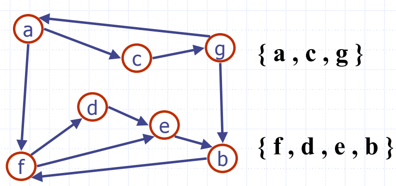
Strong Connectivity Algorithm

- 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).



Strongly Connected Components

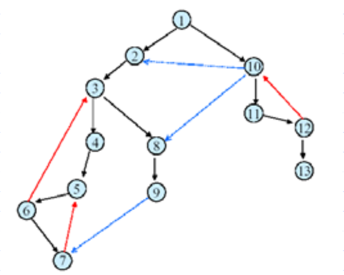
- Maximal subgraphs such that each vertex can reach all other vertices in the subgraph
- Can be computed in O(n+m) time using DFS



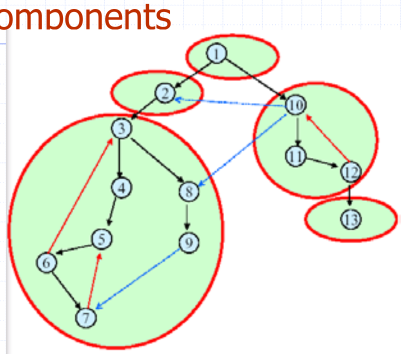
SCC algorithm

- Using DFS_Sweep for directed graphs, construct list L of reverse finish order of the vertices in the traversal.
  - Node is finished when traversal leaves it permanently.
- Do another DFS_Sweep on G^R, (G with edges reversed), with the following modification: in DFS_Sweep outer loop, start DFS calls on vertices according to the order in list L.
- Each spanning tree produced by DFS_Sweep on G^R will contain all nodes from exactly one SCC of G

Strongly Connected Components



Strongly Connected Components



SCC Algorithm, more detail

- // Phase 1
  Run DFS_Sweep on G, returning a list L of nodes in reverse finish order. Done by adding vertex v to the front of L after traversal on v is finished in DFS_Sweep.
- // Phase 2a
  Construct G^R from G by copying the vertices, and then adding the reverse of every edge from G to G^R.
- // Phase 2b
  Do a modified DFS_Sweep traversal on G^R, where list L is used to order the DFS calls. Each DFS call labels vertices traversed with a different SCC number.
- // Final Phase:
  Label vertices and edges of G.

DFS Phase 1

- Construct list L
- Similar to topological sort

Algorithm *SCC1DFS_Sweep(G)*
  Input dag *G*
  Output list *L* of vertices of G in reverse finish order.
  L ← empty list
  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*
      SCC1DFS(*G, v*)

- O(n+m) time.

Algorithm *SCC1DFS(G, v)*
  Input graph *G* and a start vertex *v* of *G*
  Output vertices of *G* in the connected component of *v* added to *L*, according to reverse finish order
  setLabel(*v, VISITED*)
  for all *e* ∈ *G.outIncidentEdges*(*v*)
    if *getLabel*(*e*) = *UNEXPLORED*
      w ← *opposite*(*v,e*)
      if *getLabel*(*w*) = *UNEXPLORED*
        setLabel(*e, DISCOVERY*)
        SCC1DFS(*G, w*)
      else
        {*e* is a forward or cross edge}
  L.insertFirst(*v*)

DFS Phase 2b

- Similar to Connected Components

Algorithm *SCC2bDFS_Sweep*(G^R*,L*)
  Input dag G^R, list *L*
  Output Labeling of vertices in G^R by scc component number
  sccNum←1
  for all *u* ∈ *G.vertices*()
    setLabel(*u, UNEXPLORED*)
  for all *e* ∈ *G.edges*()
    setLabel(*e, UNEXPLORED*)
  for all *v* ∈ *L*, {traverse *L* in order}
    if *getLabel*(*v*) = *UNEXPLORED*
      SCC2bDFS(*G, v, sccNum*)
      sccNum++

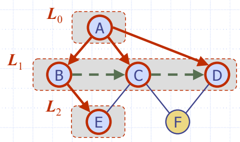
- O(n+m) time.

Algorithm *SCC2bDFS*(G^R*, v, sccNum*)
  Input graph G^R and a start vertex *v*
  Output vertices of G^R in the connected component of *v* labeled by sccNum
  setLabel(*v, VISITED*)
  Label *v* with **sccNum**
  for all *e* ∈ *G.outIncidentEdges*(*v*)
    if *getLabel*(*e*) = *UNEXPLORED*
      w ← *opposite*(*v,e*)
      if *getLabel*(*w*) = *UNEXPLORED*
        setLabel(*e, DISCOVERY*)
        SCC2bDFS(*G, w*)
      else
        {*e* is a forward or cross edge}

Correctness of SCC algorithm

- Lemma 1: In terms of vertices, SCC's of G are the same as the SCC's of G^R.
- Lemma 2: For graph G, let F be a forest generated by DFS_Sweep on G. Let S be a tree of F. Then S contains one or more complete SCC's of G. (No partial SCC's).
- Lemma 3a: Let F be the forest generated by SCC phase 2b, and S be a spanning tree in F. Let x be the root of S, and v be a descendent of x. Then there is a path from v to x in G^R.
- Lemma 3b: Let S be as in Lemma 3a. S combined with other edges in G^R form a strongly connected subgraph of G^R.

Breadth-First Search



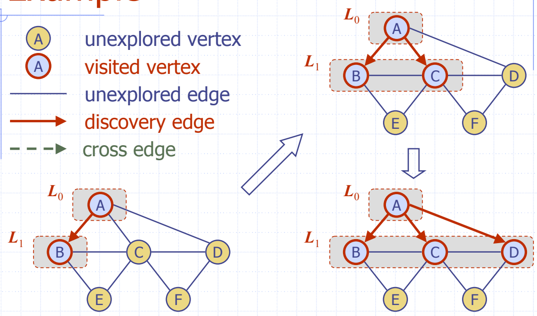
Outline and Reading

- Breadth-first search (§6.3.3)
  - Algorithm
  - Example
  - Properties
  - Analysis
  - Applications
- DFS vs. BFS (§6.3.3)
  - Comparison of applications
  - Comparison of edge labels

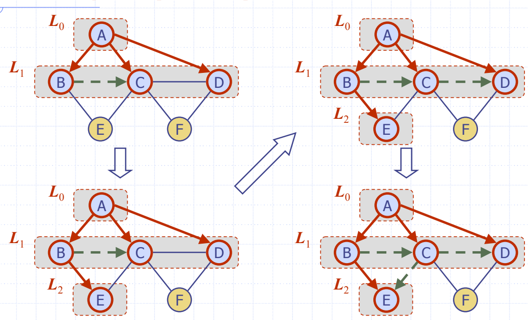
Breadth-First Search

- Breadth-first search (BFS) is
  - general graph traversal technique
  - Visits all the vertices and edges
  - with n vertices and m edges takes $O(n + m)$ time
  - Like searching a binary tree level by level
- A BFS can
  - Determine whether G is connected
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

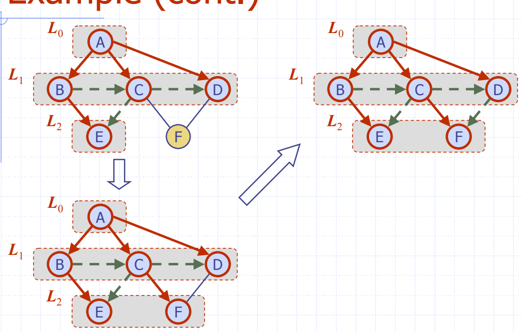
Example



Example (cont.)



Example (cont.)



BFS Algorithm

- The algorithm uses a queue to keep track of vertices

```
Algorithm BFS_Sweep(G)
   Input graph G
   Output labeling of the edges
      and 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)
  Q ← new empty queue
  Q.enqueue(s)
  setLabel(s, VISITED)
  while ¬Q.isEmpty()
     v ← Q.dequeue()
     for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
           w ← opposite(v,e)
           if getLabel(w) = UNEXPLORED
              setLabel(e, DISCOVERY)
              setLabel(w, VISITED)
              Q.enqueue(w)
           else
              setLabel(e, CROSS)
```


Properties

Notation

G_s: connected component of s

L_i: nodes at depth i in BFS tree.

Property 1

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

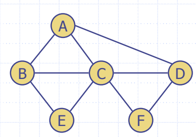
Property 2

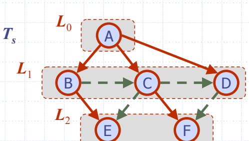
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s; T_s called BFS tree

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

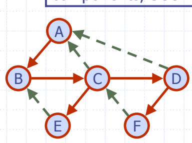
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into Q
- Inner loop of BFS runs in $O(\deg(v))$ time
- BFS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

Applications

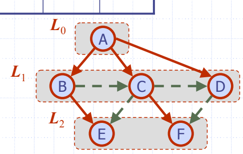
- Can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, find a minimum length path in G (if it exists)

DFS vs. BFS

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



DFS



BFS

DFS vs. BFS (cont.)

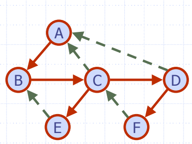
On undirected graphs

Back edge (v,w)

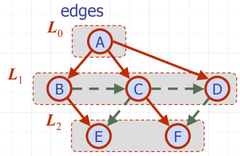
- w is an ancestor of v in the tree of discovery edges

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